Mobilna aplikacija za šah na React-Native platformi

Buha, Davor

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

Sveučilišni studij

MOBILNA APLIKACIJA ZA ŠAH NA REACT-NATIVE PLATFORMI

Diplomski rad

Davor Buha

Osijek, 2018.

SADRŽAJ

1. UVOD

Zadatak diplomskog rada je izrada platforme za igranje šaha. Pod platforma podrazumijeva se API sustav na poslužitelju te mobilna aplikacija koju je moguće koristiti na oba mobilna operacijska sustava (iOS i Android) namijenjena krajnjem korisniku za korištenje. Napravljena je mobilna aplikacija za online igranja šaha s prijateljima ili poznanicima te naravno nasumično odabranim ljudima. Tijekom izrade platforme korištena su većina znanja stečenih tijekom obrazovanja na Fakultetu elektrotehnike računarstva i informacijskih tehnologija u Osijeku. Također korištena su i znanja stečena na stručnoj praksi za vrijeme 2. godine diplomskog studija. Kao dodatan izvor znanja korišteni su online tečajevi budući da je poslužiteljski dio sustava napravljen u Go programskom jeziku, dok je mobilna aplikacija izrađena u jednom od najnaprednijih međuplatformskih frameworka današnjice – React-Nativu. Prvo će se objasniti korištene tehnologije u izradi rada. Zatim će biti opisana struktura poslužitelja i komunikacija s klijentom. Na koncu biti će opisana i sama struktura klijentske aplikacije te interakcija s poslužiteljem putem WebSocketa i odigravanje šahovske partije.

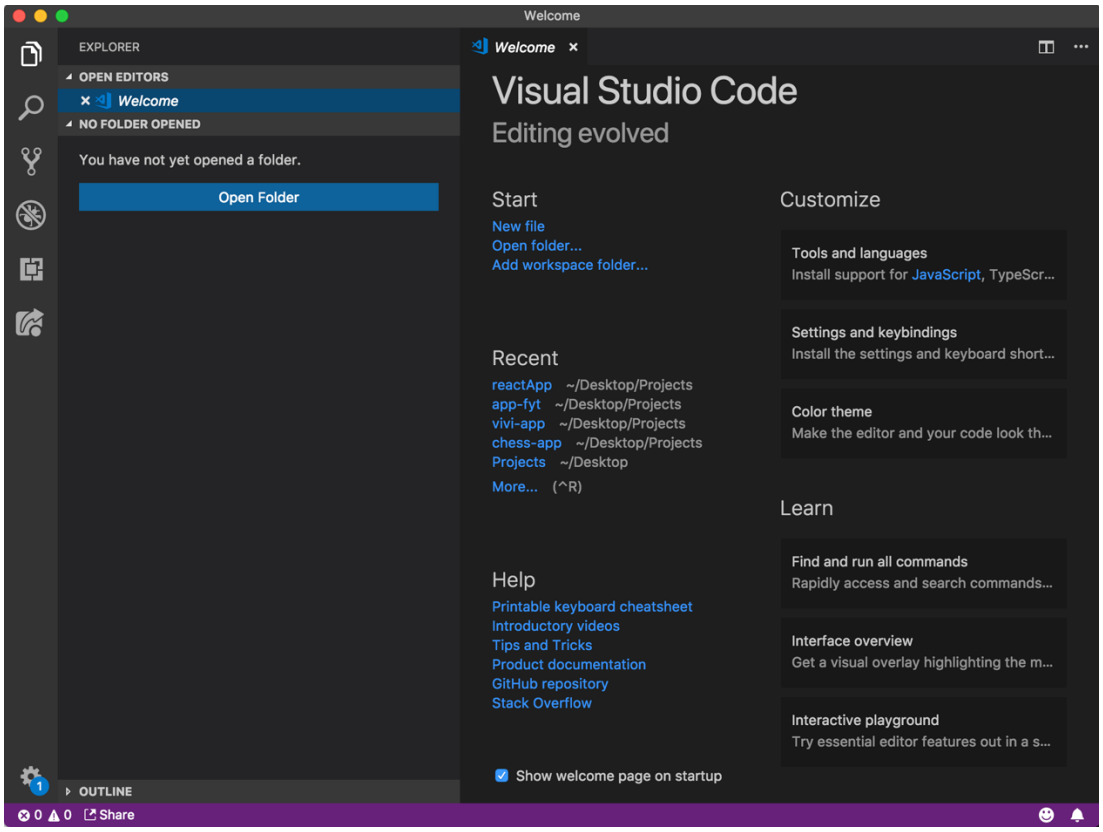
1.1. Zadatak diplomskog rada

Zadatak diplomskog rada je Mobilna aplikacija za šah na React-Native platformi. Ideja je da se krajnjem korisniku dostavi funkcionalna mobilna aplikacija putem koje će jednostavno moći vršiti interakciju s poslužiteljskim sustavom. Korisnik prilikom pokretanja aplikacije treba odabrati svoje korisničko ime te se pridružiti već kreiranim sobama za igranje ili samostalno kreirati svoju. U poslužiteljskom dijelu korišten je princip Finite-state machine (FSM) koji omogućuje kontroliranje stanja (state) svake kreirane sobe te lagano reagiranje na aktivnosti korisnika.

2. OPIS KORIŠTENIH TEHNOLOGIJA

2.1. Visual Studio Code

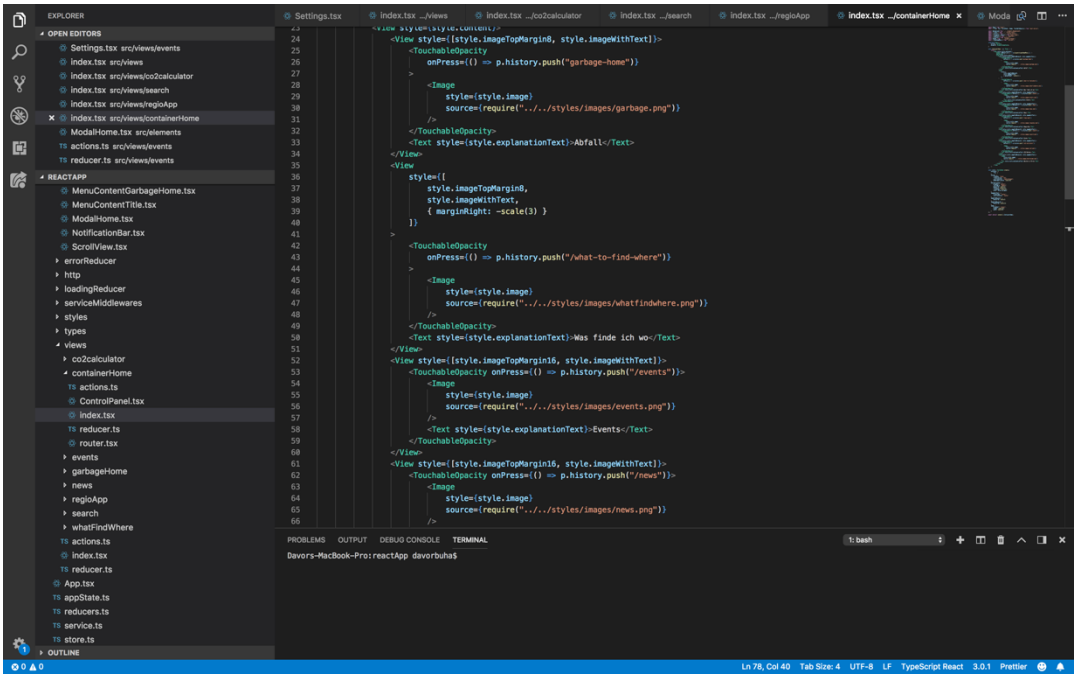
Visual Studio Code (Slika 2.1.) je besplatni open source IDE editora razvijen od strane Microsofta za operacijske sustav Windows, macOS i Linux. Ima integriranu podršku za ispravljanje grešaka u kodu, alate za verzioniranje koda (Git control), označavanje sintakse, inteligentno nadopunjavanje te refaktoriranje koda. Također je prilagodljiv, tako da korisnici mogu promijeniti izgled, prečace tipaka, postavke te integrirati svoje dodatke. Pruža pomoć pri pisanju i korištenju aktualnih tehnologija za pisanje programskog koda uključujući Go, TypeScript i React Native. [1]



Slika 2.1. Izgled Visual Studio Code IDE-a

Programski kod Visual Studio Code-a temelji se na Electronu, frameworku koji se koristi za pokretanje Node.js aplikacija na desktop računalima. Prema anketama izvršenih nad programerima

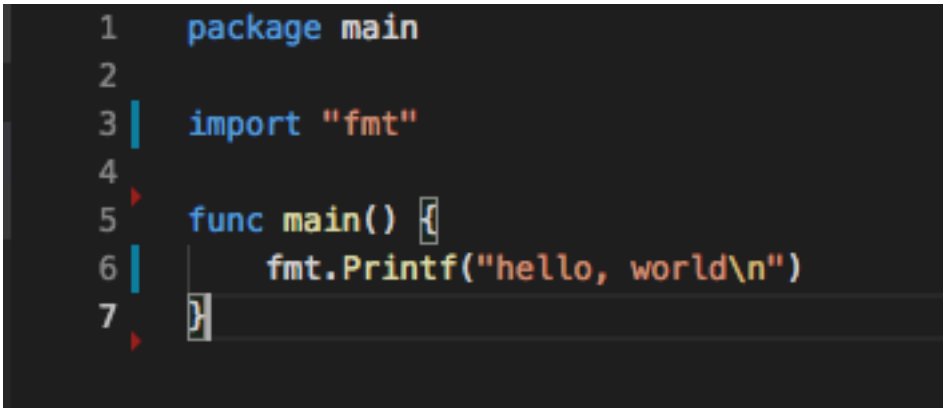
u 2018. godini, Visual Studio Code (Slika 2.2.) je rangiran kao najpopularniji alat za razvojne programere s 35% glasova programera koji ga koriste.



Slika 2.2. Izgled Visual Studio Code razvojnog okruženja

2.2. Go Programski jezik

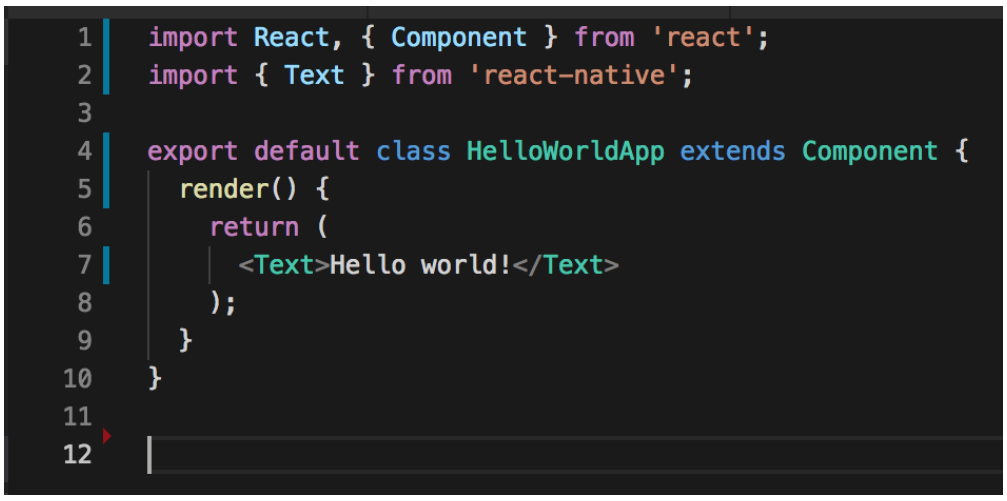
Go (često nazivan i Golang) je programski jezik kreiran od strane Google-a 2009. godine. Go sadrži statičke tipove podataka, koristi kompajler, napravljen po uzoru na C. Također ima visoku razinu sigurnosti memorije, strukturirane tipove podataka, koristi sakupljače 'smeća' nakon završetka određene procedure ili dijela programa koji više nije potreban. Dizajnere na stvaranje ovog programskog jezika potaklo je to što im se nije sviđao C++, tu su pronašli motivaciju za stvaranje programskog jezika koji nije objektno orijentiran. Trenutno je u uporabi verzija 1.11 koja je izašla 24.8.2018. Sama sintaksa Go jezika je među jednostavnijima, vrlo je čitka i pregledna. [2] Prilikom instalacije Go jezika sve što je potrebno napraviti je preuzeti datoteku s interneta, pokrenuti ju i podesiti varijable okruženja. Svaki Go projekt sastoji se od dva direktorija (src i bin). U src direktorij programeri pišu programski kod i kreiraju svoje "pakete". Dok se u bin direktoriju nalaze biblioteke programskog jezika. Svaka datoteka programa počinje s definiranjem paketa u kojem se nalazi, nakon toga slijede paketi koje koristimo unutar te datoteke, tek nakon toga definiramo "main" funkciju koja se izvršava prilikom pokretanja programa (Slika 2.3.). [3]



Slika 2.3. Izgled jednostavnog "Hello World" Go programa

2.3. React-Native

React-Native je framework napravljen od strane Facebooka za izradu nativnih mobilnih aplikacija. Omogućuje programerima pisanje koda u JavaScript-u te taj kod prevodi i krajnjem korisniku dostavlja nativno sučelje na njegovom mobilnom uređaju. Najveća prednost ovog frameworka je u tome što kod napisan u JavaScript-u može istovremeno prevesti za obje platforme (Android i iOS). Tako da nije potrebno odvojeno razvijati mobilnu aplikaciju za Android i odvojeno za iOS već prevoditelj odrađuje veći dio posla te napisani kod iz JavaScript-a prevodi u izvorni kod obje platforme što uvelike olakšava posao programerima. [4]



Slika 2.4. Izgled jednostavnog "Hello World" React Native programa

Primjer (Slika 2.4.) jednostavne React-Native "HelloWorld" komponente, Text oznaka koristi se za ispisivanje teksta na zaslon mobilnog uređaja. Za pozicioniranje komponenata unutar

korisničkog sučelja koristi se Flexbox'. Flexbox je jedan od najnaprednijih modula izgleda koji, služi za izradu responzivnih mobilnih i web aplikacija. Za dodatno uređivanje teksta (font, boja, veličina) koristi se CSS. Neki od temeljnih komponenata React-Nativea su:

- <View/> ekvivalent div-a u web aplikacijama

- <TouchableOpacity /> pretvara bilo koju komponentu unutar tagova u površinu koja reagira na dodir, te može pritiskom izvršiti neku akciju

- <Text/> služi za ispisivanje teksta

- <Image/> prima relativnu adresu slike te ju prikazuje [6]

2.4. TypeScript

TypeScript je open-source jezik za programiranje razvijen i održavan od strane Microsofta. To je strogo sintaktički superset JavaScript-a, dodaje neobavezne statičke tipove podataka na jezik. TypeScript uvelike olakšava izrade velikih aplikacija jer definiranjem statičkih tipova podataka uvelike se dobije na čitljivosti i lakšem razumijevanju koda. Budući da je TypeScript superset JavaScripta, svi JavaScript programi su također valjani TypeScript programi. Također podržava datoteke definicija (.d.ts) koje mogu sadržavati podatke o vrstama postojećih JavaScript biblioteka, slično kao što u C++ datoteke zaglavlja mogu opisati strukturu postojećih datoteka objekata. Nastao je zbog nedostataka Javascripta u razvoje velikih aplikacija, često JavaScript kod postane neuredan i teško čitljiv. Čitljivosti samog programa doprinosi definiranje tipova podataka te točni ulazni i izlazni tipovi svake od komponenti unutar programa. Neki od osnovnih tipova podataka u TypeScriptu su: string, number, undefined, null, boolean, itd. Također programer može definirati svoje tipove podataka tj. sučelje objekta (Slika 2.5.), postoji mogućnost definiranja polja koji se sastoji od različitih tipova podataka tzv. Tuple. [5]

Slika 2.5. Izgled jednostavnog sučelja koji se sastoji od dvije varijable različitog tipa

Tipovi podataka također potrebno je definirati prilikom definicije funkcije (Slika 2.6.), potrebno je odrediti točno koje tipove podataka funkcija prima te koji tip podatka će funkcija vratiti.

```
export function getNewsById(id: number, data: News[]): News | undefined {
    return data.find(item => item.id === id);
}
```

Slika 2.6. Primjer definicije ulaznih i izlaznih tipova podataka u TypeScriptu

Slika 2.6. pokazuje primjer funkcije koja prima dva tipa podatka (broj i polje objekata koji su definirani sučeljem News). U JavaScriptu postoji integrirana metoda na polju podataka koja pronalazi određeni podataka s određenim uvjetom te ukoliko je podatak pronađen vraća vrijednost podatka, u suprotnom vraća vrijednost undefined. Tako da će definirana funkcija vratiti ili tip podatka News ili undefined. TypeScript koristi kompajler koji prevodi napisani kod u JavaScript te samo jednom na početku prije pokretanja programa provjerava jesu li tipovi podataka dobro definirani. U slučaju da tipovi podataka nisu dobro definirani javlja grešku te onemogućuje pokretanje programa

3. RAZVOJ VLASTITOG RJEŠENJA

Proces izrade započet je s osmišljavanjem poslužiteljskog dijela, odnosno načina komunikacije između poslužitelja i klijenta. Nakon toga osmišljena je struktura poslužitelja, u ovom koraku poslužiteljska strana podijeljena je na dvije razine. Nakon rješavanja problema komunikacije i načina obrade podataka implementirana je logika šaha te je s tim korakom završen proces izrade poslužitelja. Završetkom poslužiteljske strane započeta je implementacija klijentskog sučelja odnosno mobilne aplikacije. Aplikacija je prvo dizajnirana te su definirane boje i slike koje će se koristiti. Nakon toga započet je proces programiranja same aplikacije.

3.1. Poslužitelj

3.1.1. Pristupne točke poslužitelja

Prvi korak izrade poslužitelja je definiranje krajnjih točki odnosno točki za komunikaciju s klijentom. Definirane su 4 pristupne točke (Slika 3.1.).

```
func main() {
    brkr = broker.New(5 * time.Minute)
    router := mux.NewRouter()
    router.HandleFunc("/create_game/{name}/move_time/{moveTime}/admin_color/{adminColor}", handleCreateGame)
    router.HandleFunc("/close_game/{id}/admin_token/{adminToken}", handleCloseGame)
    router.HandleFunc("/game_events", handleGameEventsConnections)
    router.HandleFunc("/join_game/{id}", handleJoinGameConnections)
    http.ListenAndServe(":"+os.Getenv("PORT"), router)
}
```

Slika 3.1. Pristupne točke poslužitelja

1) Pristupna točka "/game_events" koristi tehnologiju WebSocket za komunikaciju s klijentom. Ova pristupna točka obavještava klijenta o kreiranim, zatvorenim ili započetim igrama od strane ostalih klijenata. Definirana su tri tipa poruke putem te pristupne točke: "game.created" – u slučaju kreiranja igre (Slika 3.2.), "game.started" – u slučaju pokretanja igre i "game.deleted" – u slučaju zatvaranja kreirane igre.

Slika 3.2. Primjer poruke poslužitelja klijentima o kreiranju nove igre

2) Pristupna točka "/create_game/{name}/move_time/{moveTime}/admin_color/{adminColor}" koristi GET zahtjev za kreiranje nove igre. Ukoliko klijent želi kreirati novu igru potrebno je napraviti zahtjev prema navedenoj pristupnoj točki, te definirati ime igre, vrijeme trajanja poteza i boju svojih figura (crnu ili bijelu). Klijent kao odgovor od servera dobiva JSON objekt (Slika 3.3.) koji se sastoji od id-a kreirane igre te admin tokena koji služi za identifikaciju admina unutar igre.

Slika 3.3. Primjer odgovora poslužitelja na zahtjev za kreiranje nove igre

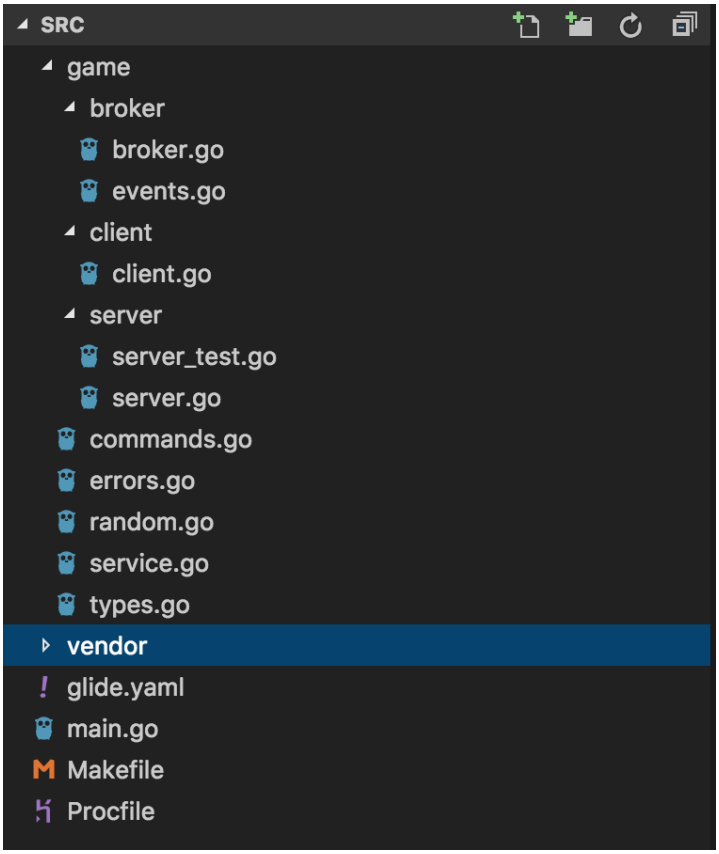
3) Pristupna točka "/close_game/{id}/admin_token/{adminToken}" koristi GET zahtjev za zatvaranje pokrenute igre, koristi se u slučaju ako drugi igrač još nije pristupio igri. Ovu pristupnu točku može koristiti jedino admin igre jer se za verifikaciju koristi admin token koji se dobije prilikom kreiranja igre.

4) Pristupna točka "/join_game/{id}" koristi WebSocket tehnologiju za pridruživanje igri, upravljanje igrom te samim time prelazak iz jednog stanja u drugo. Nakon kreiranja igre

putem GET zahtjeva admin se treba pridružiti kreiranoj igri pristupajući WebSocketu s denfiniranim id-em igre, u poruci na otvaranju komunikacije treba poslati admin token kako bi verificirali njegovo pristupanje. Nakon pristupanja admina, igra dolazi na popis kreiranih igra te se klijentima koji su se pretplatili na "/game_events" pristupnu točku dolazi događaj da je igra kreirana te da joj mogu pristupiti. Ako korisnik želi pristupiti igri koji nije kreirao to može jednostavno učiniti spajanjem na "/join_game/{id}" pristupnu točku uz navedeni id igre, nije potrebno slati poruku na otvaranju komunikacije jer verifikacija nije potrebna.

3.1.2. Slojevi poslužitelja

Poslužitelj je strukturiran u tri sloja (Slika 3.4.). Prvi sloj (broker) vodi računa o komunikaciji s klijentskim sučeljem putem WebSocketa, pakira poruke s obzirom na informacije dobivene od serverskog sloja. Vodi računa o "/game_events", "/create_game/..." i "/close_game/..." pristupnim točkama.



Slika 3.4. Struktura poslužitelja

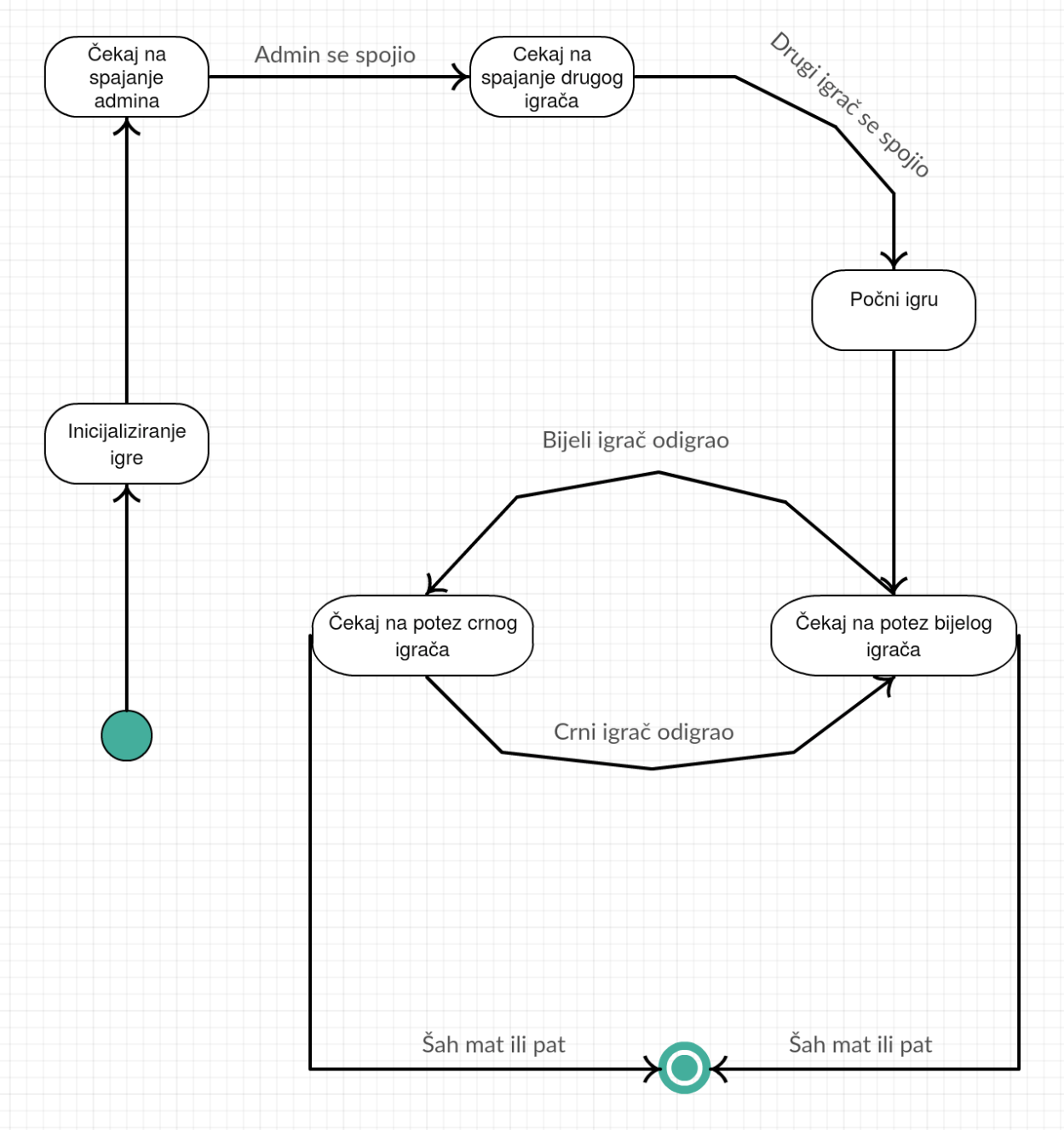
Drugi sloj (client) vodi računa isključivo o klijentima spojenim na "/join_game/{id}" pristupnu točku. Implementirane su tri metode na strukturi Client, to su JoinAdmin, JoinPlayer i MakeMove. Client sloj komunicira sa serverskim slojem putem kanala te vrši promjena na stanju servera. Za svaku uspješno uspostavljenu konekciju na pristupnu točku kreira se nova Client struktura.

```
func (c *Client) MakeMove(playersMove string) (error, interface{}) {
    c.Player.C <- game.CommandPlayMove{
        Move: game.Move(playersMove),
    }
    replay := <-c.Player.ReplayToC
    if replay.Status == game.CommandNotAllowed {
        return game.CommandNotAllowedError{Command: "CommandNotAllowed"}, nil
    }
    if replay.Status == game.CommandMoveNotAllowed {
        return errors.New("MoveNotAllowed"), nil
    }
    return nil, replay.Payload
}
```

Slika 3.5. Metoda MakeMove na strukturi Client

Metoda MakeMove (Slika 3.5.) kao argument prima igračev potez koji je primljena od igrača putem WebSocketa, te putem kanala šalje poruku PlayMove serverskom sloju. Nakon što je poruka poslana čeka se odgovor od serverskog sloja, kod se ne izvršava dok odgovor nije primljen. Kada je odgovor primljen ukoliko status odgovora ne ukazuje na grešku rezultat se vraća brokeru koji pakira poruku i šalje klijentu.

Treći sloj (server) vodi računa trenutnom stanju igre. U trećem sloju implementiran je Finite-State Machine (Slika 3.6.) koji se mijenja ovisno o naredbama koji dolaze od klijenta putem kanala.



Slika 3.6. FSM serverskog sloja

FSM na serverskom sloju sastoji se od 6 stanja koji se mogu promijeniti tijekom životnog vijeka igre. Također u serverskom sloju definirana je i struktura (Slika 3.7.) u koju se zapisuju adrese za komunikaciju s višim slojevima, postavke igrača, trenutno stanje i postavke igre.

```
type Server struct {
    ID           game.GameID
    Name         string
    B            game.BroadcastChannel
    C            game.MessageChannel
    AdminToken   game.AdminToken
    PlayerWhite  game.Player
    PlayerBlack  game.Player
    GameState    *chess.Game
    moveTimeout  time.Duration
    setupTimeout time.Duration
    state        stateFn
    StateStr     string
    FEN          string
    Started      bool
    AdminColor   string
}
```

Slika 3.7. Struktura u koju se spremaju relevantne informacije za Serverski sloj

3.1.3. Logika igre

Nakon kreiranja sobe, kreira se novi server te se on stavlja u stanje "waitForAdmin". Brojač čeka predefinirano vrijeme na admina da se spoji, ukoliko se admin ne spoji server se briše. Nakon spajanja admina server prelazi u stanje "waitForPlayer" te također čeka određeno predefinirano vrijeme na spajanje drugog igrača. Ukoliko se igrač ne spoji za to vrijeme igra se briše, a ako se spoji igra prelazi u stanje "startGame" koje postavlja sve potrebne parametre za početak igre te automatski prelazi u stanje "waitForWhiteMove" (Slika 3.8.). Trenutno stanje šahovskih figura na polju prati se pomoću FEN stringa. Iz tog zapisa može se saznati sve što je potrebno za igru, također on se šalje klijentu nakon svakog odigranog poteza kako bi imao stvaran prikaz figura na svom sučelju. U izradi logike igre korištena je biblioteka notnill/chess koja vodi računa o kretanju po figura po šahovskoj ploči

```go
func (srv *Server) waitForWhiteMove() stateFn {
    outcome := srv.GameState.Outcome()
    if outcome != chess.NoOutcome {
        return nil
    }
    srv.setGameState("waitForWhiteMove")

    timer := time.NewTimer(srv.moveTimeout)
    for {
        select {
        case <-timer.C:
            return nil

        case msg := <-srv.PlayerWhite.C:
            cmd, ok := msg.(game.CommandPlayMove)
            if !ok {
                srv.PlayerWhite.ReplayToC <- game.MessageCommandReply{
                    Status: game.CommandNotAllowed,
                }
                continue
            }

            err := srv.GameState.MoveStr(string(cmd.Move))
            if err != nil {
                srv.PlayerWhite.ReplayToC <- game.MessageCommandReply{
                    Status:  game.CommandMoveNotAllowed,
                    Payload: err,
                }
                continue
            }

            srv.PlayerWhite.ReplayToC <- game.MessageCommandReply{
                Status:  game.CommandOK,
                Payload: true,
            }
            return srv.waitForBlackMove
        }
    }
}
```

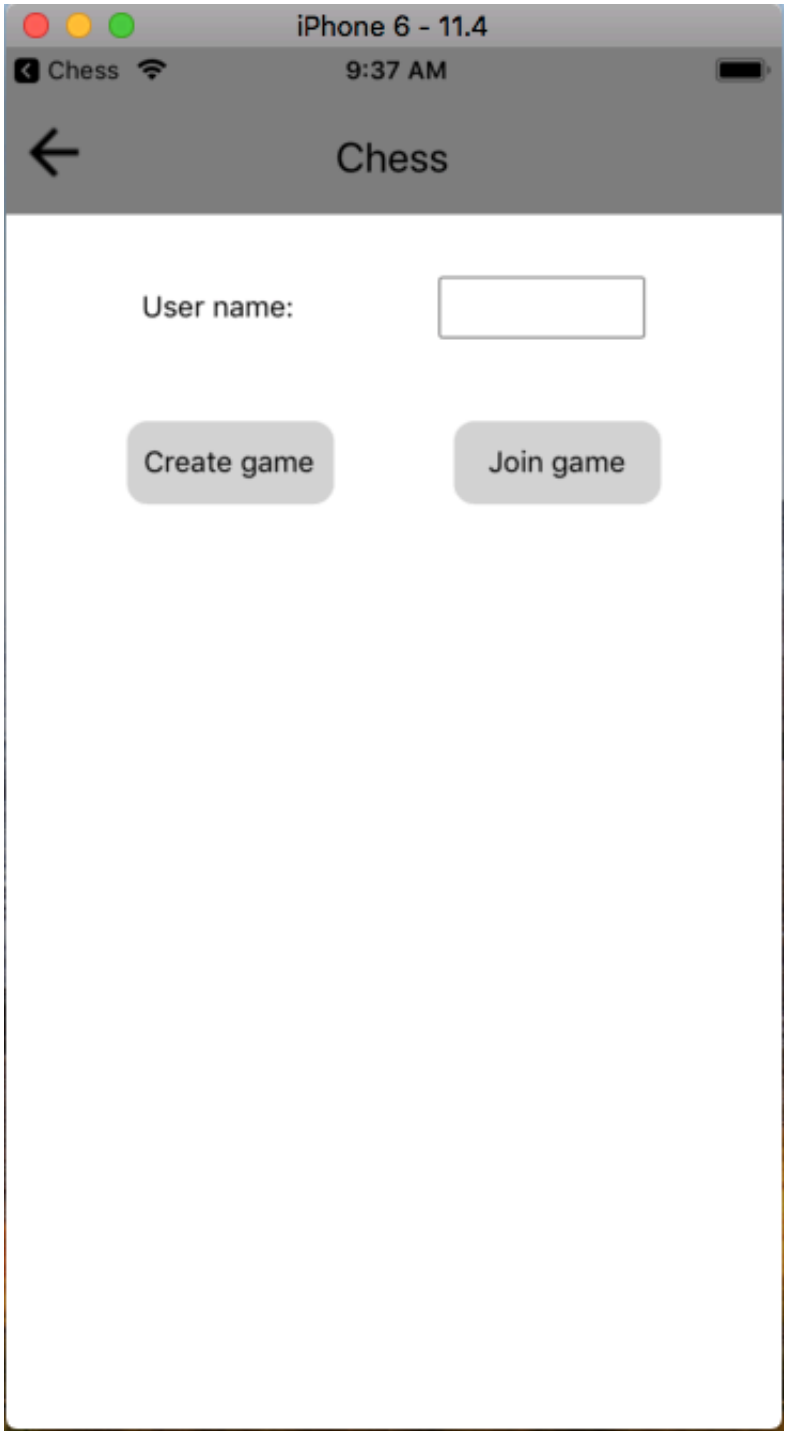
Slika 3.8. Metoda waitForWhiteMove na strukturi Server

Slika 3.8. prikazuje metodu na strukturi server koja čeka na odigravanje poteza bijelog igrača, prvo provjerava može li se igra nastaviti tj. postoji li šah mat ili pat. Ukoliko postoji šah mat proces

na sloju servera završava te se klijentskoj strani šalje odgovor o pobjedniku ili o neriješenom ishodu. U suprotnom ako ne postoji šah mat ili pat igra se nastavlja te metoda poziva blokirajuću naredbu koja čeka na potez bijelog igrača, provjerava valjanost poteza, ukoliko potez nije valjan vraća poruku s greškom te nastavlja čekati. Ako metoda kanalom od klijentskog sloja dobije valjan potez odigrava taj potez te šalje korisnicima FEN string o trenutnom stanju šahovskog polja i boju igrača čiji je red. U ovom slučaju metoda po završetku mijenja stanje u "waitForBlackMove" – čeka na potez crnog igrača, postupak se ponavlja dok ne dođe do šah mat-a ili pata. Ukoliko igrač ne napravi odigra potez u definiranom vremenu za odigravanje igra se prekida te je pobjednik protivnik.

3.2. Klijentska aplikacija

Aplikacija je izrađena u React-Native frameworku tako da je moguć rad na obje platforme (Android i iOS). Kada korisnik instalira aplikacije te ju pokrene prvo je potrebno unijeti korisničko ime (Slika 3.9.).



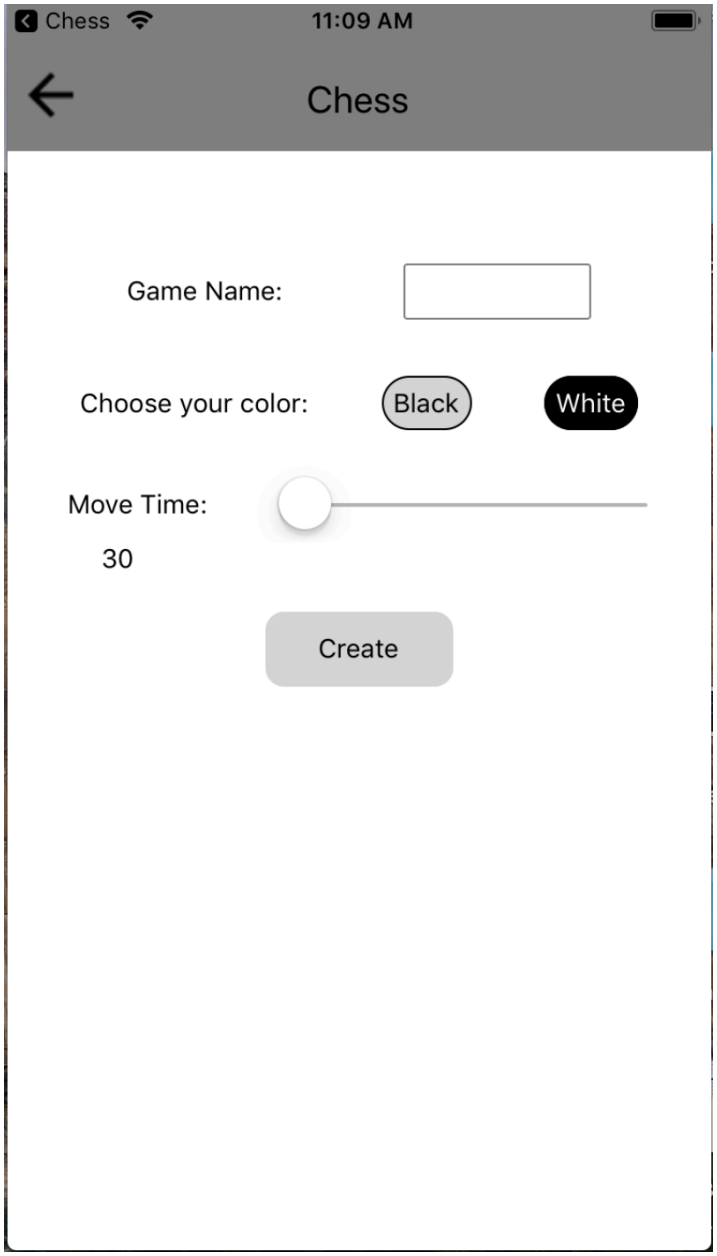
Slika 3.9. Prikaz ekrana za odabir između pridruživanja i kreiranja igre

Nakon unosa korisničkog imena, potrebno je odabrati želi li korisnik kreirati novi igru ili ući u jednu od već kreiranih od strane ostalih korisnika. Ukoliko želi sam kreirati igru biti će proslijeđen na ekran za kreiranje igre. Mijenjanje pogleda korisnika vrši se pomoću eksterne biblioteke "react-router-native" koja koristi jednostavnu logiku definira se router (Slika 3.10.) te put ("path") svake komponente, kada je put trenutne aplikacije jedan putu komponente prikazuje se ta komponenta.

```
const Router = () => {
    return (
        <NativeRouter>
            <View style={style.container}>
                <Route exact path="/" component={IntroScreen} />
                <Route path="/settingsInput" component={SettingsInput} />
                <Route path="/createGame" component={CreateGame} />
                <Route path="/gamesScreen" component={GamesScreen} />
                <Route path="/adminGame" component={AdminGame} />
                <Route path="/playerGame" component={PlayerGame} />
            </View>
        </NativeRouter>
    );
};
```

Slika 3.10. Router komponenta

Prilikom kreiranja igre potrebno je odabrati ime igre, svoju boju figura i vrijeme koliko će trajati potez (Slika 3.11.). Ograničenja kreiranja sobe su postavljena tako da duljina imena sobe mora biti između 3 i 50 znakova. Nakon kreiranja igre admin prvo se treba spojiti u igru s odgovarajućim ID-em igre i admin tokenom. Spajanje se odrađuje automatski od strane klijentske aplikacije nakon odgovora servera da je kreiranje igre završeno i dobivanja admin tokena. Nakon spajanja admina u igru, poslužitelj prelazi u stanje "waitForPlayer". U ovom trenutku admin može prestati čekati na pridruživanje drugog igrača te raspustiti igru. Ukoliko drugi igrač se pridruži stanje poslužitelja se mijenja u "startGame" tu postavljamo sve potrebne parametre za igranje i mijenjamo stanje u "waitForWhiteMove" te se čeka na potez bijelog igrača koji u šahu uvijek igra prvi.



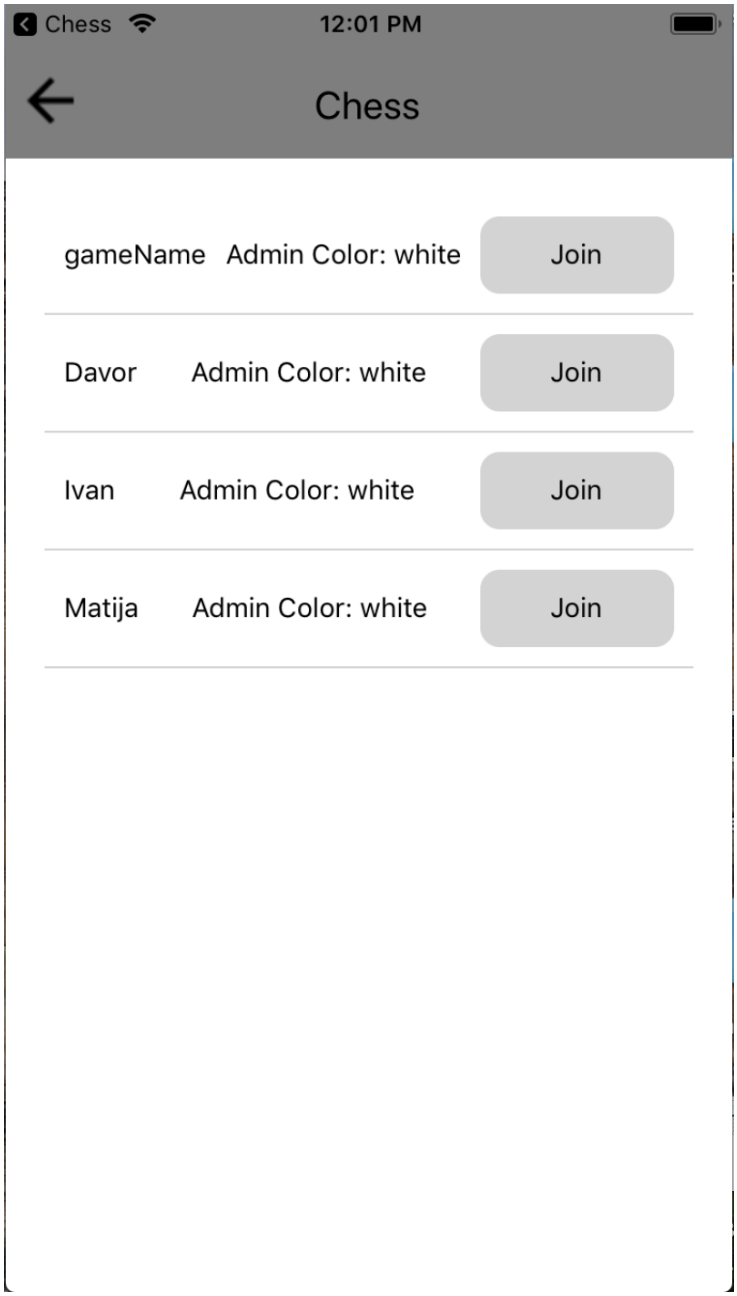
Slika 3.11. Komponenta zadužena za kreiranje nove igre

Ukoliko igrač se želi pridružiti već jednoj od kreiranih igara te odabere "Join Game" na zaslonu korisnika se prikazuje lista trenutno aktivnih igara. Aktivne igre se ažuriraju cijelo vrijeme od samog početka aplikacije, za to je zadužen "GameEvents" sloj koji putem WebSocketa prima informacije o kreiranim i obrisanim igrama. Nakon primitka događaja o kreiranju igre on mijenja Redux stanje te dodaje tu igru u listu trenutno aktivnih igara. U slučaju primitka događaja o brisanju ili početku jedne od trenutno aktivnih igara, on ju briše iz trenutnog stanja. Korisniku na ekranu se prikazuju igre u koje se može pridružiti.

```
import React, { Component } from "react";
import { View, StyleSheet } from "react-native";
import { GameEventsType } from "../types/gameEvents";
import { connect } from "react-redux";
import { gameEventAction } from "./actions";

interface Props {
    children: JSX.Element;
    dispatch: any;
}
class GameEvents extends Component<Props> {
    public ws: WebSocket | null;
    public response: GameEventsType | null;
    constructor(props: any) {
        super(props);
        this.ws = null;
        this.response = null;
    }
    componentWillMount() {
        this.ws = new WebSocket(
            "https://chessbackendfinal.herokuapp.com/game_events"
        );
        this.ws.onmessage = e => {
            console.log(e.data);
            this.response = GameEventsType.fromJSON(e.data);
            this.props.dispatch(gameEventAction(this.response));
        };
    }
    public render() {
        return <View style={style.container}>{this.props.children}</View>;
    }
}

const style = StyleSheet.create({
    container: {
        flex: 1
    }
});

export default connect()(GameEvents);
```

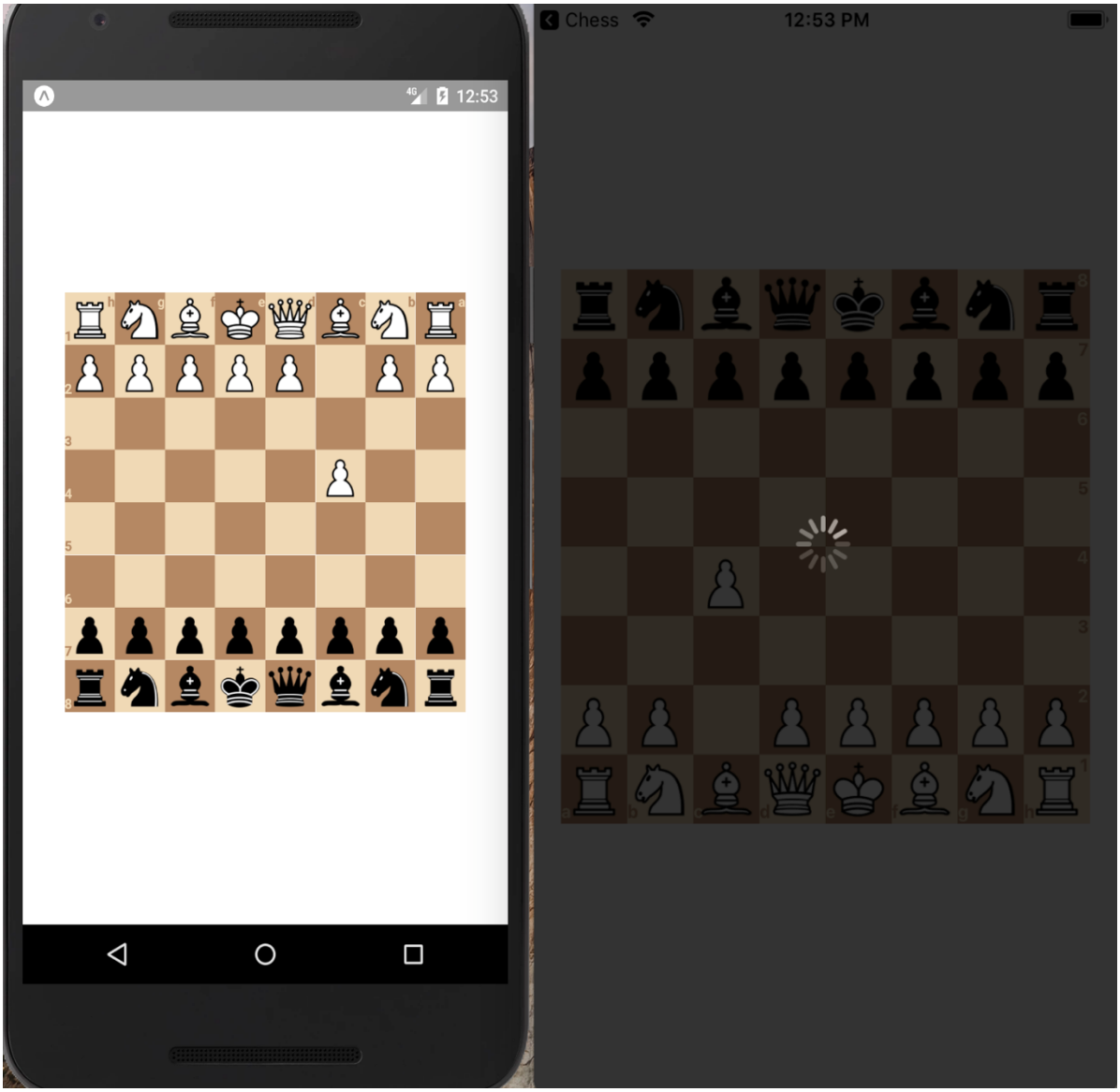
Slika 3.12. GameEvents komponenta za dodavanje i micanje igara s liste trenutno aktivnih

Nakon primitka poruke o događaju na webcocketu "GameEvents" (Slika 3.12.) poruka se prvo predaje metodi fromJSON koja dobivenu poruku pretvara u objekt klase GameEventsType te verificira jesu li tipovi podataka kao što su definirani. Nakon toga objekt se predaje akciji "gameEventAction" koja ovisno o tipu događaja okida slučaj u reduceru te mijenja trenutnu listu aktivnih igara.



Slika 3.13. Prikaz liste trenutno aktivnih igara

Korisniku se prikazuje lista trenutno aktivnih igara (Slika 3.13.) te može odabrati igru kojoj će se pridružiti, boja drugog igraća se automatski dodjeljuje suprotno onoj koju je odabrao admin igre. Nakon Pridruživanja igri, sama partija šaha započinje. Prvi je na redu igrač koji je odabrao bijelu boju figura, prilikom kreiranja igre definiran je vremenski period za odigravanje poteza (moveTime) ukoliko igrač ne odigra potez u zadanom vremenskom periodu smatra se da je on neaktivan te automatski protivnik pobjeđuje u igri. Nakon odigranog poteza, igraču se na zaslonu uređaja pokazuje ikona koja označava čekanje te se čeka na potez protivnika



Slika 3.14. Prikaz igre oba igrača

Slika 3.14. prikazuje tijek igre na obje platforme, bijeli igrač je odigrao potez te čeka na odigravanje poteza drugog igrača. Za pozicioniranje i responzivnost aplikacije koristi se flexbox te napravljena scale funkcija koja prilagođava veličinu elemenata s obzirom na dimenzije mobilnog uređaja, tako da je aplikaciju moguće pokrenuti praktički na svim Android i iOS uređajima neovisno o veličini i verziji sustava. Komunikacija između poslužitelja i klijenta za vrijeme igre koristi websocket, putem njega se šalju potezi koji su odigrani. Svaki igrač prije poteza putem websocketa dobiva trenutnu poziciju figura na polju, sve moguće poteze te oznaku da je njegov red za odigrati. Igrač bira jedan od mogućih poteza te ga šalje poslužitelju koji potez provjerava te odigrava.

Komunikacija između klijenta i poslužitelja na klijentskoj strani organizirana je pomoću tri sloja. Prvi sloj je "Loading middleware" koji prilikom svakog zahtjeva poslanog poslužitelju okida prikaz ikone koja simbolizira učitavanje te na taj način poboljšava korisnički doživljaj same aplikacije. Nakon završetka zahtjeva okida se akcija koja brise ikonu učitavanja sa zaslona korisnika te mu daje do znanja da je zahtjev uspješno završen. Sljedeći sloj je "Error middleware" koji u slučaju greške koju vraća poslužitelj prilikom slanja zahtjeva prikazuje grešku krajnjem korisniku te mu daje do znanja do kakvog tipa greške je došlo prilikom zahtjeva. Posljednji sloj je sam pristup poslužiteljskom servisu koji stvara URL te pomoću axios biblioteke pristupa poslužitelju i očekuje njegov odgovor. Asinkrona komunikacija i serverski dio u ovom radu su neophodni jer server mora voditi računa o stanjima oba igrača i obavještavati ih o promjenama na šahovskoj ploči.

4. ZAKLJUČAK

Uspješnim završetkom diplomskog rada napravljena je mobilna međuplatformska aplikacija za online igranje šaha. Aplikaciju mogu koristiti svi korisnici koji posjeduju pametan uređaj bez obzira radi li se o Android ili iOS operacijskom sustavu. Aplikacija sadrži sve potrebne funkcije koje omogućavaju nesmetanu igru te obavještava korisnika o svim greškama u unosu potrebnih stvari za igru. Također obavještava korisnika o nastalim problemima u komunikaciji, kao što su komunikacijske greške. Prilikom izrade korištene su najnovije tehnologije koje omogućuju vrlo brz i efikasan razvoj same aplikacije.

Zadatak ovog diplomskog rada je bio razvoj mobilne aplikacije za online igranje šaha, detaljno je opisan postupak kreiranja svih dijelova poslužitelja koji koristi Finite State Machine (FMS) za praćenje stanja u toku igre. Također poslužitelj je podijeljen u tri sloja, svaki svoj ima svoju zadaću te je omogućena komunikacija između slojeva jer u ovom slučaju ona je neophodna. Klijentska aplikacija također je podijeljena na tri sloja koji nam omogućavaju lakše korištenje poslužitelja te krajnjem klijentu pruža puno kvalitetniji korisnički doživljaj aplikacije.

Testiranje poslužitelja provedeno je putem različitih alata za simulaciju komunikacije websocketa i alata Postman za testiranje izvršavanja serverskih zahtjeva. Mobilna aplikacija testirana je na nekoliko uređaja različite veličine kako bi se potvrdio prilagodba na različite širine i visine korisničkog zaslona. Aplikacija funkcionira bez ikakvih grešaka na poslužiteljskoj i korisničkoj strani, samim time smatra se da je uspješno završen diplomski rad.

LITERATURA

[1] Wikipedia, Visual Studio Code (IDE) https://en.wikipedia.org/wiki/Visual_Studio_Code (stranica posjećena: 5. Srpnja 2018)

[2] Wikipedia, Golang (jezik poslužitelja) https://en.wikipedia.org/wiki/Go_(programming_language) (stranica posjećena: 5. Srpnja 2018)

[3] Golang Documentation (dokumentacija jezika poslužitelja) https://golang.org/doc/ (stranica posjećena: 6. Srpnja 2018)

[4] React Native Github (React Native framework) https://facebook.github.io/react-native/ (stranica posjećena: 22. Srpnja 2018)

[5] TypeScript Github (JavaScript superset) https://github.com/Microsoft/TypeScript/wiki (stranica posjećena: 10. Srpnja 2018)

[6] React Native (React Native framework) http://www.reactnative.com/ (stranica posjećena 25. Srpnja 2018)

[7] Go šah biblioteka (Notnill/chess) https://github.com/notnil/chess (stranica posjećena: 6. Kolovoza 2018)

SAŽETAK

Naslov: Mobilna aplikacija za šah na React-Native platformi

U ovom diplomskom radu izrađena je mobilna aplikacija za online igranje šaha. U izradi aplikacije korištene su trenutno aktualne tehnologije na strani klijenta i poslužitelja. Aplikacija sadrži sve potrebne funkcije za potpun doživljaj klijenta te da mu omogući nesmetanu interakciju s poslužiteljem. Izrada aplikacije je podijeljena u dva dijela, prvi dio je izrada poslužitelja, dok je drugi izrada klijenta. Nakon uspješne provedbe obje faze diplomskog rada svi problemi su riješeni te se možemo koristiti mobilnom aplikacijom za šah.

Ključne riječi: Golang, Javascript, TypeScript, React Native, Android, iOS, WebSocket

ABSTRACT

Title: React-Native chess mobile application

In this master thesis paper is done mobile application for multiplayer chess playing. For making this application are used most advanced and newest technologies for developing mobile applications and server side developing. Application has all needed functionalities to make player experience better and to make clear communication to the server side. Application development is divided to two stages, first stage was developing server side and second stage was developing client mobile application for interacting with server. After successful end of both stages all problems are solved. User can use chess application without any bugs or further development.

Keywords: Golang, Javascript, TypeScript, React Native, Android, iOS, WebSocket

ŽIVOTOPIS

Davor Buha rođen je 31. siječnja u Našicama. U Našicama 2009. završava osnovnu školu „Kralja Tomislava “. Iste godine upisuje srednju prirodoslovno-matematičku gimnaziju u školi “Isidora Kršnjavog” u Našicama koju završava 2013. godine. Tijekom cijelog srednjoškolskog obrazovanja ostvaruje odličan uspjeh. 2013. godine upisuje preddiplomski sveučilišni studij računarstva na Elektrotehničkom fakultetu u Osijeku. 2017. godine završava preddiplomski sveučilišni studij računarstva na Elektrotehničkom fakultetu u Osijeku te upisuje diplomski studij smjer DRD – Informacijske i podatkovne znanosti. Tijekom 2. godine diplomskog studija odrađuje praksu u inozemnom poduzeću putem Erasmus organizacije. Po završetku studija planira pronaći posao u struci kao programer mobilnih aplikacija.

_____

Davor Buha