IMPLEMENTACIJA WEB UTIČNICA NA PRIMJERU IZRADE IGRE

Amanzi, Alex

Undergraduate thesis / Završni rad

2022

Degree Grantor / Ustanova koja je dodijelila akademski / stručni stupanj: **University of Split / Sveučilište u Splitu**

Permanent link / Trajna poveznica: https://urn.nsk.hr/urn:nbn:hr:228:378787

Rights / Prava: In copyright / Zaštićeno autorskim pravom.

Download date / Datum preuzimanja: **2024-05-17**

Repository / Repozitorij:

Repository of University Department of Professional Studies

zir.nsk.hr

UNIVERSITY OF SPLIT

dabar
DIGITALNI AKADEMSKI ARHIVI I REPOZITORIJI

SVEUČILIŠTE U SPLITU

SVEUČILIŠNI ODJEL ZA STRUČNE STUDIJE

Preddiplomski stručni studij Informacijska tehnologija

ALEX AMANZI

Z A V R Š N I R A D

IMPLEMENTACIJA WEB UTIČNICA NA PRIMJERU IZRADE IGRE

Split, rujan 2022.

SVEUČILIŠTE U SPLITU
SVEUČILIŠNI ODJEL ZA STRUČNE STUDIJE

Preddiplomski stručni studij Informacijska tehnologija

Predmet: Programiranje na internetu

Z A V R Š N I R A D

Kandidat: Alex Amanzi

Naslov rada: Implementacija web utičnica na primjeru izrade igre

Mentor: Marina Rodić, viši predavač

Split, rujan 2022.

Sadržaj

Sažetak 1

Summary 1

1 Uvod 2

2 Tehnologije 3

2.1 Socket.IO 3

2.2 Express 3

2.3 Knex 4

2.4 PostgreSQL 4

2.5 React 5

2.6 Styled Components 5

3 Funkcionalnosti 7

3.1 Početni prikaz 7

3.2 Poveži četiri 8

3.3 Interaktivni kviz 10

4 Izvedba aplikacijske logike 14

4.1 Tok podataka 14

4.2 Struktura baze podataka 15

4.3 Arhitektura poslužitelja 15

4.3.1 Database 15

4.3.2 Models 16

4.3.3 Events 16

4.3.4 Utils 17

4.4 Arhitektura klijenta 17

4.4.1 Konfiguracijske datoteke 17

4.4.2 Scenes 17

4.4.3 Components 17

4.4.4 Providers 18

4.4.5 Assets 18

4.4.6 Consts 18

4.4.7 Utils 18

4.5 Implementacijski detalji 19

4.5.1 Kreiranje igre 19

4.5.2 Povezivanje u kreiranu igru 23

4.5.3 Upravljanje stanjem klijenta 25

4.5.4 Poveži četiri 27

4.5.5 Prikazivanje rezultata u interaktivnom kvizu 30

4.5.6 Stiliziranje glavnog gumba 32
5 Zaključak 34
Literatura 35

Sažetak

Ovaj završni rad opisuje izradu web aplikacije kolekcije igara s naglaskom na implementaciju web utičnica *(engl. web socket)*. Igre su zamišljene tako da se glavni sadržaj prikazuje na jednom uređaju, dok igrači koriste dodatne uređaje kao upravljače. Aplikacija sadrži dvije igre. Jedna igra je interaktivni kviz gdje je igračima cilj zavarati protivnike dajući domišljate krive odgovore na postavljeno pitanje, a usput odabrati točan odgovor na isto pitanje među mnoštvom krivih odgovora. Druga igra je implementacija društvene igre *Poveži četiri (engl. Connect four)* za dva igrača. Igrači naizmjence postavljaju novčiće u polja s ciljem da njihovi novčići naprave niz od četiri uzastopna novčića horizontalno, vertikalno ili dijagonalno.

Ključne riječi: Web utičnice, Express, React, JavaScript, Igra

Summary

Implementation of web sockets in the development of a game.

This paper describes the process of building a web application focusing on the implementation of web sockets. The web application is a hub for games which are played through multiple devices. One device is used to display the main content to players, while other devices are used by players as controllers. This application contains two games. One game is an interactive quiz where players try to deceive each other by giving false answers to a prompt while also trying to guess the correct answer themselves. The second game is an implementation of the board game *Connect four* for two players. Players take turns dropping coins onto the board with the aim of creating a horizontal, vertical or diagonal line of four of their coins.

Keywords: Web sockets, Express, React, JavaScript, Game

1 Uvod

Ideja za izradu web aplikacije nastala je iz vlastitih potreba tijekom pandemije. Igara koje podržavaju veću količinu igrača i mogućnost igranja putem videopoziva vrlo je malo. Tijekom razvoja aplikacije kviza uočena je potreba da svaki igrač ima mogućnost odgovaranja na postavljeno pitanje. Istraživanjem mogućih rješenja istaknuo se protokol web utičnica koji omogućuje komunikaciju u stvarnom vremenu *(engl. real time)*. Od tu proizlazi motivacija za daljnji razvoj aplikacije koja implementira tehnologiju web utičnica tako da ta tehnologija pogoni veliki dio sustava. Za temu je zadržan inicijalni format igre kako je za implementaciju poslovne logike igre potrebno pamtiti puno informacija koje se često mijenjaju. Dodana je funkcionalnost kolekcije igara, to jest razvijene su dvije igre i mogućnost odabira koja će se igra igrati kako bi se aplikacija mogla nadograditi s dodatnim igrama. Osim proučavanja web utičnica, cilj je bio razviti nekonvencionalnu arhitekturu poslužitelja koja se oslanja isključivo na web utičnicama.

U prvom poglavlju Tehnologije objašnjavaju se odabrani alati, razvojni okviri i biblioteke za izradu aplikacije, a uz to i njihove prednosti, mane i usporedbe s alternativama. U drugom poglavlju Funkcionalnosti objašnjena je poslovna logika aplikacije iz perspektive klijenta. U trećem poglavlju Izvedba aplikacijske logike objašnjen je tok podataka, struktura baze podataka, arhitektura poslužitelja, arhitektura klijenta i implementacijski detalji. U četvrtom poglavlju Zaključak nalazi se osvrt i moguće nadogradnje.

2 Tehnologije

U ovom su poglavlju objašnjene odabrane tehnologije i alati za izvedbu aplikacije. Tehnologije Express, Knex i PostgreSQL koriste se na poslužitelju, React i Styled components na klijentu, dok se tehnologija Socket.IO koristi i na klijentu i na poslužitelju.

2.1 Socket.IO

Socket.IO je biblioteka za JavaScript programski jezik koja se koristi za postizanje komunikacije u stvarnom vremenu između poslužitelja i klijenta ili između više klijenata u oba smjera. Alternativa utičnicama bilo bi uzastopno slanje zahtjeva prema poslužitelju sve dok se ne dogodi željena promjena što rezultira velikom količinom razmijenjenih podataka. Rad s web utičnicama bazira se na programiranju pogonjenom događajima *(engl. event-driven programming)*. Za jedan se događaj koji se definira na utičnici dodaje povratni poziv *(engl. callback)* koji se izvršava u trenutku okidanja događaja. Događaje mogu okidati klijent i poslužitelj. Socket.IO biblioteka pruža jednostavnije sučelje od onog nativnog za rad s utičnicama i različite nadogradnje. Primjerice: automatska rezerva na stare metode realiziranja istog ponašanja ako protokol web utičnica nije omogućen, analitika, grupiranje utičnica, jednostavno slanje događaja na više utičnica odjednom i slično. [1]

S druge strane, postoji i biblioteka SignalR koja je razvijena za Microsoftov razvojni okvir ASP.NET. SignalR nudi višu razinu apstrakcije and utičnicama pa ju je iz istog razloga jednostavnije koristiti. Kako je za izradu poslužitelja odabrano JavaScript okruženje, izabrana je Socket.IO biblioteka.

2.2 Express

Express je razvojni okvir za Node.js koji uz minimalno postavljanje omogućuje brz početak izrade aplikacije. Iako nije jedini razvojni okvir za Node, trenutno ga se smatra nepropisnim standardom uzimajući u obzir njegovu učestalost korištenja naspram drugih. Razlog je tome upravo jednostavnost korištenja, robusnost i širok spektar mogućnosti koje nudi. U usporedbi s drugim razvojnim okvirima poput: .Net Core, Ruby on Rails i Spring Boot, Express sigurno nije najbrži. Korištenjem Express-a moguće je razvijati klijentsku i poslužiteljsku stranu u JavaScript programskom jeziku. Dobrobit toga širok je spektar biblioteka koje razvija zajednica. Aplikacije koje su izrađene u Node.js-u jeftinije su za proizvesti jer je razvoj takvih aplikacija brži na početku. Upravo to čini Node.js odličan izbor za manje aplikacije. Kada nastane potreba za skaliranjem aplikacije potrebno je obratiti puno više pažnje da se može besprijekorno održavati.

2.3 Knex

Knex je biblioteka koja nudi funkcionalnosti graditelja upita za JavaScript što znači da omogućava komunikaciju s bazom podataka. Postoje dvije alternative graditelja upita, a to su: sirovi strukturirani jezik upita *(engl. raw structured query language, skraćenica raw SQL)* i objektno relacijsko mapiranje *(engl. object-relational mapping, skraćenica ORM)*. Graditelj upita i ORM nude apstrakciju nad sirovim SQL-om što ih čini otpornijima na greške u sintaksi i ubrizgavanje SQL-a *(engl. SQL injection)*. Također, omogućuju jednostavno upravljanje nad promjenama baze podataka. Iako je velika razina apstrakcije benefit ORM-a, to ujedno može biti i mana. Kako ORM apstrahira upite, može se dogoditi da generira dugotrajne operacije pri izvođenju. Graditelj upita izabran je radi balansa apstrakcije i kontrole, a Knex biblioteka izabrana je radi intuitivne sintakse. [2]

2.4 PostgreSQL

PostgreSQL je implementacija relacijske baze podataka otvorenog tipa kôda *(engl. open-source)* što znači da je izvorni kôd dostupan svima za pregled i nadogradnju. Relacijske baze podataka trenutno su najrašireniji tip baza podataka, a baziraju se na relacijskom modelu upravljanja podataka. Postoje i drugi tipovi baza podataka poput: graf baza podataka, ne-relacijskih baza podataka i drugih od kojih svaki ima svoju specijaliziranu namjenu. Graf baze podataka odlične su ako postoji mnogo veza među podacima pa su odlične prilikom izrade društvenih mreža. Ne-relacijske baze podataka uvode nove načine razmišljanja poput ugnježđivanja podataka za realizaciju relacija. Optimalan način spremanja podataka za razvoj ovakve aplikacije bilo bi korištenje baze podataka koja sprema podatke u memoriji. Ipak, odabran je relacijski tip baze podataka kako bi ostalo prostora za buduće nadogradnje. Uvođenjem korisničkih računa omogućio bi se razvoj funkcionalnosti poput personaliziranih konfiguracija igara. Na primjer, korisnici bi za interaktivni kviz mogli dodavati vlastita pitanja koja bi bilo potrebno spremati u bazu podataka. Nadalje, neke bi igre mogle sadržavati mogućnost prikazivanja rang liste svih igrača. Izbor relacijskih baza podataka je velik, a neke od njih su: SQL Server, MySQL, SQLite i slično. SQL Server Microsoftova je baza podataka. Njegova je potpora na drugim operacijskim sustavima ograničena te podržava samo Windows operacijski sustav. MySQL, kao i PostgreSQL, podržava i druge platforme poput Unix i MacOS. PostgreSQL zapravo nije prava relacijska baza, već objektna relacijska baza što znači da podržava kompleksnije tipove poput nizova. PostgreSQL odabran je radi razvoja aplikacije na MacOS operacijskom sustavu i jednostavnosti korištenja.

2.5 React

React je biblioteka koja se koristi za izradu korisničkog sučelja. Glavni razlog zašto je biblioteka, a ne razvojno okruženje je to što je React zapravo samo set funkcija čiji je posao brinuti se o stanju i prikazivanju istoga u objektni model dokumenta *(engl. document object model, skraćenica DOM)*, a uz to ne nameće strukturu aplikacije. Za ispisivanje hipertekst jezika za označivanje *(engl. hypertext markup language, skraćenica HTML)* u DOM, React implementira iznimno brz mehanizam, virtualni DOM koji je preslika pravog DOM-a. Glavni zadatak mu je biti poveznica React kôda i korisničkog sučelja, a tako se apstrahira JavaScript kôd za manipulaciju DOM-a. Uz to se smanjuje količina iscrtavanja elemenata na ekranu što je najzahtjevnija operacija koja se može izvršavati u pregledniku. [3] Kako bi kôd izgledao što sličniji HTML-u, React se najčešće piše koristeći JSX ekstenziju za JavaScript koja vizualno nalikuje HTML-u, ali uz to koristi sve mogućnosti JavaScript-a. React se može pisati bez JSX-a koristeći objekte koji predstavljaju elemente u DOM stablu i pozivanjem Reactove funkcije za ispisivanje. React je trenutno najrašireniji alat za izradu korisničkog sučelja, a odmah iza njega su Angular i Vue koji su razvojna okruženja. Jedna od bitnijih razlika je što React implementira jednosmjerno povezivanje podataka, dok Angular i Vue implementiraju dvosmjerno povezivanje podataka. Povezivanje podataka odnosi se na povezivanje informacija između komponente i sučelja. Kod React-a podaci mogu ići samo u jednom smjeru. [4] Važno je spomenuti dodatna dva razvojna okruženja bazirana na React-u, React Native i Next. React Native koristi se za izradu mobilnih aplikacija, a sve popularniji Next nudi mnoštvo dodatnih alata poput: iznimno jednostavnog rutiranja, mogućnosti izvršavanja klijentskog kôda na strani poslužitelja ili statički prilikom izgradnje produkcijskog kôda. React je odabran radi jednostavnosti manipuliranja informacija i ugrađenog mehanizma kontekst *(engl. context)* koji se koristi za dijeljenje stanja kroz komponente.

2.6 Styled Components

Styled components biblioteka je za React i React Native koja omogućava pisanje stilova kroz JavaScript. Poziva se funkcija iz biblioteke i prosljeđuje kôd kaskadnih stilskih listova *(engl. cascading style sheets, skraćenica CSS)* kao parameter, a kao rezultat vraća React komponentu što čini biblioteku iznimno jednostavnu za korištenje. Styled components ustvari koristi nadograđenu CSS sintaksu *(skraćenica SCSS)* što čini biblioteku još snažnijom tako da nudi dodatne mogućnosti za pisanje čitljivijeg i održivijeg kôda. Također, Styled components nudi mehanizme koji olakšavaju izradu i održavanje različitih tema boja

za aplikaciju. Jedan od najčešćih primjera bio bi kada aplikacija korisnicima nudi odabir između svijetle i tamne palete boja. Ono što je najvrjednije kod ove biblioteke upravo je to što se piše JavaScript, a ne CSS što otvara dosta mogućnosti. Takve biblioteke za stiliziranje nose ime CSS u JavaScriptu *(engl. CSS in JavaScript)*, a njihova glavna značajka je mogućnost prosljeđivanja parametara direktno u kôd za stiliziranje. Upravo ovakvo ponašanje omogućuje puno jednostavnije manipuliranje stilova i iznimno olakšava rad sa stilovima ako se radi o kompleksnijim aplikacijama ili promjenama u stvarnom vremenu. S druge strane, mana ovakvih biblioteka je što je potrebno određeno vrijeme da se JavaScript kôd pretvori u CSS što može utjecati na performanse aplikacije. Ovaj je alat odabran radi mogućnosti izrade vrlo interaktivnih elemenata poput dugmeta koji ima štih trodimenzionalnog pritiskanja dugmeta prilikom prelaska mišem preko njega.

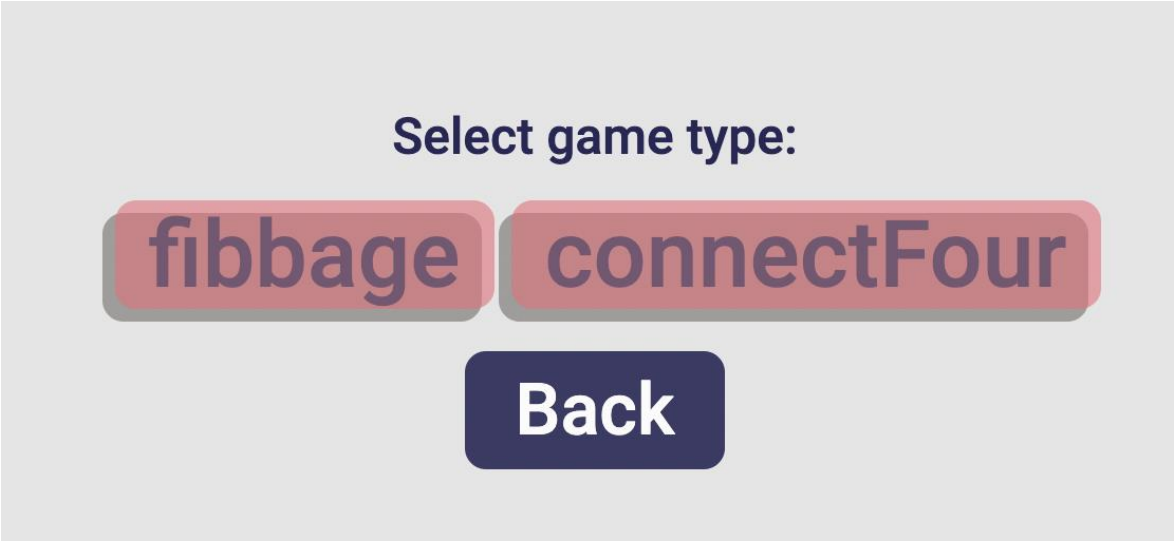
3 Funkcionalnosti

U ovom će poglavlju biti objašnjene sve funkcionalnosti aplikacije iz perspektive korisnika, a uz to i same igre.

3.1 Početni prikaz

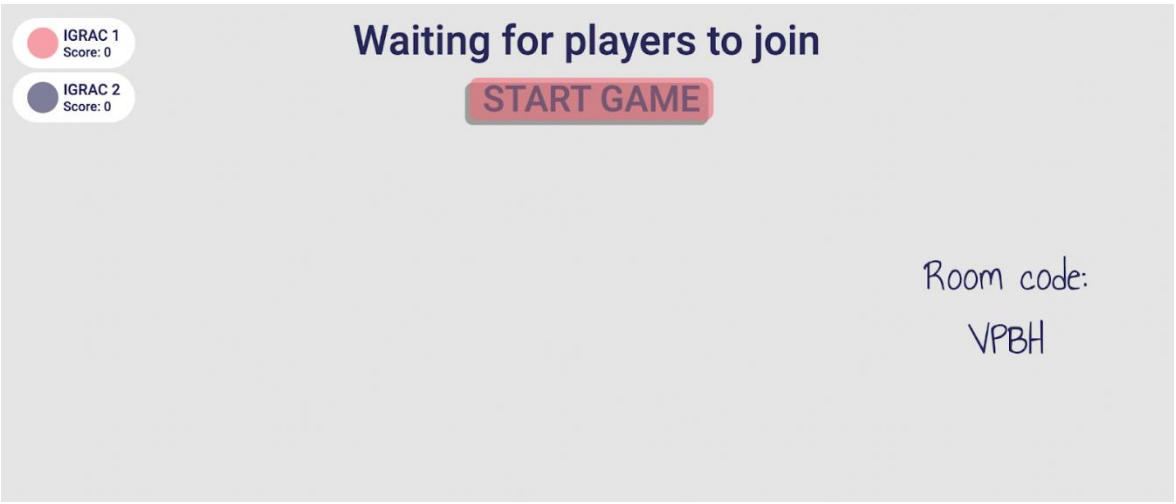
Na početnom se prikazu nalazi dugme za kreiranje igre i forma pomoću koje se igrač uključuje na već kreiranu igru. Klikom na dugme *Create game* mijenja se prikaz te se korisniku nudi odabir koju igru želi započeti. Kada se započne igra taj se korisnik tretira kao domaćin i na njegovom će zaslonu biti prikazan kôd od četiri slova pomoću kojeg se igrači uključuju u tu igru. Igrači preko drugog uređaja unose kôd u odgovarajuće polje i ispod njega svoje ime ili nadimak i tako se pridružuju igri. Na slici 1 je prikazan početni zaslon za kreiranje igre i uključivanje u postojeću, dok se na slici 2 nalazi prikaz pomoću kojeg se bira koja će se igra kreirati.

Slika 1 Početni prikaz.



Slika 2 Prikaz za biranje igre.

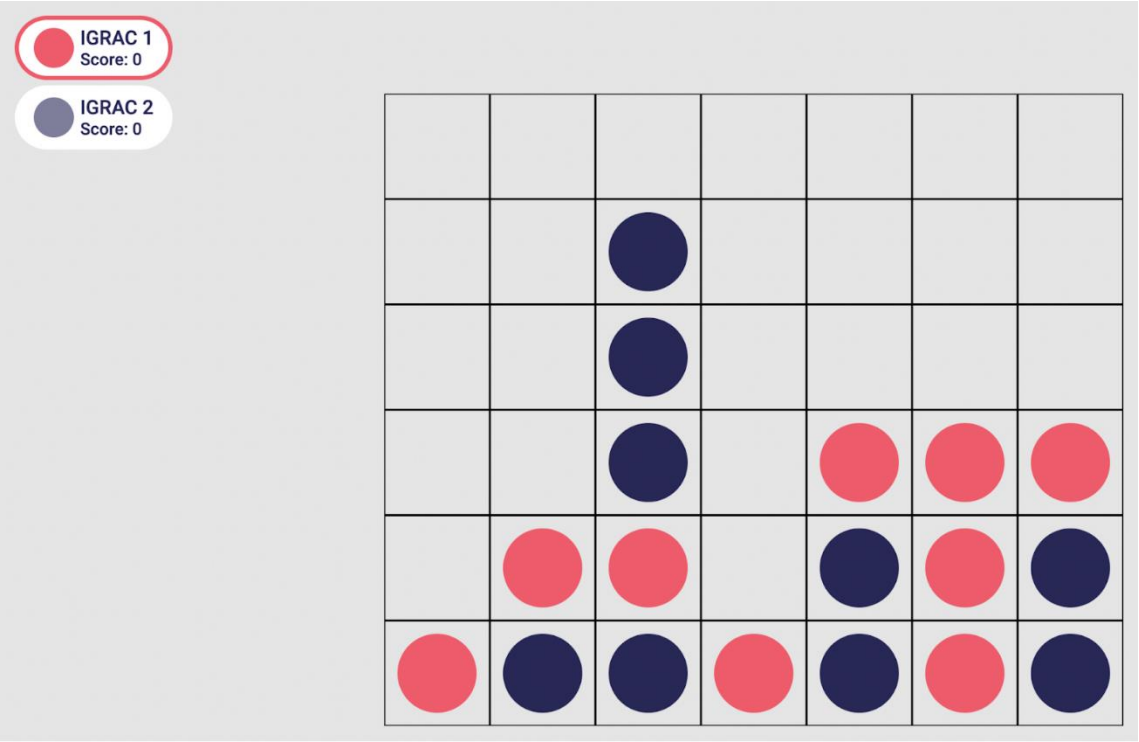
Kako se igrači priključuju u igru prikazivat će se njihove oznake na zaslonu domaćina, a ako je zadovoljen minimalan broj igrača za igru prikaže se i dugme za pokrenuti igru. Prvi igrač koji se priključio smatra se moderatorom igre i on kao takav preko svog uređaja ima mogućnost pokretanja igre. Na slici 3 nalazi se prikaz zaslona domaćina prije početka igre gdje su prikazane: oznake igrača s njihovim bodovima, dugme za početak igre i kôd za priključiti se igri.



Slika 3 Prikaz zaslona domaćina prije početka igre.

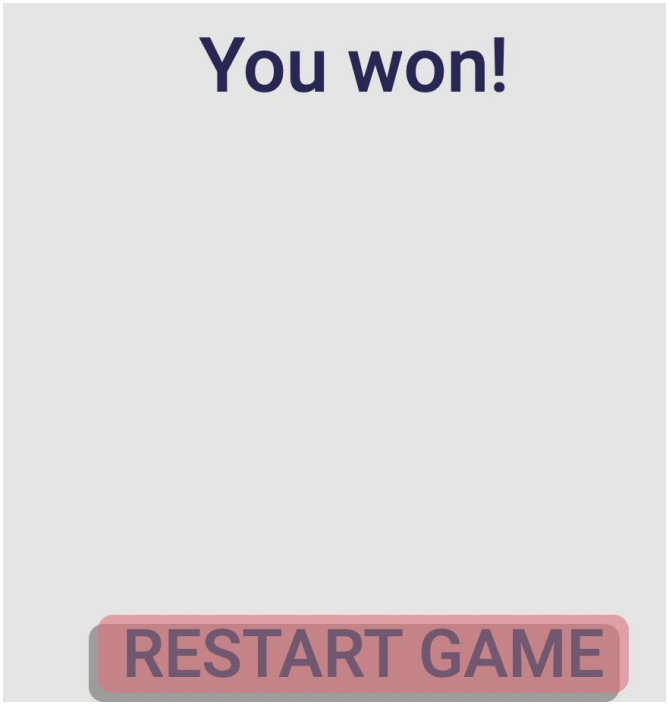
3.2 Poveži četiri

Klikom na stupce ploče igrači naizmjence ubacuju novčiće s ciljem da ih, u bilo kojem smjeru, povežu četiri uzastopno. To može biti vertikalno, horizontalno ili po dijagonali. Svaki igrač na svom uređaju vidi ploču s kojom može interaktirati, a na uređaju domaćina prikazana je ploča i trenutni igrač koji je na redu u obliku podebljane oznake igrača. Slika 4 prikazuje zaslon domaćina tijekom igre *Poveži četiri*.



Slika 4 Poveži četiri na zaslonu domaćina.

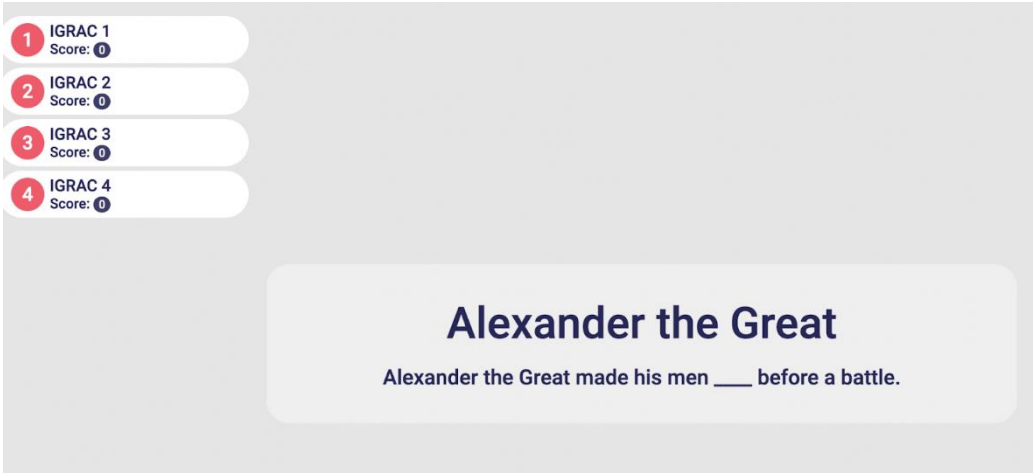
U trenutku pobjede jednog igrača, igračima se prikazuje jesu li pobijedili ili izgubili, a moderatoru i dugme za započeti novu igru. Na zaslonu domaćina ažuriraju se bodovi. Slika 5 prikazuje zaslon na uređaju pobjednika nakon završene partije.



Slika 5 Prikaz zaslona pobjedničkog igrača koji je ujedno i moderator.

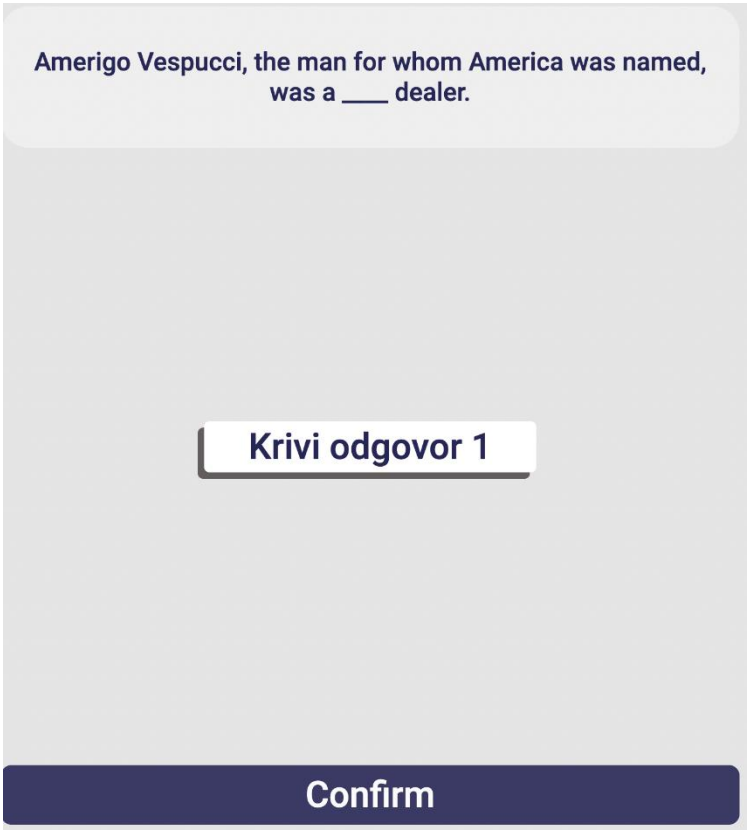
3.3 Interaktivni kviz

Tijekom interaktivnog kviza na ekranu domaćina prikazuje se postavljeno pitanje i bodovi pojedinih igrača što je prikazano na slici 6.



Slika 6 Prikaz zaslona domaćina s postavljenim pitanjem za interaktivni kviz.

Također, igračima se prikaže pitanje, ali i mogućnost unosa odgovora. Igračima je cilj dati odgovor na pitanje što bliže istini s obzirom na to da će se u sljedećem koraku prikazati svi odgovori igrača, a uz to i točan odgovor. Slika 7 prikazuje zaslon igrača s poljem za unos odgovora.



Slika 7 Prikaz zaslona igrača s poljem za unos odgovora.

Nakon davanja krivih odgovora na pitanje, na zaslonu domaćina prikazuju se svi mogući odgovori, dok se igračima prikazuju svi osim onoga kojeg je taj igrač postavio. Igrači potom moraju odabrati jedno od ponuđenih odgovora s ciljem pogađanja točnog. Na sljedećim slikama prikazani su ponuđeni odgovori iz perspektive domaćina i igrača. Slika 8 prikazuje zaslon domaćina, a slika 9 zaslon igrača. Može se primijetiti kako među odgovorima ponuđenim igraču nedostaje odgovor s brojem jedan kako je to igrač koji je postavio taj krivi odgovor.

Slika 8 Prikaz ponuđenih odgovora na zaslonu domaćina.

Slika 9 Prikaz odabira odgovora na zaslonu igrača.

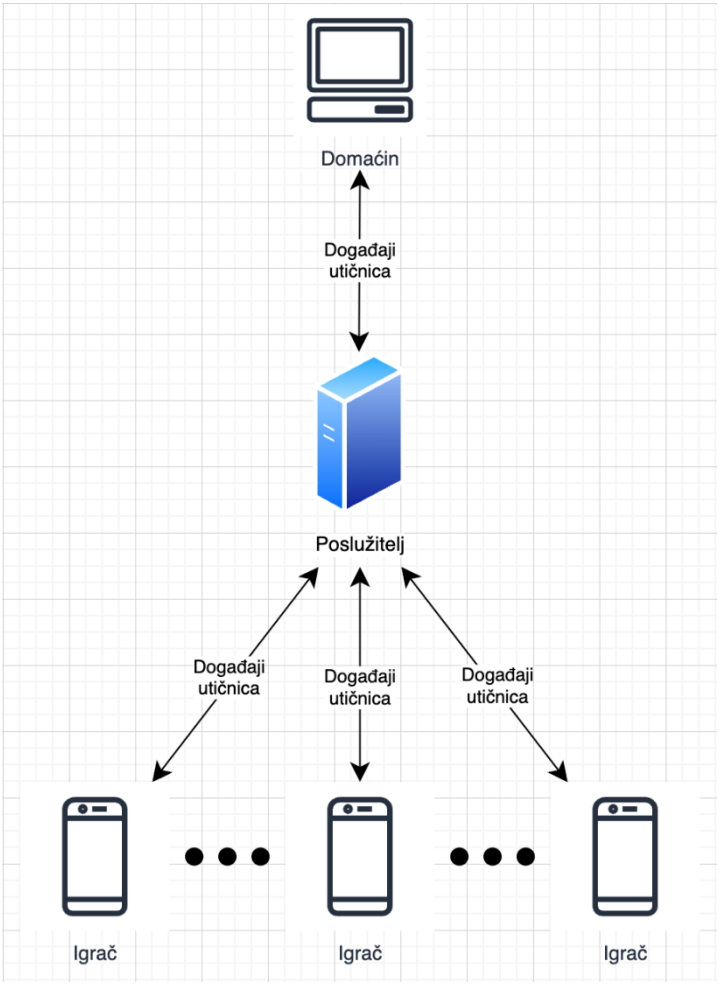
Nakon odabira odgovora na zaslonu domaćina prikazuju se rezultati. Način na koji se prikazuju rezultati je da se prikazuju redom odgovori i imena igrača koji su glasali za taj odgovor. Na primjeru sa slike 10 prikazano je da su igrači tri i četiri odabrali odgovor kojeg je postavio igrač jedan. Igrači potom dobivaju bodove ovisno o tome jesu li pogodili točan odgovor i dodatne bodove ako su uspjeli svojim krivim odgovorom zavarati nekog drugog igrača. Igra se odvija u tri runde, a svaka sljedeća runda donosi više bodova.

Slika 10 Prikaz rezultata odgovora i igrača koji su glasali za isti.

4 Izvedba aplikacijske logike

4.1 Tok podataka

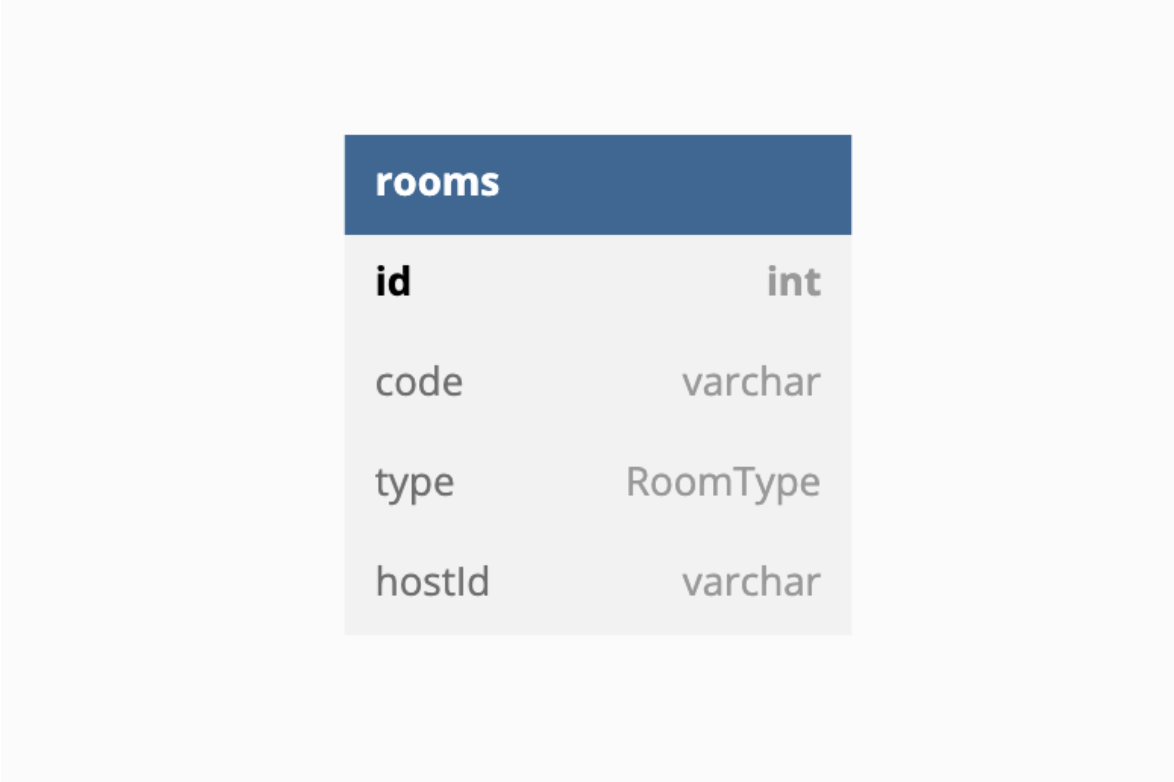
Interakcija s aplikacijom zamišljena je tako da je potrebno istovremeno koristiti različite uređaje koji mogu imati jednu od dvije uloge. Jedan uređaj, na primjer računalo, ima ulogu prikazivanja zajedničkih informacija svim igračima te ga se smatra domaćinom *(engl. host)*. Takav će uređaj prikazivati podatke poput: stanja bodova svih igrača, trenutno postavljeno pitanje u kvizu, igraču ploču u igri *poveži četiri* i slično. Zadatak je ovakvog klijenta, osim prikazivanja informacija, upravljanje cijelog toka igre. Druga vrsta klijenta predstavlja pojedinog igrača, na primjer mobilni uređaj, a uloga takvog klijenta je služiti kao upravljač korisniku kako bi korisnik mogao sudjelovati u igri, to jest odraditi svoj potez. Koristeći web utičnice domaćin okida događaje na poslužitelju, a potom poslužitelj okida događaje na jednom ili više uređaja igrača. Igrači potom biraju svoje poteze te se, opet pomoću utičnica, podaci propagiraju do domaćina. Tok podataka vizualiziran je na slici 11.



Slika 11 Tok podataka.

4.2 Struktura baze podataka

Za spremanje potrebnih podataka za izradu aplikacije korištena je jedna tablica u bazi podataka čija je struktura prikazana na slici 12.



Slika 12 Struktura baze podataka.

Kako bi bilo moguće da se različiti klijenti spoje na istu igru, a uz to i podržati da se istovremeno igraju različite igre, potrebno je spremati iduće podatke: unikatni identifikator partije, to jest sobe za igru, tip sobe koji definira koja se igra igra i identifikator domaćina. Identifikator domaćina potreban je radi načina komunikacije. Poslužitelj mora znati koji je točno uređaj domaćina u trenutku kada klijenti igrača uzrokuju događaj. Također, vrijedi i obrnuto da poslužitelj mora znati koji su identifikatori klijenata igrača što je objašnjeno u potpoglavlju Kreiranje igre poglavlja Implementacijski detalji.

4.3 Arhitektura poslužitelja

Arhitektura poslužitelja organizirana je po mapama tako da jedna mapa definira funkcionalnosti svih datoteka koje se nalaze unutra.

4.3.1 Database

Database mapa sadrži potreban kôd za povezivanje s bazom podataka, upravljanje migracijama i tablicama na niskoj razini koristeći biblioteku Knex. Ova se mapa brine o tome da se poslužitelj uspješno poveže s bazom podataka i da je ona točno onakve strukture

kakvu poslužitelj očekuje. U datoteci *knexfile.js* nalazi se konfiguracija baze podataka koja sadrži informacije o vrsti baze podataka, načinu povezivanja i podatke o migracijama. S obzirom na to da su jedini podaci koji se zapisuju u bazu podataka oni o aktivnim igrama, to jest sobama za igru, migracije nisu u potpunosti implementirane, nego je pripremljena mogućnost za nadogradnju. Knex kroz sučelje naredbenog retka *(engl. command line interface)* nudi mogućnost automatskog generiranja migracija uspoređivanjem trenutne strukture baze podataka i one definirane u kôdu. To znači da bi se za nadogradnju većina kôda pisala unutar *package.json* datoteke kako bi ga bilo što jednostavnije za koristiti. Migracije se zapisuju u mapi migrations koja je izdvojena od izvornog kôda aplikacije. [5]

4.3.2 Models

Mapa *models* sadrži sve definicije tablica baze podataka i svih popratnih podataka koji su bitni za tu tablicu. U datoteci *Room.js* upravo se to i nalazi, a iz nje se izvoze *(engl. export)* funkcije i varijable. Izvozi se ime tablice i funkcija koja kreira shemu tablice koji se koriste unutar database mape prilikom pokretanja poslužitelja. Zamrznuti se objekti kreiraju pomoću funkcije *Object.freeze*, a u JavaScriptu najčešće se koriste za realiziranje fiksnih tipova bolje poznatim imenom *enum* u drugim programskim jezicima. Razlog zbog čega se upravo takvi objekti koriste za to je što se ne mogu promijeniti tijekom izvršavanja aplikacije. Najbitnija je funkcija koja se izvozi iz ove datoteke ona koja prilikom izvođenja vraća graditelja upita za tu tablicu. Graditelj upita radi koristeći uzorak ulančavanja funkcija *(engl. function chaining pattern)* što znači da su povratne vrijednosti funkcija objekt nad kojima se može dodatno pozivati određeni set funkcija. [6]

4.3.3 Events

U *events* mapi odvija se glavni dio poslovne logike, a uz to i upravljanje utičnicama. Datoteka *index.js* sadrži dvije funkcije *handleCreate* i *handleJoin* koje su ulazne točke za klijente domaćina i igrača. Ovdje su definirani svi povratni pozivi na događaje utičnica koje sve igre dijele međusobno poput: kreiranja igre, ulaženja u postojeću igru, oznaka početka igre i slično. U events mapi se također nalaze i druge dvije mape imena *client* i *host* od kojih svaka sadrži dvije datoteke *connectFour.js* i *fibbage.js*. Svrha je tih mapa i datoteka implementirati povratne pozive za događaje specifične za određenu igru i specifične za domaćina ili igrače. Ovakav pristup je odabran kako bi se poslovna logika mogla smisleno odvojiti.

4.3.4 Utils

Zadnja, a uz to i najjednostavnija mapa na poslužitelju je *utils* mapa koja sadrži funkcije za manipulaciju podataka na nižoj razini apstrakcije poput manipuliranja nizova ili riječi *(engl. string)*. Za svrhu ovog projekta, kreirana je datoteka *string.js* koja sadrži funkciju za nasumično generiranje kôda sobe za igru.

4.4 Arhitektura klijenta

Za izradu klijentske strane korišteno je grupiranje datoteka u mape po funkcionalnostima s iznimkom pet datoteka koje stoje samo unutar *src* mape, a služe za konfiguraciju klijenta. Mape koje se koriste su: *assets*, *components*, *consts*, *providers*, *scenes* i *utils*.

4.4.1 Konfiguracijske datoteke

Ulazna datoteka *index.js* služi za inicijaliziranje React biblioteke unutar HTML elementa čija je vrijednost atributa *id* jednaka *root*. Navedeno znači da je za pokretanje React komponenti potrebno servirati HTML datoteku s takvim elementom, a ona se nalazi unutar *public* mape. Datoteke koje se nalaze unutar *public* mape ne prolaze kroz proces svezivanja *(engl. bundle)*. Datoteka *index.js* poziva komponentu *App* koja se nalazi u datoteci *App.js* koja onda poziva datoteku *Routes.js* za rutiranje klijenta na različite stranice. U datoteci *App.css* definirani su stilovi za prebrisati zadane stilove web preglednika i konfiguracija fontova koji se koriste kroz aplikaciju. U posljednjoj datoteci *config.js* definirane su konfiguracijske varijable poput rute poslužitelja, konekcije na web utičnicu i dodatne varijable za jednostavniji rad i testiranje.

4.4.2 Scenes

Mapa *scenes* sadrži komponente stranica aplikacije, a aplikacija je podijeljena na tri stranice. Glavna stranica na koju korisnik prvo dođe i pomoću koje bira hoće li kreirati igru ili se priključiti već kreiranoj nalazi se unutar *Dashboard* mape. Ako je korisnik kreirao igru, aplikacija korisnika vodi na stranicu za domaćina koja se nalazi unutar *RoomHost* mape, a ako je korisnik unio kôd i ime za priključiti se kreiranoj igri, aplikacija korisnika vodi na stranicu za igrača čija se mapa zove *RoomClient*.

4.4.3 Components

Mapa *components* sadrži stilizirane komponente i komponente igara. Datoteka *styled.js* definira stilove dijeljene među komponentama tako da koristi biblioteku Styled components i izvozi isključivo stilizirane komponente bez funkcionalnosti. Mape

ConnectFour i *Fibbage* sadrže komponente igara podijeljenih na komponente za domaćina i igrače u mapama *Client* i *Host*. Osim funkcionalnih komponenti, svaka od tih mapa ima i svoju *index.styled.js* datoteku koja definira specifične stilove za tu grupu te stilove rasporeda elemenata koje nije moguće definirati kroz dijeljenu datoteku *styled.js*.

4.4.4 Providers

Providers mapa implementira jezgru poslovne logike igara i upravlja cijelim stanjem istih što ju čini jednom od najzahtjevnijim dijelova aplikacije. Ime *providers* dano je po uzorku posluživanja *(engl. provider pattern)* koji se implementira kroz datoteke ove mape. On je potreban kako bi se svim komponentama jednostavno pružilo podatke o stanju i funkcije za manipulaciju istoga. U protivnom, bilo bi potrebno prosljeđivati podatke kroz komponente što bi rezultiralo neorganiziranim, zbunjujućim i neodrživim kôdom. Ovakav pristup odabran je radi potrebe čuvanja kompleksne logike o igrama na strani klijenta. [7] Za realizaciju ovakvog pristupa korišten je mehanizam kontekst iz biblioteke React. Jedna od alternativa koja se jako često koristi je biblioteka Redux čiji je izvorni tvorac isti kao i Reactov. Glavna je razlika, a i dobrobit biblioteke Redux, što se prilikom korištenja konteksta kada se mijenja dijeljeno stanje uzrokuje ponovno prikazivanje *(engl. re-render)* svih komponenti u stablu gdje je definirano korištenje takve metode. Redux biblioteka ponovno prikazuje samo one komponente u kojima se koriste takva stanja. Pristup pomoću konteksta odabran je radi toga što je kontekst ugrađen u React biblioteku pa nije potrebno dodavanje još jednog paketa. [8]

4.4.5 Assets

Mapa *assets* služi kao skladište datoteka poput fontova i slika.

4.4.6 Consts

Consts mapa služi za spremanje konstantnih varijabli bitnih za aplikaciju. U njoj se nalazi jedna datoteka imena *enums.js* u kojoj se nalaze fiksni tipovi za klijenta poput tipa igre i stanja pojedinih igara sa svrhom jednostavnijeg korištenja i imenovanja.

4.4.7 Utils

Kao i kod poslužitelja, *utils* mapa koristi se za manipuliranje jednostavnijim podacima. U njoj se nalaze dvije datoteke, *array.js* i *wait.js*. Datoteka *array.js* sadrži funkcije za miješanje elemenata niza i za grupiranje niza po svojstvu njegovih elemenata. Datoteka *wait.js* sadrži funkciju koja je prikazana na odsječku kôda, ispis 1.

```
export const wait = (ms) => {
  return new Promise((resolve) => setTimeout(resolve, ms));
};
```

Ispis 1 funckija wait.

Svrha je ove funkcije pojednostaviti korištenje *setTimeout* globalne metode koja se koristi pomoću povratnih poziva. Funkcija vraća obećanje *(engl. promise)* što je upravo razlog zašto je jednostavnija za korištenje. Kako se radi o obećanju, unutar asinkrone funkcije dodaje se ključna riječ *await* prije poziva ove funkcije kako bi se pričekalo daljnje izvršavanje prije nego je isteklo proslijeđeno vrijeme u milisekundama. Način na koji ovo radi je da se kreira obećanje koje izvršava funkciju *setTimeout,* a za povratni se poziv šalje *resolve* funkcija koja označuje kraj obećanja što znači da obećanje završava kada istekne proslijeđeno vrijeme.

4.5 Implementacijski detalji

4.5.1 Kreiranje igre

Kreiranje igre započinje na strani klijenta. Pomoću funkcije *handleCreateGame* sprema se stanje željene igre pomoću kojeg će se na klijentu prikazivati odabrana igra. Put poslužitelja šalje se događaj *host/send/create*. Kako je interakcija između klijenata i poslužitelja velika, potrebno je bilo osmisliti sustav imenovanja događaja koji se sastoji od tri dijela. Prvi dio definira utječe li događaj na domaćina ili na igrača, drugi dio definira šalje li se događaj s klijenta poslužitelju ili klijent sluša događaj poslužitelja, a treći je dio ime samog događaja. Događaj se šalje pomoću biblioteke Socket.IO, metodom *emit*. Ispis 2 prikazuje *handleCreateGame* funkciju.

```
const handleCreateGame = (type) => {
   setRoomType(type);
   socket.emit('host/send/create', {
     socketId: socket.id,
     type,
   });
 };
```

Ispis 2 Kreiranje igre na klijentu.

Koristeći metodu *on* iz biblioteke Socket.IO, na poslužitelju se sluša okidanje događaja *host/send/create*. Kada se on izvrši, poslužitelj prima tip igre i identifikator utičnice klijenta koji će uskoro postati domaćin. Pomoću *resolver* objekta za entitet sobe kreira se nova soba u bazi podataka koja će imati nasumično generiran kôd za pristup igri. Ako se dogodi pogreška na bazi podataka, šalje se događaj tako da klijent može prikazati odgovarajuću

poruku, a u protivnom se domaćinu šalje kôd kreirane sobe za igru. Ispis 3 prikazuje odgovor na događaj poslan s klijenta za kreiranje sobe i slanja povratnih informacija klijentu domaćinu.

```
socket.on('host/send/create', async ({ socketId, type }) => {
   const room = await RoomsResolver.mutation.createRoom(type, socketId);

   if (!room) {
     socket.emit('host/receive/room-create-error');

     return;
   }

   socket.emit('host/receive/room-create-success', {
     roomCode: room.code,
   });
   …
};
```

Ispis 3 Kreiranje igre na poslužitelju.

Kôd za priključivanje u igru se generira pomoćnom funkcijom *getRandomString* koja za proslijeđenu željenu duljinu riječi vraća nasumično odabrana slova. To radi tako da se prolazeći kroz *for* petlju odabire slovo na nasumičnom indeksu iz konstante koja sadrži sva moguća slova. Koristeći metodu *random* nad ugrađenim objektom *Math* koji služi za matematičke operacije u JavaScriptu, dobije se pseudoslučajni decimalni broj između nula i jedan gdje je nula uključena u mogućnost povratne vrijednosti, a jedan nije. Taj se rezultat množi s brojem ponuđenih slova i sve se skupa zaokružuje na manji cijeli broj. S obzirom na to da se radi s indeksima, potrebno je uključiti nulu i nema potrebe za dodatno rukovanje s rezultatom jer je potrebno ostaviti mogućnost da rezultat bude nula. U protivnom, rezultat bi se zbrajao s jedan. Ispis 4 prikazuje pomoćnu funkciju *getRandomString*.

```
export const getRandomString = (length) => {
 let result = '';
 const characters = 'ABCDEFGHIJKLMNOPQRSTUVWXYZ';

 for (let i = 0; i < length; i++) {
   result += characters.charAt(Math.floor(Math.random() * characters.length));
 }

 return result;
};
```

Ispis 4 Generiranje kôda za priključivanje u igru.

Kako je poslužitelj morao slušati događaje klijenta, tako i klijent mora slušati okidanja događaja s poslužitelja. Osim što je potrebno rukovati podacima, jako bitan faktor je okruženje klijenta. Kako se radi o React biblioteci, potrebno je pripaziti na ponovno prikazivanje komponenti. Prilikom promjene bilo koje vrijednosti stanja komponente React ponovo izvršava kôd te komponente i svih onih koje se nalaze ispod te u DOM stablu. To znači da je moguće postaviti više slušatelja događaja *(engl. event listeners)* što bi uzrokovalo problem da kada bi se jednom okinuo događaj, više puta bi se izvršio kôd za odgovor. U React se biblioteci to rješava koristeći *useEffect* kuku. Kuka *useEffect* služi za ponovno izvršavanje kôda koji se zove posljedica *(engl. effect)* kada se vrijednosti proslijeđenih ovisnosti *(engl. dependency)* promijene. U kuku se prosljeđuju dva parametra. Jedan je funkcija koja će se izvršiti kao posljedica, a drugi je niz ovisnosti. Ako je niz ovisnosti prazan, posljedica se izvršava samo jednom prilikom postavljanja komponente. Proslijeđena funkcija može vraćati jednu funkciju koja definira kôd koji će se izvršiti odmah nakon promjene ovisnosti, ali prije ponovnog izvršavanja posljedice. Ovakvu funkciju zovemo funkcijom čišćenja *(engl. cleanup function)*. S obzirom na sve navedeno, za slušati događaje utičnica na klijentu potrebno je pomoću funkcije čišćenja pobrinuti se da će se slušatelj događaja ugasiti prilikom demontiranja komponente. [9] Na odsječku kôda, ispis 5 prikazano je slušanje događaja na klijentu kroz primjer događaja uspješnog kreiranja sobe za igru.

```
useEffect(() => {
   socket.on('host/receive/room-create-success', ({ roomCode }) => {
     setCreatedRoomCode(roomCode);
   });

   return () => {
     socket.off('host/receive/room-create-success');
   };
 }, []);
```

Ispis 5 Slušanje događaja na klijentu.

Nakon dobivene potvrde o uspješno kreiranoj sobi klijenta se preusmjerava na rutu za domaćina koja prikazuje odabranu igru. Ispis 6 prikazuje komponentu na ruti domaćina i prikazivanje igre, a uz to i provjeru je li korisnik došao na rutu tako da ju je upisao direktno u traku za rutu ili kroz aplikaciju. Za jednostavniji rad i testiranje omogućeno je direktno dolaženje na rutu ako je tako postavljeno kroz konfiguracijske varijable.

```
const RoomHost = () => {
 const history = useHistory();
 const { roomType } = useParams();

 if ((!dev || !playersDevMode) && history.action !== 'REPLACE') {
   return <Redirect to='/' />;
 }

 switch (roomType) {
   case ROOM_TYPE.fibbage:
     return (
       <FibbageProvider>
         <FibbageHost />
       </FibbageProvider>
     );
   case ROOM_TYPE.connectFour:
     return (
       <ConnectFourProvider>
         <ConnectFourHost />
       </ConnectFourProvider>
     );
   default:
     return null;
 }
};
```

Ispis 6 Komponenta domaćina.

Kako bi se odvojio kôd događaja pojedinih igara, dodan je odsječak kôda za odabir koji će se događaji slušati. Oni su izdvojeni u datoteke koje izvoze funkcije sa slušateljima događaja. Osim objekata *socket* i *io* koji su potrebni za upravljanje događajima potrebno je slati i listu svih identifikatora utičnica igrača. Kako se ti podaci prosljeđuju odmah prilikom kreiranja igre, nije moguće niz identifikatora utičnica direktno proslijediti u odgovarajuću funkciju jer prilikom promjene se ista neće detektirati i unutar funkcije događaja za igru. Iz tog je razloga potrebno proslijediti funkciju koja dohvaća vrijednost niza i nju koristiti. Na odsječku kôda, ispis 7 prikazano je biranje funkcije sa slušateljima događaja specifičnih za odabranu igru i prosljeđivanje informacije o identifikatorima igrača koji igraju.

```
let playerSockets = [];

const getPlayerSockets = () => {
  return playerSockets;
};

switch (room.type) {
  case roomTypeEnum.fibbage:
    fibbageHostEvents(socket, io, getPlayerSockets);
    break;
  case roomTypeEnum.connectFour:
    connectFourHostEvents(socket, io, getPlayerSockets);
    break;
  default:
    break;
};
```

Ispis 7 Odabir slušatelja događaja za igru.

4.5.2 Povezivanje u kreiranu igru

Srž logike povezivanja u kreiranu igru identična je kao i za kreiranje, a prolazi se kroz iste korake. Razlika je u tome što je potrebno provjeriti postoji li soba s proslijeđenim kôdom i da domaćin provjeri može li se igrač uopće spojiti u igru. Igrač se ne može spojiti u igru ako je maksimalan broj igrača dosegnut ili ako je igra već započeta. To je realizirano tako da se domaćinu, čiji je identifikator spremljen u bazu, šalje događaj koji označava igračev pokušavaj spajanja u igru. Nadalje, klijent domaćina odrađuje sve potrebne provjere i šalje odgovarajući događaj poslužitelju koji ga za kraj prosljeđuje natrag klijentu igrača. Ispis 8 prikazuje odgovor poslužitelja na igračev zahtjev za priključivanje igri, dok su na slici 13 prikazani potrebni koraci u komunikaciji između klijenta i poslužitelja.

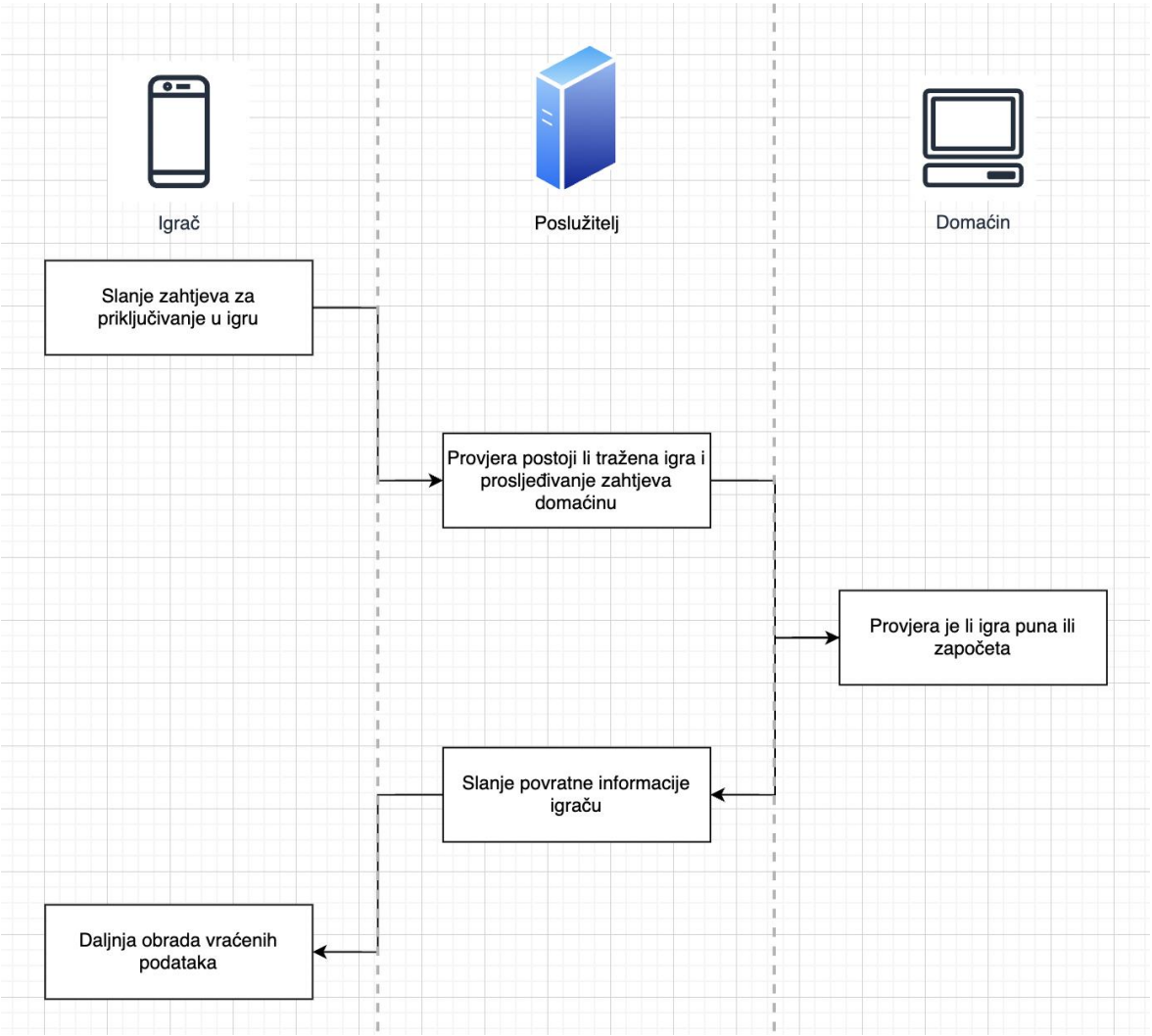
```
socket.on('client/send/join', async ({ roomCode, username, socketId }) => {
   const room = await RoomsResolver.query.room(roomCode);

   if (!room) {
     socket.emit('client/receive/join-error', {
       message: 'Invalid room code',
     });

     return;
   }

   io.to(room.hostId).emit('host/receive/player-join', {
     name: username,
     socketId,
   });
   …
};
```

Ispis 8 Slušatelj događaja priključivanja u igru.



Slika 13 Komunikacija potrebna za priključivanje u igru.

4.5.3 Upravljanje stanjem klijenta

Kao što je spomenuto u poglavlju o arhitekturi klijenta, stanjem igara se u cijelosti upravlja pomoću kontekst mehanizma React biblioteke. Kako bi se kôd podalje grupirao u smislene cjeline, koristio se reduktor uzorak *(engl. reducer pattern)*. On pruža dodatnu razinu apstrakcije prilikom upravljanja stanjem tako da se stanje mijenja isključivo kroz funkciju reduktora. U funkciju se prosljeđuje objekt akcije koji sadrži informaciju o tipu promjene stanja i dodatne podatke, a kao povratnu vrijednost vraća novo stanje nakon promjene. [10] Ispis 9 prikazuje dio reduktor funkcije za dodavanje novog igrača.

```
const reducer = (state = initialState, action) => {
 switch (action.type) {
   case actionType.ADD_PLAYER:
     if (state.playerOne === null) {
       return {
         ...state,
         playerOne: action.player,
       };
     }

     if (state.playerTwo === null) {
       return {
         ...state,
         playerTwo: action.player,
       };
     }

     return { ...state };
  …
};
```

Ispis 9 Dodavanje novog igrača kroz reduktor.

Kako bi se funkcija reduktora povezala s uzorkom poslužitelja, a uz to i u potpunosti konfigurirao reduktor, React biblioteka pruža kuku koja se zove *useReducer*. Njezinim se pozivom dobiva stanje aplikacije koje će se koristiti i funkcija pomoću koje se pristupa reduktoru. Ovo je potrebno kako bi reduktor funkcija uvijek manipulirala trenutnim stanjem i povratnom vrijednošću ažurirala isto. Ispis 10 prikazuje poziv *useReducer* kuke i funkciju pomoću koje se poziva reduktor akcijom za dodavanje igrača.

```
const [state, dispatch] = useReducer(reducer, initialState);

 const addPlayer = (player) => {
   dispatch({
     type: actionType.ADD_PLAYER,
     player: {
       ...player,
       score: 0,
     },
   });
 };
```

Ispis 10 Poziv reduktora.

4.5.4 Poveži četiri

Zbog jednostavnijeg manipuliranja podacima i prikazivanja na zaslon, korišten je dvodimenzionalni niz kao tip podatka za predstavljanje ploče. Elementi vanjskog niza predstavljaju stupce ploče za igru, dok elementi unutarnjeg niza predstavljaju pojedino polje ploče. Prolazeći kroz oba niza i koristeći njihove indekse jednostavno se dolazi do podatka o koordinatama polja. Elementi polja inicijalno nemaju vrijednost jer su polja prazna, a kasnije poprimaju vrijednost koja predstavlja jednog ili drugog igrača kako bi se prikazao ispravan novčić. Ispis 11 sadrži prvobitno popunjavanje ploče koristeći konfiguracijske varijable za veličinu ploče.

```
export const constructNewBoard = () => {
 return new Array(connectFourConfig.boardSize.columns).fill(
   new Array(connectFourConfig.boardSize.rows).fill(null)
 );
};
```

Ispis 11 Popunjavanje ploče.

Kada igrač koji je na redu klikne na jedan od stupaca ubacuje se novčić na prvo slobodno mjesto u stupcu. Kako bi se to dogodilo, potrebno je pomoću utičnice domaćinu proslijediti podatak o odabranom stupcu te domaćin ažurira podatak o ploči pomoću reduktora. Koristeći dvodimenzionalni niz ovo je realizirano tako da se u nizu proslijeđenog stupca očitava prvo prazno polje, to jest ono koje nema vrijednost. Bitno je naglasiti da prilikom korištenja reduktora nije dozvoljeno izravno mijenjati stanje, već je potrebno kreirati kopije svih referentnih tipova varijabli jer će u protivnom nastati greške. Ispis 12 sadrži ažuriranje ploče nakon ubacivanja novčića.

```
const rowIndex = state.board[action.columnIndex].findIndex(
  (cell) => cell === null
);

if (rowIndex === -1) {
  return { ...state };
}

const boardAfterDropCoin = state.board.map((column) => [...column]);
const playerThatDropppedCoin =
  state.currentEvent === CONNECT_FOUR_EVENT_TYPE.playerOneTurn
    ? PLAYER.one
    : PLAYER.two;

boardAfterDropCoin[action.columnIndex][rowIndex] = playerThatDropppedCoin;
```

Ispis 12 Ubacivanje novčića u stupac.

Odmah nakon ubacivanja novčića u stupac potrebno je provjeriti je li igrač pobijedio. Provjera pobjede riješena je tako da se za bačeni novčić uzimaju nizovi s podacima svih polja za stupac, redak i dijagonale te se provjerava postoji li barem u jednom od tih nizova slijed od četiri ista novčića. Provjeravanje je slijeda trivijalno. Koristeći petlju i brojač provjerava se postoji li slijed od četiri uzastopna. Kôd provjere prikazan je na ispisu 13.

```
const CAP = 4;

const getIsWinFromOrderedArray = (array, player) => {
 if (array.length < CAP) {
   return false;
 }

 let counter = 0;
 let isWin = false;

 array.forEach((cell) => {
   if (cell === player) {
     counter++;

     if (counter === CAP) {
       isWin = true;

       return;
     }
   } else {
     counter = 0;
   }
 });

 return isWin;
};
```

Ispis 13 Provjeravanje slijeda novčića.

Generiranje nizova za stupce i retke je također trivijalno uzimajući u obzir da se radi o dvodimenzionalnom nizu, ali priča je zanimljivija kad su u pitanju dijagonale. Postoje dvije, uzlazna i silazna. Uzlazna se kreira tako da se krene od koordinata ubačenog novčića. Iteriranjem tako da se smanjuju indeksi koordinata dobiva se podatak o polju koje se nalazi na istoj dijagonali kao ubačeni novčić koje je u najdonjoj lijevoj točki mogućoj za tu dijagonalu. Potom se ponovno iterira po ploči, ovaj put uzlazno inkrementirajući koordinate. Time se generira niz dijagonale. Ispis 14 sadrži kôd za provjeru pobjede na uzlaznoj dijagonali. Za provjeru pobjede na silaznoj dijagonali potrebno je obrnuti smjer prema kojim se krajevima dijagonale iterira.

```
const getIsAscendingDiagonalWin = (board, columnIndex, rowIndex, player) => {
 let current = {
   column: columnIndex,
   cell: rowIndex,
 };

 while (board[current.column - 1]?.[current.cell - 1] !== undefined) {
   current.column--;
   current.cell--;
 }

 const diagonal = [];

 while (board[current.column]?.[current.cell] !== undefined) {
   diagonal.push(board[current.column][current.cell]);

   current.column++;
   current.cell++;
 }

 return getIsWinFromOrderedArray(diagonal, player);
};
```

Ispis 14 Provjera pobjede na uzlaznoj dijagonali.

4.5.5 Prikazivanje rezultata u interaktivnom kvizu

Kako bi doprinijelo uzbudljivosti iščekivanja točnog odgovora na pitanje u interaktivnom kvizu, dodano je animirano prikazivanje pojedinog mogućeg odgovora i imena svih igrača koji su ga odabrali. Za realizaciju takve animacije prati se indeks trenutno fokusiranog odabira po nizu koji sadrži informacije o odgovorima i igračima koji su ih odabrali. Taj se podatak šalje komponenti oznake odabira, *AnswerTag* kao zastavica koja komponenti označava da je trenutak da započne animaciju. Komponenta *AnswerTag* može imati jedno od tri stanja. Inicijalno stanje prije animacije. Stanje u kojem se u modalnom prozoru prikazu imena igrača koji su odabrali taj odgovor. Krajnje stanje u kojem je modalni prozor zatvoren, ali uz naziv odgovora prikazane su oznake koje označavaju količinu igrača koji su odabrali taj odgovor. Ispis 15 prikazuje slušanje promjene vrijednosti zastavice za izmjenu stanja aplikacije i definiranje vremena koliko je potrebno sačekati da se sva imena igrača prikažu te promjena stanja u krajnje.

```
const disappearsIn =
PLAYER_DELAY * (playerIdsSelected?.length || 0) + INITIAL_DELAY;

useEffect(() => {
const handleDisplays = async () => {
  if (shouldStartDisplayingResults) {
    await wait(INITIAL_DELAY);
    setEvent(EVENT.focusedResults);

    await wait(disappearsIn + INITIAL_DELAY / 2);
    setEvent(EVENT.valueAndResults);
  }
};

handleDisplays();
}, [shouldStartDisplayingResults, playerIdsSelected, disappearsIn]);
```

Ispis 15 Upravljanje stanjem prikazivanja rezultata.

Za samu animaciju korišteno je CSS svojstvo *animation* kojemu se odgoda i trajanje prosljeđuje korištenjem biblioteke Styled components. Na ispisu 16 nalazi se ispis modalnog prozora koji nestaje nakon *disappearsIn* milisekundi. Animacija za prikazivanje imena igrača ima odgodu početka definiranu konstantom *PLAYER_DELAY* koja se množi s rednim brojem igrača kako bi se oni prikazivali jedan po jedan.

```
if (event === EVENT.focusedResults) {
  return (
    <AnswerTagContainerAbsolute disappearsIn={disappearsIn}>
      <AnswerTagValue>{value}</AnswerTagValue>
      <AnswerTagPlayersContainer>
        {playerNamesSelected.map((player, index) => (
          <AnswerTagPlayerTag
            key={player.socketId}
            delay={PLAYER_DELAY * (index + 1)}>
            {player.name}
          </AnswerTagPlayerTag>
        ))}
      </AnswerTagPlayersContainer>
    </AnswerTagContainerAbsolute>
  );
}
```

Ispis 16 Ispis modalnog prozora i prosljeđivanje parametara animacije.

4.5.6 Stiliziranje glavnog dugmeta

Glavno dugme u aplikaciji stilizirano je tako da izgleda što sličnije pravom trodimenzionalnom dugmetu, a prelaskom mišem preko njega daje dojam da je zapravo pritisnut u prostoru. Kako bi se dobio takav rezultat, potrebno je bilo koristiti prozirnost boja i transformacijske CSS funkcije. Kako se prozirnosti boja ne bi neželjeno miješale nije korišteno svojstvo *opacity*, već je korištena *alpha* vrijednost prilikom definiranja boja teksta i pozadine. Kada je riječ o heksadecimalnom zapisu boje, *alpha* vrijednost očitava se iz sedme i osme znamenke zapisa te se u stvari preslikava *rgba* zapis boje u heksadecimalni zapis. Vrijednosti crvene, zelene i plave boje idu od nula do dvjesto pedeset i pet, dok *alpha* vrijednost ide od nula do jedan. U heksadecimalnom se zapisu *alpha* vrijednost definira uzimajući postotak. Dugme se sastoji od dva elementa. Jedan definira dimenzije i sjenu, a na drugome je prikazana labela i on je realiziran kao apsolutno pozicionirani *after* element. Prilikom prelaženja mišem preko dugmeta mijenja se *transform* svojstvo apsolutno pozicioniranog elementa. Za animaciju transformiranja odabrana je funkcija *translate3d* kako bi se element prebacio u kompozitni sloj *(engl. composite layer)* procesa ispisivanja u svrhu optimizacije. Kompozitni sloj zadnji je korak u procesu ispisivanja elemenata što znači da ako se tu dogodi promjena, samo će se taj korak ponoviti što rezultira optimizacijom animacija. Kao i sa svim optimizacijama potrebno je pripaziti da se ne primjenjuju svugdje gdje je moguće, nego točno tamo gdje treba jer se inače prouzrokuje kontra efekt. [11] U ispisu 17 prikazana je stilizirana komponenta glavnog dugmeta.

```
export const ButtonPrimary = styled.button`
 margin: 4px;
 background-color: #635e5e99;
 color: transparent;
 padding: 0 16px;
 font-size: 44px;
 position: relative;
 border-radius: 10px;

 &:hover {
   background-color: #635e5eb3;

   &::after {
     background-color: #ee5d6cb3;
     color: #282856b3;
     transform: translate3d(-2px, 2px, 0);
   }
 }

 &::after {
   ${({ content }) =>
     content &&
     css`
       content: '${content}';
     `}
   transition: background 0.2s ease-in-out, transform 0.2s ease-in-out, color
0.2s ease-in-out;
   position: absolute;
   background-color: #ee5d6c99;
   color: #28285699;
   width: 100%;
   height: 100%;
   bottom: 6px;
   left: 6px;
   border-radius: 10px;
 }
`;
```

Ispis 17 Komponenta stiliziranog dugmeta.

5 Zaključak

Tijekom razvoja aplikacije jedan od najvećih izazova predstavljalo je strukturiranje kôda za razmjenu podataka putem utičnica. Kako je za realizaciju igara potrebno konstantno izmjenjivati podatke popraćene kompleksnom logikom, ključan je dio bio osmisliti održivi sustav imenovanja događaja i razdvajanja kôda. Kako se web utičnice češće koriste za manje funkcionalnosti aplikacija, primjera i materijala za izradu ovakvih arhitektura ima jako malo. Retrospektivno gledajući, održivost se mogla poboljšati koristeći čvrsto tipkani *(engl. strongly typed)* jezik poput TypeScripta koji ne bi utjecao na tehnologije, ali bi olakšao daljnji rad time što bi podaci koji prolaze kroz utičnice bili striktno definiranih tipova. Najveći je izazov ipak bio testiranje aplikacije i svih funkcionalnosti, pogotovo prilikom izrade kviza zato što je bilo potrebno simulirati više uređaja u isto vrijeme. To je donekle riješeno konfiguracijskim varijablama i simulacijom podataka igrača, no pravo je testiranje bilo nužno odraditi na stvarnim klijentima.

Aplikacija ima dosta prostora za napredak i razvoj. Primjerice, uvođenje novih igara gdje su mogućnosti beskonačne. Uvođenje korisničkih računa i rang liste personaliziralo bi igre, a uz to i poticalo kompetitivnost igrača. Ukoliko bi postojali korisnički računi, utoliko bi bilo moguće pružiti korisnicima funkcionalnost da sami dodaju pitanja za kviz i tako personaliziraju iskustvo.

Literatura

[1] S interneta, https://socket.io/docs/v4 (posjećeno 14. rujna 2022.)

[2] Thoma M: „Raw SQL vs Query Builder vs ORM“, s interneta, https://levelup.gitconnected.com/raw-sql-vs-query-builder-vs-orm-eee72dbdd275 (posjećeno 14. rujna 2022.)

[3] S interneta, https://reactjs.org/docs/faq-internals.html (posjećeno 14. rujna 2022.)

[4] GeeksForGeeks: „ReactJS Data Binding“, s interneta, https://www.geeksforgeeks.org/reactjs-data-binding/ (posjećeno 14. rujna 2022.)

[5] S interneta, https://knexjs.org/guide/migrations.html (posjećeno 14. rujna 2022.)

[6] Gupta M: „JavaScript Function Chaining”, s interneta, https://medium.com/technofunnel/javascript-function-chaining-8b2fbef76f7f (posjećeno 14. rujna 2022.)

[7] Hallie L, Osmani A: „Patterns.dev”, s interneta, https://www.patterns.dev/posts/provider-pattern/ (posjećeno 14. rujna 2022.)

[8] GeeksForGeeks: „What’s the difference between useContext and Redux ?”, s interneta, https://www.geeksforgeeks.org/whats-the-difference-between-usecontext-and-redux/ (posjećeno 14. rujna 2022.)

[9] S interneta, https://reactjs.org/docs/hooks-effect.html (posjećeno 14. rujna 2022.)

[10] S interneta, https://redux.js.org/tutorials/fundamentals/part-2-concepts-data-flow#reducers (posjećeno 14. rujna 2022.)

[11] Moreno González M: „Layers, layers, layers… Be careful!”, s interneta, https://medium.com/masmovil-engineering/layers-layers-layers-be-careful-6838d59c07fa (posjećeno 14. rujna 2022.)