Web-mjesto kao višeagentni sustav

Koprek, Zvonimir

Master's thesis / Diplomski rad

2024

Degree Grantor / Ustanova koja je dodijelila akademski / stručni stupanj: **University of Zagreb, Faculty of Organization and Informatics / Sveučilište u Zagrebu, Fakultet organizacije i informatike**

Permanent link / Trajna poveznica: https://urn.nsk.hr/urn:nbn:hr:211:366542

SVEUČILIŠTE U ZAGREBU
FAKULTET ORGANIZACIJE I INFORMATIKE
VARAŽDIN

Zvonimir Koprek

WEB MJESTO KAO VIŠEAGENTNI SUSTAV

DIPLOMSKI RAD

Varaždin, 2024.

SVEUČILIŠTE U ZAGREBU

FAKULTET ORGANIZACIJE I INFORMATIKE

V A R A Ž D I N

Zvonimir Koprek

Matični broj: 0016135996

Studij: Organizacija poslovnih sustava

WEB MJESTO KAO VIŠEAGENTNI SUSTAV

DIPLOMSKI RAD

Mentor:

doc. dr. sc. Bogdan Okreša Đurić

Varaždin, rujan 2024.

Zvonimir Koprek

Izjava o izvornosti

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

Autor potvrdio prihvaćanjem odredbi u sustavu FOI Radovi

Sažetak

U ovom diplomskom radu srž je izgradnja aplikacije koja se može podijeliti u dva dijela. Dio klijenta zadužen je za uspostavu web-stranice s podacima prikupljenima iz višeagentnog sustava u dijelu poslužitelja. U uvodnom dijelu rada donosi se kratak pregled trenutnog stanja u svijetu tehnologije i ideje korištenja agenata kao pomoć pri izvedbi platformi na webu. U teorijskom dijelu rada predstavljaju se korišteni alati i tehnologije prilikom izrade projekta te osnovni pojmovi domene višeagentnih sustava. U praktičnom dijelu se opisuje i analizira izrađena aplikacija te se referenciraju dijelovi iz teorije za lakše razumijevanje. Zaključak je zadužen za zaokruživanje cijele priče te iznošenje dojmova autora rada prilikom njegovog sastavljanja, ponajviše u praktičnom dijelu.

Ključne riječi: Višeagentni sustav, okruženje, SPADE, Python, Flask, React, web mjesto, web-stranica

Sadržaj

1. Uvod

Brzi napredak web-tehnologija transformirao je način na koji pojedinci, poduzeća i organizacije komuniciraju, surađuju i posluju putem interneta. Kako se internet nastavlja razvijati, raste potreba za dinamičnim, inteligentnim i prilagodljivim sustavima koji mogu zadovoljiti sve složenije zahtjeve korisnika. Jedan od pristupa koji je stekao popularnost u posljednjih nekoliko godina je razvoj web-platformi kao višeagentnih sustava (engl. *multi-agent systems - MAS*). Višeagentni sustavi čine autonomne, međusobno djelujuće agente koji rade zajedno ili se natječu kako bi postigli individualne ili kolektivne ciljeve. U kontekstu okruženja na internetu, ti agenti mogu predstavljati korisnike, usluge, izvore podataka ili čak apstraktne entitete, doprinoseći fleksibilnijoj, skalabilnijoj i robusnijoj arhitekturi sustava. Ovaj rad istražuje koncept dizajna i implementacije web mjesta kao višeagentnog sustava, ističući potencijalne prednosti, izazove i primjene takvog pristupa.

Web-platforma dizajnirana kao višeagentni sustav može se konceptualizirati kao ekosustav agenata, od kojih svaki ima specifične uloge, sposobnosti i ciljeve. Ovi agenti mogu predstavljati različite entitete, kao što su korisnička sučelja, moduli za obradu podataka, vanjske usluge ili čak cijele aplikacije. Iskorištavanjem principa autonomije, komunikacije i suradnje, ovi agenti mogu raditi zajedno kako bi ispunili zahtjeve web-platforme, bilo da se radi o obradi korisničkih zahtjeva, upravljanju protokom podataka ili koordinaciji složenih zadataka. Decentralizirana priroda višeagentnog sustava omogućuje poboljšanu skalabilnost jer se novi agenti mogu dodati ili ukloniti prema potrebi, bez ometanja ukupnog sustava. Štoviše, sposobnost agenata da se prilagode promjenama u okruženju ili ponašanju korisnika poboljšava odaziv i otpornost platforme.

Cilj ovog rada je doprinijeti istraživanjima o višeagentnim sustavima testirajući njihovu primjenu u okruženjima na internetu. Pružit će se detaljna analiza principa dizajna, strategija implementacije i potencijalnih primjena web-platformi kao višeagentnih sustava. Na taj način, ovaj rad nastoji demonstrirati izvedivost i prednosti ovog pristupa, nudeći uvide i praktična rješenja za programere, istraživače i početnike u ovom području. U konačnici, ovaj rad nastoji utrti put inteligentnijim, prilagodljivijim i otpornijim web-platformama koje mogu zadovoljiti sve složenije potrebe korisnika u sve složenijem digitalnom krajoliku.

2. Metode i tehnike rada

2.1. Opis korištenih alata

Kako ovo nije malen projekt i kako je plan autora da ga se nakon upotrebe u ovom radu dalje nastavlja razvijati, koristio se relativno velik broj pomoćnih alata i tehnologija. Ovo je poglavlje u kojem se oni svi navode i ukratko predstavljaju, prvo sa strane klijenta, pa zatim i poslužitelja.

2.1.1. Klijent

2.1.1.1. Microsoft Visual Studio Code

Microsoft Visual Studio Code (VS Code) je moćan i svestran uređivač izvornog (engl. *source*) koda koji je stekao široku popularnost među programerima zbog svog robusnog skupa značajki, proširivosti i kompatibilnosti s različitim platformama. Objavljen od strane Microsofta 2015. godine, VS Code je brzo postao popularan zbog svoje lagane dizajnerske strukture i besprijekorne integracije brojnih alata koji podržavaju različite programske jezike i okvire [1].

Jedna od ključnih značajki VS Codea je bogata podrška za otklanjanje pogrešaka. On pruža integrirani alat za otklanjanje pogrešaka za različite jezike, uključujući JavaScript, Python i C#, omogućujući programerima da obavljaju zadatke poput postavljanja prijelomnih točaka, prolaska kroz kod korak po korak i pregleda varijabli bez napuštanja uređivača [2]. Ova integracija pojednostavljuje razvojni proces smanjujući potrebu za prebacivanjem između različitih alata, što je često uzrok smanjenja produktivnosti.

Proširivost VS Codea je još jedna značajna prednost. Uređivač podržava ogromno tržište proširenja, omogućujući programerima prilagodbu njihovog razvojnog okruženja prema vlastitim potrebama. Ova proširenja obuhvaćaju alate specifične za jezik, poput provjere i formatiranja koda, kao i napredne značajke poput integriranih terminalskih okruženja i Git verzioniranja [3]. Ova fleksibilnost čini VS Code pogodnim za širok raspon projekata, od jednostavnih skripti do složenih poslovnih aplikacija.

Osim toga, VS Codeova IntelliSense značajka pruža inteligentno dovršavanje koda na temelju tipova varijabli, definicija funkcija i uvezenih modula. Ova značajka značajno poboljšava učinkovitost kodiranja i smanjuje vjerojatnost pogrešaka pružanjem prijedloga u stvarnom vremenu dok programeri pišu kod. Podrška uređivača za više programskih jezika i njegova sposobnost rada na različitim operativnim sustavima, uključujući Windows, macOS i Linux, čine ga svestranim alatom za raznoliku zajednicu programera [4].

Otvoreni kod (engl. *open-source*) VS Codea također doprinosi njegovoj širokoj prihvaćenosti. Kao open-source alat, zajednica oko VS Codea aktivno sudjeluje u njegovom razvoju, doprinoseći kontinuiranom poboljšanju i dodavanju novih značajki. Ovaj kolaborativni pristup osigurava da uređivač ostaje ažuriran s najnovijim programskim trendovima i tehnologijama [5].

Microsoft Visual Studio Code je izuzetno učinkovit alat koji je postao neizostavan u

modernom programerskom arsenalu alata. Kombinacija snažnih značajki, podrške za različite platforme i aktivne zajednice čini ga neprocjenjivim resursom kako za početnike, tako i za iskusne programere.

2.1.1.2. JavaScript

Visoko-razinski, interpretirani programski jezik nazvan JavaScript trenutno je kamen temeljac u svijetu interneta. Prvotno ga je kreirao Brendan Eich 1995. godine tijekom svog rada u Netscape Communicationsu, a JavaScript je bio osmišljen kako bi omogućio dinamičan sadržaj na web-stranicama [6] . Tijekom godina, evoluirao je u jedan od najraširenijih programskih jezika na svijetu, s primjenama koje nadilaze samo skriptiranje na strani klijenta, uključujući razvoj na strani poslužitelja, razvoj mobilnih aplikacija, pa čak i desktop aplikacije.

JavaScript se integrira s web-tehnologijama kao što su HTML i CSS, što omogućuje programerima stvaranje interaktivnih i responzivnih korisničkih sučelja. Ova integracija dovela je do razvoja moćnih okvira i knjižnica poput Reacta, Angulara i Vue.js, koji pojednostavljuju proces izgradnje složenih web-aplikacija [7]. Ovi okviri pružaju strukturirane načine za upravljanje stanjem aplikacije, rukovanje korisničkim interakcijama i učinkovito ažuriranje korisničkog sučelja, čime se povećava produktivnost programera i poboljšava korisničko iskustvo.

Uvođenje Node.js-a 2009. godine označilo je značajno proširenje mogućnosti JavaScripta omogućujući mu da se pokreće na strani poslužitelja. Node.js koristi model vođen događajima s neblokirajućim ulazno-izlaznim (engl. *input-output, I/O*) operacijama, što ga čini laganim i učinkovitim za izgradnju skalabilnih mrežnih aplikacija [8]. Ovo je otvorilo vrata za korištenje JavaScripta u razvoju kompletnih rješenja (full-stack), omogućujući programerima korištenje jednog jezika i na strani klijenta i na strani poslužitelja, što može pojednostaviti razvojne procese i smanjiti složenost upravljanja različitim kodnim bazama.

Štoviše, evolucija ECMAScript (ES) standarda, koji definira specifikacije za JavaScript, igrala je ključnu ulogu u rastu ovog jezika. Značajna ažuriranja poput ES6 (poznat i kao ECMAScript 2015) uvela su značajke poput streličnih funkcija, klasa i modula, što je JavaScript približilo drugim modernim programskim jezicima i poboljšalo njegovu upotrebljivost za velike aplikacije [9].

Unatoč konkurenciji drugih jezika i tehnologija, JavaScript nastavlja svoju sveprisutnost u web-preglednicima uz svoju aktivnu i opsežnu zajednicu. Njegova prilagodljivost i stalna evolucija osiguravaju da ostaje relevantan i moćan alat za programere u različitim domenama.

2.1.1.3. HTML

Hypertext Markup Language (HTML) je temeljni jezik koji se koristi za stvaranje i strukturiranje sadržaja na World Wide Webu. Prvotno ga je predložio Tim Berners-Lee 1991. godine, a HTML pruža osnovni okvir za web-stranice definiranjem skupa elemenata koji predstavljaju različite vrste sadržaja, poput teksta, slika i poveznica [10]. Ovi elementi su definirani oznakama koje su zatvorene u uglate zagrade (npr. `<p>` za odlomak ili `<h1>` za naslov najviše razine), a koje web-pregledniku govore kako prikazati sadržaj na stranici.

Razvoj i standardizaciju HTML-a nadgleda World Wide Web Consortium (W3C), osiguravajući da HTML ostane kompatibilan s različitim preglednicima i uređajima. HTML je prošao kroz nekoliko iteracija od svog nastanka, a svaka verzija donosi nove značajke i mogućnosti kako bi zadovoljila rastuće potrebe web-razvoja. Najznačajnije ažuriranje, HTML5, službeno je objavljeno od strane W3C-a 2014. godine i predstavlja veliki iskorak u pogledu funkcionalnosti i upotrebljivosti [10].

HTML5 donosi niz novih elemenata i atributa koji poboljšavaju semantičku strukturu web-stranica, rukovanje multimedijom i omogućuju bolju izvedbu web-aplikacija. Ključne novosti uključuju elemente `<article>`, `<section>` i `<footer>`, koji pružaju smisleniju strukturu web-dokumentima, čineći ih lakšima za navigaciju i korisnicima i tražilicama [10]. Osim toga, HTML5 je uveo element `<canvas>`, koji omogućuje dinamičko, skriptabilno renderiranje 2D oblika i bitmapiranih slika, olakšavajući razvoj složenih grafičkih prikaza izravno unutar web-stranica.

Široko prihvaćanje HTML5 dovelo je do ujednačenijeg i učinkovitijeg procesa web-razvoja, smanjujući oslanjanje na tehnologije trećih strana i olakšavajući stvaranje bogatih, interaktivnih i pristupačnih web-iskustava. Kako HTML nastavlja evoluirati, onostaje ključni alat za web-programere, pružajući strukturni temelj nužan za moderne web-aplikacije i osiguravajući dosljedna korisnička iskustva na različitim platformama.

2.1.1.4. CSS

Cascading Style Sheets (CSS) je jezik za stiliziranje koji se koristi za opisivanje prikaza dokumenta napisanog u HTML-u ili XML-u. Razvili su ga Håkon Wium Lie i Bert Bosa 1996. godine, CSS omogućuje odvajanje sadržaja od dizajna, što omogućuje web-programerima da kontroliraju izgled web-stranica bez promjene osnovne HTML strukture [11]. Ovo odvajanje pojednostavljuje proces dizajniranja, poboljšava održavanje i povećava fleksibilnost web-dizajna.

CSS pruža niz mogućnosti za stiliziranje, uključujući kontrolu rasporeda, upravljanje bojama, odabir fontova i responzivni dizajn. Stilovi se primjenjuju putem selektora koji ciljaju specifične HTML elemente i primjenjuju definirana pravila na njih. Naprimjer, CSS pravilo može promijeniti veličinu fonta svih `<h1>` elemenata ili postaviti boju pozadine `<div>` elementa [12]. Kaskadna priroda CSS-a znači da se stilovi mogu nasljeđivati od roditeljskih elemenata, nadjačati specifičnijim pravilima ili kombinirati na složene načine kako bi se postigao željeni dizajn.

Jedan od značajnih napredaka u CSS-u je uvođenje CSS3, koje je donijelo nove značajke i module koji su značajno proširili njegove mogućnosti. CSS3 je uveo poboljšane mehanizme za raspored kao što su Flexbox i Grid, koji omogućuju složenije i fleksibilnije rasporede stranica u usporedbi s tradicionalnim metodama [12]. Ove značajke omogućuju programerima stvaranje responzivnih dizajna koji se besprijekorno prilagođavaju različitim veličinama ekrana i uređajima, poboljšavajući korisničko iskustvo na različitim platformama.

Dodatno, CSS3 uključuje nove osobine za animacije i prijelaze, što omogućuje stvaranje dinamičnih i interaktivnih web-elemenata. Ove mogućnosti omogućuju glađe vizualne efekte i poboljšane korisničke interakcije, doprinoseći modernijem i zanimljivijem web-dizajnu [12].

Modularni pristup CSS3 također znači da programeri mogu koristiti samo značajke koje su im potrebne, olakšavajući postupno usvajanje novih sposobnosti.

Evolucija CSS-a imala je dubok utjecaj na web-dizajn, omogućujući stvaranje vizualno privlačnih i korisnički prijateljskih web-stranica s većom učinkovitošću. Kako se web-standardi nastavljaju razvijati, CSS ostaje ključni alat za web-programere, nudeći sredstva za implementaciju složenih dizajna i responzivnih rasporeda uz održavanje jasne odvojenosti između sadržaja i prikaza.

2.1.1.5. React

React.js, poznat kao React, popularna je JavaScript biblioteka za izgradnju korisničkih sučelja, gdje su responzivnost i dinamično korisničko iskustvo od izuzetne važnosti. Razvijen u Facebooku i objavljen kao *open-source* projekt 2013. godine, React je od tada postao široko prihvaćen alat u zajednici frontend programera zbog svoje učinkovitosti i fleksibilnosti [7]. Omogućuje programerima stvaranje višekratnih UI komponenti koje mogu upravljati vlastitim stanjem, što dovodi do održivijih i skalabilnijih projekata.

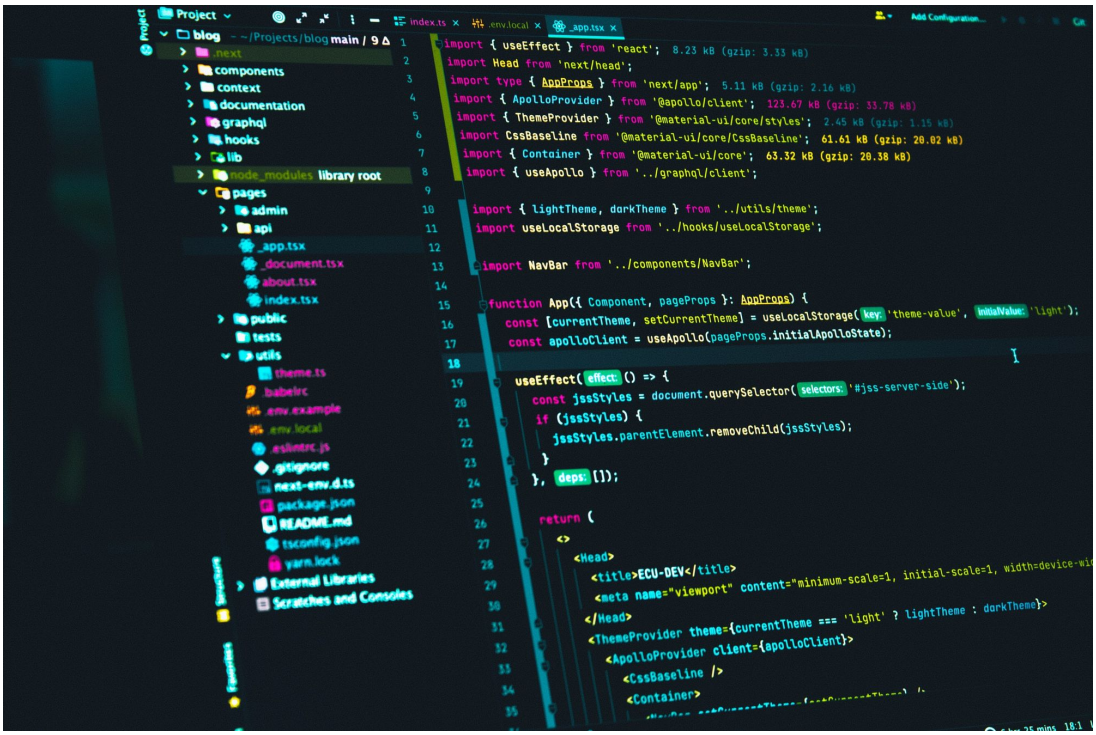
Jedan od osnovnih koncepata Reacta je Virtual DOM (Document Object Model). Za razliku od tradicionalnog razvoja u web-okruženju gdje su promjene u DOM-u spore i skupe u pogledu performansi, React koristi Virtual DOM za optimizaciju prikaza. Virtual DOM je zapravo kopija stvarnog DOM-a koju React održava u memoriji. Kada se stanje komponente promijeni, React prvo ažurira Virtual DOM i zatim izvodi algoritam za usporedbu kako bi odredio minimalan broj promjena potrebnih za ažuriranje stvarnog DOM-a. Ovaj pristup značajno poboljšava performanse i osigurava glađe korisničko iskustvo [13].

Još jedna važna značajka Reacta je njegova arhitektura temeljena na komponentama. U Reactu, korisničko sučelje se razlaže na male, višestruko iskoristive komponente, od kojih svaka enkapsulira svoju vlastitu logiku i ponašanje prikaza. Komponente se mogu sastaviti zajedno kako bi se izgradila složena UI sučelja, a promjene u stanju jedne komponente mogu se prenijeti na druge komponente prema potrebi. Ova modularnost potiče ponovnu upotrebu koda i olakšava upravljanje i testiranje pojedinih dijelova aplikacije [7].

React je također uveo JSX (JavaScript XML), sintaktičku ekstenziju koja omogućuje programerima pisanje koda nalik HTML-u unutar JavaScript datoteka. JSX pojednostavljuje proces stvaranja React elemenata i komponenti pružajući intuitivniju i čitljiviju sintaksu. Iako JSX nije obavezan za korištenje Reacta, široko je prihvaćen jer kombinira snagu JavaScripta s izražajnošću HTML-a, čineći kod lakšim za razumijevanje i održavanje [13].

Od svog objavljivanja, React se nastavio razvijati sa značajnim ažuriranjima i poboljšanjima. Uvođenje React Hooks u verziji 16.8 donijelo je novi način upravljanja stanjem i nuspojavama u funkcionalnim komponentama, pružajući alternativu komponentama temeljenim na klasama i dodatno poboljšavajući fleksibilnost biblioteke [13]. Ekosustav Reacta također uključuje alate kao što su React Router za upravljanje usmjeravanjem i Redux za upravljanje stanjem, koji se besprijekorno integriraju s osnovnom bibliotekom za izgradnju robusnih aplikacija.

Široko usvajanje Reacta u industriji naglašava njegovu učinkovitost u stvaranju visokih



Slika 1: Programiranje u Reactu; preuzeto iz https://tinyurl.com/2f5nac9w

performansi i dinamičnih web-aplikacija. Njegova snažna podrška zajednice i opsežni ekosustav doprinose njegovoj stalnoj popularnosti i čine ga vrijednim alatom za moderni web-razvoj. Kako se web i dalje razvija, stalna ažuriranja i inovacije Reacta vjerojatno će igrati ključnu ulogu u oblikovanju budućnosti razvoja korisničkih sučelja.

2.1.1.6. Node.js

Node.js je moćno, open-source okruženje za izvršavanje JavaScript koda na serveru. Razvio ga je Ryan Dahl i objavio 2009. godine, a Node.js koristi V8 JavaScript engine, koji je izvorno razvijen za Google Chrome, za izvršavanje JavaScript koda s visokom performansom i učinkovitošću. Za razliku od tradicionalnih *server-side* platformi koje se oslanjaju na višedretveno upravljanje, Node.js koristi neblokirajuću, *event-driven* arhitekturu, koja mu omogućuje da upravlja s više istodobnih veza s minimalnim opterećenjem. Ovaj model čini Node.js posebno pogodnim za aplikacije koje zahtijevaju real-time interakciju, kao što su chat aplikacije ili aplikacije za ažuriranja uživo [8].

Ključna značajka Node.jsa je njegov sustav za upravljanje paketima, npm (Node Package Manager), koji pruža pristup ogromnom spremištu *open-source* biblioteka i alata. Kroz npm, programeri mogu lako integrirati module i pakete trećih strana u svoje projekte, što ubrzava razvoj i smanjuje potrebu za izgradnjom funkcionalnosti od nule. Ovaj ekosustav značajno je pridonio širokoj prihvaćenosti i svestranosti Node.jsa [8].

U kontekstu web-razvoja, Node.js se često koristi u kombinaciji sa softverskim okvirima poput Next.jsa. Next.js, softverski okvir temeljen na React-u, koristi Node.js za omogućavanje *server-side* renderiranja i generiranja statičkih stranica, poboljšavajući performanse i SEO

mogućnosti za React aplikacije. Pružajući robusno *server-side* okruženje, Node.js podržava Next.js u isporuci optimiziranih, skalabilnih web-aplikacija, što ga čini popularnim izborom među programerima [8].

Sve u svemu, event-driven arhitektura Node.jsa, zajedno s njegovim opsežnim ekosustavom biblioteka i alata, čini ga neprocjenjivim resursom za moderni web-razvoj. Njegova integracija sa softverskim okvirima poput Next.jsa dodatno ilustrira njegovu ulogu u omogućavanju učinkovitog i skalabilnog *server-side* renderiranja za dinamične web-aplikacije.

2.1.1.7. Next.js softverski okvir

Next.js je napredni open-source softverski okvir za izgradnju React.js aplikacija, razvijen od strane Vercela i prvi put objavljen 2016. godine. Stekao je široku prihvaćenost zbog svoje sposobnosti da poboljša React aplikacije uz podršku za server-side rendering (SSR), statičku generaciju stranica (SSG) i druge moćne značajke [14]. Next.js je dizajniran kako bi riješio nekoliko uobičajenih izazova u web-razvoju, kao što su optimizacija performansi i SEO, pružajući robusni set alata koji proširuju mogućnosti Reacta.

Jedna od najistaknutijih značajki Next.js-a je podrška za *server-side* renderiranje. SSR omogućuje da se stranice renderiraju na serveru prije nego što budu poslane klijentu, što može značajno smanjiti početne vrijeme učitavanja i poboljšati SEO. Ova značajka je posebno korisna za dinamičan sadržaj koji mora biti ažuriran kada se isporučuje korisnicima. Prethodnim renderiranjem stranica na poslužitelju, Next.js osigurava da korisnici primaju potpuno renderiranu stranicu, što poboljšava korisničko iskustvo i pristupačnost [14].

Next.js podržava i statičku generaciju stranica, gdje se stranice renderiraju u vrijeme izgradnje, a ne pri svakom zahtjevu. Ovaj pristup je pogodan za sadržaj koji se ne mijenja često, kao što su blog objave ili dokumentacija. Statička generacija stranica dovodi do bržeg učitavanja stranica i boljih performansi budući da se sadržaj isporučuje kao unaprijed izgrađena statička datoteka. Next.js-ov hibridni pristup omogućuje programerima da odaberu najbolju strategiju renderiranja za svaku stranicu ili komponentu, kombinirajući prednosti SSR-a i SSG-a [14].

Štoviše, Next.js uključuje značajke poput automatskog razdvajanja koda, optimiziranog rukovanja slikama i inkrementalne statičke regeneracije koje doprinose poboljšanju performansi i učinkovitosti. Automatsko razdvajanje koda osigurava da se učita JavaScript nužan samo za trenutnu stranicu, smanjujući vrijeme učitavanja. Optimizirano rukovanje slikama poboljšava performanse učitavanja i korisničko iskustvo automatskim posluživanjem slika u najboljem formatu i veličini [14].

Sve u svemu, kombinacija *server-side* renderiranja, statičke generacije stranica i ostalih značajki čini Next.js odličnim izborom za izgradnju skalabilne aplikacije visokih performansi. Njegova besprijekorna integracija s Reactom omogućuje programerima da iskoriste njegove mogućnosti dok zadržavaju fleksibilnost i moć Reacta, a upravo tim pristupom se radilo tijekom izrade aplikacije za ovaj rad.

2.1.1.8. TypeScript

TypeScript je statički tipizirani superset JavaScripta, razvijen od strane Microsofta i prvi put objavljen 2012. godine. Poboljšava proces razvoja dodavanjem opcionalne statičke tipizacije JavaScriptu, omogućujući programerima da otkriju greške tijekom prevođenja umjesto izvođenja [15]. Ova značajka čini TypeScript posebno vrijednim u aplikacijama velikih razmjera, gdje je ključno održavati kvalitetu koda i smanjiti broj grešaka. U ovom radu, TypeScript je korišten u kombinaciji s React.js-om, iskorištavajući njegove snažne tipizacijske mogućnosti za poboljšanje pouzdanosti i održivosti aplikacije.

Jedna od glavnih prednosti TypeScripta je njegova sposobnost pružanja sigurnosti tipova. Definiranjem tipova za varijable, funkcije i objekte, programeri mogu spriječiti uobičajene pogreške poput neusklađenosti tipova i nedefiniranih varijabli. Ovo proaktivno otkrivanje grešaka dovodi do robusnijeg koda i uvelike smanjuje vjerojatnost pogrešaka tijekom izvođenja. U kontekstu React.js, TypeScript omogućuje korištenje tipiziranih tzv. *propova* i *statea*, osiguravajući da se komponente koriste dosljedno i ispravno u cijeloj aplikaciji [15].

TypeScript također podržava moderne JavaScript značajke i besprijekorno se integrira s postojećim JavaScript kodom. Ova unazadna kompatibilnost omogućuje programerima postupno uvođenje TypeScript-a u svoje projekte bez potrebe za prepisivanjem postojećih baza koda. Štoviše, podrška TypeScripa za ES6 i novije verzije osigurava da programeri mogu koristiti najnovije JavaScript značajke uz prednosti statičke tipizacije. Ova kompatibilnost s JavaScript softverskim okvirima kao što je React.js pridonijela je rastućoj popularnosti TypeScripta u zajednici web-developera [16].

Još jedna značajna prednost TypeScripta je njegova podrška za razvojne alate. TypeScript pruža poboljšane značajke u uređivačima koda, kao što su automatsko dovršavanje, alati za refaktoriranje i inteligentna navigacija kodom, što poboljšava produktivnost programera. Ovi alati su posebno korisni u složenim aplikacijama gdje su razumijevanje baze koda te brze i točne promjene ključne. Integrirana razvojna okruženja (IDE) poput Microsoftovog Visual Studio Codea, nude izvrsnu podršku za TypeScript, dodatno poboljšavajući iskustvo razvoja [16].

TypeScriptovo uvođenje statičke tipizacije u JavaScript, zajedno s njegovom besprijekornom integracijom s modernim JavaScript značajkama i softverskim okvirima čini ga neophodnim alatom za izgradnju skalabilnih i održivih web-aplikacija. Kombinacija sigurnosti tipova, unazadne kompatibilnosti i robusne podrške za razvojne alate učinila je TypeScript preferiranim izborom među programerima.

2.1.1.9. Tailwind

Tailwind CSS je visoko prilagodljiv, nisko-razinski CSS softverski okvir koji pruža korisničke klase za izgradnju prilagođenih dizajna bez potrebe za pisanjem tradicionalnog CSS-a. Prvi put objavljen 2017. godine, Tailwind je brzo stekao popularnost zbog svog jedinstvenog pristupa stiliziranju. Za razliku od konvencionalnih CSS softverskih okvira koji nude unaprijed dizajnirane komponente, Tailwind se fokusira na pružanje korisničkih klasa koje se mogu kombinirati za stvaranje prilagođenih korisničkih sučelja izravno u oznaci nekog elementa [17].

Jedna od ključnih prednosti Tailwind CSS-a je njegov pristup "utility-first". Umjesto oslanjanja na unaprijed definirane komponente, programeri koriste korisničke klase za primjenu specifičnih stilova na elemente. Ova metodologija nudi veću kontrolu i fleksibilnost u dizajnu, jer programeri mogu graditi responzivna, prilagođena korisnička sučelja bez napuštanja HTML-a ili JSX datoteka. Naprimjer, primjena stilova za razmak, tipografiju, boje i raspored izravno u oznaci omogućuje brzu iteraciju i eksperimentiranje tijekom razvoja [17]. Ovaj pristup je posebno koristan kada se radi sa softverskim okvirima temeljenim na komponentama kao što je React.js, gdje se Tailwindove korisničke klase mogu besprijekorno integrirati u strukture komponenti, što promiče učinkovitiji tijek rada.

Nadalje, Tailwind CSS-u je fokus na performanse i optimizaciju. Tailwind uključuje ugrađeni alat nazvan PurgeCSS koji automatski uklanja neiskorišteni CSS tijekom procesa izgradnje, što rezultira minimalnom i optimiziranom konačnom CSS datotekom. Ovo smanjenje veličine datoteke izravno utječe na vrijeme učitavanja i ukupne performanse web-aplikacije. Kada se koristi Tailwind s Reactom, ova učinkovitost postaje još izraženija, budući da Reactova arhitektura temeljena na komponentama prirodno nadopunjuje Tailwindov *utility-first* pristup. Kombinacija ovih tehnologija osigurava da aplikacija nije samo vizualno konzistentna već i optimizirana za performanse [18].

Tailwind također briljira u održivosti i skalabilnosti. Kako projekti rastu, složenost upravljanja CSS-om može postati značajan izazov. Tailwind to ublažava promicanjem konzistentne upotrebe korisničkih klasa, što smanjuje potrebu za složenim, prilagođenim CSS pravilima. Ova konzistentnost pojednostavljuje proces ažuriranja i održavanja dizajna aplikacije tijekom vremena. Osim toga, Tailwindova konfiguracijska datoteka omogućuje opsežnu prilagodbu, omogućujući timovima da definiraju svoj sustav dizajna, uključujući sheme boja, razmake i tipografiju, osiguravajući da se svi programeri pridržavaju istih dizajnerskih principa [17].

Zaključno, Tailwind CSS, kada se koristi u kombinaciji s Reactom, nudi snažan skup alata za izgradnju modernih, responzivnih web-aplikacija. Njegov *utility-first* pristup, u kombinaciji s optimizacijom performansi i značajkama održivosti, čini ga idealnim izborom za projekte koji zahtijevaju i fleksibilnost i učinkovitost.

2.1.1.10. ESLint

ESLint je alat za provjeru JavaScript koda, osmišljen je za prepoznavanje i ispravljanje problema u JavaScript kodu, uključujući uobičajene programske greške, stilističke probleme i bugove. Razvijen od strane Nicholasa C. Zakasa 2013. godine, ESLint je postao ključni alat u ekosustavu web-razvoja, posebno za projekte koji koriste moderne JavaScript softverske okvire poput Reacta i TypeScripta.

Neke od glavnih prednosti korištenja ESLinta su njegova fleksibilnost i proširivost. ESLint omogućuje programerima definiranje i provođenje standarda kodiranja putem konfiguracije seta pravila, koja se mogu prilagoditi ili proširiti putem dodataka. Ova značajka je posebno korisna u složenim projektima koji uključuju više tehnologija, kao što su React.js i Tailwind CSS, gdje je održavanje konzistentnog stila kodiranja gotovo neophodno. Naprimjer, mogućnost ES-

Linta da provodi pravila specifična za TypeScript osigurava da su tipovi ispravno označeni, smanjujući vjerojatnost pogrešaka povezanih s tipovima u kodu [19].

Osim toga, ESLint se besprijekorno integrira s modernim razvojnim okruženjima i alatima. Kada se koristi s React.js-om i TypeScriptom, ESLint pruža povratne informacije programerima u stvarnom vremenu, ističući probleme izravno u uređivaču koda. Ova trenutna povratna sprega pomaže u ranom otkrivanju grešaka u procesu razvoja, čime se poboljšava ukupna kvaliteta koda. Štoviše, ESLintova kompatibilnost s Prettierom, popularnim alatom za formatiranje koda, omogućuje istovremeno provođenje pravila za stil i formatiranje koda, osiguravajući da baza koda ostane čista i konzistentna [20].

U kontekstu ovog diplomskog rada, ESLint je igrao ključnu ulogu u održavanju kvalitete i čitljivosti koda. Korištenjem ESLinta s Reactom, Tailwindom i TypeScriptom, osigurano je da se kod pridržava najboljih praksi, smanjujući potencijal za greške i poboljšavajući održivost. Integracija ESLinta u razvojni proces aplikacije rada ne samo da je olakšala provođenje standarda kodiranja već je i pojednostavila proces pregleda koda.

2.1.2. Poslužitelj

2.1.2.1. Python

Python je visoko-razinski, interpretirani programski jezik poznat po svojoj jednostavnosti i čitljivosti, što ga čini jednim od najpopularnijih jezika za širok spektar primjena, uključujući web-razvoj, analizu podataka, umjetnu inteligenciju i još mnogo toga. Guid van Rossum ga je prvi put objavio 1991. godine. Pythonova filozofija dizajna naglašava čitljivost koda korištenjem praznog prostora i jasne, jednostavne sintakse. U ovom radu, Python je korišten kao programski jezik za backend, iskorištavajući njegove opsežne biblioteke i okvire za učinkovito upravljanje logikom na strani poslužitelja i interakcijom s React frontendom [21].

Za svoju popularnost, Python može zahvaliti i širokom ekosustavu biblioteka i okvira koji uvelike pojednostavljuju proces razvoja. Okviri poput Django i Flask pružaju snažne alate za izgradnju skalabilnih web-aplikacija. Django, visoko-razinski web-okvir za Python, posebno je prikladan za brzi razvoj i čist, pragmatičan dizajn. Uključuje ORM (engl. *Object-Relational Mapping*) sustav koji omogućuje programerima interakciju s bazama podataka koristeći Python kod umjesto SQL upita, čime se smanjuje složenost upravljanja bazama podataka. S druge strane, Flask je mikro-okvir koji nudi fleksibilnost i kontrolu nad komponentama koje se koriste u web-aplikaciji, što ga čini idealnim za projekte koji zahtijevaju prilagođeni pristup [22].

Pythonova jednostavnost i lakoća korištenja također značajno doprinose njegovoj širokoj primjeni u backend razvoju. Njegova sintaksa jako nalikuje običnom engleskom jeziku, što smanjuje krivulju učenja za nove programere i ubrzava proces razvoja. Štoviše, Pythonova interpretirana priroda omogućuje brzu iteraciju i testiranje, što je ključno u agilnim razvojnim okruženjima. Ova značajka je posebno korisna u backend razvoju, gdje mogućnost brzog testiranja i implementacija promjena koda može značajno utjecati na vremenske rokove projekta [21].

Još jedna prednost korištenja Pythona za backend razvoj je snažna podrška zajednice i opsežna dokumentacija. Python zajednica je jedna od najvećih i najaktivnijih u svijetu programiranja, pružajući obilje resursa, vodiča i dodatnih paketa koji se lako mogu integrirati u projekte. Ovaj sustav podrške ne samo da olakšava rješavanje problema nego i potiče primjenu najboljih praksi u kodiranju i upravljanju projektima.

U ovom projektu Pythonova uloga programskog jezika za backend bila je ključna za osiguravanje učinkovitog upravljanja zadacima na strani poslužitelja. Njegov izbor također je pridonio održivosti i skalabilnosti aplikacije, zahvaljujući čistoj sintaksi i velikom broju dostupnih biblioteka.

2.1.2.2. Flask

Flask je softverski okvir za Python, koji je razvio Armin Ronacher i objavljen 2010. godine. Dizajniran je da bude jednostavan i fleksibilan, pružajući osnovne alate potrebne za razvoj web-aplikacija bez nametanja određenog načina rada. Flask se često naziva mikro-okvirom jer ne zahtijeva specifične alate ili biblioteke te nema ugrađenu validaciju obrazaca, sloj za apstrakciju baze podataka ili druge komponente koje neki drugi softverski okviri standardno nude. Ovaj minimalistički pristup omogućuje programerima potpunu kontrolu nad komponentama koje žele koristiti, što Flask čini izuzetno prilagodljivim i jednostavnim za skaliranje [23].

Jednostavnost Flaska čini ga idealnim izborom za programere koji preferiraju izgradnju web-aplikacija od temelja, birajući samo one komponente koje su im potrebne. Ovaj pristup je posebno koristan u projektima koji zahtijevaju visok stupanj prilagodbe ili kada se integriraju s postojećim sustavima. Flask pruža solidnu osnovu za web-aplikacije upravljanjem tzv. routingom, zahtjevima i odgovorima te nudi ugrađeni razvojni poslužitelj. Također podržava proširenja koja mogu dodati potrebnu funkcionalnost, poput integracije s bazama podataka, validacije obrazaca ili autentifikacije. Ta se proširenja besprijekorno integriraju s Flaskom, zadržavajući njegovu jednostavnost i fleksibilnost, dok istovremeno proširuju njegove mogućnosti [23].

Flask posjeduje podršku za RESTful preusmjeravanje zahtjeva, što ga čini izvrsnim izborom za razvoj REST API-ja, koji su ključni za moderne web-aplikacije koje se oslanjaju na arhitekturu klijent-poslužitelj. Lagana priroda Flaska omogućuje učinkovito upravljanje RESTful API-jima, a njegov jednostavan dizajn pojednostavljuje proces routinga i upravljanja HTTP zahtjevima. Dodatno, Flaskova kompatibilnost s WSGI (engl. *Web Server Gateway Interface*) i njegova sposobnost rada na većini operativnih sustava čine ga svestranim alatom za web-razvoj [24].

U kontekstu ovog rada, Flask je odabran kao okvir za backend zbog svoje fleksibilnosti i jednostavne integracije s drugim tehnologijama. On se u ovom slučaju nije koristio na standardan način gdje upravlja putanjama web mjesta u frontend dijelu, već se isključivo rabi zbog biblioteke SocketIO koja je opisana u nastavku.

2.1.2.3. WebSocket

WebSocket je komunikacijski protokol koji omogućava dvosmjerne komunikacijske kanale putem jedne dugotrajne veze između klijenta i poslužitelja. Za razliku od tradicionalne HTTP komunikacije, koja slijedi model zahtjeva i odgovora gdje klijent inicira svaku interakciju, WebSocket omogućava i klijentu i poslužitelju da šalju podatke u bilo kojem trenutku, što ga čini idealnim za aplikacije u stvarnom vremenu. Ova tehnologija posebno je korisna u scenarijima koji zahtijevaju trenutna ažuriranja podataka, kao što su online igre, aplikacije za chat uživo i alati za suradnju [25].

WebSocket protokol standardiziran je od strane IETF-a kao RFC 6455 2011. godine, što je označilo značajnu promjenu u načinu na koji web-aplikacije rješavaju komunikaciju u stvarnom vremenu [25]. Prije WebSocketa, postizanje komunikacije u stvarnom vremenu u web-aplikacijama obično je uključivalo složene zaobilazne metode, poput dugog ispitivanja (engl. *long polling*) ili događaja poslanih s poslužitelja (engl. *server-sent events*) koji su bili neučinkoviti i skloni kašnjenju. S druge strane, WebSocket uspostavlja trajnu vezu između klijenta i poslužitelja, omogućujući kontinuiranu razmjenu podataka s minimalnim opterećenjem.

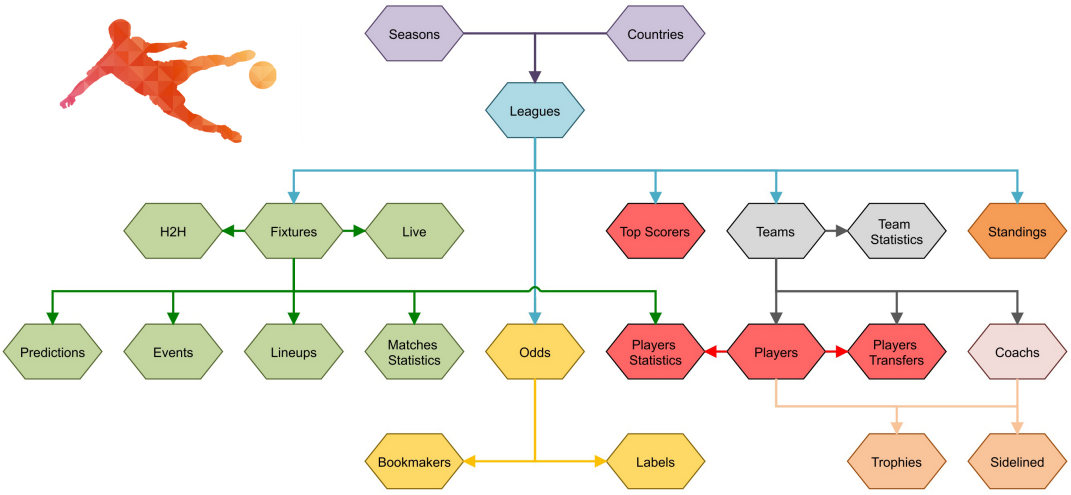
Jedna od ključnih prednosti WebSocketa je njegova učinkovitost u obradi komunikacije u stvarnom vremenu. Budući da WebSocket radi preko jedne veze, on smanjuje potrebu za višestrukim HTTP zahtjevima i odgovorima, čime se smanjuje kašnjenje i korištenje propusnosti. Ova učinkovitost postiže se putem WebSocket rukovanja (enlg. *handshake*), procesa koji započinje standardnim HTTP zahtjevom, ali vezu nadograđuje na WebSocket protokol. Nakon što je veza uspostavljena, podaci se mogu razmjenjivati u oba smjera s niskim kašnjenjem, što ga čini idealnim za aplikacije koje zahtijevaju brza i učestala ažuriranja, poput financijskih platformi za trgovanje ili ažuriranja sportskih rezultata uživo [25].

Još jedan važan aspekt WebSocketa je njegova kompatibilnost s postojećim web-tehnologijama. WebSocket se lako može integrirati u web-aplikacije korištenjem JavaScripta na strani klijenta i raznih okvira i biblioteka na strani poslužitelja. Ova kompatibilnost učinila je WebSocket popularnim izborom među programerima jer se može koristiti bez potrebe za rekonstrukcijom postojeće infrastrukture [25]. U slučaju ovog rada, WebSocket se koristi uz pomoć biblioteke naziva Flask SocketIO autora Miguel Grinberg.

WebSocket je u svrhe ovog rada implementiran kako bi omogućio ažuriranja u stvarnom vremenu između klijenta i poslužitelja. Izbor je bio motiviran potrebom za jednostavnom komunikacijom s niskim kašnjenjem u scenarijima gdje korisnici promjene moraju vidjeti odmah. Integracija WebSocketa omogućila je sustavu učinkovitu obradu podataka uživo, osiguravajući da korisnici mogu komunicirati s aplikacijom na fluidan i responzivan način.

2.1.2.4. API-Football

API-Football moćan je alat za programere koji žele integrirati opsežne podatke o nogometu u svoje aplikacije. Ovaj API nudi sveobuhvatan niz značajki, uključujući pristup rezultatima uživo, događajima na utakmicama, statistikama timova i igrača, poretku liga i povijesnim podacima. Sa svojim robusnim i fleksibilnim krajnjim točkama, programerima omogućuje jed-



Slika 2: Struktura podataka na API-Football; preuzeto iz [27]

nostavno dohvaćanje i prikazivanje informacija vezanih uz nogomet u stvarnom vremenu, što ga čini ključnim resursom za sportske aplikacije, platforme za klađenje i tzv. fantasy nogometne lige [26]. Na slici 2 u grafičkom obliku prikazana je struktura podataka koji se mogu dobiti s API-Football.

Jedna od najistaknutijih značajki API-Footballa je isporuka podataka u stvarnom vremenu, što osigurava da korisnici imaju najnovije informacije nadohvat ruke. API podržava širok raspon nogometnih liga i natjecanja diljem svijeta, pružajući detaljnu statistiku utakmica poput golova, asistencija, kartona, zamjena i mnogo više. Osim toga, API nudi krajnje točke za podatke o timovima i igračima, omogućujući izradu detaljnih profila i praćenje statistika, što je osobito vrijedno za analitičke aplikacije i platforme za angažman obožavatelja [26].

API je izgrađen na RESTful principima, što ga čini dostupnim i jednostavnim za korištenje programerima. Podržava razne metode zahtjeva, uključujući GET i POST, kako bi se omogućilo različite vrste dohvaćanja podataka i interakcije. Dokumentacija koju pruža API-Football je detaljna i relativno intuitivna za korištenje, s primjerima zahtjeva i odgovora koji pomažu programerima da brzo integriraju API u svoje projekte. Potrebno je provesti proces autentifikacije, a postoje ograničenja brzine i rukovanje pogreškama, osiguravajući da programeri mogu izgraditi pouzdane aplikacije koristeći ovaj API [26]. Latencija (engl. *latency*) je prema njihovoj dokumentaciji otprilike 357 milisekundi.

Podaci se uz pomoć ovog API-ja dohvaćaju na poslužitelj, gdje ih agenti oblikuju u potreban JSON format te nakon toga prosljeđuju klijentu preko WebSocketa.

2.1.2.5. SPADE

SPADE (engl. *Smart Python Agent Development Environment*) je moćna biblioteka dizajnirana za razvoj višeagentnih sustava (VAS) u Pythonu. Pruža sveobuhvatan okvir koji pojednostavljuje proces stvaranja, implementacije i upravljanja agentima u različitim sustavima. SPADE je izgrađen na uz XMPP- (engl. *Extensible Messaging and Presence Protocol*) i omo-

gućuje agentima asinkronu komunikaciju putem mreže, što ga čini idealnim za aplikacije koje zahtijevaju suradnju između više autonomnih agenata [28].

Jedna od ključnih značajki SPADE-a je njegova fleksibilnost u upravljanju životnim ciklusom agenata, uključujući stvaranje, aktivaciju, deaktivaciju i ukidanje. Agenti u SPADE-u mogu biti programirani za obavljanje raznih zadataka autonomno, komunicirati s drugim agentima koristeći mehanizme razmjene poruka i donositi odluke na temelju svoje okoline. Korištenje XMPP-a kao osnovnog komunikacijskog protokola osigurava da agenti mogu djelovati na decentralizirani način, što je ključno za izgradnju skalabilnih i otpornijih višeagentnih sustava [28].

SPADE također podržava razvoj složenih ponašanja unutar agenata, omogućujući im izvršavanje više zadataka istovremeno. Programeri mogu definirati ponašanja koristeći jednostavne Python klase i metode, što olakšava implementaciju sofisticirane logike bez potrebe za detaljnim poznavanjem mrežnih slojeva. Osim toga, integracija SPADE-a s postojećim Python bibliotekama poboljšava njegovu primjenjivost u raznim domenama kao što su robotika, IoT i distribuirana umjetna inteligencija [29].

3. Razrada teme

3.1. Inteligentni agenti i okruženja

Prilikom traženja općenite definicije agenta, nailazi se na dosta izvora koji tvrde kako još ne postoji univerzalno prihvaćeno objašnjenje. Štoviše, na tom polju još uvijek ima dovoljno prostora za debate i kontroverze. Neki konsenzus se unatoč tome ipak morao postići, pa tako imamo naširoko rasprostranjeno mišljenje kako jednog agenta ponajprije definira njegova autonomija. Tema je ovo kojoj u sklopu koje predstoji još mnogo rasprava, a glavni razlog tome je što različiti atributi asocirani s agentima imaju različite važnosti ovisno o domeni primjene [30, 2. poglavlje]. U nastavku će biti spomenuto ono što je prema autorima iz literature i iz ovog rada najvjerodostojnije opisano značenje tog pojma.

Inteligentni agenti predstavljaju temeljni koncept u umjetnoj inteligenciji, pri čemu se radi o entitetima koji su sposobni percipirati svoje okruženje i djelovati u njemu kako bi postigli određene ciljeve. Dizajn inteligentnog agenta podrazumijeva razumijevanje načina na koji ti agenti komuniciraju sa svojim okruženjem te kako donose odluke na temelju svojih percepcija. Inteligentan je agent onaj koji autonomno percipira svoje okruženje putem senzora, obrađuje te informacije i poduzima akcije koristeći pokretačke mehanizme ili aktuatore kako bi postigao najbolji mogući ishod prema unaprijed definiranom mjerilu izvedbe [31, str. 36]. Interakcija između agenta i njegovog okruženja ključna je za koncept inteligencije u AI-u.

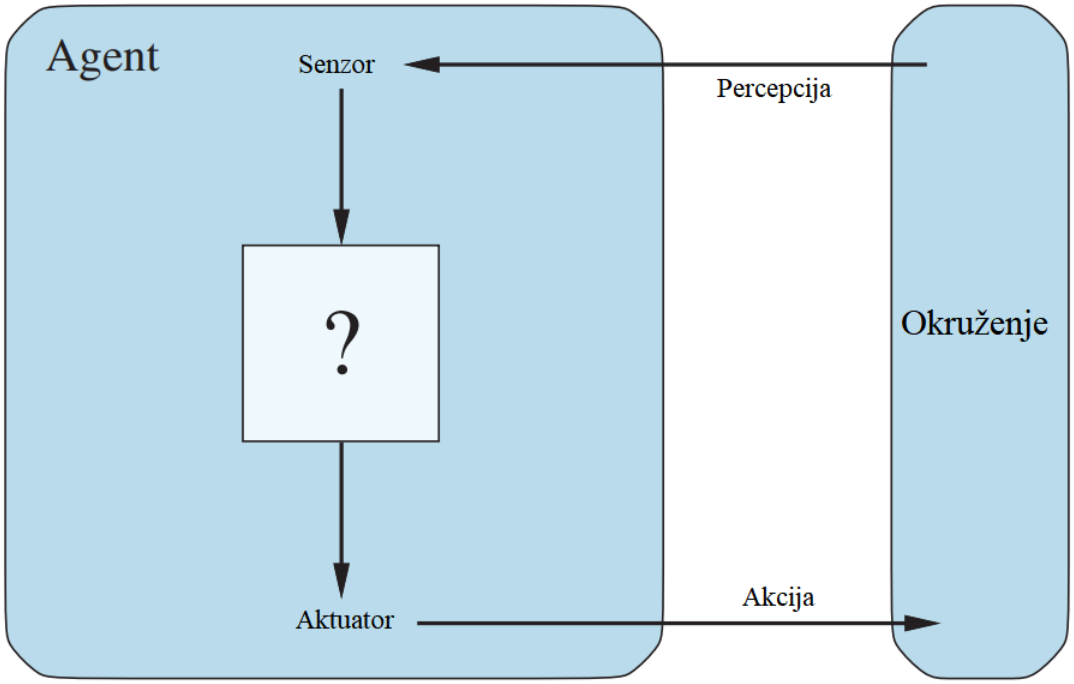
S druge strane, okruženje je sve što je izvan agenta, a što on može doživljavati i na što može djelovati. Okruženja se mogu značajno razlikovati, a njihove karakteristike značajno utječu na dizajn i ponašanje agenata.

Na slici 3 jasno je prikazan odnos agenta i njegove okoline. Primjera radi, gleda li se entitet čovjeka kao agent, njegovi senzori ili osjetila bila bi oči, uši, koža i ostali organi, a aktuatori noge, ruke, glas. Analogno, kod agenta u obliku robota senzori su kamere, toplomjer, barometar, mikrofon i slično, dok su aktuatori motori ili recimo uređaji kao zvučnici. [31, str. 36]

Od važnijih pojmova valja svakako spomenuti funkciju agenta koja je, matematički gledano, opis njegovog ponašanja i ona specificira akciju koja se poduzima kao odgovor na bilo koju sekvencu podražaja. Funkcija je dakle apstraktan matematički opis, koja se iznutra konkretno implementira kao program agenta. Norvig i Russel [31, str. 37] za pojašnjenje ove teorije navode primjer kućnog robotskog usisavača. Za njega bi funkcija agenta vrlo jednostavno mogla riječima biti opisana kao: ukoliko je komadić prostora prljav, usisavaj, a inače prijeđi na drugi komadić prostora. Opisane funkcije su tablično prikazane na slici 4, a program u obliku programskog koda u isječku 1.

3.1.1. Racionalnost agenta

Racionalni agent je onaj koji djeluje kako bi maksimizirao svoje mjerilo performansi, uzimajući u obzir svoje znanje o okruženju i ograničenja svojih akcija za svaki slijed percepcije. Pritom je važno napomenuti da racionalnost ne podrazumijeva sveznanje (engl. *omniscience*);



Slika 3: Shematski dijagram interakcije agenta i okoline; preuzeto iz [31]

Slijed percepcije	Akcija
[A, Čist]	Desno
[A, Prljav]	Usisavaj
[B, Čist]	Lijevo
[B, Prljav]	Usisavaj
[A, Čist], [A, Čist]	Desno
[A, Čist], [A, Prljav]	Usisavaj
...	...
[A, Čist], [A, Čist], [A, Čist]	Desno
[A, Čist], [A, Čist], [A, Prljav]	Usisavaj
...	...

Slika 4: Slijed percepcije i akcije robotskog usisavača; preuzeto iz [31]

```
function REFLEX-VACUUM-AGENT( [location,status]) returns an action

    if status = Prljav then return Usisavaj
    else if location = A then return Desno
    else if location = B then return Lijevo
```

Isječak koda 1: Primjer programa agenta za robotskog usisavača

umjesto toga, ono znači da agent donosi najbolju moguću odluku s obzirom na informacije koje u tom trenutku ima. Na primjer, u nepristupačnom okruženju, racionalni agent mora donositi odluke na temelju nesigurnih ili nepotpunih podataka, s ciljem postizanja najveće očekivane koristi. Prema Norvigu i Russelu [31, str. 39-42], kao što gornja definicija navodi, racionalnost ovisi o sljedeće 4 stvari:

- Mjerilo performansi koje definira kriterij uspjeha
- Agentovo predznanje o okolini
- Akcije koje agent može izvesti
- Agentov dotadašnji slijed percepcije

U svijetu robotskog usisavača, redom bi to izgledalo kao što slijedi.

- Mjerilo performansi dodjeluje 1 bod za svaki čisti komadić prostora u svakom koraku vremena, u životnom vijeku od 1000 koraka
- Geografija prostora spada u predzanje, a raspored nečistoće i početna lokacija agenta ne. Čisti komadići ostaju čisti i usisavanje čisti trenutni komadić. Akcije lijevo i desno pomiču agent za jedan komadić prostora, osim tamo gdje bi izašao izvan dozvoljenog prostora, gdje se ne pomiče
- Jedine dostupne akcije su desno, lijevo i usisavaj
- Agent ispravno percipira svoju lokaciju i nečistoće

Učinkovitost inteligentnog agenta obično se mjeri njegovom racionalnošću, a u ovakvoj postavki agent je racionalan. Međutim, postoje nepokriveni slučajevi zbog kojih bi djelovao iracionalno, kao što je recimo slučaj kada se sve počisti. Pošto mu tada nije drugačije zadano, naš agent u obliku robotskog usisavača bi se nepotrebno nastavio kretati.

3.1.2. Inteligencija agenta

Prema Norvigu i Russelu, Inteligencija kod agenata proizlazi iz njihove sposobnosti donošenja odluka koje vode do optimalnih ishoda u različitim okruženjima. Postoji nekoliko faktora koji doprinose inteligenciji. Sposobnost agenta da točno i potpuno percipira svoje okruženje, sam proces donošenja odluka koji mora biti robustan i prilagodljiv promjenjivim okolnostima. Inteligentni agenti također su karakterizirani svojom sposobnošću učenja iz iskustva, što im omogućava da s vremenom poboljšaju svoju izvedbu. Tu je i sposobnost agenta da razmišlja o budućim posljedicama svojih akcija i da planira sukladno tome [31, str. 47-48].

S druge strane, Wooldridge [30, poglavlje 2.1.] navodi kako pitanje "što je to inteligentan agent" nije lagano za odgovoriti, ali se može objasniti tako da se popišu sve sposobnosti koje od agenta očekujemo da posjeduje.

- Reaktivnost - percipiranje okoline i pravovremeno reagiranje na promjene koje se u njoj dogode kako bi se zadovoljili ciljevi dizajna
- Proaktivnost - sposobnost primjenjivanja ponašanja usmjerenog prema nekom cilju preuzimanjem inicijative, da bi se zadovoljili ciljevi dizajna
- Socijalnost - sposobnost održavanja interakcije s drugim pripadnicima svoje vrste ili s ljudima, da bi se zadovoljili ciljevi dizajna

Dizajn inteligentnog agenta mora biti prilagođen specifičnim karakteristikama okruženja u kojem djeluje. Na primjer, u pristupačnom, deterministčkom okruženju, jednostavni refleksni agent može biti dovoljan. Međutim, u nepristupačnom ili stohastičkom okruženju, agent temeljen na modelu ili korisnosti vjerojatno će biti učinkovitiji. Izbor arhitekture agenta ovisi o faktorima kao što su promatrana priroda, determinističnost, epizodična priroda, dinamična svojstva i prisutnost drugih agenata u okruženju. Ovi faktori diktiraju složenost procesa donošenja odluka agenta i njegovu sposobnost da učinkovito postigne svoje ciljeve [31, str. 50-53].

Sposobnost učenja je također važna komponenta inteligencije kod agenata. Agent koji može učiti iz svojih iskustava i prilagođavati svoje ponašanje u skladu s tim znatno je učinkovitiji od onoga koji to ne može. Učenje omogućava agentima da s vremenom poboljšaju svoju izvedbu, čak i u okruženjima koja se mijenjaju ili gdje svi podaci u početku nisu dostupni. Primjerice, agent koji uči može započeti s jednostavnim modelom svog okruženja i usavršavati taj model kako prikuplja više podataka, što vodi do boljeg donošenja odluka u budućnosti. Ova sposobnost prilagodbe ključna je za suočavanje sa složenošću i nesigurnostima koje su inherentne u stvarnim okruženjima [31, str. 53-57].

Inteligentni agenti i njihova interakcija s okruženjima čine temelj moderne umjetne inteligencije. Razumijevanje načina na koji agenti percipiraju, donose odluke i djeluju u različitim okruženjima esencijalno je za dizajniranje sustava u kojem mogu autonomno i učinkovito djelovati. Složenost okruženja značajno utječe na vrstu potrebnog agenta, od jednostavnih refleksnih agenata do složenijih agenata temeljenih na modelu ili korisnosti. Sposobnost agenata da uče i prilagođavaju se dodatno povećava njihovu inteligenciju, omogućavajući im da se nose s raznim izazovima u dinamičnim i nesigurnim okruženjima. Kako umjetna inteligencija nastavlja evoluirati, principi inteligentnih agenata ostat će važni za razvoj naprednijih i sposobnijih sustava [31, str. 57-60].

3.1.3. Svojstva okruženja

Razumijevanje svojstava okruženja zadataka ključno je za dizajn i implementaciju inteligentnih agenata. Okruženja zadataka definiraju probleme koje agent mora riješiti i kontekst u kojem djeluje. Okruženje utječe na dizajn agenta i određuje složenost njegovog ponašanja. Stuart Russell i Peter Norvig raspravljaju o raznim svojstvima koja karakteriziraju okruženja zadataka [31, str. 43-47].

3.1.3.1. Pristupačna i nepristupačna

Jedno od ključnih svojstava okruženja zadataka je pitanje je li ono potpuno pristupačno ili ne. U cjelovito promatranom okruženju, senzori agenta pružaju potpun pristup relevantnim aspektima stanja okoline. To znači da agent ima sve potrebne informacije za donošenje informirane odluke u bilo kojem trenutku. Primjeri pristupačnih okruženja uključuju šah, gdje je cijela ploča vidljiva za oba igrača i stanje okoline je potpuno poznato.

Suprotno tome, nepristupačno okruženje ne pruža agentu potpune informacije o stanju svijeta. Agent mora donositi odluke na temelju nepotpunih, neurednih ili neizvjesnih podataka. To zahtijeva od agenta da održava unutarnje stanje ili uvjerenje o nevidljivim aspektima okruženja. Primjer djelomično promatranog okruženja je vožnja u magli, gdje je vidljivost ograničena, a vozač mora sam zaključivati o položaju drugih vozila i prepreka na temelju ograničenih vizualnih informacija i prošlih iskustava.

3.1.3.2. Determinističko i stohastičko

Drugo važno svojstvo je pitanje je li okruženje determinističko ili stohastičko. U determinističkom okruženju, sljedeće stanje okruženja u potpunosti je određeno trenutnim stanjem i radnjama agenta. Ova predvidljivost pojednostavljuje proces donošenja odluka, jer agent može biti siguran u ishode svojih akcija. Klasični problemi planiranja, poput rješavanja labirinta, često pretpostavljaju determinističko okruženje u kojem svaka akcija vodi do predvidljivog ishoda.

S druge strane, stohastičko okruženje uključuje elemente slučajnosti, što onemogućava predviđanje ishoda akcije s potpunom sigurnošću. U takvim okruženjima, agent mora uzeti u obzir vjerojatnosti različitih ishoda i planirati u skladu s tim. Na primjer, u kockarskim igrama poput pokera, neizvjesnost u ishodima izvlačenja karata dodaje složenost, tjerajući agenta da razmišlja pod uvjetima neizvjesnosti.

3.1.3.3. Epizodično i neepizodično

Još jedna značajna razlika u okruženjima zadataka je između epizodnih i neepizodičnih okruženja. U epizodičnom, iskustvo agenta podijeljeno je u diskretne epizode, od kojih svaka sadrži jednu radnju i kasnije opažanje. Važno je napomenuti da je svaka epizoda neovisna o prethodnima, što znači da akcije agenta ne utječu na buduće situacije. To omogućuje agentu da djeluje isključivo na temelju trenutnog opažanja, bez razmatranja posljedica svojih akcija na buduća stanja. Primjer epizodnog okruženja je prepoznavanje slika, gdje se svaka slika klasificira neovisno o drugima.

Nasuprot tome, neepizodično okruženje zahtijeva od agenta da uzme u obzir utjecaj svojih akcija na buduća stanja. Radnje su međusobno ovisne, a ishod jedne radnje može utjecati na buduće situacije s kojima će se agent susresti. Ova međusobna ovisnost u problem uvodi vremensku dimenziju jer agent mora planirati unaprijed i uzeti u obzir dugoročne posljedice. Šah je klasičan primjer sekvencijalnog okruženja, gdje svaki potez utječe na cijeli tijek igre.

3.1.3.4. Statično i dinamično

Statično okruženje ostaje nepromijenjeno dok agent razmišlja, što znači da se agent ne mora brinuti o vremenskim ograničenjima ili drugim agentima koji mijenjaju okruženje dok odlučuje što će učiniti. Mnogi zadaci rješavanja zagonetki, poput Rubikove kocke, primjeri su statičnih okruženja.

Međutim, dinamična okruženja nastavljaju se razvijati čak i dok agent razmišlja. U takvim okruženjima, agent mora biti sposoban donositi odluke u stvarnom vremenu, uzimajući u obzir mogućnost da se okruženje može promijeniti neovisno o njegovim akcijama. Naprimjer, samovozeći automobil djeluje u dinamičnom okruženju gdje se druga vozila, pješaci i prometni signali neprestano mijenjaju, zahtijevajući od agenta da stalno ažurira svoje razumijevanje okoline i donosi odluke u skladu s time.

3.1.3.5. Diskretno i kontinuirano

Okruženja zadataka također se mogu klasificirati na temelju toga jesu li diskretna ili kontinuirana. U diskretnom okruženju, broj različitih stanja i akcija je konačan i prebrojiv. Šah je tu ponovo odličan primjer jer, iako ih zaista ima mnogo, posjeduje konačan broj mogućih postavki ploče i poteza.

Nasuprot tome, kontinuirano okruženje ima beskonačan broj mogućih stanja i akcija. To je tipično u fizičkim zadacima poput navigacije robota, gdje robot teoretski može zauzeti bilo koju poziciju unutar kontinuiranog prostora i napraviti sitne prilagodbe svojim akcijama. Kontinuirana okruženja često zahtijevaju sofisticiranije matematičke modele i algoritme za rješavanje složenosti beskonačnih mogućnosti .

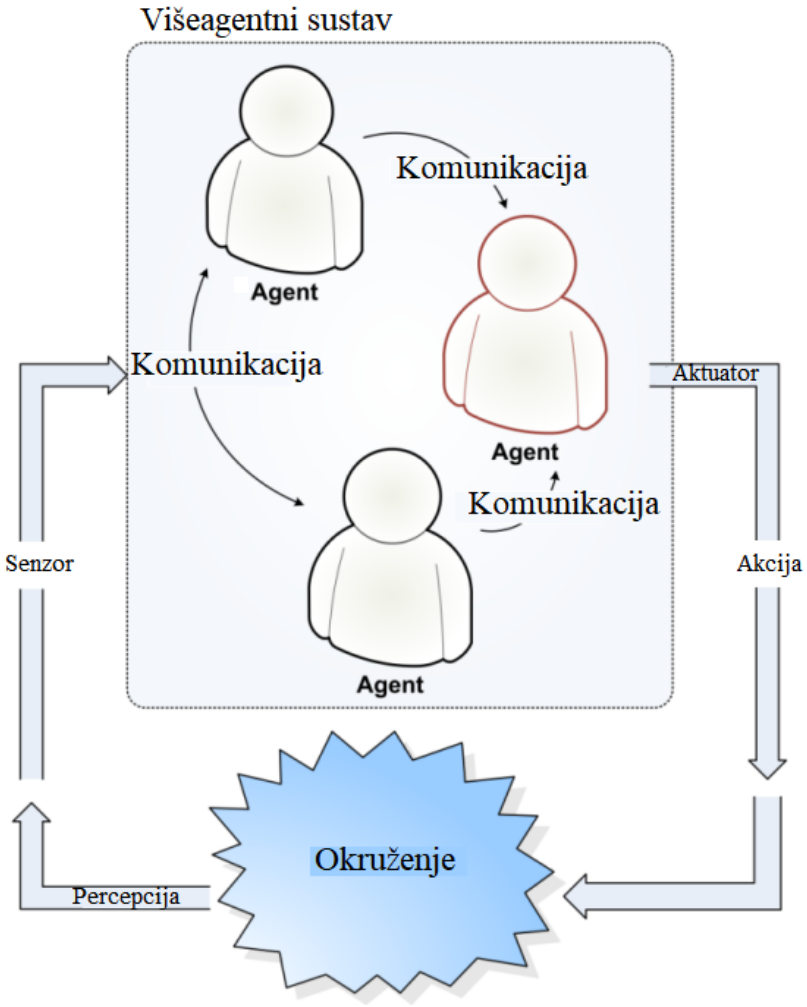
3.1.3.6. Jedan agent i više agenata

Konačno, broj agenata u okruženju također je svojstvo. U okruženju s jednim agentom on djeluje sam, bez prisutnosti drugih agenata koji bi ga mogli ometati ili s njime surađivati. Klasične društvene igre poput Solitera primjer su okruženja s jednim agentom.

Nasuprot tome, okruženje s više agenata uključuje više agenata koji mogu ili surađivati ili se natjecati. U kompetitivnim okruženjima s više agenata, poput mnogih strateških igara, prisutnost drugih agenata povećava složenost jer svaki agent mora pokušati predvidjeti i neutralizirati akcije drugih. U kooperativnim okruženjima, agenti rade zajedno kako bi postigli zajednički cilj, što zahtijeva komunikaciju i koordinaciju. Okruženja s više agenata uobičajena su u mnogim stvarnim scenarijima, poput prometnih sustava ili tržišnih ekonomija, gdje međusobno djeluje više entiteta.

3.2. Općenito o višeagentnim sustavima

Višeagentni sustavi predstavljaju važnu disciplinu usko povezanu s umjetnom inteligencijom, koja se fokusira na interakcije više agenata unutar zajedničkog okruženja. Svaki agent



Slika 5: Grafički vizualiziran VAS; preuzeto iz stranice

je autonomna jedinica kojemu se može dodijeliti sposobnost percepcije okoline, donošenja odluke i poduzimanja akcije na temelju svojih percepcija i ciljeva [31, str. 599]. Ovi sustavi su posebno korisni u složenim okruženjima gdje se zadaci mogu podijeliti među više agenata, od kojih svaki doprinosi postizanju zajedničkog ili individualnog cilja.

U VAS-u, okruženje može biti ili kooperativno ili nekooperativno. U kooperativnim okruženjima, agenti surađuju kako bi postigli zajednički cilj, često dovodeći do optimiziranih rješenja koja su korisna za sve sudionike. Suprotno tome, u nekooperativnim okruženjima, agenti slijede vlastite ciljeve, koji mogu biti u sukobu s ciljevima drugih agenata. Ova razlika značajno utječe na složenost sustava i strategije koje agenti koriste za postizanje svojih ciljeva [31, str. 601].

Proučavanje interakcija u VAS-u uvelike je pod utjecajem teorije igara, koja pruža okvir za razumijevanje strateškog donošenja odluka među agentima. Teorija igara omogućuje agentima da predvide akcije drugih i prilagode svoje strategije u skladu s tim, što je posebno relevantno u nekooperativnim okruženjima. Na primjer, u scenariju konkurentnog tržišta, svaki agent (kao što je kupac ili prodavač) izrađuje strategiju na temelju predviđenih akcija drugih agenata kako bi maksimizirao svoju korist. Ovaj aspekt VAS-a čini ih primjenjivim u raznim do-

menama, uključujući ekonomiju, robotiku i društvene znanosti [31, str. 601]. U poglavlju 3.2.1 se jedan takav primjer razrađuje u detalje.

Jedan od glavnih izazova u VAS-u je koordinacija. U kooperativnim okruženjima, agenti moraju uskladiti svoje akcije kako bi izbjegli sukobe i osigurali da njihovi zajednički napori dovedu do željenog ishoda. To zahtijeva sofisticirane algoritme sposobne za upravljanje ovim interakcijama, osiguravajući da agenti djeluju na način koji je u skladu s ciljevima cijelog sustava. Na primjer, u robotskom sustavu, više robota možda će trebati koordinirati svoje pokrete kako bi sastavili proizvod na proizvodnoj liniji. U takvim scenarijima, sposobnost agenata da komuniciraju i prilagođavaju svoje akcije u stvarnom vremenu je ključna [31, str. 603-605].

U praksi, VAS-i su uspješno implementirani u raznim područjima. U robotici, naprimjer, VAS omogućuje raspoređivanje više robota za obavljanje zadataka koje jedan robot ne bi mogao samostalno izvršiti. Svaki robot djeluje kao agent unutar sustava, doprinoseći ukupnom cilju, poput operacija pretrage i spašavanja, gdje više robota pokriva različita područja. Slično tome, u upravljanju mrežama, VAS može optimizirati protok prometa omogućujući više agenata da upravljaju različitim segmentima mreže, čime se poboljšava ukupna učinkovitost [31, str. 605].

U konačnici, višeagentni sustavi predstavljaju dinamično područje koje evoluira unutar umjetne inteligencije. Omogućavanjem autonomnim jedinicama da djeluju u složenim okruženjima, VAS pruža moćna rješenja za širok spektar primjena. Daljnji razvoj ovog područja obećava unapređenje našeg razumijevanja kooperativnih i kompetitivnih ponašanja u umjetnim i prirodnim sustavima, nudeći nove mogućnosti za inovacije u raznim industrijama.

3.2.1. Dilema zatvorenika

Dilema zatvorenika je možda najpoznatiji primjer u području teorije igara, koji ilustrira složenost donošenja odluka u nekooperativnim scenarijima. Ova dilema se često prikazuje u kontekstu dviju osoba, obično nazvanih zatvorenici, koje su suočene s odlukom hoće li surađivati jedna s drugom ili izdati drugu osobu. Srž dileme leži u činjenici da svaki zatvorenik mora donijeti svoju odluku bez znanja o tome što će drugi odabrati, a ishod za svakog ovisi o izboru drugog.

Klasična postavka uključuje dva zatvorenika koja su uhićena i ispitivana odvojeno. Tužitelj nudi svakom zatvoreniku pogodbu: ako samo jedan svjedoči protiv drugog, bit će pušten na slobodu, dok će drugi služiti desetogodišnju kaznu. Međutim, ako oba zatvorenika svjedoče jedan protiv drugog, svaki će dobiti petogodišnju kaznu. Nasuprot tome, ako obojica šute, služit će samo jednogodišnju kaznu za manji prekršaj [31, str. 606-607].

Iz perspektive teorije igara, dilema se prikazuje u matrici isplata gdje su izbori "svjedočiti" ili "odbiti". Matrica je strukturirana tako da je za oba zatvorenika svjedočenje dominantna strategija, bez obzira na to što drugi zatvorenik učini, svaki će minimizirati svoju kaznu svjedočenjem. To je zato što, ako bilo koji zatvorenik svjedoči, smanjiti će kaznu s deset na barem pet godina. Opisano je prikazano na slici 6. Jedan zatvorenik je imena Ali, a drugi Bo.

Prema izvoru, uobičajena pretpostavka u teoriji igara jest da će racionalni igrač uvijek

	Ali: svjedoči	Ali: odbij
Bo: svjedoči	$A = -5, B = -5$	$A = -10, B = 0$
Bo: odbij	$A = 0, B = -10$	$A = -1, B = -1$

Slika 6: Matrica dileme zatvorenika; preuzeto iz [31]

odabrati dominantnu strategiju i izbjeći onu nad kojom druge dominiraju. Ova situacija vodi do onoga što je poznato kao ravnoteža dominantne strategije gdje oba igrača biraju svoju dominantnu strategiju, što rezultira time da obojica svjedoče i dobivaju petogodišnju kaznu [31, str. 606-607]. Ironija dileme zatvorenika je u tome što bi međusobna suradnja (oboje odbijaju svjedočiti) dovela do boljeg ishoda za oba zatvorenika (jedna godina svaki), ali racionalni vlastiti interes vodi ih do lošijeg zajedničkog ishoda. Ovaj ishod ilustrira izazove u nekooperativnim situacijama gdje individualna racionalnost vodi do kolektivno tek polovično optimalnog rezultata.

Dilema je ovo koja ima duboke implikacije izvan grane kaznenog prava. Koristi se za modeliranje raznih stvarnih scenarija u ekonomiji, političkim znanostima i društvenim interakcijama gdje se pojedinci ili entiteti suočavaju sa sličnim dilemama. Naglašava potencijalne sukobe između individualne racionalnosti i kolektivne dobrobiti, pokazujući kako kompetitivno ponašanje ne mora uvijek dovesti do optimalnih rezultata za sve uključene strane.

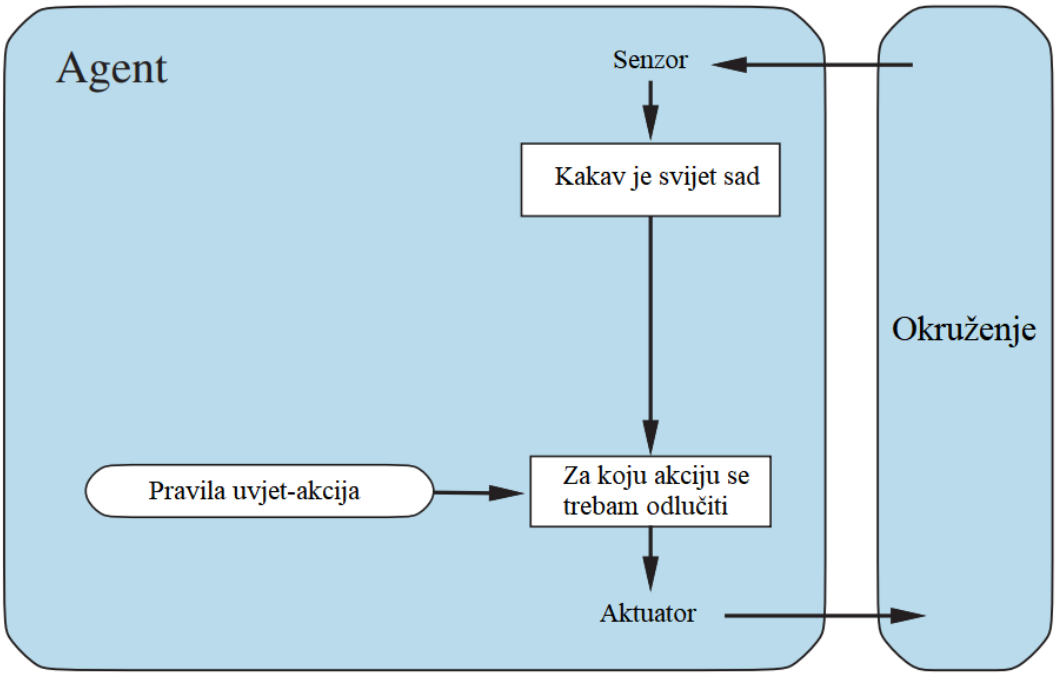
Nadalje, dilema zatvorenika može se proširiti na iterirane verzije, gdje se igra više puta između istih sudionika. U tim scenarijima, strategije poput "Tit for Tat", gdje igrač započinje suradnjom, a zatim oponaša prethodni potez protivnika, mogu dovesti do održive suradnje, nadilazeći iskušenja kratkoročne izdaje radi dugoročnog dobitka.

3.3. Vrste agenata

S obzirom na to da agenti mogu imati različita ponašanja, strukture, sposobnosti doživljavanja okruženja i interakcije s njome, može se doći do zaključka kako u tom pogledu postoji određena klasifikacija. Ovisno o stručnom izvoru, broj vrsta agenata nije uvijek isti, kao ni imena koja su im dodijeljena. U ovom poglavlju navedene su one značajnije, a među kojima je i skupina reaktivnih agenata, koja je u središtu pažnje ovog rada.

3.3.1. Reaktivni agenti

Reaktivni, ili prema nekim izvorima refleksni agenti djeluju na temelju svojih trenutnih percepcija okoline, bez korištenja unutarnje simboličke reprezentacije ili povijesti prošlih akcija. Ovi agenti ne sudjeluju u složenim procesima razmišljanja, već reagiraju na podražaje na izravan i često unaprijed programiran način. Subsumptivna arhitektura, koju je razvio Rodney Brooks, jedan je od najpoznatijih okvira za izgradnju reaktivnih agenata. Ova arhitektura temelji se na ideji da inteligentno ponašanje ne mora nužno zahtijevati eksplicitne reprezentacije ili apstraktno razmišljanje, već može proizaći iz interakcije jednostavnijih ponašanja. Primjerice,



Slika 7: Shematski dijagram reaktivnog agenta; preuzeto iz [31]

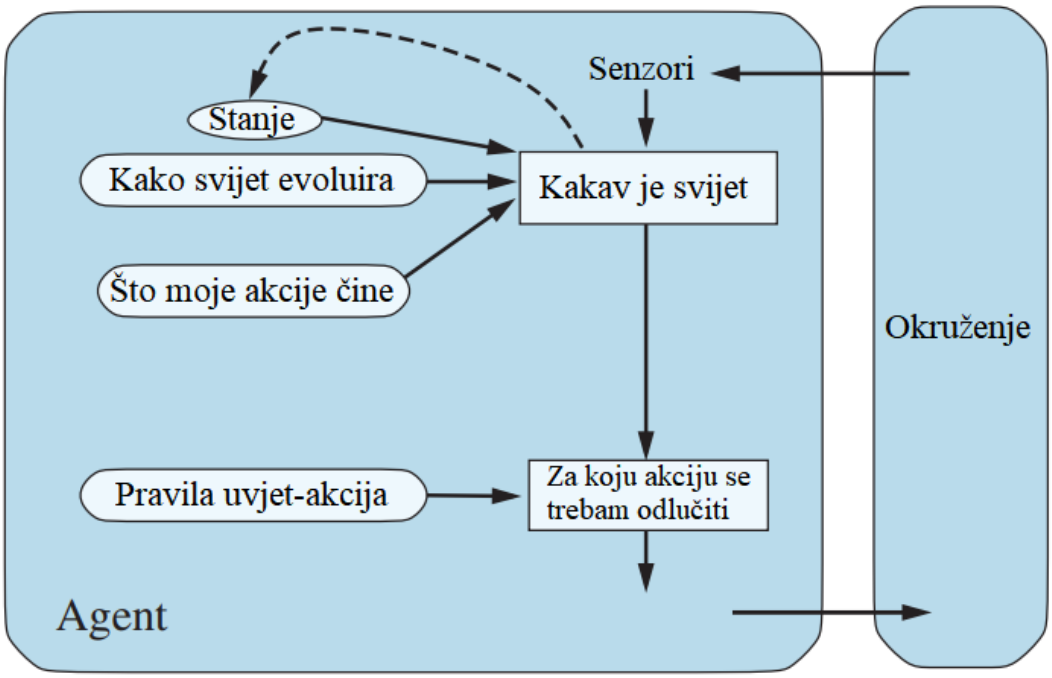
termostat koji uključuje grijanje kada temperatura padne ispod određene točke klasičan je primjer reaktivnog agenta. On reagira na trenutnu temperaturu, ne uzimajući u obzir prethodna stanja ili predviđene buduće promjene [31, str. 49-51].

Ovo je najjednostavniji oblik agenata koji se spominje u literaturi. Usprkos tome, integracija ove vrste agenta u nekim situacijama ima najviše ili čak jedino smisla. Takav je slučaj ovaj rad, gdje agenti nemaju međusobnu interakciju na klasičan način, već isključivo neizravnim dijeljenjem stečenih i oblikovanih podataka. Koordinirani su na način da se svaki od njih pokreće na jedan podražaj iz okoline stvoren baš za njega, tj. svaki od njih ima jedan zadatak koji mora riješiti bez značajno složene logike i gdje je pri tom optimizacija uloženog vremena od najvećeg značaja. Shematski prikaz ovakvog principa funkcioniranja je prisutan na slici 7

3.3.2. Reaktivni agenti s modelom

Ovo je zapravo napredniji oblik jednostavnih refleksnih agenata. Dok jednostavni refleksni agenti donose odluke isključivo na temelju trenutnog opažanja, reaktivni agenti s modelom održavaju unutarnje stanje koje se ažurira tijekom vremena kako bi odražavalo razumijevanje agenta o okolini. Ovo unutarnje stanje informirano je trenutnim opažanjem agenta i modelom kako se svijet razvija neovisno o akcijama agenta [31, str. 51-53].

Struktura ovog tipa agenta obično uključuje nekoliko ključnih komponenti: trenutni status okruženja, tranzicijski model koji opisuje kako se ono mijenja i skup pravila uvjet-akcija. Ovi agenti koriste tranzicijski model za ažuriranje svog unutarnjeg stanja na temelju najnovijeg opažanja, što im omogućuje donošenje odluka koje uzimaju u obzir i trenutni status i predviđeno buduće stanje okoliša. Na primjer, autonomni taksi može koristiti svoje senzore za detekciju prepreka i ažuriranje svog unutarnjeg stanja kako bi reflektirao ne samo ono što trenutno može



Slika 8: Shematski dijagram reaktivnog agenta s modelom; preuzeto iz [31]

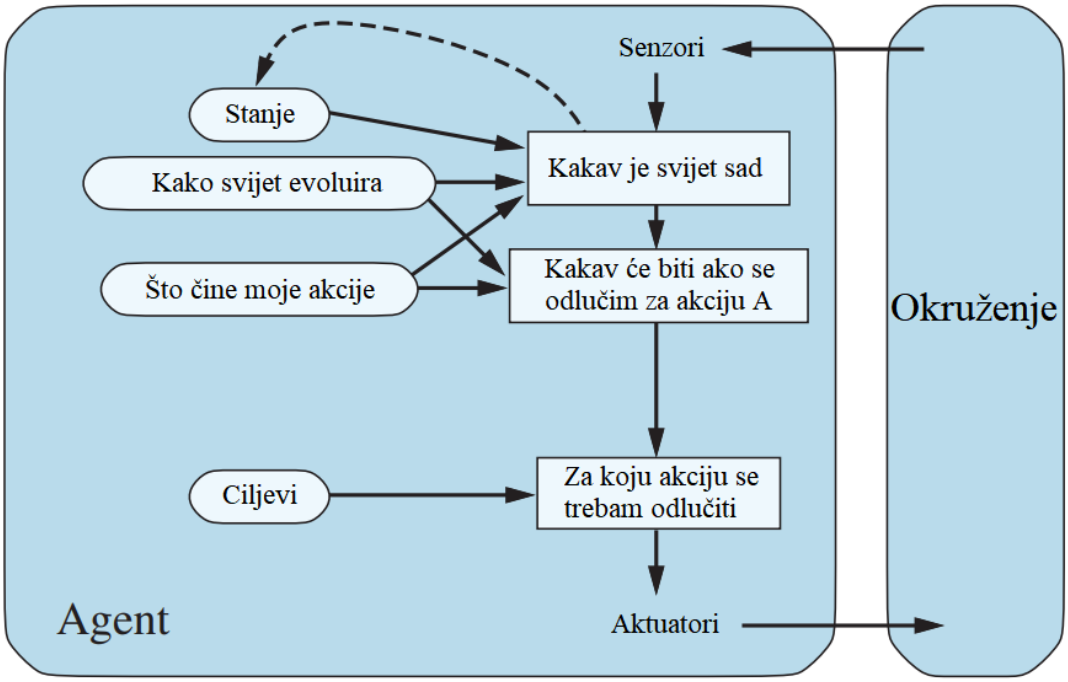
vidjeti, već i ono što pretpostavlja da bi moglo biti ispred, poput prometne gužve skrivene iza velikog kamiona [31, str. 51-53]. Ilustracija je dostupna na slici 8

Reaktivni agenti s modelom posebno su korisni u nepristupačnim okruženjima gdje agent ne može izravno opažati cijelo stanje svijeta. Održavanjem unutarnjeg modela, ovi agenti mogu donositi informirane pretpostavke o neopaženim dijelovima okoline i birati akcije koje imaju veće šanse za uspjeh. Ovaj pristup povećava sposobnost agenta da se nosi s neizvjesnošću i donosi učinkovitije odluke u usporedbi s jednostavnim refleksnim agentima, koji se oslanjaju isključivo na trenutna opažanja bez ikakvog pamćenja ili predviđanja [31, str. 51-53].

3.3.3. Agenti temeljeni na cilju

Ovakva vrsta agenata proširuje mogućnosti refleksnih agenata s modelom uključivanjem ciljeva u svoj proces donošenja odluka. Dok reaktivni agenti s modelom donose odluke na temelju trenutnog stanja i kako se ono razvija, agenti na temelju ciljeva razmatraju buduća stanja i odabiru akcije koje ih približavaju željenom cilju. To zahtijeva da agent ne samo da predviđa učinke svojih akcija, već i da procjenjuje koliko ti učinci doprinose postizanju njegovih ciljeva [31, str. 53-54].

Kod agenata temeljenih na ciljevima, unutarnji model okoliša kombinira se sa ciljem. Agent odabire akcije uzimajući u obzir koje akcije će vjerojatno dovesti do ostvarenja njegovih ciljeva. Na primjer, u slučaju autonomnog taksija, cilj bi mogao biti postizanje određene destinacije. Agent mora birati između različitih dostupnih ruta, uzimajući u obzir čimbenike poput prometa, udaljenosti i vremena, te zatim odabrati akciju koja najbolje ostvaruje njegov cilj. Ova vrsta donošenja odluka složenija je od one kod refleksnog agenta jer zahtijeva od agenta da



Slika 9: Shematski dijagram agenta s ciljem; preuzeto iz [31]

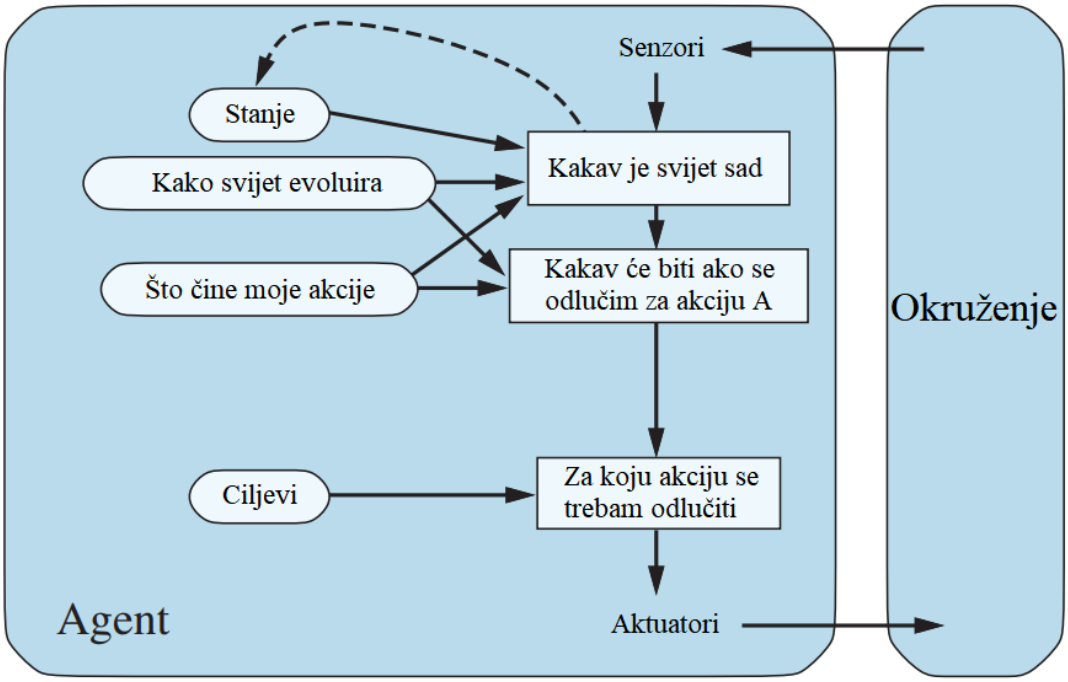
predvidi posljedice svojih akcija i planira nizove akcija kako bi postigao svoj cilj [31, str. 53-54].

Jedna od glavnih prednosti agenata na temelju ciljeva je njihova fleksibilnost. Budući da se proces donošenja odluka temelji na eksplicitnim ciljevima, relativno je lako promijeniti ponašanje agenta mijenjanjem njegovih ciljeva. Primjerice, ako se destinacija autonomnog taksija promijeni, agent na temelju ciljeva može prilagoditi svoje ponašanje bez potrebe za promjenama u osnovnim pravilima uvjet-akcija. Ova prilagodljivost čini agente na temelju ciljeva prikladnima za dinamična okruženja gdje se ciljevi mogu mijenjati tijekom vremena [31, str. 53-54].

3.3.4. Agenti temeljeni na korisnosti

Agenti temeljeni na korisnosti predstavljaju još sofisticiraniji pristup donošenju odluka uvođenjem koncepta korisnosti, koji kvantificira poželjnost različitih stanja svijeta. Dok agenti na temelju ciljeva donose odluke na temelju toga hoće li akcija postići cilj, agenti na temelju korisnosti procjenjuju relativnu vrijednost različitih ishoda i biraju akcije koje maksimiziraju očekivanu korisnost.

Kao što ime sugerira, kod agenata na temelju korisnosti, funkcija korisnosti igra ključnu ulogu. Ta funkcija dodjeljuje numeričku vrijednost svakom mogućem stanju svijeta, predstavljajući agentove preferencije za to stanje. Agent zatim bira akcije koje maksimiziraju njegovu očekivanu korisnost, uzimajući u obzir vjerojatnost različitih ishoda, prikazano na slici 10. Autonomni taksi u ovom slučaju može razmotriti ne samo hoće li stići na odredište, već i čimbenike poput udobnosti vožnje, učinkovitosti goriva i sigurnosti. Funkcija korisnosti omogućuje agentu da uravnoteži te čimbenike, donoseći kompromise između suprotstavljenih ciljeva kako bi pos-



Slika 10: Shematski dijagram agenta temeljenog na korisnosti; preuzeto iz [31]

tigao najbolji ukupni ishod [31, str. 54-55].

Agenti na temelju korisnosti posebno su korisni u složenim okruženjima gdje se više ciljeva može sukobiti. Agent bi mogao birati između brze, ali rizične rute i sporije, ali sigurnije rute. Korištenjem funkcije korisnosti, agent može procijeniti kompromise između brzine i sigurnosti i odabrati opciju koja pruža najvišu ukupnu korisnost. Ovaj pristup omogućuje suptilnije donošenje odluka od pukog nastojanja da se postigne cilj jer uzima u obzir kvalitetu ishoda kao takvog [31, str. 54-55].

Nadalje, agenti na temelju korisnosti mogu se nositi sa situacijama u kojima ciljevi nisu strogo binarni (ostvareni ili neostvareni), već se mogu ostvariti u različitim stupnjevima. Ova fleksibilnost omogućuje da se učinkovito djeluje u okruženjima u kojima optimalno ponašanje ovisi o uravnoteženju više čimbenika, pri čemu svaki utječe na ukupnu korisnost [31, str. 54-55].

3.3.5. Hibridni agenti

Hibridni agenti kombiniraju karakteristike reaktivnih, proaktivnih i ponekad drugih tipova agenata, kao što su deliberativni ili učeći agenti. Ovi agenti su dizajnirani da djeluju u složenim okruženjima gdje čisto reaktivni ili čisto proaktivni pristupi mogu biti nedovoljni. Na primjer, hibridni agenti mogu koristiti reaktivne strategije za rutinske zadatke, dok proaktivne strategije koriste za dugoročne ciljeve. U naprednijim sustavima, hibridni agenti mogu prelaziti između različitih načina rada ovisno o situaciji, poput korištenja reaktivnih ponašanja u situacijama kritičnim za vrijeme i ponašanja temeljenih na planiranju kada to vrijeme dopušta. Arhitekture za hibridne agente često uključuju slojevite strukture, gdje svaki sloj odgovara različitoj vrsti

ponašanja. Arhitektura TouringMachines, naprimjer, je hibridna arhitektura agenta koja integrira različita ponašanja za učinkovitije rješavanje složenih zadataka [30, str. 97-101].

3.4. Društvenost agenata

Društvenost agenta odnosi se na sposobnost agenata da komuniciraju s drugim agentima unutar sustava s više agenata, stvarajući složene društvene strukture i ponašanja. U kontekstu umjetne inteligencije (AI), društveni agenti dizajnirani su da djeluju u okruženjima gdje moraju međusobno surađivati ili se natjecati s drugim agentima. Ove interakcije često zahtijevaju od agenata da pregovaraju, komuniciraju i usklađuju svoje ciljeve s ciljevima drugih, što društvenost čini ključnom komponentom sustava s više agenata.

Društvenost kod agenata posebno je važna u okruženjima gdje akcije jednog agenta mogu izravno utjecati na ishode drugih. To zahtijeva da agenti posjeduju ne samo individualnu racionalnost, već i društvenu svijest razumijevanje namjera, ciljeva i ponašanja drugih agenata. To uključuje strategije poput kompromisa, blefiranja ili davanja ustupaka, koje su sve pod utjecajem razumijevanja društvene dinamike agenta [31, str. 171]. Na primjer, u distribuiranom AI sustavu, agenti bi mogli surađivati kako bi postigli zajednički cilj, što zahtijeva koordinaciju i suradnju. Kako bi to postigli, agenti često slijede društvene konvencije ili norme, tj. definirana ponašanja koja stvaraju glađe interakcije. Ove norme pomažu u smanjenju sukoba i ubrzavaju proces donošenja odluka u sustavima s više agenata pružajući predvidljiv okvir za ponašanje agenta.

Koncept društvenih zakona još je jedan ključni aspekt društvenosti agenta. Društveni zakoni su pravila koja reguliraju interakcije među agentima, osiguravajući da njihovo kolektivno ponašanje bude organizirano i učinkovito. Tako se recimo prometna pravila mogu smatrati društvenim zakonom koji regulira kako se autonomna vozila, koja djeluju kao agenti, trebaju ponašati na cesti. Ova pravila pomažu u sprječavanju nesreća i osiguravaju da promet teče glatko. U AI-u, dizajn takvih društvenih zakona ključan je za održavanje reda i sprječavanje kaosa u okruženjima gdje djeluje mnogo agenata [31, str. 599-605].

Jedan od izazova u razvoju društvenih agenata je osigurati da se mogu prilagoditi promjenjivoj društvenoj dinamici. Agenti nekad moraju biti sposobni učiti nove društvene norme i prilagoditi svoje ponašanje u skladu s tim. Ova prilagodljivost je ključna u dinamičkim okruženjima gdje se društvene konvencije mogu razvijati tijekom vremena. Na primjer, na online tržištu, kupci i prodavači moraju se prilagoditi promjenjivim tržišnim uvjetima, kao što su nove strategije određivanja cijena ili promjene u ponašanju potrošača. To zahtijeva razinu društvene inteligencije koja omogućuje agentima da prepoznaju i reagiraju na promjene u društvenom okruženju.

3.5. Sposobnost učenja

Učenje agenta odnosi se na sposobnost agenta da poboljša svoje performanse tijekom vremena stjecanjem novih znanja ili poboljšanjem postojećih. Učenje je važan aspekt inteli-

gentnog ponašanja, omogućujući agentima da se prilagode svojoj okolini, optimiziraju svoje akcije i rješavaju nepredviđene situacije. U domeni AI-a, agenti sa sposobnošću učenja su dizajnirani da djeluju u okruženjima gdje u početku posjeduju ograničeno znanje, ali mogu unaprijediti svoje sposobnosti kroz stjecanje iskustva. Prema Wooldridgeu, bilo koja od gore navedenih vrsta agenata se može nadograditi sa sposobnošću učenja. [31, str. 56-58].

Agent koji uči obično se sastoji od četiri glavne komponente: element učenja (engl. *learning element*), element performanse (engl. *performance element*), kritika (engl. *critic*) i generator problema (engl. *problem generator*). Element performanse odgovoran je za odabir akcija na temelju trenutnog znanja agenta, dok je element učenja zadužen za poboljšanje znanja i sposobnosti donošenja odluka agenta. Kritika pruža povratne informacije o performansama agenta, pomažući mu da identificira područja za poboljšanje. Konačno, generator problema sugerira istraživačke akcije koje bi mogle dovesti do novih saznanja ili uvida [31, str. 56-58].

Proces učenja kod agenata može se široko razdijeliti na nadzirano učenje, nenadzirano učenje i učenje pojačanjem. U nadziranom učenju, agenti uče iz skupa označenih primjera koje pruža vanjski učitelj. Ova metoda se često koristi u zadacima klasifikacije, gdje se agent obučava da prepozna obrasce u podacima. Nenadzirano učenje, s druge strane, uključuje otkrivanje skrivenih struktura u podacima bez vanjskog navođenja. Ovaj pristup često se koristi u zadacima grupiranja, gdje agent grupira slične stavke na temelju njihovih karakteristika. Učenje s pojačanjem posebno je relevantno za sustave temeljene na agentima, jer uključuje učenje iz posljedica akcija. U ovom pristupu, agent uči interakcijom sa svojom okolinom i primanjem povratnih informacija u obliku nagrada ili kazni. Cilj agenta je maksimizirati svoju kumulativnu nagradu tijekom vremena odabirom akcija koje vode do povoljnih ishoda. Učenje pojačanjem široko se koristi u scenarijima gdje agent mora naučiti politiku donošenja odluka u dinamičnom i neizvjesnom okruženju [31, str. 653-782].

Prednost agenata koji mogu učiti je njihova sposobnost djelovanja u početno nepoznatim okruženjima. Za razliku od tradicionalnih AI sustava koji se oslanjaju na unaprijed definirana pravila, agenti sa sposobnošću učenja mogu se prilagoditi novim situacijama ažuriranjem svoje baze znanja. Ova prilagodljivost je ključna u okruženjima koja se stalno mijenjaju ili gdje agent susreće nove situacije koje nisu bile predviđene tijekom početnog programiranja. Tako primjerice, agent koji uči u robotskom sustavu može naučiti navigirati kroz nepoznat teren postupno poboljšavajući svoje algoritme za pronalaženje puta na temelju pokušaja i pogrešaka [31, str. 56-58].

Međutim, dizajniranje učinkovitih agenata sa sposobnošću učenja predstavlja nekoliko izazova. Jedna od glavnih poteškoća je osigurati da agent može uravnotežiti istraživanje i eksploataciju. Istraživanje uključuje isprobavanje novih akcija kako bi se otkrili njihovi učinci, dok se eksploatacija fokusira na odabir akcija za koje je poznato da donose visoke nagrade. Pronalaženje prave ravnoteže između ova dva pristupa ključno je kako bi se osiguralo da agent ne zapne u polovično optimalnim ponašanjima ili propusti bolje prilike [31, str. 163].

3.6. Praktični dio

Nakon teorijskog dijela u kojem su opisani temeljni koncepti i pojmovi u domeni više-agentnih sustava, vrijeme je za prelazak na praktičan dio u kojemu je opisana aplikacija koja, među ostalim, služi kao primjer implementacije agenata u konkretnom sustavu.

3.6.1. O aplikaciji

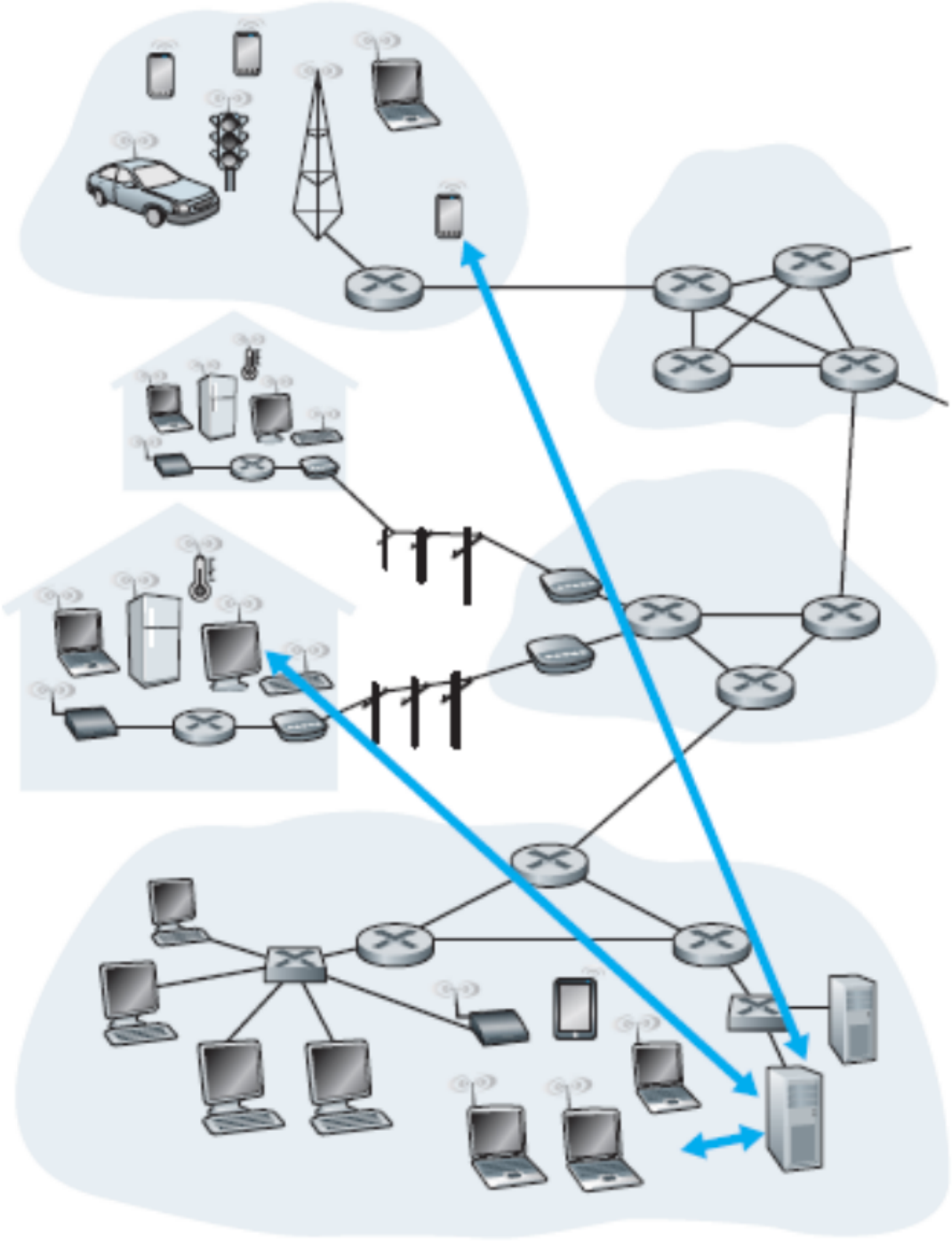
Aplikacija koja je predmet analize u ovom poglavlju nazvana je *MatchPitch*. Radi se zapravo o sustavu sačinjenom od dva dijela. Klijent (engl. *client*) na kojemu je frontend ili sučelje aplikacije preko tehnolgije WebSocket opisanog u pogavlju 2.1.2.3 komunicira sa poslužiteljem (engl. *server*) i ondje prisutnim *backendom* u kojemu je prisutna gotovo sva logika aplikacije. Klijent se može preuzeti s ove poveznice, a poslužitelj s ove poveznice.

3.6.1.1. Klijent-server arhitektura

Prema Kuroseu i Rossu, ovakva vrsta arhitekture podrazumijeva uvijek aktivnog domaćina (engl. *host*), naziva poslužitelj, koji poslužuje druge domaćine na temelju njihovih zahtjeva, a ti domaćini se zovu klijenti. Klasičan primjer je web-aplikacija za koju uvijek aktivan poslužitelj rješava zahtjeve nastale na web-preglednicima (engl. *web browser*) pokrenutima na računalima domaćina. Prilikom toga, kad poslužitelj od klijenta primi zahtjev za određeni objekt, on odgovara na način da natrag na isti način pošalje taj traženi objekt. Objekti mogu biti različitih tipova i sadržaja. U našem slučaju, klijent prvo šalje poruku s tekstualnim sadržajem na poslužitelj, a odgovor mu stiže u JSON podatkovnom obliku [32, poglavlje 2.1.1.]. Tim podacima se upotpunjuje izgled web-stranice.

Važnije karakteristike ovakve arhitekture su da klijenti međusobno nikad izravno ne komuniciraju, već se sve odvija preko poslužitelja. Primjerice u aplikacijama koje sadrže chat funkcionalnosti, premda se može činiti da nije tako, poslužitelj je taj preko kojeg u svakom scenariju ide komunikacija dvaju ili više klijenata. Nastale poruke prvo idu do računala poslužitelja, koji ih potom preusmjerava do potrebnih adresa klijenata. Osim toga, svi poslužitelji imaju fiksnu i dobro poznatu adresu koju nazivamo IP adresa (engl. *IP address*), zbog čega bi oni u principu u svakom trenutku trebali moći odgovarati na zahtjeve klijenata, kad ih oni kontaktiraju [32, poglavlje 2.1.1.]. Na slici 11 prikazana je ta interakcija za koju bi se, kad bi se išlo u detalje, vidjelo da zapravo nije toliko jednostavna. Da bi neki podatak došao od klijenta do poslužitelja, mora proći kroz više slojeva, a to su fizički sloj (engl. *physical layer*), sloj podatkovne veze (engl. *data link layer*), mrežni sloj (engl. *network layer*), transportni sloj (engl. *transport layer*), sloj sesije (engl. *session layer*), prezentacijski sloj (engl. *presentation layer*), aplikacijski sloj (engl. *application layer*). Svaki od njih predstavlja jedan dio puta na spomenutoj relaciji. U detalje se, što se tiče domene mreže računala, ovdje neće ići, pošto to izlazi izvan okvira tematike ovog rada.

Često je slučaj da jedan poslužitelj u nekom sustavu neće moći zadovoljiti zahtjeve svih klijenata ako ih ima mnogo. Zbog tog razloga se izgrađuju podatkovni centri (engl. *data center*)



Slika 11: Klijent-server arhitektura, preuzeto iz [32]

Slika 12: MatchPitch logotip, vlastita izrada

gdje je prisutan dovoljan broj domaćina koji onda sačinjavaju moćan virtualni poslužitelj. E-commerce platforme kao Amazon, eBay, Alibaba ili društvene mreže kao Facebook, Twitter i Instagram su samo neki od mnogih primjera. U našem slučaju za sad to dakako nije potrebno, već je dovoljno da se jedan klijent i poslužitelj pokrenu na različitim terminalima unutar istog računala.

3.6.1.2. Ideja, primjena i preduvjeti aplikacije

U ovom poglavlju opisana je ideja izrade i svrha postojanja aplikacije MatchPitch. Iz naziva nije teško prepoznati da se radi tipu proizvoda koji se koristi u domeni sporta i klađenja. Kako se svijet i tehnologija postepeno razvijaju, čovječanstvo sve više teži k tome da što više informacija i mogućnosti bude dostupno nadohvat ruke. Kada se priča o utakmicama, vjernim pratiteljima je od velike važnosti brzina, lakoća i točnost dobivanja informacija o primjerice rezultatima i koeficijentima oklada u kladionicama. Upravo zbog tog razloga jako su popularne aplikacije koje uz malo ili nimalo napora serviraju informacije koje se mogu konzumirati u gotovo bilo kojem trenutku dana. One također moraju biti intuitivno raspoređene na ekranima uređaja i uz njih se mora pružiti mogućnost određene interakcije na relaciji korisnik-sučelje kako bi se što prije moglo doći do traženog.

Sve navedeno može se naći u aplikacijama koje nogometne rezultate prikazuju uživo. Vrsta je to proizvoda u koju spada MatchPitch. Naravno, u današnje vrijeme nije ni malo lagano izmisliti u potpunosti originalan proizvod koji će određenoj skupini ljudi biti od koristi, pa tako na tržištu već možemo naći primjere kao što su SofaScore, FlashScore, FootMob i slični. Iz takvih je primjera autor aplikacije ovog rada djelomično uzeo inspiraciju, a preostali dio je došao iz vlastite želje za određenim poboljšanjima, ispravcima nedostataka ili naprosto promjenama koje bi rezultirale izradom proizvoda prema malo drugačijim standardima i zamislima. Vlastito izrađen logotip u alatu Canva je prikazan na slici 12.

3.6.1.3. Projektna struktura

Ovo poglavlje gdje se objašnjava i prikazuje razmještaj datoteka unutar projekta posebice je važno za dio klijenta jer se već na njemu može vidjeti veliki doprinos softverskog okvira Next.js i React.

Strukturiranje projekta je ključni aspekt razvoja skalabilnih i održivih aplikacija u Reactu i Next.js. S obzirom na fleksibilnost koju ovi alati pružaju, važno je usvojiti strukturu projekta

koja prati najbolje prakse, a istovremeno zadovoljava specifične potrebe aplikacije.

Način na koji je projekt strukturiran može značajno utjecati na njegov životni ciklus razvoja. Dobro organizirana baza koda poboljšava čitljivost, pojednostavljuje održavanje i omogućava programerima učinkovitiju suradnju. Ovo je posebno važno u većim projektima gdje više programera radi na različitim značajkama istovremeno. Logičkim grupiranjem povezanih datoteka i komponenti, struktura projekta pomaže u izolaciji odgovornosti i sprječava da baza koda postane kaotična kako aplikacija raste [14].

U Next.js, struktura projekta nije strogo definirana, omogućujući programerima da organiziraju bazu koda prema vlastitim preferencijama i potrebama projekta. Ipak, pridržavanje određenih konvencija i najboljih praksi može pružiti solidnu osnovu koja smanjuje složenost i olakšava skaliranje.

Tipičan Next.js projekt započinje s minimalnom postavkom, koja obično uključuje direktorije kao što su *pages*, *public* i *styles*. Kako aplikacija raste, mogu se dodati dodatni direktoriji kako bi se organizirale komponente, alati i resursi na način koji održava bazu koda čistom i lako navigabilnom [33].

Direktorij pages je ključna značajka Next.js, omogućujući usmjeravanje (engl. *routing*) temeljeno na datotekama. Svaka datoteka u ovom direktoriju odgovara putanji (engl. *route*) u aplikaciji, pri čemu ime datoteke određuje put URL-a (engl. Uniform Resource Locator). Naprimjer, datoteka pod nazivom *about.js* unutar direktorija pages automatski kreira rutu */about*. Ovaj sustav eliminira potrebu za eksplicitnim konfiguracijama ruta i zadržava logiku usmjeravanja intuitivnom i jednostavnom [33].

Pored direktorija *pages*, uobičajeno je vidjeti i direktorij *components*, gdje se pohranjuju višekratne UI komponente. Ovaj direktorij obično sadrži React komponente koje se koriste na više stranica ili značajki aplikacije. Grupiranje ovih komponenti na jednom mjestu potiče ponovnu upotrebu i osigurava dosljednost korisničkog sučelja aplikacije [34].

Objašnjena struktura vidljiva je na slici 13, gdje se osim spomenutih direktorija koristi vlastiti *functions*. U njemu su spremljeni stilovi koje je moguće upotrijebiti više puta kroz projekt i postavka za omogućavanje povezivanja klijenta s poslužiteljem preko WebSocketa u datoteci *Socket.ts*. Svaki *page.tsx* u određenoj mapi definira što će se prikazati kada se pokrene neka putanja, dakle zadana početna stranica. *Styles* direktorij u slučaju ovog projekta ne postoji pošto se CSS stilovi koriste na drugi način zbog biblioteke Tailwind. Postoje jedino oni osnovni, koji su definirani na razini cijelog projekta, a oni su prisutni u *globals.css*. Naravno, prikazano stablo projekta je samo onaj centralni i najvažniji dio naziva *src* direktorij u kojemu se događa sva značajna logika klijenta. Od onoga što na slici nije vidljivo valja izdvojiti *public* direktorij s preuzetim slikama u *jpeg* ili *png* te ikonama u *svg* formatima, ali i *ttf* datoteke gdje je preuzet glavni font aplikacije naziva Anek Devanagari u *regular* rezu. Tu spadaju i datoteke korištenih biblioteka u *node modules* direktoriju te konfiguracijske datoteke za *next.js* i *tailwind*. Bilo bi previše uključiti ih sve u isti stablasti dijagram.

Za potrebe ovog rada nije bilo potrebno raditi više od 4 različitih putanja, odnosno web-stranica. Svaka od njih u putanji osim u nastavku navedenog izgleda sadrže ID-jeve (engl.

identification) potrebnog entiteta, na temelju kojeg se šalju zahtjevi prema poslužitelju i on potom traži od API-ja da se dostave podaci o dotičnom, npr. ("*player?id=376*").

- Početna stranica ("/")
- Stranica lige ("/league")
- Stranica igrača ("/player")
- Stranica tima ili kluba ("/team")

Što se tiče dijela s poslužiteljem, prikazanog na slici 14, stvari su poprilično jednostavne. Klase svih *SPADE* agenata smještene su u zasebne datoteke u *Agents* direktoriju, odakle ih se uvozi (engl. *import*) u logički gledano glavnu datoteku tog dijela aplikacije, *Main.py*. Globalne varijable i funkcije smještene su u *config.py*, dok se u *Leagues.py* čuvaju podaci u JSON obliku za prvenstva, koji su u sučelju na početnoj stranici prvi stupac s lijeva. *Dockerfile* i *requirements.txt* su tu za slučaj potrebe izgradnje *Docker* kontejnera (engl. *Docker container*).

Kao što se na slikama 13 i 14 može primjetiti, prilikom dodavanja datoteka u klijentu i poslužitelju slijedile su se drugačije konvencije imenovanja (engl. *naming convention*). U klijentu se koristio tzv. *Pascal Case*, spojene riječi s velikim početnim slovima, a u poslužitelju tzv. *Snake Case*, pri kojemu se imenuje riječima odvojenim donjom crtom (engl. *underscore*). Radilo se na ovaj način radi bolje preglednosti u razvojnom okruženju prilikom rukovanja s mnoštvom datoteka.

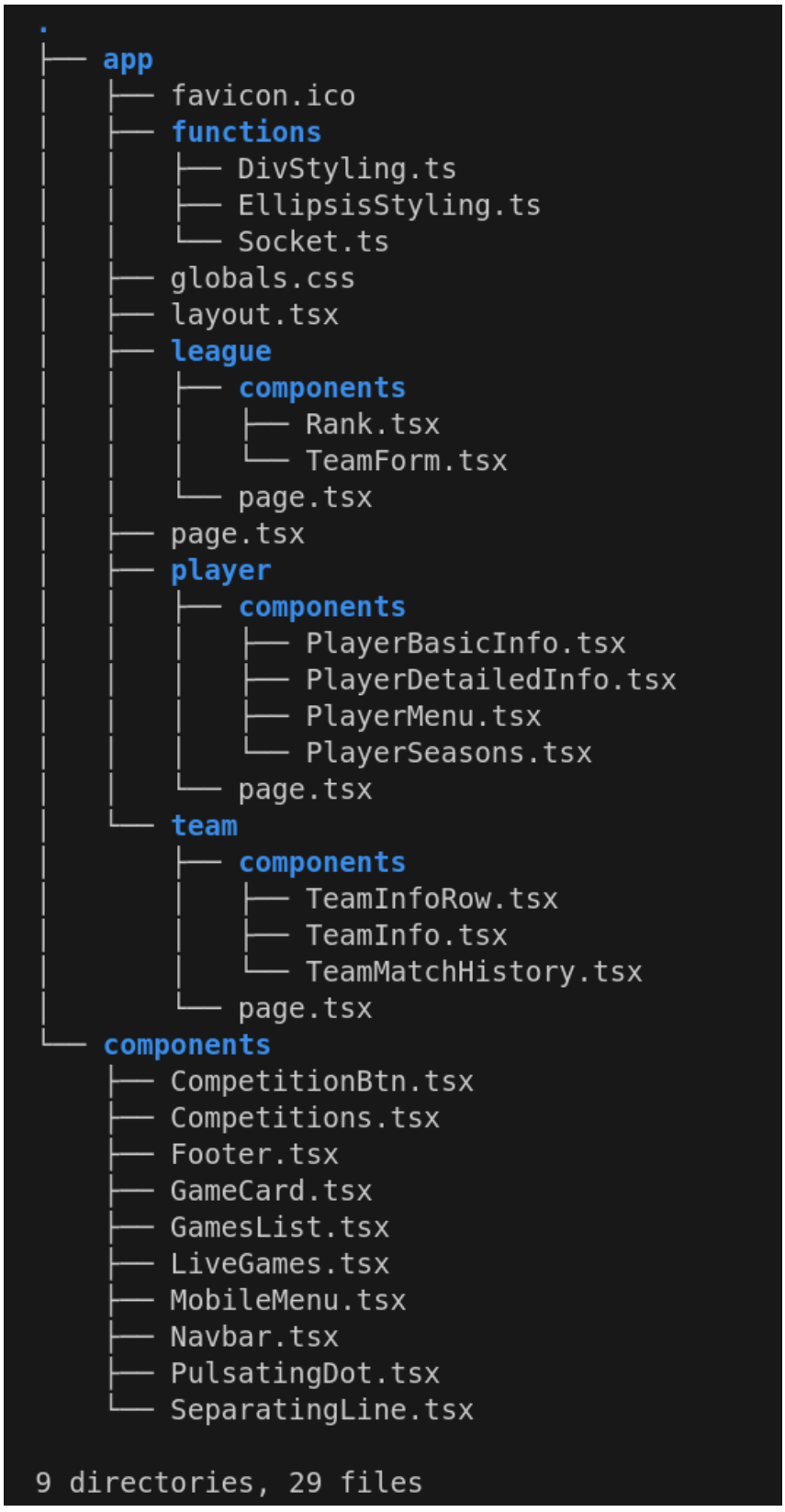
3.6.1.4. Opis sučelja i funkcionalnosti

Već je spomenuto kako je izvedba sučelja dobrim dijelom tipizirana s obzirom na ostale aplikacije ove vrste. Pošto se koristi React, nije bilo teško organizirati stranicu prema njezinim dijelovima ili komponentama, stoga će se u ovom poglavlju ukratko proći kroz svaku od njih.

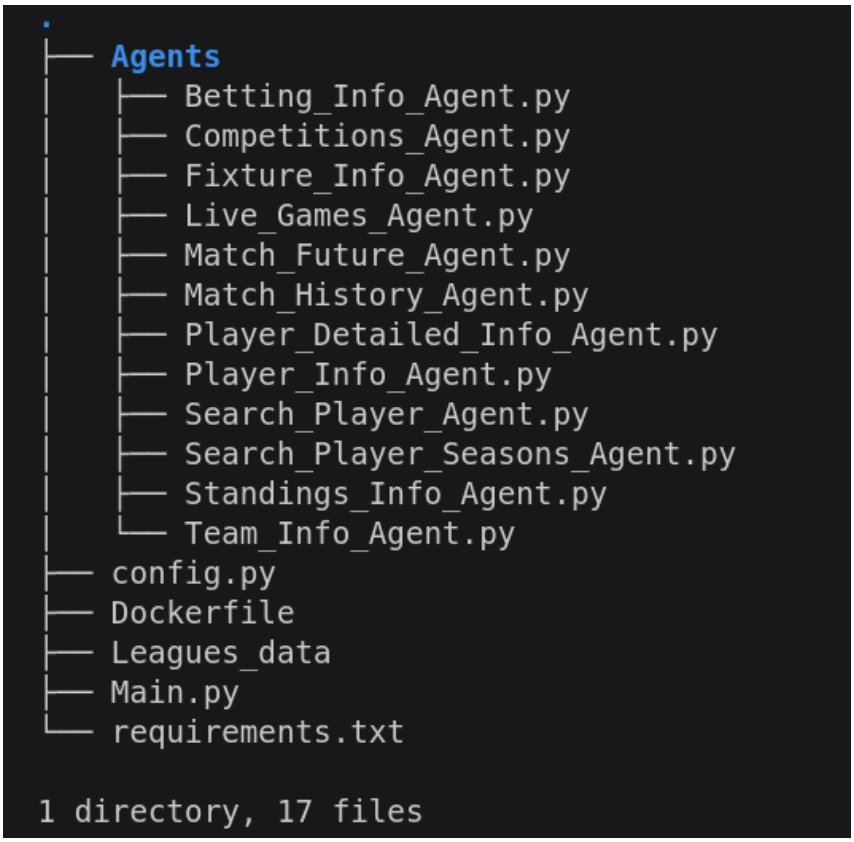
Na samom vrhu svake stranice se standardno nalazi navigacijska traka (engl. *navbar*) u kojoj je s jedne strane smješten vlastito kreiran logotip aplikacije, a s druge strane element tražilice (engl. *search bar*). Za sad se u njoj mogu pretraživati samo igrači, no kako je u planu nastavak razvoja i proširenje funkcionalnosti aplikacije, uskoro će se moći i klubovi, treneri te lige. Na dnu stranice svakako nalazi se podnožje (engl. *footer*), gdje je također pozicioniran logotip s lijeve strane, dok je s desne potpis autora.

Ono što se od jedne do druge stranice razlikuje jest što se nalazi u između navigacijske trake i podnožja. Cilj je bio gdje god je bilo moguće primijeniti isti kostur elemenata, pa tako su u svim stranicama, osim u onoj za pojedinu ligu, postavljena tri stupca okomitog poravnanja koji u sebi sadržavaju podatke određene kategorije.

Krenimo od početne stranice (engl. *home page*), vidljive na slici 15. Na njoj se u prvome s lijeva prikazuju već spomenuta nogometna natjecanja koja su grupirana prema državama u kojima se taj sport održava. Kada se klikne na pojedinu državu, otvara se padajući izbornik (engl. *dropdown menu*) iz kojeg je klikom na neko natjecanje moguće otići na pripadnu stranicu.



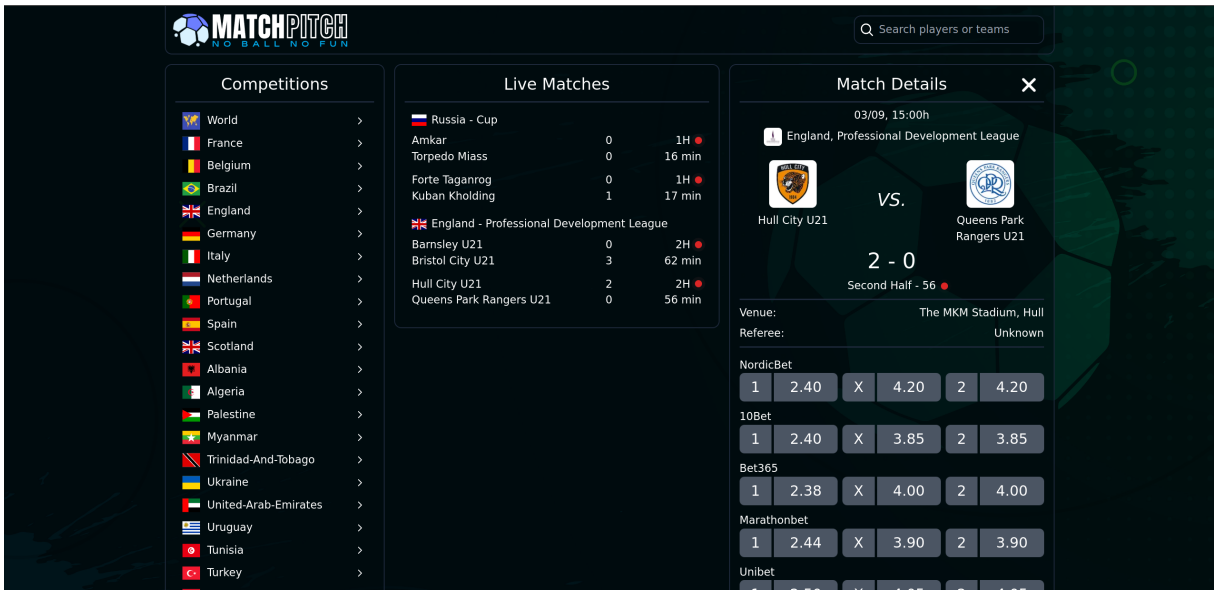
Slika 13: Projektna struktura klijenta u *src* direktoriju u stablastom dijagramu, vlastita izrada



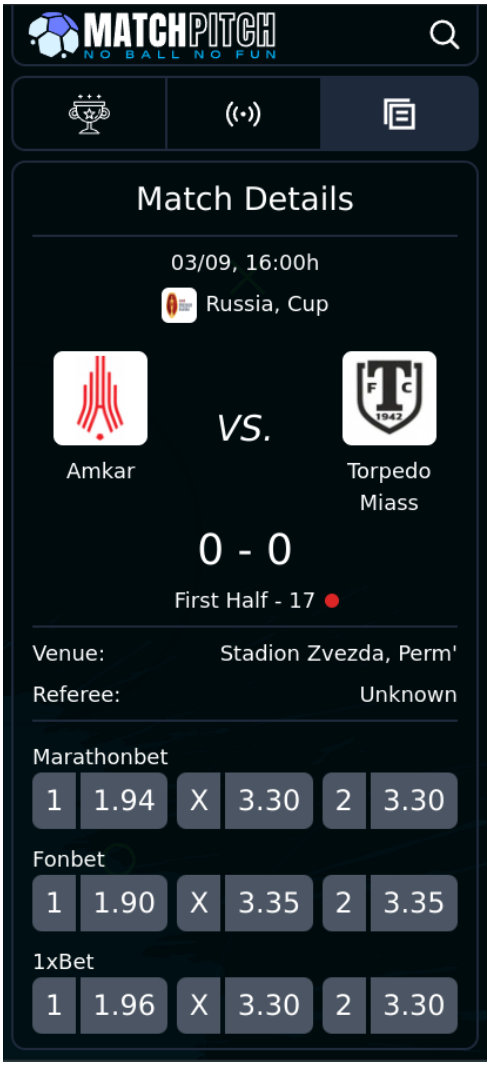
Slika 14: Projektna struktura poslužitelja u stablastom dijagramu, vlastita izrada

Trenutno su dostupne stranice samo za prvenstvena natjecanja, dok će se kupovi dodati u idućoj nadogradnji. Ide li se nadesno, dolazi se do stupca u kojemu su prikazane utakmice koje se održavaju uživo. Radi se o utakmicama koje su grupirane po natjecanjima i za koje agenti podatke dohvaćaju i prosljeđuju točno svakih 10 sekundi, dokle god je ta stranica otvorena. Nekad se dakako može dogoditi da nema nikakvih dostupnih podataka o takvim utakmicama, što će se navesti prigodnim tekstom. Zadnji stupac se otvara samo kad se odabere neka utakmica i to je primjer komponente koja se koristi višekratno. Radi se o komponenti *GameCard* koja prikazuje detalje odabrane utakmice i sa koje se odabirom jednog od dva sudionika može izravno otići na pripadajuću stranicu tima. Osim toga, na njezinom dnu je kutak za kladioničare gdje se prikazuju koeficijenti svih dostupnih kladioničarskih poduzeća i to samo brojke vezane za pobjedu domaćina (1), neriješeni ishod (X) i pobjedu gosta (2), što je osnovno.

Kako je osiguravanje responzivnosti web-stranica danas jedna od najvažnijih značajki pri razvoju, ona je morala i ovdje biti izvedena. U mobilnoj verziji početne stranice dostupnoj na slici 16 vidljivo je kako se samo jedan od tri stupca prikazuje u jednom pogledu, a koji je stupac trenutno aktivan može se birati uz pomoć izbornika odmah ispod navigacijske trake. Za mobilnu verziju taj način je prisutan kod svih stranica, osim kod stranice lige gdje to nije potrebno zbog samo jednog stupca.



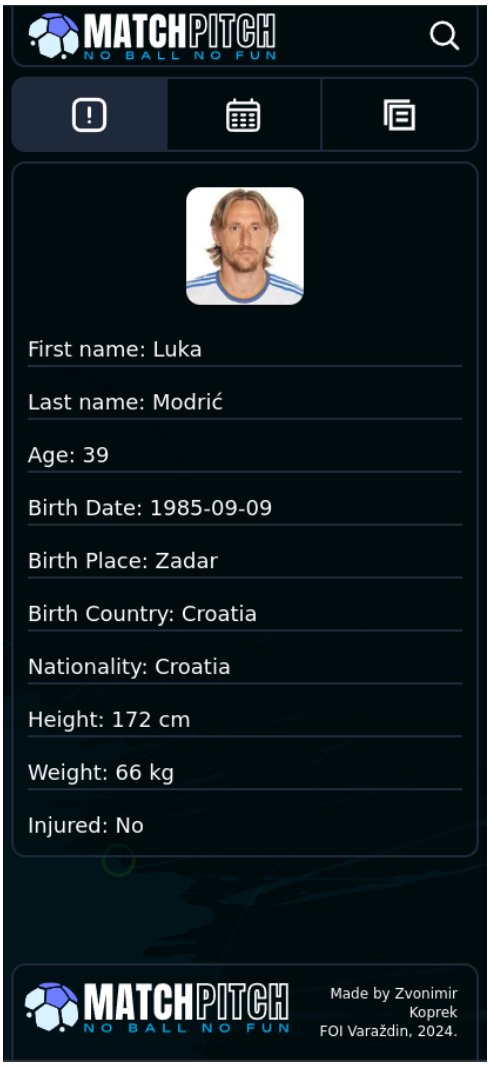
Slika 15: Slika zaslona početne stranice, vlastita izrada



Slika 16: Slika zaslona početne stranice u mobilnoj verziji, vlastita izrada

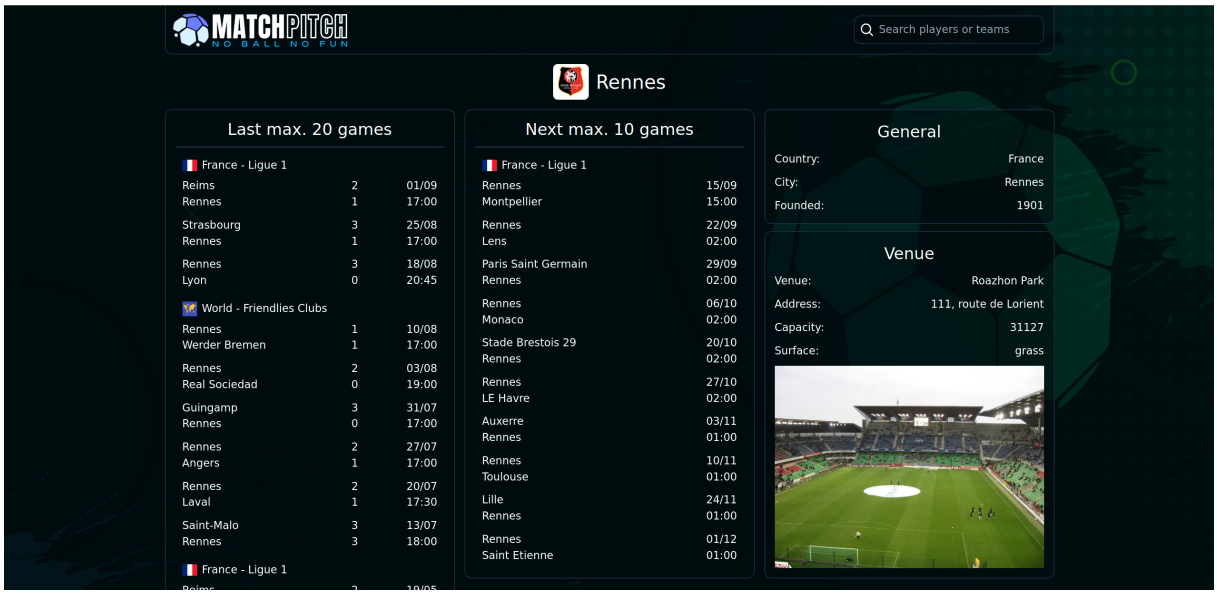
Na stranici igrača na slikama 17 i 18 prvi stupac s lijeva je zadužen za prikaz općenitih informacija o pojedincu. Drugi stupac prikazuje sve sezone za koje postoje podaci o igraču, a kada se jedna odabere dobije se treći stupac gdje se dobije kronološki pregled statističkih podataka ovisno o natjecanju u kojem igrač nastupao i o klubu.

Slika 17: Slika zaslona stranice igrača, vlastita izrada



Slika 18: Slika zaslona stranice igrača u mobilnoj verziji, vlastita izrada

Stranica kluba, vidljiva na slikama 19 i 20 također ima podatke raspoređene po stupcima. Prvi s lijeva je tu da prikaže povijest odigranih utakmica, zatim se u onome pored njega prikazuju buduće utakmice, a ako se iz jednog od ta dva odabere jedan dvoboj, dobije se posljednji s detaljima. Taj posljednji je popunjen detaljima utakmice i tu se ponovo koristi komponenta *GameCard*. Postoji opcija zatvaranja, a ukoliko se zatvori, dobije se kao i prilikom učitavanja stranice stupac s općenitim informacijama o dotičnom klubu i stadionu na kojem igra.



Slika 19: Slika zaslona stranice tima, vlastita izrada

Slika 20: Slika zaslona stranice tima u mobilnoj verziji, vlastita izrada

Za kraj ovog poglavlja spomenut će se još stranica koja je drugačija od ostalih, a to je ona za ligu. Na slikama i vidljiva je jedna tablica sa standardnim podacima za neku ligu, a kad se s mišem prođe preko rednih brojeva na početku svakog retka dobije se informacija o tome u koje europsko natjecanje to mjesto vodi na kraju sezone. Osim stanja prikazanog po učitavanju tablice, može se dobiti i konačno stanje u prošlim sezonama.

Slika 21: Slika zaslona stranice lige, vlastita izrada

Slika 22: Slika zaslona stranice lige u mobilnoj verziji, vlastita izrada

3.6.2. Tokovi procesa u aplikaciji

Općenito gledano, pošto se u ovoj aplikaciji isključivo radi s reaktivnim agentima opisanim u poglavlju 3.3.1, logika na temelju koje njihov rad funkcionira nije odviše složena. 12 agenata koji svoj životni ciklus provode unutar računala poslužitelja mogu se svrstati u dvije kategorije, oni s jednokratnim ponašanjem i oni s periodičnim ponašanjem. U ovom poglavlju se iz tog razloga analizira rad triju agenata, *Match History Agent* i *Live Games Agent*, koji su odabrani kao predstavnici tih dviju kategorija.

3.6.2.1. Agenti s jednokratnim ponašanjem

Match History Agent je SPADE agent koji sudjeluje u procesu dobavljanja podataka za stranicu tima, točnije za njezin prvi stupac s lijeva. On se pokreće unutar Main.py skripte na temelju pristigle poruke unutar tzv. *webscoket eventa* koji je ovom prigodom nazvan "*get match history*", a iniciran od strane klijenta prilikom učitavanja stranice nekog tima. Iz *URL-a* se dohvaća jedinstveni identifikator tima i preko klijenta se u sadržaju poruke prosljeđuje do

poslužitelja. U isječku 2 je to prikazano.

```
@GLOBAL.socketio.on('get_match_history')
def handle_get_match_history(msg):
    async def start_agent():
        GLOBAL.team_ID = msg
        get_match_history_agent = Match_History_Agent(
            "Match_History_Agent@localhost", "secret")
        await get_match_history_agent.start()
        await get_match_history_agent.fetch_behaviour.join()
    asyncio.run(start_agent())
```

Isječak koda 2: Pokretanje agenta *Match History Agent*

Ono što zatim slijedi je pokretanje ponašanja *Fetch Behaviour* unutar samog agenta te odmah dohvaćanje potrebnih podataka preko HTTPS veze sa API-Football. Šalje se GET zahtjev (engl. request) i traže se podaci za 20 zadnjih utakmica nekog tima. Tijelo dobivenog odgovora se iz bajtova parsira (engl. *decode*) u tip *string*, a potom i u *Python Dictionary* objekt kojim nad kojim se mogu vršiti operacije, što je prikazano u isječku 3. Biblioteka asyncio se ovdje koristi kako bi se osiguralo da se funkcija pokreće u primjerenom *event loopu* i time izbjegla greška oko asinkronog dijela programskog koda.

```
conn = http.client.HTTPSConnection(
    "api-football-v1.p.rapidapi.com")
conn.request("GET", "/v3/fixtures?team=" +
             GLOBAL.team_ID + "&last=20", headers=GLOBAL.headers)

res = conn.getresponse()
string = res.read().decode('utf-8')
data = json.loads(string)
previous_game_competition = None
```

Isječak koda 3: Dohvaćanje i parsiranje podataka sa API-FOOTBALL

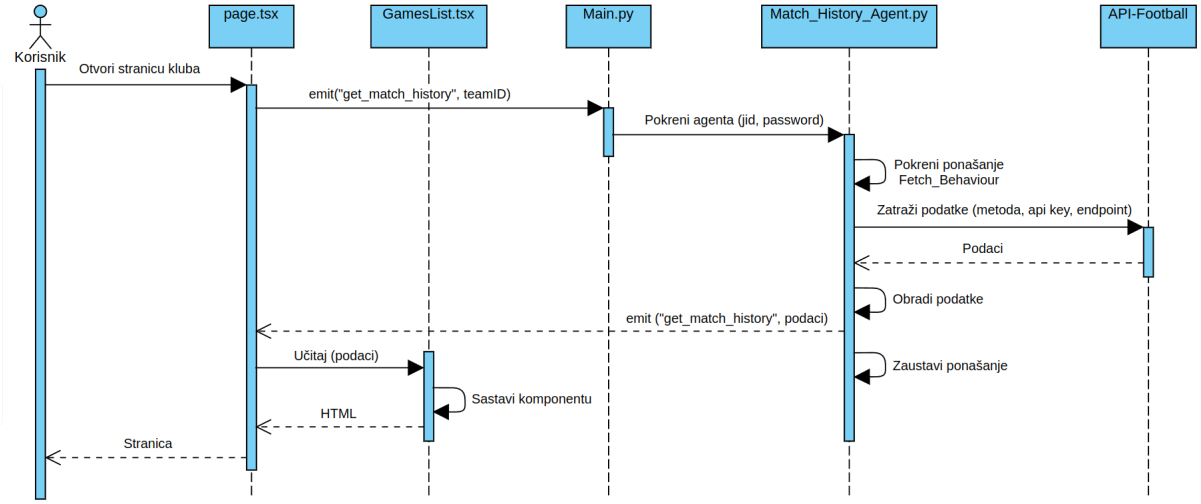
Za svaki element, tj. utakmicu se zatim odvija *for* petlja u kojem se prvo provjerava je li ona bila u sklopu istog natjecanja kao i ona prošla. Ako jest, to se označi posebnom vrijednošću koja kasnije klijentu omogućava lakše grupiranje po natjecanjima. Nakon toga, pošto je termin dvoboja originalno zapisan u UTC+2 formatu, potrebna je prilagodba lokalnom vremenu. To se odvija preko funkcije *UTCToLocalDateTime* u *config.py*. Navedena radnja namjerno nije stavljena u tijelo posebnog agenta jer se mora pozivati više desetaka puta tijekom jednog osvježavanja podataka, a pokretanje i zaustavljanje agenta tom prilikom imalo bi prevelik negativan vremenski utjecaj. U varijablu koja se kasnije klijentu šalje natrag kao odgovor na taj *event* dodaje se sastavljeni objekt u svakoj iteraciji. Klijent odgovor dobije u JSON obliku.

Ovim procesom se zapravo osigurava da se dalje prenose samo oni podaci koji su potrebni. Naime, API-FOOTBALL po svojoj usluzi daje mnoge podatke koji se u ovom projektu ne koriste, a na ovaj način se dobiveni JSON "sreže" na samo ono što će se klijentu u pregledniku dati na uvid.

```
previous_game_competition = None

GLOBAL.match_history_response.clear()

for item in data["response"]:
    if previous_game_competition != item["league"]["name"]:
        previous_game_competition = item["league"]["name"]
    else:
        item["league"]["name"] = "Same"

    local_date, local_time = GLOBAL.UTCToLocalDateTime(
        item["fixture"]["date"])

    GLOBAL.match_history_response.append({
        'id': item["fixture"]["id"], 'country': item["league"]["country"],
        'countryFlag': item["league"]["flag"], 'competition':
        ↪  item["league"]["name"],
        'homeTeam': item["teams"]["home"]["name"], 'homeTeamLogo':
        ↪  item["teams"]["home"]["logo"],
        'awayTeam': item["teams"]["away"]["name"], 'awayTeamLogo':
        ↪  item["teams"]["away"]["logo"],
        'goalsHome': item["goals"]["home"], 'goalsAway': item["goals"]["away"],
        'date': local_date, 'time': local_time
    })

emit("get_match_history", GLOBAL.match_history_response)
```

Isječak koda 4: Operacije nad dohvaćenim podacima prije prosljeđivanja klijentu

Nakon povratnog emitiranja (engl. *emit*) podataka preko WebSocket, agentovo ponašanje se gasi i on životni ciklus time završava. Na računalu klijenta se tada još, da bi se dobio konačan HTML kod za preglednik, JSX kod mora kompajlirati (engl. *compile*) u običan JavaScript kod, a zatim i renderirati (engl. *render*) u konačan oblik koji se korisniku prikaže na ekranu. Programski kod klijentove strane ovdje neće biti prikazan jer je sva važna logika sadržana i riješena na poslužitelju. Ovim procesom može se sastaviti dijagram slijeda (engl. *sequence diagram*) koji još vjernije prikazuje što se sve i kojim redom događa, a dostupan je na slici 23.



Slika 23: Dijagram slijeda otvaranja stranice tima, vlastita izrada

Na opisani način funkcioniraju svi ostali agenti, osim *Live Games Agent*, *Fixture Info Agent* te *Betting Info Agent*. Zašto je tome tako, objašnjeno je u idućem poglavlju.

3.6.2.2. Agenti s periodičnim ponašanjem

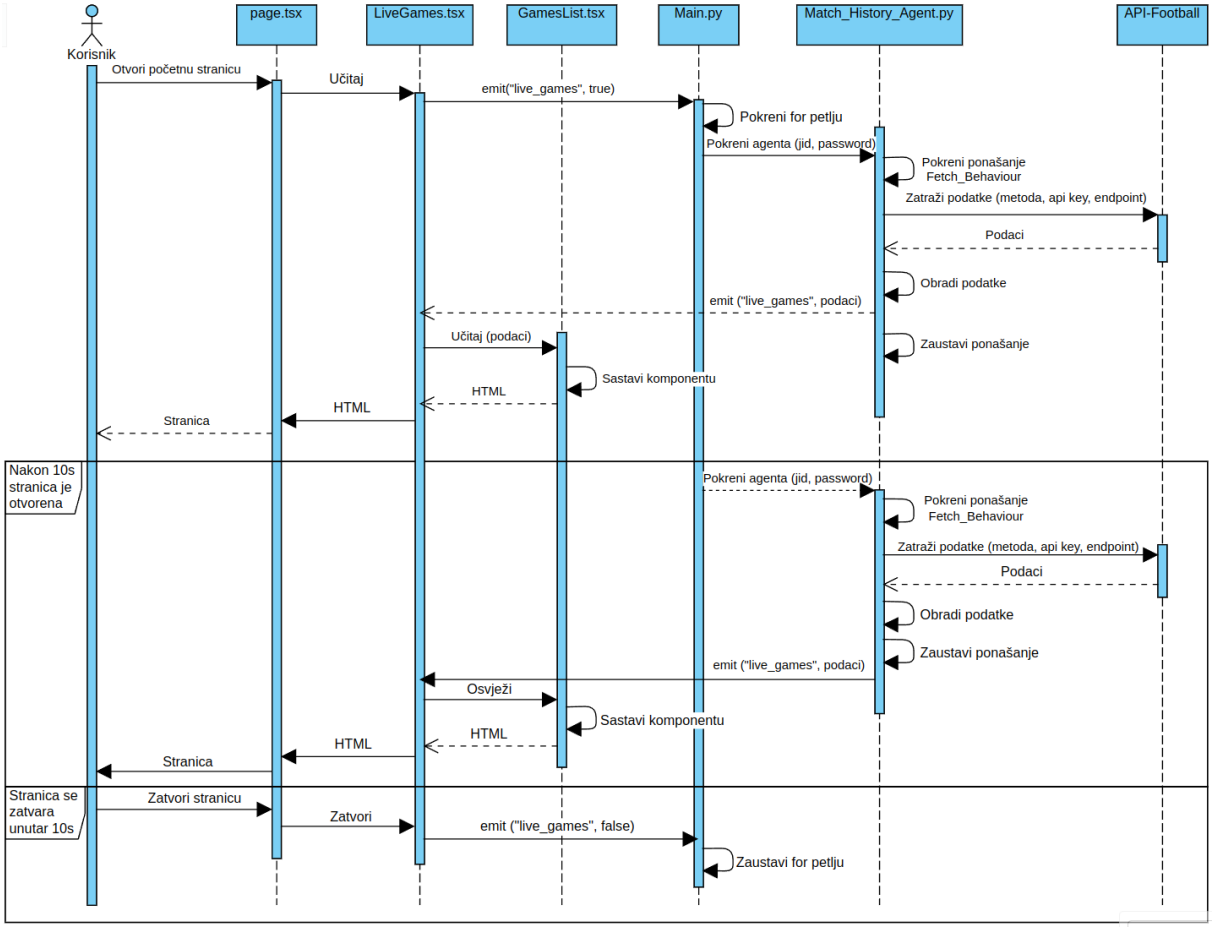
Periodično ponašanje agenata nije bilo moguće izvesti na standardan način uz pomoć SPADE-ove *PeriodicBehaviour* klase zbog pojave greške svaki puta kada bi se trebao emitirati novi WebSocket paket. Umjesto toga, da bi se postigao identičan rezultat, koristi se funkcija koja se pokreće u zasebnoj dretvi kako ne bi smetala izvođenju glavne dretve. U toj dretvi se koristi *sleep()* metoda 100 puta po 0.1 sekundu, što je sveukupno neznatno više 10 sekundi spavanja i prilikom svakog "buđenja" se provjerava je li stranica na kojoj bi se trebali osvježiti podaci još otvorena ili nije. To jest, gleda se je li korisnik na web-pregledniku još uvijek na toj stranici ili nije. Dokle god jest, svakih 10 sekundi se iznova pokrene jednokratno ponašanje agenta *Live Games Agent* i on u svom ponašanju tada odradi praktički isto što i agent kojem je dovoljno da se samo jednom pokrene. Dretva se pokreće putem metode *start background task* biblioteke SocketIO [35].

Način na koji se provjerava je li stranica još otvorena ili nije je također potrebno proanalizirati. To se radi uz pomoć *Event* objekta iz *threading.Event* klase, koja omogućava realizaciju komunikacije između dretvi koristeći najbolju praksu. Ta komunikacija u ovom slučaju koristi se da bi novostvorena dretva mogla prepoznati kad je vrijeme da se prekine. Nakon prestanka izvođenja svakog *sleep* u trajanju od 0.1 sekunde, provjerava se je li varijabla *exit event* tipa *Event* postavljena na true, što bi značilo da je dobiven signal od klijenta da se stranica zatvorila. Ako jest, izlazi se iz funkcije koja tu petlju vrti i time se dretva "ubija". Ako nije, to znači da se slobodno može nastaviti s radom dretve i to će eventualno rezultirati novim pokretanjem agenta *Live Games Agent*. Funkcija koju se upravo predstavilo vidljiva je u isječku 5.

```
def live_games_refresher():
    while not GLOBAL.exit_event.is_set():
        if GLOBAL.live_games_initial_emit:
            GLOBAL.live_games_initial_emit = False
        else:
            for _ in range(100):
                eventlet.sleep(0.1)
                if GLOBAL.exit_event.is_set():
                    print("STOPPING LIVE GAMES AGENT")
                    return

        with GLOBAL.app.test_request_context():
            async def start_agent_thread():
                live_games_agent = Live_Games_Agent(
                    "Live_Games_Agent@localhost", "secret")
                await live_games_agent.start()
                await live_games_agent.fetch_behaviour.join()
            asyncio.run(start_agent_thread())
    print("STOPPING LIVE GAMES AGENT")
```

Isječak koda 5: Dretva zadužena za izvedbu periodičnog ponašanja agenta Live Games Agent

U samom ponašanju agenta nema značajne razlike u odnosu na bilo kojeg drugog agenta. To znači da se potrebni podaci dobivaju preko API-FOOTBALL te se nakon vršenja operacije izdvajanja samo onih potrebnih u JSON obliku oni emitiraju do klijenta koji ih prikazuje na stranici. Slijed događaja je također prikazan na dijagramu slijeda na slici 24.



Slika 24: Dijagram slijeda otvaranja početne stranice, vlastita izrada

3.6.3. Analiza integracije višeagentnog sustava

U nastavku se analizira konkretna primjena teorijskog dijela iz poglavlja 3.1 u praktični dio kojeg sačinjava izrađena i opisana aplikacija. Iako stvorena WebSocket veza nije doslovno unutar tijela agenta gdje on izvodi svoje ponašanje, funkcija koja čeka i sluša emitiranja klijenta može se smatrati njegovim osjetilom. Svaki *socketIO event* ima posebnog agenta koji reagira kada se na njega primi neku poruku.

Reakcija svakog agenta na podražaj je u potpunosti sadržana u njegovom ponašanju koje nema potrebe pratiti povijest poduzetih akcija ili posljedice akcija koje će se izvesti u budućnosti. Isključivo se uzima trenutno stanje svijeta i reagira se na podražaj na izravan i unaprijed programiran način. Prisutno je dakle jednostavno ponašanje, bez apstraktnog razmišljanja. Agenti međusobno ne komuniciraju, ali zajedničkim djelovanjem doprinose istom cilju, a to je dopremanje podataka za web-stranicu računalu klijenta koji ju je otvorio radi dobivanja informacija. Prema tome, u aplikaciji rada prisutan je višeagentni sustav u kojemu agenti ne surađuju na klasičan način, već im je zadatak u vremenski prihvatljivom roku ispravno izvesti svoj dio rada i odgovornosti za posao koji su delegirani.

Što se tiče okruženja agenata, ono je sastavljeno od korisnika web-stranice i njegovog računala te API usluge. Ono je pristupačno, pošto se uvijek dobivaju relevantne informacije u

vidu dobivanja podataka poput identifikatora igrača za kojeg se podaci moraju naći, te samih podataka o njemu. Epizodično je, pošto djelovanja agenata ničime ne mogu utjecati na buduće stanje pa on ni ne mora paziti na svoje akcije koje bi mogle utjecati na buduća stanja okoline. Okruženje se u našem slučaju tijekom agentovog rada može promijeniti. Ono je dinamično zbog primjerice slučaja kada se dobije informacija da je stranica rezultata uživo u trenutku provjere još uvijek otvorena, nakon čega agent dobije "zeleno svijetlo" za pokretanje, no za vrijeme agentovog rada korisnik promijeni ili zatvori stranicu. U tom slučaju agent će svoje ponašanje izvesti do kraja na način da neće emitirati sastavljene podatke jer za to nema više potrebe. On stoga mora "razmišljati" o svojoj akciji. Primjer je ovo okruženja koje je kontinuirano jer okolina može zauzeti beskonačno mnogo stanja zbog nepredvidljivosti dobivenih nogometnih podataka koji se neprestano mijenjaju.

3.6.4. Prednosti i nedostaci pristupa

Jasno je kako ovakav pristup izradi web mjesta ima i određene nedostatke, pa će se prvo oni navesti. Za ispravno funkcioniranje aplikacije potrebna su tri entiteta, računalo korisnika, poslužitelj te API usluga. Takav sustav nije naročito vremenski optimiziran, pošto se uvijek mora prolaziti kroz dio gdje se podaci dodatno zahtijevaju i prosljeđuju te preoblikuju. Time se gubi vrijeme, više energije se troši, a prilikom razvoja u poslovnom okruženju bi i ukupni troškovi posljedično bili viši.

Također, zbog odabira pružatelja API usluge podaci s kojima se rukuje nisu od samog početka u posve poželjnoj strukturi. Primjerice, prilikom jedne izgradnje komponente *GameCard* potrebno je svaki puta izvesti dva API poziva, jedan za općenite informacije o dvoboju, a drugi put za dobivanje informacija iz kladionica. Time taj proces dulje traje, ali je i skuplje jer API usluga nije besplatna i ukupni trošak se gleda prema broju upućenih zahtjeva u nekom vremenskom razdoblju.

No, bez obzira na to, prednosti su očite. Podjela odgovornosti prema agentima je vrlo korisna stvar i njihova decentralizirana priroda također. U bilo kojem trenutku se aplikacija može nadograditi dodavanjem nove funkcionalnosti za koju će biti zadužen novostvoreni agent. Jasno je što je čiji zadatak.

Osim toga, sa stajališta sigurnosti, uvijek je prihvatljivije da se sav važan kod nalazi na poslužitelju. To uključuje rukovanje API-jima i pozadinsku logiku. Ključ za API je vrlo osjetljiv podatak kojeg valja čuvati na posebnom mjestu, što je malo teže za izvesti ako je većina projektnih datoteka izložena na strani klijenta. Što god je vidljivo i dostupno vanjskom svijetu, može potencijalno prouzrokovati veću ranjivost. Ovako, kad je sve čuvano na udaljenom računalu u ulozi poslužitelja, morala bi se dogoditi veća katastrofa i proboj u mrežu da se dođe do informacija u tuđem vlasništvu.

4. Zaključak

Ovaj diplomski rad poslužio je da bi se jednim projektom demonstriralo kako to izgleda kada se višeagentni sustav integrira u pozadinu web mjesta. To naravno ima svoje prednosti i nedostatke koji su opisani, ali razvoj tehnologije u današnjem svijetu prisiljava ljude koji na nju mogu imati utjecaja da razmišljaju kako se pravovremeno adaptirati i ponuditi uvijek barem malo više od onoga što drugi nude. Korištenje principa agenata u nekom sustavu je zasigurno jedan od trenutno modernijih načina koji imaju svijetlu budućnost.

U radu se krenulo opisivanjem metoda i tehnika rada na način da se prošlo kroz svaki značajniji alat korišten tijekom izrade aplikacije. Ukratko ih se opisalo i svrstalo u dio projekta u kojem pripadaju, a to može biti klijent ili poslužitelj. Nakon toga, u razradi teme dotaknuli smo se općenitih pojmova i teorije usko povezane s radom jer bez njezinog razumijevanja nema previše smisla u čitanju praktičnog dijela. Agenti i njihove vrste te okruženja i njihova svojstva najvažniji su bili za izdvojiti, ali i pojmovi poput društvenosti i učenja koji uvelike utječu na ponašanje i koordinaciju sudionika u višeagentnom sustavu.

U praktičnome dijelu red je došao na opis same aplikacije. Što zapravo aplikacija predstavlja, čemu služi i kako funkcionira i uz koje pristupe i metode se nju izrađivalo. Prikazali su se njezini najvažniji dijelovi pomoću slika zaslona u verzijama na većim i manjim ekranima da bi se lakše predstavio trud koji je potrebno uložiti tijekom web-razvoja. U konačnici te cjeline, taj praktični dio se povezalo s teorijskim i objasnilo se koje se koncepte upotrebljavalo, a koje nije i zašto.

Sve u svemu, bilo je zabavno raditi na ovakvom relativno neuobičajenom projektu koji je spojio izuzetno poznate tehnologije u svijetu frontenda s nekim malo manje poznatim alatima u svijetu backenda. Želja autora ovog rada koji od malena redovito prati nogometna događanja i često koristi pripadajuće aplikacije bila je izvesti jednu vlastitu, a vjerojatno jedna od najboljih prilika za to pojavila se kada se trebala odabrati tematika diplomskog rada.

Popis literature

[1] Microsoft. „Visual Studio Code - Code Editing. Redefined.” (2024.), adresa: https://code.visualstudio.com/ (pogledano 19. 8. 2024.).

[2] Microsoft. „Debugging in Visual Studio Code.” (2024.), adresa: https://tinyurl.com/4nebuvcm (pogledano 19. 8. 2024.).

[3] Microsoft. „Visual Studio Code Marketplace.” (2024.), adresa: https://tinyurl.com/yckbuwtn (pogledano 19. 8. 2024.).

[4] Microsoft. „IntelliSense in Visual Studio Code.” (2024.), adresa: https://tinyurl.com/34uhj443 (pogledano 19. 8. 2024.).

[5] GitHub. „Microsoft / vscode.” (2024.), adresa: https://github.com/microsoft/vscode (pogledano 19. 8. 2024.).

[6] D. Flanagan, *JavaScript: The definitive guide: Activate your web pages*. " O'Reilly Media, Inc.", 2011.

[7] Meta. „React - A JavaScript library for building user interfaces.” (2024.), adresa: https://reactjs.org/ (pogledano 19. 8. 2024.).

[8] OpenJS Foundation. „Node.js.” (2024.), adresa: https://nodejs.org/en/about (pogledano 20. 8. 2024.).

[9] N. C. Zakas, *Understanding ECMAScript 6: the definitive guide for JavaScript developers*. No Starch Press, 2016.

[10] „HTML.” (2024.), adresa: https://en.wikipedia.org/wiki/HTML (pogledano 20. 8. 2024.).

[11] H. W. Lie i B. Bos. „Cascading Style Sheets: Designing for the Web.” (1996.), adresa: https://www.w3.org/TR/CSS1/ (pogledano 20. 8. 2024.).

[12] Wikipedia. „CSS.” (2024.), adresa: https://en.wikipedia.org/wiki/CSS (pogledano 21. 8. 2024.).

[13] React Dokumentacija. „React Dokumentacija.” (2024.), adresa: https://reactjs.org/docs/getting-started.html (pogledano 21. 8. 2024.).

[14] Vercel. „Next.js: React Framework.” (2024.), adresa: https://nextjs.org/docs (pogledano 21. 8. 2024.).

[15] Microsoft. „TypeScript for the New Programmer.” (2024.), adresa: https://tinyurl.com/3casxy3t (pogledano 21. 8. 2024.).

[16] React Documentation. „Using TypeScript with React.” (2024.), adresa: https://reactjs.org/docs/static-type-checking.html (pogledano 21. 8. 2024.).

[17] Tailwind CSS. „Tailwind CSS Documentation.” (2024.), adresa: https://tailwindcss.com/docs (pogledano 22. 8. 2024.).

[18] K. Ukey. „Why you should use TailwindCSS in your ReactJS project.” (2024.), adresa: https://dev.to/kunalukey/why-you-should-use-tailwindcss-in-your-reactjs-project-51kf (pogledano 22. 8. 2024.).

[19] ESLint. „ESLint Documentation.” (2024.), adresa: https://eslint.org/docs/latest/use/core-concepts (pogledano 23. 8. 2024.).

[20] P. Ccari. „How to Use ESLint and Prettier in a TypeScript Project.” (2024.), adresa: https://blog.logrocket.com/linting-typescript-eslint-prettier (pogledano 23. 8. 2024.).

[21] Wikipedia. „Python.” (2024.), adresa: https://en.wikipedia.org/wiki/Python_(programming_language) (pogledano 23. 8. 2024.).

[22] D. S. Foundation. „The Web framework for perfectionists with deadlines.” (2024.), adresa: https://www.geeksforgeeks.org/differences-between-django-vs-flask/ (pogledano 23. 8. 2024.).

[23] Flask Documentation. „Flask Documentation.” (2024.), adresa: https://en.wikipedia.org/wiki/Flask_(web_framework) (pogledano 23. 8. 2024.).

[24] M. Garcia. „Deploying a Python Flask Example Application Using Heroku.” (2022.), adresa: https://realpython.com/flask-by-example-part-1-project-setup/ (pogledano 23. 8. 2024.).

[25] Mozilla. „Writing WebSocket servers.” (2024.), adresa: https://developer.mozilla.org/enUS/docs/Web/API/WebSockets_API/Writing_WebSocket_servers (pogledano 23. 8. 2024.).

[26] API-Football. „Football API Documentation.” (2024.), adresa: https://www.api-football.com/documentation-v3 (pogledano 23. 8. 2024.).

[27] API-Football. „Architecture.” (2024.), adresa: https://www.api-football.com/documentation-v3#section/Architecture (pogledano 23. 8. 2024.).

[28] SPADE. „The SPADE agent model.” (2020.), adresa: https://spade-mas.readthedocs.io/en/latest/model.html (pogledano 23. 8. 2024.).

[29] SPADE. „Advanced Behaviours.” (2020.), adresa: https://tinyurl.com/3f3a4kj9 (pogledano 23. 8. 2024.).

[30] M. Wooldridge, *An Introduction to Multiagent Systems*, 2nd. UK: John Wiley ‘I&’ Sons ltd., 2009.

[31] P. Norvig i S. Russell, *Artificial Intelligence: A Modern Approach*, 4th. Pearson, 2020.

[32] J. Kurose i K. Ross, *Computer networks: A top down approach featuring the internet*, 2010.

[33] „Pages and Layouts.” (2024.), adresa: https://nextjs.org/docs/pages/building-your-application/routing/pages-and-layouts (pogledano 30. 8. 2024.).

[34] „Quick start.” (2024.), adresa: https://react.dev/learn (pogledano 30. 8. 2024.).

[35] „API Reference.” (2024.), adresa: https://flask-socketio.readthedocs.io/en/latest/api.html (pogledano 30. 8. 2024.).

Popis slika

Popis isječaka koda