Web aplikacije u realnom vremenu

Levačić, Filip

SVEUČILIŠTE U ZAGREBU

FAKULTET ORGANIZACIJE I INFORMATIKE

V A R A Ž D I N

Filip Levačić

Web aplikacije u realnom vremenu

DIPLOMSKI RAD

Varaždin, 2019.

SVEUČILIŠTE U ZAGREBU

FAKULTET ORGANIZACIJE I INFORMATIKE

V A R A Ž D I N

Filip Levačić

Matični broj: 46355/17-R

Studij: Informacijsko i programsko inženjerstvo

Web aplikacije u realnom vremenu

DIPLOMSKI RAD

Mentor:

Doc. dr. sc. Darko Andročec

Varaždin, srpanj 2019.

Filip Levačić

Izjava o izvornosti

Izjavljujem da je moj završni/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 radu će biti predstavljene Web aplikacije u realnom vremenu. Rad započinje teorijom oko samih aplikacija ove vrste, njihove povijesti, razloga njihovog nastanka te mogućih primjena. Proći će se kroz sve važnije tehnologije koje su se koristile kroz povijest ovih aplikacija, prikazati kratka demonstracija kroz programski primjer te dati komentar na njihovu relevantnost u trenutku pisanja rada. Obradit će se nekoliko različitih biblioteka u raznim programskim jezicima koje se koriste kod izrade ovakvih aplikacija, objasniti prednosti, mane i razloge korištenja određene biblioteke, prikazati kraći programski primjer u obliku jednostavne *chat* aplikacije te dati komentar na njihovu relevantnost u trenutku pisanja rada. Zadnji programski primjer je veći i izrađen korištenjem biblioteke SignalR. Primjer aplikacije je aplikacija za praćenje sportskih rezultata, te će prikazati mogućnosti biblioteke SignalR kao jedne od češće korištenih biblioteka u izradi ove vrste aplikacija, kao i mogućnosti samih Web aplikacija u realnom vremenu. Nakon završnog primjera slijedit će završni komentar i zaključak teme.

Ključne riječi: Web aplikacija, realno vrijeme, AJAX, polling, WebSocket, SignalR

Sadržaj

1. Uvod

Web je u svojoj ranoj fazi postojanja bio vrlo različit od onoga što danas poznajemo, te je zamišljen za rješavanje specifičnog problema dijeljenja dokumenata između udaljenih računala. Inženjeri koji su ga razvijali nisu mogli znati u što će se njihova tvorevina pretvoriti u budućnosti te nisu mogli računati na sve moguće slučajeve korištenja takve računalne mreže, što dovodi do toga da su se Internet i Web morali kontinuirano proširivati raznim dodacima da bi se osigurale mogućnosti koje je poželjelo mnoštvo novih korisnika. Neke od poznatijih „rupa“ se tiču sigurnosti same mreže, tj. njezinim nedostatkom, koji se rješava još i danas, pa do raznih pokušaja dodavanja interaktivnosti pojedinačnim Web stranicama koje su u početku predstavljale samo statične dokumente. Mogućnost Web-a koja će se obrađivati u ovom radu su Web aplikacije koje rade u realnom vremenu.

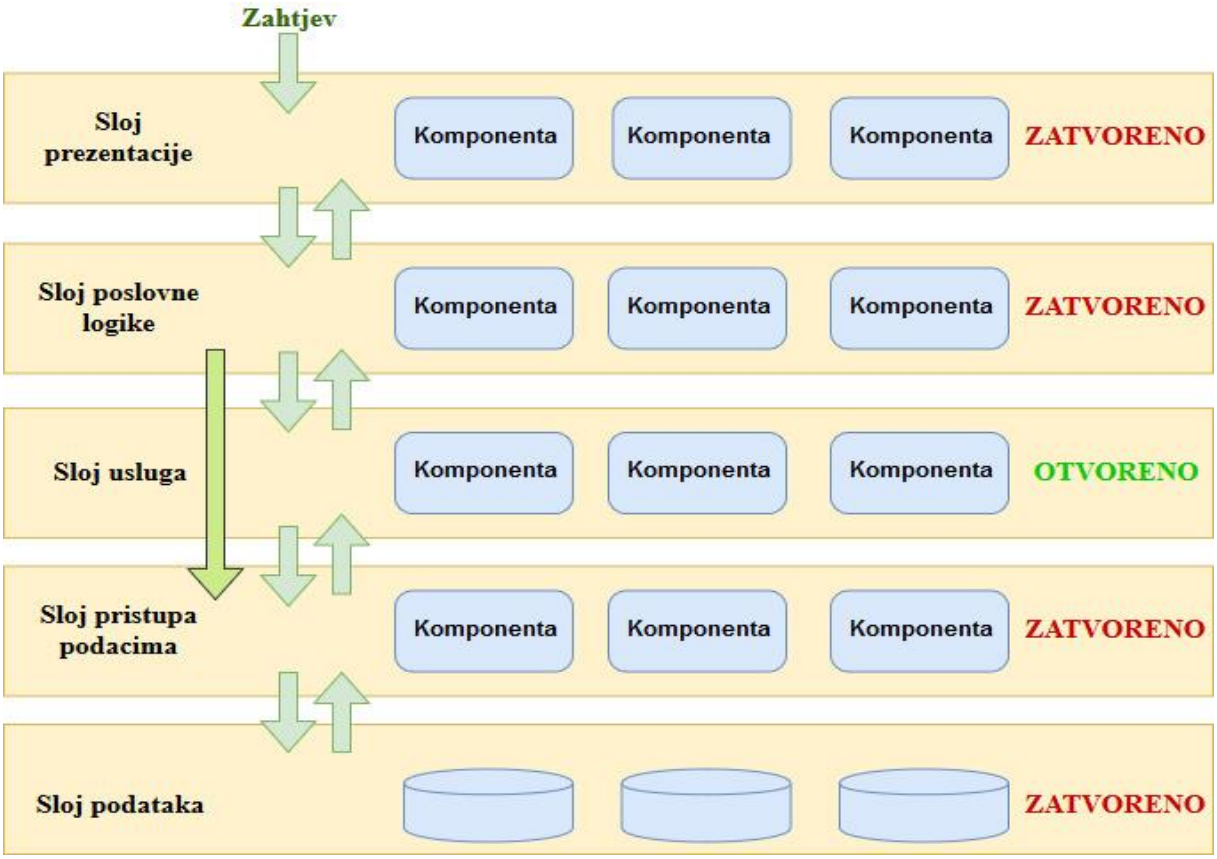
Jedan od nedostataka Web-a bio je taj da je Web stranicu bilo potrebno ponovno učitati kako bi ona dobila nove podatke od poslužitelja. Mogućnost dobivanja podataka bez ponovnog učitavanja stranice otvorila je velik broj primjena koje su drastično promijenile način na koji koristimo Web. E-mail se odjednom činio spor u usporedbi sa aplikacijama za čavrljanje, a mogućnost kontinuiranog prikazivanja novih podataka značila je velike promjene u aplikacijama koje prikazuju vremenske podatke, financijske podatke sa burza, podatke o stanju nadziranih sustava, itd. Danas je ovaj način rada Web aplikacija opće prihvaćen te je korišten od strane velikog broja Web stranica, od *chat* funkcionalnosti na društvenim mrežama do pretraživanja u realnom vremenu na Web tražilicama. Razvoj Web aplikacija u realnom vremenu počeo je vrlo sporo jer je bio ograničen od strane pretraživača Interneta, njegovom infrastrukturom i drugim faktorima. Dok su se u početku koristile određene tehnike i trikovi kako bi se dobila željena trenutačna komunikacija zbog potrebe zaobilaženja nedostataka mreže ili pretraživača Interneta, danas postoje standardizirane tehnologije koje podržavaju svi popularniji pretraživači Interneta, te se koriste u velikom broju javno dostupnih biblioteka i alata za razvoj Web aplikacija. Infrastruktura Interneta također je napredovala te danas vrlo rijetko stoji kao prepreka u razvoju ovakvih vrsta aplikacija.

U ovom radu objasnit će se pojam Web aplikacija u realnom vremenu, te će se pojasniti sve važnije tehnologije koje su se kroz povijest Interneta koristile kako bi se takve aplikacije mogle razvijati. Svaka od tehnologija bit će popraćena kraćim programskim primjerom u svrhu boljem razumijevanja određene tehnologije. Nadalje, objasnit će se neke važnije biblioteke napisane u nekoliko programskih jezika koje se koriste u razvoju ovakvih aplikacija, te će za njih biti prikazan manji primjer. Najveći dio praktičnog dijela rada zauzimat će veći primjer aplikacije za praćenje sportskih rezultata izrađen sa bibliotekom SignalR.

2. Arhitektura Web aplikacija

Prije nego što možemo započeti proučavati Web aplikacije u realnom vremenu, potrebno je reći nešto o arhitekturi Web aplikacija. Danas postoji mnogo arhitekturalnih uzoraka koji se koriste u izgradnji aplikacija, a svaka od njih zamišljena je sa određenim zahtjevima aplikacije koja se u tom trenutku razvijala te ograničenjima na koja su nailazili razvojni inženjeri koji su osmislili arhitekturu, bilo da se radilo o ograničenjima infrastrukture ili alata koje su koristili. Web aplikacije prirodno teže prema određenim uzorcima zbog vlastitih zahtjeva ili ograničenja, primjerice česte logičke i/ili fizičke razdvojenosti određenih dijelova aplikacije, ali nove tehnologije omogućuju veću fleksibilnost u odabiru arhitekture koja se koristi u razvoju Web aplikacija. Važan korak u izradi svake aplikacije je odabir ili čak dizajn nove arhitekture ukoliko je to potrebno jer će ta odluka utjecati na daljnji razvoj aplikacije.

Najčešće korištena arhitektura naziva se višeslojna ili n-slojna arhitektura. Kao što njeno ime ukazuje, ona razdvaja aplikaciju na više cjelina ili slojeva, ovisno o potrebi aplikacije. Svaki sloj idealno predstavlja potpuno razdvojenu cjelinu koja prima poruke od jednog sloja, transformira podatke te prenosi poruke drugom sloju, bez znanja o internoj poslovnoj logici tih slojeva. Poruke se prenose na standardiziran način kako bi se povezivanje slojeva umanjilo što je to više moguće. Takvo odvajanje slojeva omogućuje nezavisan rad na različitim slojevima u isto vrijeme, tj. promjene na jednom sloju ne utječu na rad drugog sloja ukoliko sama komunikacija između njih ostaje netaknuta. Za različite slojeve mogu se koristiti potpuno različite tehnologije koje bolje podupiru funkciju samog sloja tako da se svaki sloj izgradi na najbolji mogući način. Broj samih slojeva je varijabilan te se slojevi dodaju i oduzimaju u fazi izrade proizvoda prema potrebama tog proizvoda, a i samom komunikacijom između slojeva može se manipulirati na više načina osim standardne promjene formata poruka koje se razmjenjuju. U klasičnom i idealnom modelu višeslojne arhitekture, svaki sloj je zatvoren, što znači da razmjenjuje poruke samo sa slojevima iznad i ispod sebe, no u praksi to nije uvijek idealno rješenje. Često se događa situacija da sloj sa podacima koje dobije ne radi ništa, već ih samo proslijedi sljedećem sloju. Ovdje se gubi na brzini i jednostavnosti komunikacije, pa je neki sloj moguće implementirati kao otvoreni sloj. Otvoreni sloj je prema potrebi moguće preskočiti, tj. sloj iznad otvorenog sloja ne šalje poruke tom sloju ukoliko zna da ih on neće mijenjati, već ih direktno šalje sloju koji se nalazi nakon otvorenog sloja. Ukoliko je sloj zatvoren, njega je nemoguće preskočiti te predstavlja sloj koji radi ključne promjene nad podacima koje se ne smiju preskočiti. Na sljedećoj slici prikazan je primjer modela višeslojne arhitekture koji se može susresti u praksi, a sadrži dovoljno elemenata za objašnjenje cjelokupne arhitekture (Microsoft Docs (Azure), 2017).



Slika 1: Višeslojna arhitektura s otvorenim slojem (Richards, 2015)

Ovaj model višeslojne arhitekture sastoji se od ukupno pet slojeva. Svaki od tih slojeva prima poruku od sloja iznad sebe te prenosi poruku sloju ispod sebe, sa posebnim slučajem kod otvorenog sloja. Svaki sloj sastoji se od više komponenti. Te komponente mogu predstavljati zasebne tehnologije, manje aplikacije, interne baze podataka, servise, itd. U ovom modelu to nije važno jer pratimo samo komunikaciju između slojeva i njihov zajednički rad. Prvi sloj u nizu je sloj prezentacije. To je sloj koji je izravno vidljiv korisnicima aplikacije te koji služi za komunikaciju sa korisnikom, tj. prikaz informacija korisniku na njemu razumljiv način te prihvaćanje korisnikovog unosa i naredbi koji se zatim prosljeđuju idućem sloju. Sloj prezentacije prima korisnikov zahtjev koji može imati oblik poslane tekstualne naredbe, klika na gumb, glasovne naredbe, itd. Ovaj sloj sastoji se od tehnologija i logike koji taj zahtjev pretvaraju u oblik razumljiv idućem sloju te ga njemu i prenose. Sljedeći sloj je sloj poslovne logike. Ovdje se nalazi veći dio poslovne logike aplikacije te on zna kako reagirati na koju naredbu. Odluku o tome koji oblik će poruka imati te kojim će se komunikacijskim kanalom ona slati mora donijeti razvojni inženjer. Također, mora se donijeti odluka o raspodjeli određenih poslova između slojeva. Npr., validacija korisnikovog unosa može se obraditi direktno u sloju prezentacije, ali može se obaviti i u sloju poslovne logike koji zatim odgovarajuću poruku vraća sloju prezentacije koji pak je u prikladnom obliku prikazuje korisniku.

Nakon što sloj poslovne logike obavi svoj dio posla, dolazimo do još jedne odluke. Sljedeći sloj, sloj usluga, je otvoren sloj te se može preskočiti, pa je potrebno znati u kojim slučajevima se on koristi. U ovom slučaju on može predstavljati uslugu transformacije podataka u određeni format koji je potreban sljedećem sloju, sloju pristupa podacima. Ukoliko sloj poslovne logike kao poruku šalje jednostavan podatak koji ne može imati više oblika, sloj usluga moguće je preskočiti te poruku poslati izravno sloju pristupa podacima, jer bi u suprotnom sloj usluga poruku samo prenosio, bez transformacije, zbog čega se gubi na optimiziranosti aplikacije. Ukoliko pak je podatak složeniji, npr. ako je to datum koji može imati više formata, sloj poslovne logike može poslati taj podatak sloju usluge koji će ga pretvoriti u pravilan format nakon čega će poruku proslijediti sloju pristupa podacima. Naravno, ovo nije jedino rješenje, te je sloj usluga moguće uklopiti u bilo koji od slojeva sa kojima on komunicira. Odvajanjem određenih usluga u zaseban sloj dobijemo veću fragmentiranost sustava što može biti i dobro i loše, ovisno o kontekstu. Učahurivanje zasebnih funkcionalnosti u posebne cjeline razbija monolitne aplikacije te olakšava nadogradnju i održavanje sustava jer se taj sloj može mijenjati nezavisno o drugim slojevima, ali se kroz povećanje broja slojeva usporava komunikacija, obrada zahtjeva te ukupna brzina rada aplikacije.

Spomenuti sloj pristupa podacima usko je povezan sa slojem podataka. Sloj podataka velikom većinom predstavlja bazu podataka koje aplikacija koristi. Implementacija baze podataka može biti potpuno nezavisna od ostatka aplikacije kao i svaki drugi sloj, ali su ponekad potrebni dodatni koraci kako bi se aplikacija učinila boljom i sigurnijom, a za to služe funkcionalnosti sloja pristupa podacima. Kao česti primjer tehnologija ovog sloja spominju se ORM (eng. *Object Relational Mapping*) rješenja koja pomažu u komunikaciji objektno-orijentiranih programskih rješenja sa relacijskim bazama podataka, ali ovdje se mogu smjestiti sve tehnologije koje olakšavaju komunikaciju između baze podataka i poslovne logike aplikacije. Ovdje se mogu svrstati i druge funkcionalnosti poput validacije ulaznih podataka kao zaštite protiv napada poput *SQL injection*-a i slično. Opet napominjem da je sve funkcionalnosti sloja pristupa podataka moguće smjestiti u sloj poslovne logike, što može imati i prednosti i nedostatke koje je potrebno pomno razmotriti u fazi planiranja aplikacije.

Ovo je model arhitekture sa pet slojeva, ali moguće je po potrebi dodavati i oduzimati slojeve po želji, tj. prema potrebama aplikacije. Prednosti i nedostaci mogu se naći i kod arhitektura sa malo i arhitektura sa puno slojeva, pa je potrebno naći neku ravnotežu koja odgovara aplikaciji koju gradimo. Ovdje je potrebno imati na umu i više od samog modela, jer i drugi faktori mogu znatno utjecati na uspjeh višeslojne aplikacije. Najočitiji faktor je brzina komunikacije između slojeva. Različiti slojevi su logički odvojeni, ali oni mogu biti i fizički odvojeni. Ovdje se kao primjer često navode *terminal* aplikacije koje su predstavljale samo sloj prezentacije te komunicirale sa fizički udaljenim poslužiteljem kod obrade zahtjeva. Ovdje

počinje i razmatranje Web aplikacija kao višeslojnih aplikacija. Sama priroda Web aplikacije, gdje klijentska aplikacija šalje zahtjev poslužitelju koji zatim vraća odgovor, razdvojena je, tako da je daljnje razdvajanje u slojeve očiti daljnji korak.

Općenito se kod izrade aplikacija, pa tako i Web aplikacija, najčešće koristi troslojna arhitektura koja se sastoji od sloja prezentacije, sloja poslovne logike te sloja podataka. Prije pojašnjen model arhitekture sa pet slojeva zapravo je proizašao iz troslojne arhitekture odvajanjem određenih funkcionalnosti iz sloja poslovne logike u zasebne slojeve. Kod aplikacija koje ne koriste mrežnu komunikaciju slojevi su samo logički odvojeni u samom kodu aplikacije te nemaju nikakav vidljiv učinak na korisničko iskustvo, ali kod Web aplikacija slojevi su često i fizički odvojeni pa aplikacije ovise i o brzini mreže kojom slojevi komuniciraju. Određeni slojevi mogu se nalaziti na istoj fizičkoj lokaciji kako bi se komunikacija ubrzala. Ovdje se najčešće koriste pojmovi tankog i debelog klijenta. U slučaju arhitekture tankog klijenta, samo sloj prezentacije nalazi se na strani klijenta, dok se sloj poslovne logike i sloj podataka nalazi na strani poslužitelja. U slučaju debelog klijenta, sloj prezentacije i sloj poslovne logike nalaze se na strani klijenta, dok se na strani poslužitelja nalazi samo sloj podataka. Naravno, svaki sloj moguće je dodatno podijeliti i nove slojeve rasporediti na stranu koja najbolje odgovara zahtjevima aplikacije. Sloj usluga se često implementira u obliku zasebnih Web servisa koji se nalaze u posebnom logičkom sloju i fizičkoj lokaciji, te je pristup ovom sloju omogućen većem broju aplikacija što olakšava održivost sustava.

Oba modela imaju svoje prednosti i nedostatke, te je danas najčešće zastupljena neka ravnoteža između te dvije ideje. Arhitektura tankog klijenta znači puno manje komuniciranja sa poslužiteljem što pomaže u brzini izvođenja aplikacije, te je prikladna ukoliko se sva ili barem veći dio poslovne logike može smjestiti na klijentsku stranu bez potencijalne štete. Prvi problem na koji razvojni inženjeri ovdje nailaze je teško ažuriranje aplikacije. Kada je potrebno promijeniti poslovnu logiku koja se nalazi na klijentskoj strani, potrebno je ažurirati svaku instaliranu aplikaciju što često dovodi do prekida rada aplikacije za vrijeme ažuriranja. Ukoliko se poslovna logika koja se mijenja nalazi na poslužiteljskoj strani, dovoljno je ažurirati promjene na svim poslužiteljima koji komuniciraju sa klijentima, a njih će uvijek biti znatno manje od broja klijenata. Drugi problem je sigurnost aplikacije. Kritični podaci u različitim arhitekturama prolaze kroz različite kanale i razmjenjuju se na različite načine što ih dovodi u opasnost od određenih vrsta napada. Nijedna arhitektura ovdje nema preveliku prednost jer je napad moguće izvesti i na klijentskoj i na poslužiteljskoj strani, kao i na komunikacijski kanal između njih. Veći problem je čuvanje poslovne tajne, tj. procesa sadržanih u sloju poslovne logike koji se žele zadržati u vlasništvu graditelja aplikacije, a njih je puno lakše dobiti reverznim inženjerstvom ukoliko se oni nalaze na klijentskoj strani (Ivankov, 2018).

Web tehnologije razvijaju se relativno kratko, od početka nastanka Web-a, ali one se već sada mogu dovoljno lako podijeliti prema slojevima Web aplikacije kojoj pripadaju. Svaka od tih grupa tehnologija danas je dovoljno velika da možemo izabrati onu koja najbolje odgovara onome što želimo postići. U slučaju sloja prezentacije, najvažnije je spomenuti HTML i CSS tehnologije koje su temelj Web-a i Web aplikacija te ih je nemoguće zaobići. To su opće prihvaćene tehnologije koje se svakodnevno razvijaju i oko kojih se zatim razvijaju sve ostale tehnologije Web-a. Mnoge biblioteke i razvojni okviri olakšavaju korištenje ovih tehnologija te proširuju naizgled vrlo male mogućnosti koje one nude. Također, JavaScript skriptni jezik je vrlo važna tehnologija klijentske strane koja se uglavnom koristi u domeni sloja prezentacije, ali može preuzeti i određene zadaće sloja poslovne logike. JavaScript je jedna od najstarijih opće prihvaćenih tehnologija Web-a te je služio u izgradnji mnogih često korištenih tehnologija poput jQuery-a, Node.js, itd. ECMAScript specifikacija za skriptne jezike izrađena je radi standardizacije JavaScript-a, ali podržava i razvoj drugih implementacija poput TypeScript-a.

Tehnologije sloja poslovne logike uglavnom se odnose na poslužiteljske tehnologije i programske jezike. Najpoznatije rješenje je programski jezik PHP, ali danas postoje mnoga rješenja u raznim programskim jezicima koja rješavaju problem izvršavanja poslovne logike na poslužitelju. Ti programski jezici nude i rješenja za sloj prezentacije tako da se ti slojevi često isprepliću, što ne znači nužno da će aplikacija biti loša. Rješenje je potrebno odabrati i prema njihovoj internoj arhitekturi, npr. da li želimo da sloj prezentacije i sloj poslovne logike komuniciraju izravnim porukama kao kod MVC arhitekture ili *binding*-zima kao što se to radi u MVVM arhitekturi. Mnogi poznati razvojni okviri poput React-a i Angular-a daju mogućnosti za razvoj klijentske strane, tzv. *front-end*-a, te dodavanje poslovne logike na poslužiteljskoj strani, tzv. *back-end*-u. Svi ovi razvojni okviri teže lakšem i bržem razvoju Web aplikacija, što dovodi do zanimljivih rješenja poput Node.js, koji omogućava izvršavanje JavaScript skripti kao rješenja za poslovnu logiku na poslužiteljskoj strani. Sloj podataka uglavnom predstavlja neku bazu podataka kojoj nije važno da li radi na Web-u ili ne, ali je potrebno prilagoditi sloj pristupa podacima (ili pripadajuću logiku) da radi sa ostatkom aplikacije. Zbog velikog izbora tehnologija, današnja poduzeća često će odabirati između onih koje njihovi razvojni inženjeri najbolje poznaju, a ne nužno onih koje su najprikladnije za rješavanje problema.

Kod razvoja Web aplikacija koje rade u realnom vremenu, vrlo je važno na umu imati cjelokupnu arhitekturu sustava koju treba razviti sa ciljem izrade takve aplikacije, tj. odabrati arhitekturu koja će raditi zajedno sa rješenjem za komunikaciju u realnom vremenu, a ne protiv nje. Znanje o tome koje podatke je potrebno slati i u kojem trenutku može značiti puno uštođenog vremena u komunikaciji između klijenta i poslužitelja što može značiti granicu između onog što shvaćamo kao komunikaciju u realnom vremenu i onog što se čini kao klasična, spora Web stranica.

3. Aplikacije u realnom vremenu

Prije nego se može početi pričati o Web aplikacijama u realnom vremenu, potrebno je pojasniti teoriju i pojmove računanja u realnom vremenu, kao i samog realnog vremena. Realno vrijeme (eng. *real time*), ponekad prevedeno i kao realistično ili potrebno vrijeme, najlakše se može objasniti kao najduže vrijeme potrebno za izvršenje zadatka u kojem bismo mogli reći da cjelokupni sustav radi prema zadanim specifikacijama. Ovaj pojam često se, donekle pogrešno, koristi kao vrijeme u kojem zadatak mora biti izvršen kako bi se ljudima činilo da je izvršen odmah, uglavnom zbog velike količine aplikacija kojima danas to i je realno vrijeme. Realno vrijeme može biti bilo koje trajanje, ovisno o potrebni aplikacije. Npr., meteorološka stanica u sklopu bove može imati ugrađenu aplikaciju koja ima zadatak izmjeriti određene parametre lokalnih vremenskih prilika te ih poslati u meteorološku stanicu. Takva aplikacija može imati realno vrijeme od nekoliko sekundi u slučaju da je potrebno strogo paziti na vremenske uvjete određenog područja, ili pak nekoliko minuta ukoliko učestalo praćenje nije potrebno. S druge strane, aplikacija ugrađena u obrambene sustave koji reagiraju na neprijateljske projektile mora imati iznimno kratko realno vrijeme kako bi ona bila upotrebljiva.

Računanje u realnom vremenu (eng. *real time computing*) opisuje sustave koji se temelje na opremi i aplikacijama koji imaju zadano realno vrijeme. Takve sustave karakteriziraju tri komponente i njihova međuovisnost (Shin & Ramanathan, 1994):

- Vrijeme izvršavanja
- Pouzdanost
- Okolina sustava

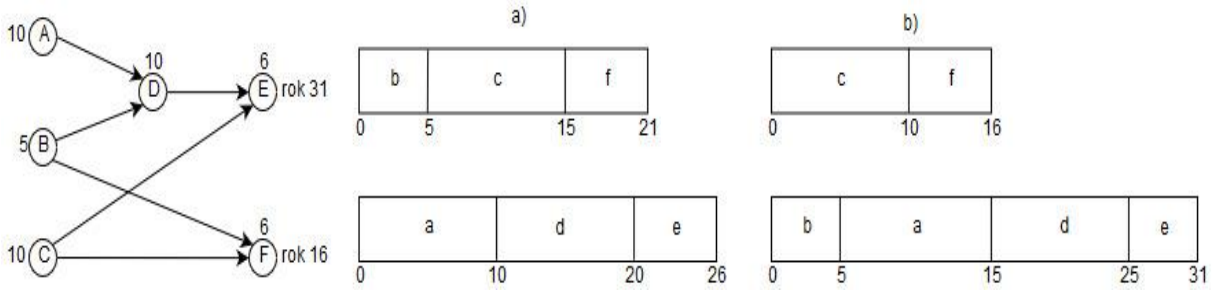
Vrijeme je najvažniji resurs sustava u realnom vremenu, jer sustav ne ovisi samo o točnosti poruka i njihovog sadržaja koji se prenosi kroz sustav, već i da li su te poruke prenesene na vrijeme. Svaki zadatak u sustavu ima svoje realno vrijeme čiji kraj označava rok (eng. *deadline*) tog zadatka. Smatra se da zadatak nije obavljen uspješno ukoliko prenese pogrešnu vrijednost ili ako ne prenese vrijednost do svog zadanog roka, bez obzira na uspješno obavljanje same logike zadatka. Pouzdanost se također veže na vrijeme izvršavanja jer svaki dio sustava pretpostavlja da će drugi dijelovi sustava izvršiti svoj zadatak točno i na vrijeme, jer ako oni to ne učine to može rezultirati pogrešnim radom sustava ili čak njegovim gašenjem, što pak može dovesti do velikih financijskih gubitaka, kao i gubitaka ljudskih života kod određenih sustava. Okolina sustava označava sve što okružuje sustav, tj. vanjski dijelovi kojima se sustav bavi i bez kojih on ne bi imao svrhu. Sustav kontrole leta bio bi besmislen bez zrakoplova, kao i sustav za upravljanje zrakoplovom, te se zbog toga zrakoplov smatra dijelom tog sustava. Više komponenti može činiti okolinu sustava, npr. sustav kontrole leta

također je ovisan i o drugim sustavima poput onih u zračnim lukama, te o ljudima koji sa njima rade i koji generiraju nove ulazne podatke na temelju onih dobivenih iz sustava.

Svaki sustav u realnom vremenu sastoji se od nekoliko zadataka koji obavljaju svoju zadaću. Zadaci koji se izvršavaju u određenom vremenskom periodu su periodični zadaci. Oni često imaju svoj rok do kojeg se moraju izvršiti kako bi sustav ostao funkcionalan, a realno vrijeme tog zadatka uglavnom će biti manje od vremenskog razmaka od jednog roka do drugog. Ukoliko bi realno vrijeme bilo veće od tog razmaka, podaci bi se brzo izmiješali i postali gotovo neupotrebljivi jer se gubi pouzdanost sustava. Ti zadaci često prikupljaju podatke sa nekog izvora iz okoline i izvode određene kalkulacije, te na temelju njih mogu odraditi još neku akciju. Npr. termostat mjeri temperaturu svake minute, te ako je ona veća ili manja od postavljene željene temperature (koju je postavio čovjek kao dio okoline), on automatski mijenja postavke klima uređaja kako bi se održala željena temperatura. Sustavi sa vrlo kratkim realnim vremenom, poput autopilota u mnogim vozilima, moraju imati vrlo visoku pouzdanost kako bi pravilno funkcionirali, jer bi u suprotnom bili ugroženi ljudski životi. Zadaci koji se ne izvršavaju u određenom vremenskom periodu nazivaju se aperiodični zadaci. Oni se izvršavaju na zahtjev korisnika, npr. kod ručnog dobavljanja stanja sustava, ili kao odgovor na neki događaj poput kvara sustava kada se može dobaviti stanje sustava izvan predviđenog vremena i na temelju njega odraditi korektivne akcije. Oni mogu također imati realno vrijeme i svoj rok iako je to rjeđe viđeno nego kod periodičnih zadataka, te tada moraju izvršiti svoj zadatak što prije mogu jer drugi dijelovi sustava svejedno ovise jedni o drugima.

Rokovi zadataka u realnom vremenu se dijele na nekoliko kategorija s obzirom na posljedice njihovog nepridržavanja. Najvažniji rokovi su čvrsti (eng. *hard*) rokovi. Ukoliko zadatak ima čvrst rok te on svoj rok propusti prije nego izvrši svoj zadatak, posljedice za sustav mogu biti katastrofalne, što može značiti veliku štetu s obzirom na domenu sustava o kojem se govori. Periodični zadaci često se nalaze u ovoj kategoriji jer opskrbljuju ostatak sustava sa podacima koji su potrebni za rad. Sljedeća kategorija rokova su strogi (eng. *firm*) rokovi. Ukoliko zadatak ima strogi rok, propuštanje roka neće uzrokovati nikakvu štetu, ali podaci koje je on poslao ili akcije koje je taj zadatak pokušao poduzeti smatraju se beskorisnima. Zadaci u sustavima sa promjenjivom okolinom često imaju ovakve rokove jer su njihovi rokovi zadani u vremenu za koje se smatra da podaci i odabrane akcije ostaju relevantni u odnosu na stanje okoline, tj. ukoliko rok prođe podaci će već biti zastarjeli, a akcija odabrana na temelju tih podataka neće rezultirati željenim ishodom. Svi ostali rokovi su meki (eng. *soft*) rokovi, te propuštanje tih rokova neće rezultirati većom štetom, ali je i dalje poželjno da se zadaci pridržavaju tih rokova. Aperiodični zadaci često imaju meke rokove jer brz odgovor na mnoge manje važne događaje nema preveliku važnost za sustav u tom trenutku, ali korisnost odgovora se svejedno umanjuje što se duže on ne koristi.

Pouzdanost u ovakvim sustavima je također iznimno važna, ali kolika razina pouzdanosti je potrebna ovisi o vrsti aplikaciji i zadacima. Zadatak se smatra pouzdanim ukoliko uvijek možemo predvidjeti njegov rezultat, tj. za određene ulazne podatke uvijek se dobiju točno određeni izlazni podaci. Za sustav sa više komponenti koje međusobno komuniciraju ovo je iznimno važno, čak i u sustavima čiji zadaci nemaju zadane rokove, jer ako zadatak dobiva ulazne parametre od drugog zadatka, on mora znati što može očekivati. Nepoznati ili nepredviđeni ulazni parametar može dovesti do pomutnje određenog zadatka koji zatim može sa sobom srušiti cijeli sustav. Nadalje, u sustavima u realnom vremenu raspored zadataka također je vrlo bitan. Zbog toga što svaki zadatak može ovisiti o nekom drugom zadatku, pravilan redoslijed izvršavanja zadataka nužan je za optimalno izvođenje funkcije sustava. Nepravilan raspored zadataka može dovesti do velike neiskorištenosti vremena u kojem zadaci čekaju jedni na druge dok su se mogli izvoditi slijedno, te rezultirati višestruko dužim vremenom izvršavanja.



Slika 2: Primjer sustava u realnom vremenu i dvaju rasporeda zadataka tog sustava u kojem će primjer A zadovoljiti rok od 30 jedinica vremena, dok primjer B neće (Shin & Ramanathan, 1994)

Zbog česte fragmentiranosti sustava gdje su komponente fizički razdvojene, mnogi sustavi imaju neki oblik vlastitog digitalnog sata prema kojem se svaki zadatak i komponenta sustava orijentira te sprječava nastanak kaosa ukoliko bi svaki zadatak pratio vlastiti sat na kojem se temelji njegov početak i rok izvršenja. Razdvojenost komponenti sustava donosi još jedan problem, a to je da kod izrade sustava treba u obzir uzeti i vrijeme komunikacije između njegovih komponenti, što dodatno smanjuje vrijeme u kojem zadatak mora biti izvršen jer njegovo realno vrijeme ostaje isto, bez obzira na brzinu prijenosa poruka. Poruka se u razmjeni može i izgubiti, ponekad cijela, a ponekad samo dio, što također treba uzeti u obzir te uvesti mjere koje će sustav štititi od ispada i u tom slučaju. Danas je, zbog brzine i sigurnosti internetskih veza, to puno manji problem, ali nekad su specijalizirani i važni sustavi imali posebno dizajnirane i izrađene komunikacijske kanale kako bi se postigle što veća brzina i pouzdanost prijenosa podataka.

3.1. Razlike Web aplikacija i tradicionalnih sustava u realnom vremenu

Kako ulazimo u područje Web aplikacija u realnom vremenu, ovdje definirani pojmovi često su degradirani u njihovo najčešće korišteno značenje, pa treba držati par stvari na umu. Internet je vrlo brzo postao glavno mjesto korištenja aplikacija i alata, mnogi od kojih su nekad bili prisutni samo u *offline* inačicama poput alata za uređivanje teksta, a zbog svojih mogućnosti postao je glavno mjesto korištenja aplikacija koje koriste Internet u svojem radu. Mnoge od aplikacija dostupnih putem Interneta rade sa rokovima kraćima od sekunde, pa su takve aplikacije ubrzo u javnosti postale poznate kao aplikacije u realnom vremenu te se taj pojam održao, unatoč svojoj netočnosti. Brzina sustava nema veze sa računanjem u realnom vremenu, u smislu da čak i najbrže računalo nije samo po sebi sustav u realnom vremenu. Isto tako, sustav u realnom vremenu sa zadacima sa realnim vremenom od nekoliko dana vrlo je spor, ali to ne znači da to nije sustav u realnom vremenu. Kod Web aplikacija to je naročito bitno primijetiti, jer ista Web aplikacija može biti vrlo brza ili vrlo spora, ovisno o brzini veze kojom raspolažemo. Ovdje ponavljam prije spomenutu činjenicu da je duljina realnog vremena ovisna o aplikaciji, tj. realno vrijeme aplikacije je ovisno o potrebama i funkciji te aplikacije. Ovdje se može govoriti i o upotrebljivosti aplikacije sa stajališta korisnika – aplikacija možda ispunjava sve svoje zahtjeve uključujući i duljinu realnog vremena, ali je korisnik ne doživljava kao takvu jer neka druga, slična aplikacija radi bolje i brže.

Kad se govori o Web aplikacijama u realnom vremenu (eng. *real time Web applications*), govori se o aplikacijama sa kojima je korisnik u stalnoj interakciji te izravno vidi rezultate te aplikacije, a vrijeme koje je aplikaciji potrebno da te rezultate dobije je dovoljno kratko da ga korisnik tumači kao trenutačno. Zato se „izvršavanje u realnom vremenu“ često, donekle pogrešno, smatra istim izrazom kao i „trenutačno izvršavanje“. Donekle pogrešno zato jer je i ta izjava točna ukoliko je realno vrijeme dovoljno kratko, a upravo to predstavlja današnje Web aplikacije u realnom vremenu – Web aplikacije koje svoj zadatak obavljaju u vremenskom razdoblju koje ljudi percipiraju kao trenutačno. O duljini realnog vremena koje je potrebno da bi se iluzija trenutačnog izvršavanja dobila vode se česte rasprave zbog različite namjene i domene aplikacija. Web aplikacija koja svakih pola sekunde pokazuje položaj zrakoplova i njihov put sa dovoljne visine čini se trenutačnom zbog relativno male duljine koju zrakoplov prođe na takvoj mapi, no u slučaju bilo koje kompetitivne *online* igre (koje nisu nužno *Web* aplikacije, ali služe kao dobar primjer), gdje pobjeda korisnika ovisi o brzini njegove reakcije, promjena stanja aplikacije svakih pola sekunde nije ni približno dovoljna, unatoč tome što se isto realno vrijeme u drugoj aplikaciji percipira kao trenutačno.

Web aplikacijom u realnom vremenu često se smatra vrlo responzivna Web aplikacija. Responzivnost računalnog sustava se može podijeliti prema sljedećim kategorijama (Miller, 1968):

- Odgovor u manje od 100 milisekundi
- Odgovor u manje od 1 sekunde
- Odgovor u manje od 10 sekundi

Ukoliko računalni sustav odgovor vrati u manje od 100 milisekundi, korisnik ga doživljava kao trenutačnim. Iako poželjno, takva brzina nije uvijek potrebna. Ljudi nisu navikli na ovakvu brzinu reakcije čak i kod interakcije sa drugim osobama, pa se može staviti manji naglasak na brzinu reakcije ukoliko ona nije potrebna, kao u prije spomenutim *online* igrama gdje ljudi često reagiraju instinktivno, pa je brži odziv sustava bolji. Ukoliko računalni sustav odgovori u roku od jedne sekunde ili manje, korisnik i dalje ima osjećaj slobodne interakcije sa sustavom, ali i ovo se može činiti prebrzim. Miller je zaključio da bi najbolja opcija bila konzistentan odziv sustava od dvije sekunde, te se ta vrijednost danas često koristi kao cilj kod izrade Web aplikacija. Svaki sustav koji je sporiji od navedenog korisnici smatraju sporim te sa stajališta korisničkog iskustva takav sustav nije kvalitetan, a ako se na odgovor čeka deset sekundi ili više korisnik postaje potpuno nezainteresiran za komunikaciju sa sustavom što želimo izbjeći. Čak je i ovdje potrebno naglasiti da spomenuta „zlatna sredina“ od dvije sekunde nije uvijek dobra, te je najbolje istražiti kakvo vrijeme odgovora aplikacije bi najbolje odgovarala aplikaciji. U već spomenutim *online* igrama ona svakako nije dovoljno brza, ali dvije sekunde činiti će se vrlo brzim ukoliko sustav nove podatke dobiva tek svakih 10 sekundi.

Čak i ako se zanemari drugačije shvaćanje realnog vremena kod Web aplikacija u realnom vremenu i tradicionalnih sustava u realnom vremenu, postoje mnoge razlike koje se mogu primijetiti i koje karakteriziraju Web aplikacije u realnom vremenu, ali postoje i mnoge sličnosti. Sustavi se i dalje sastoje od komponenti od kojih jedna često preuzima ulogu voditelja sustava te svojim djelovanjem sinkronizira cijeli sustav, a kod Web aplikacija u realnom vremenu to je poslužitelj. Poslužitelj ovdje često predstavlja sloj poslovne logike, dok klijenti preuzimaju ulogu sloja prezentacije, pa su vidljive naznake troslojne arhitekture koja je toliko često prisutna kod Web aplikacija. Ovdje su rokovi zadataka vrlo rijetko čvrsti ili strogi, već uglavnom govorimo o mekim rokovima. Poruke se šalju u velikoj količini na velik broj različitih komponenti, ali rijetko možemo biti sigurni u to kad će odgovor na tu poruku stići, ili čak ako će odgovor uopće biti poslan. Ako za primjer uzmemo društvene mreže, to je lako vidljivo. Objavljivanjem statusa i drugim aktivnostima na društvenim mrežama mi šaljemo poruke drugim komponentama sustava, ali odgovor na njih često je redundantan jer nas više zanima isti tip informacija koje mi šaljemo, ali iz smjera drugih komponenti. Nadalje, naše akcije možda će pokrenuti neki novi zadatak na nekoj od komponenti te poslati odgovor, ali često mi taj

odgovor ne očekujemo ili ga čak ni ne primjećujemo. Npr., naše akcije pretraživanja možda će u budućnosti rezultirati time da sustav uključi pametno filtriranje sadržaja prema našim prijašnjim pretraživanjima, ali mi toga nećemo biti svjesni jer nećemo dobiti eksplicitnu poruku da se to dogodilo, ali ćemo moći vidjeti rezultate tog zadatka. Čak i poruke drugih komponenti često neće biti vidljive jer će sustav zaključiti da one za nas nisu važne te ih neće prikazati, a zbog vrlo velikih količina poruka sve društvene mreže moraju imati sličan način rada.

Već se ovdje može vidjeti svojevrsna podjela takvih aplikacija na one gdje poslužitelj dominira komunikacijom u sustavu, tj. komponente klijenti samo prikazuju informacije koje mu šalje poslužitelj koji ih šalje čim ih on dobije, te one aplikacije u kojima su klijent i poslužitelj komponente iste razine te komuniciraju izravno i trenutačno, tj. klijent češće ima potrebu obavijestiti poslužitelja o nekom događaju ili mu čak slati nove podatke koje on mora ažurirati. Prva vrsta aplikacija zapravo je tradicionalni oblik troslojne arhitekture koji je nadograđen određenim tehnikama i tehnologijama kako bi se komunikacija između klijenta i poslužitelja činila trenutačnom. Ovakva vrsta aplikacije dobra je za scenarije poput aplikacije za praćenje vremenskih prilika, gdje se podaci skupljaju na jednom mjestu i gotova informacija se kontinuirano šalje klijentskom računalu, bez potrebe prikupljanja podataka ili dodatne komunikacije sa njime. Druga vrsta aplikacije je ono što danas vrlo često srećemo kod socijalnih mreža – velika mreža računala, klijentskih i poslužiteljskih, koje konstantno, izravno i trenutačno komuniciraju jedna sa drugima i razmjenjuju podatke. Za ovakve vrste aplikacija tehnologije korištene u prijašnjem primjeru već postaju zastarjele, te se razvijaju nove tehnologije koje omogućuju lakšu dvosmjernu komunikaciju između dvije komponente sustava u zadovoljavajućem vremenu, tj. vremenu koje se korisniku čini trenutačnim kako on ne bi izgubio interes za korištenje aplikacije.

Ovdje se također mogu naći podijeljena mišljenja o tome što zapravo karakterizira Web aplikaciju u realnom vremenu, tj. što Web aplikacije u realnom vremenu dijeli od aplikacija koje ih samo oponašaju. Sukobi oko tehnologija, njihovih klasifikacija te načina korištenja ovdje uzimaju glavnu riječ. Tehnologije poput *polling*-a, gdje klijentsko računalo kontinuirano, u određenom vremenskom periodu, poslužitelju šalje poruku sa upitom o tome da li je došlo do promjene stanja, i ako je, da se to stanje ažurira i na klijentsko računalo, često se smatra svojevrsnom simulacijom prave Web aplikacije u realnom vremenu jer su to već postojeće tehnologije adaptirane da se dobije iluzija trenutačnog izvršavanja. Ovakva rješenja još postoje na zastarjelim sustavima kojima su novije tehnologije nedostupne zbog ograničenosti sustava u kojima rade. Nasuprot tome, tehnologije poput WebSocket protokola, koji se danas drži kao standard za Web aplikacije u realnom vremenu, a koji omogućuje perzistentnu dvosmjernu komunikaciju između dva računala, su tehnologije stvorene sa ciljem da pomognu u izradu Web aplikacija u realnom vremenu te se u tu svrhu uglavnom i koriste. Ipak, ovdje je često

riječ upravo u sukobu tehnologija, mišljenja i preferenci razvojnih inženjera, te treba imati na umu da se svaka od tehnologija koristi s istim ciljem, ali pod drugim okolnostima. Nadalje, čak i najnovije tehnologije zasnovane su na starijim tehnologijama poput AJAX-a te na temelju principa i primjera dobre prakse naučenima kod razvoja starijih tehnologija. Zbog toga ću u ovom radu, u dijelu o komunikacijskim tehnikama, objasniti i nekoliko tehnika koje se danas više ne koriste toliko često, ali su dobar primjer snalažljivosti razvojnih inženjera kada trebaju napraviti proizvod prema određenoj specifikaciji, ali nemaju pristup tehnologijama koje su im potrebne ili koje bi bile najbolje za korištenje u tom trenutku.

3.2. Područja korištenja Web aplikacija u realnom vremenu

Područje korištenja Web aplikacija u realnom vremenu podudara se sa područjem korištenja koje pokrivaju tradicionalni sustavi u realnom vremenu, osim naravno sustava koji nisu spojeni na Web poput već spomenutih kontrola za upravljanje zrakoplovom. No područje korištenja Web aplikacija u realnom vremenu znatno se proširilo u zadnjih nekoliko desetljeća u kojima Web postoji kao takav jer se svakim danom uviđaju novi potencijalni načini korištenja Web-a i aplikacija na Web-u. Standardne aplikacije koje često služe kao primjer ovakvih vrsta aplikacija, zbog čega ću ih ja i koristiti za predložak većine programskih primjera, su aplikacije za trenutačnu komunikaciju (C. Holmberg, S. Hakansson, 2015). To je nekad uključivalo aplikacije za trenutačni prijenos poruka poput aplikacija za čavrljanje (eng. *chat*), ali zbog razvoja Web-a i infrastrukture Interneta danas su vrlo česte aplikacije za *video chat* koje omogućavaju video konferencije ili čak prijenos video i audio sadržaja velikom broju ljudi poput *streaming*-a, a ponekad i kombinacija multimedijskog sadržaja i trenutačne komunikacije između korisnika aplikacije koji konzumiraju taj sadržaj. Tržište kojim su nekad dominirale TV i radio postaje sve više osvajaju razne *streaming* stranice koje daju sve usluge tradicionalnih medija poput prijenosa sportskih i drugih događanja uživo, pregleda filmova i serija, itd. Poznate TV kuće počele su prepoznavati važnost Interneta u današnjem poslovanju pa su počele uključivati Web aplikacije kao dio svoje ponude, a te aplikacije uvijek pružaju *streaming* usluge ili druge funkcionalnosti koje zahtijevaju korištenje Web aplikacija u realnom vremenu. Zanimljivo je gledati razvoj Interneta i Web aplikacija kroz *chat* aplikacije koje su nekad činile cjelokupnu funkcionalnost Web aplikacija, dok se danas *chat* aplikacije koriste u suradnji sa drugim funkcionalnostima Web stranica koje *chat* funkcionalnost smo komplementira, što je vidljivo u gotovo svakoj popularnoj društvenoj mreži. Razne *streaming* stranice uspješno su spojile dvije funkcionalnosti, *streaming* multimedijskog sadržaja i popratnu aplikaciju za čavrljanje, od kojih se obje oslanjaju na trenutačnu komunikaciju putem Interneta, te su tim putem postale jednima od najvećih i najutjecajnijih Web aplikacija.

Slika 3: Web streaming aplikacija Twitch *sa kombinacijom video sadržaja i* chat *funkcionalnosti*

Još jedna vrsta aplikacije koja se sve češće može vidjeti na Internetu je aplikacija za praćenje geografske lokacije. Trenutačno dobavljanje podataka omogućava funkcionalnosti poput trenutnog praćenja vremenskih prilika, navigaciju preko karte u realnom vremenu, praćenje lokacije određenog objekta u realnom vremenu, itd. Ovakve vrste aplikacija često su samo dio neke veće aplikacije, što je često aplikacija koja je dio nekog IoT (Internet stvari, eng. *Internet of Things*) sustava. IoT sustavi sastoje se od većeg broja komponenti od kojih su mnoge razni senzori postavljeni na različite lokacije. Oni skupljene podatke šalju na neku središnju lokaciju koja zatim komunicira sa Web aplikacijom razmjenom poruka u realnom vremenu. Mnogi IoT sustavi bi izgubili svoju vrijednost ukoliko bi komunikacija bila sporija, pa je ovakav model i način rada aplikacije ključan u mnogim takvim sustavima. Web aplikacije u IoT sustavima uglavnom imaju ulogu skupljanja i prikazivanja podataka koje mu šalju drugi uređaji i senzori u IoT sustavu, te slanje komandi tim uređajima na temelju analize dobivenih podataka, automatski ili na zahtjev korisnika aplikacije. IoT sam po sebi ima ogroman potencijal, čime Web aplikacije u realnom vremenu samo dobivaju na važnosti zbog svoje uloge u IoT sustavima. Kada se spominju IoT sustavi, razgovor se uglavnom okreće raznim dodacima za kućanstvo poput automatskog paljenja i gašenja svjetala ili sustavima za nadzor, no IoT sustavi imaju ogroman potencijal za korištenje u industriji. Povezivanje svih uređaja u tvornici te prikupljanje podataka sa svakog od tih uređaja zahtjeva središnju aplikaciju koja prati dobivene podatke i na temelju njih može reagirati na iznenadne promjene poput novih zahtjeva proizvodnje ili nesreće u radu što dovodi do boljeg i sigurnijeg proizvodnog procesa. Nadalje, radnici mogu imati vlastite uređaje poput mobilnih uređaja ili tableta sa vlastitim aplikacijama te se na taj način integrirati u sustav te iskoristiti mogućnosti trenutačne komunikacije i pregleda statusa cjelokupnog postrojenja do maksimuma.

Web aplikacije u realnom vremenu imaju velik potencijal i u poslovnoj domeni, izvan korištenja u IoT sustavima. Bilo koji sustav koji zahtjeva praćenje podataka u realnom vremenu potencijalan je sustav koji se može nadograditi Web aplikacijom u realnom vremenu. Sustavi za upravljanje poslovnim sustavima, koji prate podatke o poslovanju pojedine organizacije, uz pomoć trenutačne komunikacije mogu prikazivati trenutne podatke o poslovanju u obliku raznih grafova i drugih grafičkih prikaza koji su ljudima lakši za korištenje. Sustavi za praćenje cijena dionica ili kriptovaluta, od kojih se oboje mijenjaju brzo i često te predstavljaju vrlo važnu informaciju osobama koje se njima bave, također su često popraćeni Web aplikacijom u realnom vremenu. Poslovni događaji koji zahtijevaju komunikaciju gotovo su se revolucionalizirali dolaskom Interneta. Svakodnevne poslovne potrebe sastanaka promijenile su se dolaskom video konferencija i drugih aplikacija za trenutačnu komunikaciju eliminirajući potrebu osobnog sastajanja. Nadalje, događanja poput aukcija promijenila su se iz korijena. Aukcije rijetkih i skupih stvari za koje se zanima malen broj ljudi ili aukcije za dobivanje poslova danas je puno lakše organizirati, no čak i lokalne i česte aukcije poput onih na kojima se prodaje hrana ili cvijeće danas imaju velike koristi od ovakvih aplikacija (Carter, 2019). Aukcije koje su se nekad organizirale vrlo rijetko zbog toga što se potencijalni sudionici nalaze po cijelom svijetu danas je trivijalno organizirati i održati uz pomoć specijaliziranih aplikacija. Zdravstvo je još jedna domena koja ima veliku potencijalnu korist od Web aplikacija u realnom vremenu zbog mogućnosti lakog praćenja statusa pacijenata na bilo kojem uređaju. Nadalje, pomoć na daljinu (eng. *remote assistance*) uz pomoć Web aplikacija u npr. kirurškim zahvatima omogućuje pomoć specijalista koji možda fizički ne može biti prisutan zahvatu, ali takva funkcionalnost zahtijeva vrlo pouzdanu Web aplikaciju sa vrlo kratkim realnim vremenom jer kašnjenje ili kvar aplikacije može značiti gubitak nečijeg života.

Struka razvojnog inženjera također ima velike koristi od Web aplikacija u realnom vremenu. Jedna od često viđenih aplikacija ovog tipa su aplikacije za kolaborativno uređivanje dokumenata. Naravno, ova vrsta aplikacije ima veliku korist i izvan struke softverskog inženjerstva, ali mogućnost zajedničkog popunjavanja dokumentacije, izvještaja i drugih dokumenata je od velike koristi. Dijeljenje ekrana (eng. *screen sharing*), tj. zajednički rad više geografski razdvojenih osoba na jednom računalu također može biti vrlo korisno. Implementacija, nadzor i održavanje softverskog proizvoda također je olakšana Web aplikacijama u realnom vremenu. One omogućuju praćenje trenutnog stanja udaljenog sustava, mogućnost promjene konfiguracije sustava ukoliko se za to pokaže potreba, slanje izvještaja o kvaru udaljenog sustava, usporedbu stanja sustava sa ostalim sustavima iste vrste, itd. Naravno, mnoge već spomenute aplikacije također su vrlo korisne u ovoj industriji. Uloga Web aplikacija u realnom vremenu u spomenutim domenama svakim danom raste i postaje sve važnijom, a novim idejama i aplikacijama one se šire i na ostale domene.

4. Komunikacijske tehnike

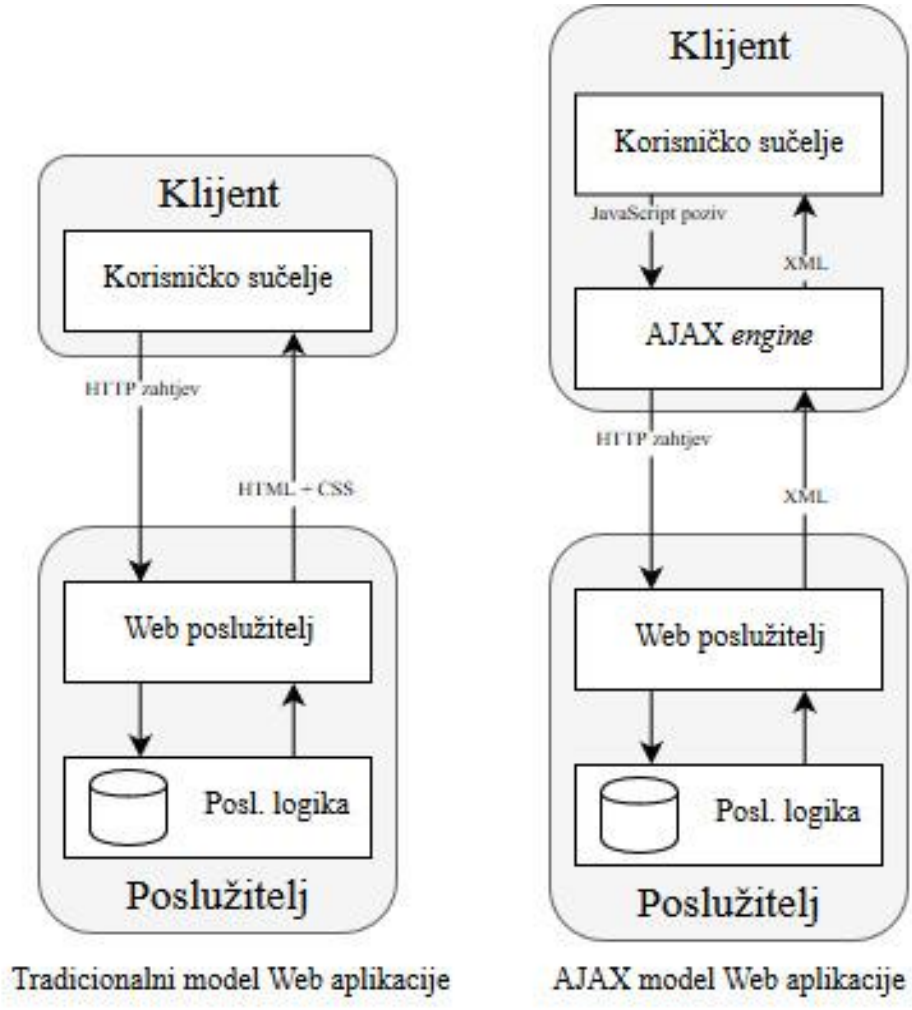
Web aplikacije u realnom vremenu bile su cilj razvojnih inženjera od samih početaka Interneta i Weba. Unatoč tome što na početku one nisu bile moguće zbog ograničenih mogućnosti Web preglednika i malog broja tehnologija za razvoj Web aplikacija, broj, kompleksnost i uporabljivost tih tehnologija je ubrzano rastao. Ubrzo je postalo moguće kombinirati te tehnologije u još kompleksnije tehnologije, bolje od sume korisnosti njihovih dijelova. Jedna od tih tehnologija, i najvažnija tehnologija u ovoj domeni, je AJAX. AJAX, skraćeno za asinkroni JavaScript i XML (eng. *asynchronous JavaScript and XML*) je tehnologija koja omogućuje Web aplikaciji da šalje asinkrone zahtjeve poslužitelju te od njega na isti način primi podatke. To je omogućilo revoluciju u izgradnji interaktivnih Web stranica koje su postale puno responzivnije i dinamičnije.

Unatoč AJAX-ovoj spomenutoj definiciji kao tehnologije, on se može bolje opisati kao skup različitih tehnologija koje zajedno dodaju novu funkcionalnost. Popis tih tehnologija je sljedeći (Garrett, 2005):

- Prezentacijske tehnologije (HTML, CSS)
- *Document Object Model* (DOM)
- Tehnologije za rad sa podacima (XML, JSON)
- XMLHttpRequest objekt
- JavaScript skriptni jezik

Osnovni dio prezentacijskog dijela AJAX-a su standardne HTML i CSS prezentacijske tehnologije i sve njihove izvedenice. One se upotpunjuju DOM-om (eng. *Document Object Model*), svojevrsnim API-em koji gleda Web stranicu kao stablo na kojem je svaki element stranice posebna grana i koji za sebe ima priključen određene događaje i svojstva. DOM nam omogućuje pristup svakom elementu Web aplikacije te manipulaciju stanja tih elemenata kroz njihova svojstva te promjenu njihova ponašanja kroz njihove odgovore na određene događaje (Robie, 1998). Nadalje, AJAX se sastoji od nekog sustava za spremanje, razmjenu i manipulaciju podataka poput XML-a i pripadnih tehnologija. AJAX-ovo ime često nije potpuno točno već se koristi samo zbog konvencije, jer se umjesto XML-a mogu naći i druge tehnologije koje rade sa podacima, od kojih je na Web-u najčešći i najvažniji JSON. Sljedeći važan element AJAX-a je XMLHttpRequest, objekt koji sadrži metode za slanje podataka od Web preglednika do poslužitelja, te primanje podataka koje poslužitelj šalje Web pregledniku (W3C (XMLHttpRequest dokumentacija), 2016). Ovdje se također, unatoč AJAX-ovom imenu, mogu koristiti i druge tehnologije koje rade sa podacima, poput JSON-a ili HTML-a. Ovaj objekt također je osnova za veliku većinu Web servisa jer opisuje parametre poput vrste zahtjeva

poput GET, POST, PUT i DELETE, od kojih svaki ima svoju funkciju. Zadnji dio AJAX-a je JavaScript koji spaja sve ostale tehnologije i omogućuje njihovu kvalitetnu i točnu interakciju unutar Web aplikacije. Sve ove tehnologije spojene su u takozvani AJAX *engine* koji ostvaruje asinkronu komunikaciju sa poslužiteljem „iza kulisa“ Web aplikacije. Usporedbu tradicionalne Web aplikacije i njene komunikacije sa poslužiteljem sa onom u kojoj se koristi AJAX može se vidjeti na sljedećoj slici:



Slika 4: Usporedba modela tradicionalne Web aplikacije i one koja koristi AJAX (Garrett, 2005)

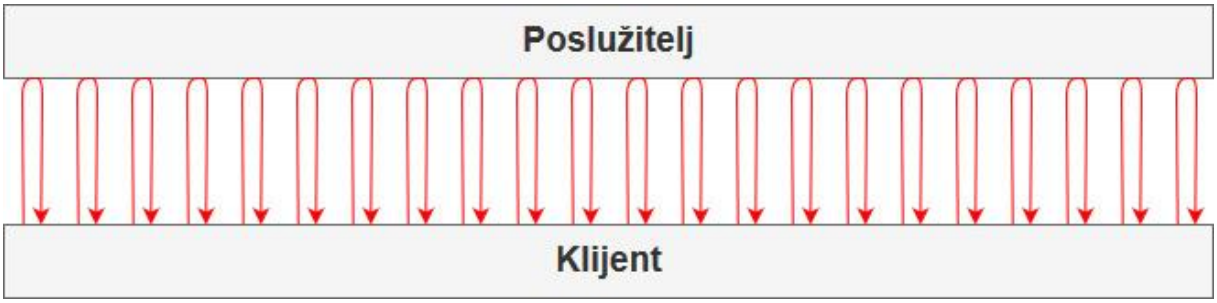
Model tradicionalne Web aplikacije vidljiv je na lijevom modelu. Na klijentskoj strani nalazi se korisničko sučelje koje je vidljivo korisniku i sa kojim on komunicira. Na korisnikov zahtjev, tj. kao reakciju na neku njegovu akciju na Web stranici, klijentski dio aplikacije šalje HTTP zahtjev poslužiteljskom dijelu aplikacije. Tamo se dobavljaju i prema potrebi transformiraju podaci, te se šalje odgovor klijentskoj strani aplikacije u obliku kombinacije HTML i CSS podataka koje klijent može izravno prikazati korisniku. Takav pristup zahtjeva ponovno učitavanje Web stranice kako bi odgovor i promjena koju taj odgovor stvara bili vidljivi jer ne postoji nikakav mehanizam kojim bi poslužitelj mogao natjerati klijenta da osvježi podatke koje prikazuje, ili koji bi rekao klijentu neka osvježi korisničko sučelje kada dobije nove

podatke. Model na desnoj strani slike prikazuje Web aplikaciju koja koristi AJAX *engine* kao ključnu komponentu klijentske strane aplikacije. Umjesto da klijent poslužitelju direktno šalje zahtjev uporabom elemenata korisničkog sučelja, on se pozivom JavaScript metode šalje AJAX *engine*-u. On zahtjev korisnika zapakira u HTTP zahtjev nakon eventualnih transformacija podataka ukoliko su one potrebne te ga šalje poslužiteljskoj strani aplikacije. Poslužitelj gotovo sve radi na isti način kao i kod prvog modela, ali ovog puta ne šalje odgovor u obliku kojeg klijent može direktno prikazati, već šalje zahtijevane podatke u XML obliku, ili bilo kojem drugom spomenutom obliku pohrane podataka. Ti podaci stižu do AJAX-ovog *engine*-a te se ondje pretvaraju u prezentacijske podatke koji se prikazuju korisniku, ali koji se vraćaju na korisničko sučelje koristeći pozive JavaScript funkcija. Ovakav način rada omogućuje da korisničko sučelje ostaje konstantno, tj. da ga nije potrebno osvježavati kako bi se dobili novi podaci, jer ovaj put imamo mehanizam koji uz pomoć JavaScript metoda može korisničkom sučelju dati podatke i reći mu kada da se osvježi i prikaže nove podatke. Taj mehanizam je, naravno, AJAX *engine*, koji također preuzima i dio poslovne logike koji se tiče slanja zahtjeva poslužiteljskoj strani aplikacije. AJAX *engine* nikad nije vidljiv korisniku već se nalazi unutar Web stranice u obliku skrivenog elementa.

AJAX je sam po sebi donio veliku revoluciju u proces razvoja Web aplikacija. Svaki od njegovih dijelova polako je prihvaćen i standardiziran od strane W3C konzorcija, a i sam AJAX ubrzo je postao popularan te se danas može naći u velikoj većini Web preglednika te je poznat svakom ozbiljnijem razvojnom inženjeru koji radi sa Web aplikacijama. No, unatoč svemu onome što AJAX nudi, on je samo osnova u mnogim kasnije razvijenim tehnologijama kao što su i JavaScript, XML i ostale tehnologije samo osnova za razvoj samog AJAX-a. Veće brzine Interneta te njegova stabilnija infrastruktura i veća pouzdanost ubrzo je dovela do toga da se asinkrona komunikacija između klijentske i poslužiteljske strane aplikacije može iskoristiti u svrhu razvoja Web aplikacija u realnom vremenu. AJAX rješava velik dio problema u komunikaciji raznih dijelova aplikacije u realnom vremenu, no problemi poput rasporeda zadataka, pravovremene komunikacije između različitih dijelova aplikacije te stabilnosti takvog sustava nisu riješene u osnovnim AJAX-ovim funkcionalnostima. Zbog toga su razvojni inženjeri kroz godine razvoja razvili mnoge tehnologije kojima su rješavali te probleme na način na koji je to njima bilo potrebno s obzirom na projekt koji im je u tom trenutku bio u fokusu. Zato danas imamo velik broj tehnologija, neke od kojih su rudimentarne, a neke razvijene sa ciljem standardizacije kako bi se rudimentarna rješenja mogla zaboraviti jer nisu bila prikladna u velikom broju slučajeva ili se smatralo da će vrlo uskoro postati zastarjela. Organizacije poput W3C konzorcija također su prepoznale potencijal ovakvih tehnologija te radile na standardizaciji onih koje su imale najviše potencijala, te ću u nastavku opisati nekoliko najvažnijih i najzanimljivijih od tih tehnologija.

4.1. Polling/Dugi polling

Polling je najosnovnija komunikacijska tehnika Web aplikacija u realnom vremenu proizašlih od vremena kada se AJAX počeo koristiti, no to je i starija tehnika koja se koristila i u druge svrhe. Riječ *polling* mogla bi se na hrvatski jezik prevesti kao prozivanje, no taj se pojam gotovo uopće ne koristi već se koristi originalni izraz, pa ću ga i ja koristiti u ovom radu. Polling kao komunikacijska tehnika obuhvaća kontinuirano cikličko slanje zahtjeva za podacima od strane klijenta prema poslužitelju. Zbog toga što sustav ne zna kada poslužitelj dobiva nove podatke, on treba kontinuirano slati nove zahtjeve poslužitelju kako bi bio spreman primiti nove podatke što prije, tj. kaže se da klijent *poll*-a ili proziva poslužitelja. Ovakav sustav vrlo je jednostavno za implementirati i može se izraditi u jednostavnoj aplikaciji koristeći samo obične *socket*-e, što je jedan od razloga zbog kojeg se polling nekad vrlo često koristio, no dolaskom AJAX-a on se počeo koristiti u slanju zahtjeva i primanju odgovora u Web aplikacijama. Zahtjev koji se šalje može biti minimalan jer se prema poslužitelju ne šalju nikakvi kompleksni podaci, a vrijeme između dva zahtjeva, tj. polling interval, moguće je konfigurirati prema potrebi sustava, tj. skratiti ga ukoliko znamo da poslužitelj često dobiva nove podatke ili ga produžiti na određeni interval za koji znamo da traje onoliko koliko treba novim podacima da stignu do poslužitelja. Komunikacija između klijenta i poslužitelja u sustavu koji koristi polling može se vidjeti na modelu na sljedećoj slici.

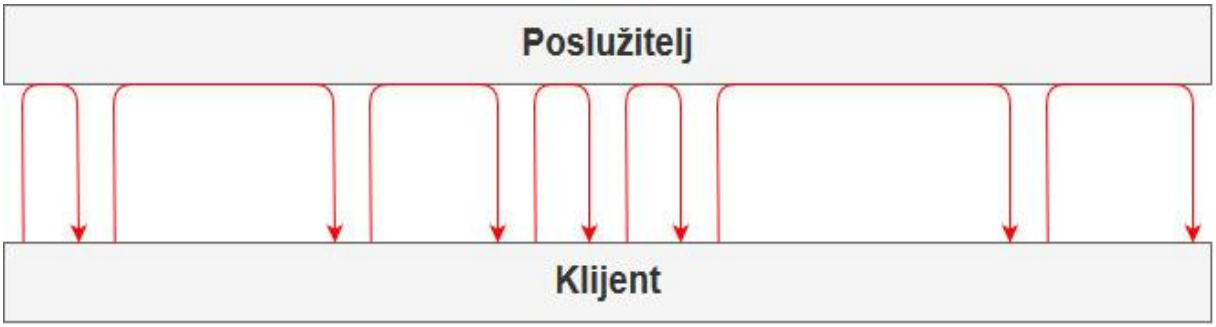


Slika 5: Model komunikacije koja koristi polling

Ovakav sustav vrlo je primjeren za jednostavne sustave sa osnovnom tehnikom i opremom, no u sustavima koji imaju veće mogućnosti polling ima neke lako uočljive nedostatke. U jednostavnijim sustavima zahtjev za novim podacima može biti vrlo malen, nekad samo veličine jednog bita, no u sustavima poput Interneta čak i jednostavni zahtjevi nose različite metapodatke čime stalna komunikacija ove vrste postaje skupa, pogotovo ako infrastruktura nije u dobrom stanju i zahtjevi vrlo dugo putuju mrežom. Nadalje, potrebni su česti zahtjevi kako bi se Web aplikacija činila interaktivnom i kako bi je ljudi doživljavali kao Web aplikaciju u realnom vremenu, no to je vrlo neefikasno i skupo ukoliko većina tih zahtjeva rezultira odgovorom da poslužitelj nema nikakvih novih podataka. Ukoliko znamo da podaci na poslužitelj stižu u određenom intervalu, cijena polling-a neće biti toliko visoka, no to je vrlo

rijedak slučaj, pogotovo kada aplikaciju koristi veći broj korisnika koji u isto vrijeme dodaju nove podatke na poslužitelj. U tom slučaju bolje je koristiti neku od komunikacijskih tehnika u kojoj poslužitelj kontaktira klijenta kada dobije nove podatke, što nekad nije bilo moguće, no stoji kao razlog zašto polling danas više nije efikasna komunikacijska tehnika. Do problema također dolazimo ukoliko novi podaci na poslužitelj stignu odmah nakon zahtjeva klijenta, pa se novi podaci šalju tek u sljedećem intervalu što može biti štetno za aplikaciju ukoliko je taj interval dugačak čime se gubi smisao cijele aplikacije, a to je da se podaci dobiju što je to prije moguće. Ukoliko pak smanjimo interval tako da bude npr. kraći od sekunde, velik broj klijenata generirat će stotine zahtjeva po sekundi i zagušiti stranicu što ne smijemo dopustiti.

Sve navedene probleme rješava nadogradnja na polling koja se naziva *long polling*, kojeg ću u daljnjem tekstu nazivati dugi polling. Umjesto niza zahtjeva koje klijent šalje prema poslužitelju, šalje se samo jedan zahtjev kada je to potrebno. Kada klijent primi podatke, on odmah pošalje novi zahtjev za podacima prema poslužitelju, koji se tada sprema kod poslužitelja te on na njega odgovori kada dobije nove podatke. Na koji način je zahtjev spremljen kod poslužitelja ima veliku važnost jer o tome ovisi koliko će skupa ova komunikacijska tehnika biti za sustav. Kada novi podaci stignu, poslužitelj ih šalje klijentu koji ih primi te odmah pošalje novi dugi *poll*, tj. novi zahtjev prema poslužitelju, i tako unedogled ili do kada klijent više nije aktivan. Najveća prednost dugog polling-a je izbjegavanje nepotrebno velikog broja zahtjeva koji se šalju prema poslužitelju, tako da je ova tehnika nešto što je upotrebljivo i u današnjim okolnostima. Također neće doći do situacije u kojem klijent propušta nove podatke sa poslužitelja te ih mora čekati do sljedećeg ciklusa, tj. do slanja sljedećeg zahtjeva, jer se novi zahtjev šalje odmah nakon primanja podataka i koji čeka na poslužitelju do sljedeće pojave novih podataka. Kod ove komunikacijske tehnike nije potrebno razmišljati o najboljoj duljini intervala za određenu aplikaciju jer cikličkog izvođenja i periodičnog slanja zahtjeva jednostavno više nema (Realtime API, 2019). Ta komunikacija vidljiva je na modelu prikazanom na sljedećoj slici:



Slika 6: Model komunikacije koja koristi dugi polling

Unatoč očitim prednostima naspram običnog polling-a, dugi polling svejedno ima svoje nedostatke. Glavni nedostatak je potreba spremanja zahtjeva ili podataka o klijentima koji čekaju na nove podatke na poslužitelju. Koliko je to skupo za sustav ovisi o tome kako je poslužitelj implementiran, no česta implementacija sa bazenom (eng. *pool*) dretvi znači da će svaki klijent u nekom trenutku trošiti određeni dio resursa poslužitelja jer će im se ili slati novi podaci ili će biti spremljeni kao klijenti koji čekaju nove podatke. Hardver postaje bolji iz dana u dan, no broj korisnika Web-a koji se povećava svakim danom te pristup izradi Web aplikacija koji je više fokusiran na čitljivost koda i njegovo održavanje nego na performanse sustava znači da ne možemo računati na to da će više instanci poslužitelja riješiti problem. Drugi problem je da je to još uvijek neefikasan sustav u kojem se šalje dva puta više zahtjeva nego je to potrebno jer klijent još uvijek mora sam zatražiti nove podatke da ih dobije, umjesto da mu ih poslužitelj sam pošalje kada oni stignu. No unatoč navedenim nedostacima, dugi polling još uvijek je u upotrebi kao svojevrsna zadnja opcija, tj. komunikacijska tehnika koji se koristi ukoliko nijednu drugu (i bolju) komunikacijsku tehniku nije moguće koristiti. Na taj način je koristi i SignalR, biblioteka za izradu Web aplikacija u realnom vremenu o kojoj ću više pričati kasnije, a ovdje ću ukratko objasniti način na koji ona koristi dugi polling. Na sljedećem isječku koda vidljiva je konfiguracija aplikacije koja nalaže korištenje samo dugog pollinga kao komunikacijske tehnike.

```
app.UseSignalR(configure =>
{
	var desiredTransports = HttpTransportType.LongPolling;
	configure.MapHub<ChatHub>("/hub", (options) =>
	{
		options.Transports = desiredTransports;
		options.LongPolling.PollTimeout = TimeSpan.FromSeconds(2);
	});
});
```

Ovdje sam izradio *sample chat* aplikaciju koja služi kao primjer SignalR biblioteke u njihovoj dokumentaciji (Microsoft Docs (SignalR), 2019), a aplikaciju sam pridodao radu te se može naći u direktoriju sa primjerima pod nazivom *SignalRSample* i pokrenuti uz pomoć Visual Studio razvojnog okruženja. SignalR aplikaciju moguće je konfigurirati sa nekoliko različitih postavki. HttpTransportType je enumeracija koja sadrži sve komunikacijske tehnike koje se nude u SignalR biblioteci, te sam ja ovdje odabrao samo LongPolling opciju kako bih pregazio automatsku opciju koja odabire najbolju moguću komunikacijsku tehniku, a ovdje bi dugi polling bio zadnji na listi. Kao željeni tip transporta, tj. komunikacijsku tehniku, stavio sam varijablu *desiredTransports* koja sada sadrži samo dugi polling. Nakon pokretanja aplikacije u dva različita prozora u Web pregledniku kako bih dobio dva klijenta aplikacije, otvorio sam

Network karticu unutar alata za razvojne inženjere koji su dostupni u velikoj većini Web preglednika kako bih mogao pratiti komunikaciju moja dva klijenta, tj. kako bih vidio zahtjeve koje šalje i prima klijent kojeg promatram spomenutim alatima. U ovom primjeru, kao i u drugim primjerima, koristim Google Chrome Web preglednik te izostavljam sve nepotrebne informacije sa prikaza unutar razvojnih alata. Komunikacija između moja dva simulirana klijenta vidljiva je na sljedećoj slici:

Slika 7: Snimka komunikacije koja koristi dugi polling

Na lijevoj strani vidljivi su nazivi zahtjeva, na sredini je vidljivo trajanje čekanja zahtjeva, dok je na desnoj strani vidljiv vodopadni model komunikacije. Na slici se može vidjeti da je, svaki put kada je klijent primio podatke, on odmah poslao novi zahtjev poslužitelju te počeo čekati na nove podatke. Čekanje klijenta označeno je zelenom bojom na vodopadnom modelu. Taj period prekinut je novim podacima nakon čega počinje novi period čekanja. Jedna od opcija konfiguriranja sustava koji koristi dugi polling je vrijeme ispadanja zahtjeva (eng. *poll timeout*), tj. vremena nakon kojeg će se zahtjev smatrati izgubljenim i nakon kojeg će se poslužitelju slati novi zahtjev. Ta opcija se koristi u slučaju ako poslužitelj izgubi ili ignorira zahtjev zbog neke greške, pa da klijent pritom ne bude primoran ponovno pokrenuti aplikaciju i slati početni zahtjev kako bi održao komunikaciju sa poslužiteljem. Potrebno je biti oprezan te ovu opciju ne postaviti prekratkom jer se u tom trenutku dobije svojevrsna simulacija običnog pollinga jer šaljemo vrlo velik broj zahtjeva prema poslužitelju bez zaprimanja novih podataka, te samim time gubimo sve prednosti koje dugi polling ima naspram običnog pollinga. Primjer komunikacije klijenta i poslužitelja kada vrijeme ispadanja zahtjeva postavimo na dvije sekunde, vrijeme koje je postavljeno u primjeru koda u *TimeSpan* obliku, vidljiv je ovdje:

Slika 8: Snimka komunikacije koja koristi dugi polling sa poll timeout *opcijom postavljenom na dvije sekunde*

4.2. Forever Frame

Forever Frame je komunikacijska tehnika koja se danas zbog svojih mnogih nedostataka praktički uopće ne koristi kod izrade Web aplikacija u realnom vremenu, te se uglavnom može naći samo u starim sustavima i aplikacijama. On spada u skupinu komunikacijskih tehnika koje se nazivaju *Comet* komunikacijske tehnike, u koje se često pribrajaju i polling i dugi polling, a koje koriste česte HTTPS zahtjeve kako bi Web pregledniku dali podatke bez da ih on zasebno traži (Roth, 2008). Odlučio sam posvetiti jedan dio rada ovoj komunikacijskoj tehnici jer je dobar primjer slučaja u kojem razvojni inženjeri uspiju iskoristiti mogućnosti pojedinih tehnologija i sustava za svrhu za koju originalno nisu izrađeni kako bi se zaobišli nedostaci platforme i postigao krajnji cilj. Glavni razlog postojanja Forever Frame komunikacijske tehnike su ograničenja Web preglednika Internet Explorer. Zbog toga što je on vrlo dugo bio sastavni dio Windows operacijskog sustava, koristio ga je vrlo velik broj korisnika te je svaka Web aplikacija koja je htjela imati velik broj korisnika morala ponuditi i način na koji će se njihova aplikacija pokretati i na Internet Explorer-u. Ostali Web preglednici poput Mozilla Firefox-a i Google Chrome-a vrlo su brzo u svoj rad ukomponirali mogućnosti HTML5, između ostalog i naprednije komunikacijske tehnike koje su nadišle *Comet* skupinu, no Internet Explorer to nije napravio te se za njega moralo osmisliti drugačije rješenje.

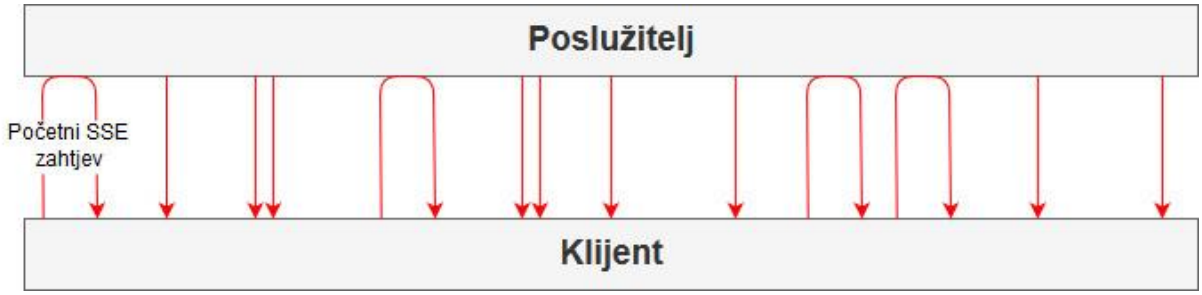
To rješenje je došlo u obliku Forever Frame-a, komunikacijske tehnike koja se oslanja na skrivenu *iFrame* HTML oznaku (eng. *tag*) koja otvara komunikaciju prema poslužitelju (Griffin, 2015). *iFrame* je HTML oznaka koja se koristi kako bi se u HTML dokument umetnuo (eng. *embed*) neki drugi sadržaj sa druge lokacije kojoj se pristupa preko Web adrese. Ono što ta oznaka još omogućava je da ta udaljena lokacija kao sadržaj pošalje JavaScript skriptu koja može pozivati JavaScript funkcije sa svojim podacima kao argumentima na strani na kojoj se nalazi *iFrame* oznaka. Ta oznaka namijenjena je npr. umetanju slika na Web stranicu bez potrebe da se one preuzimaju i spremaju na poslužitelju ili da se u stranicu umetne sadržaj neke druge Web aplikacije. No brzo je postalo očito da se kroz korištenje funkcionalnosti pozivanja JavaScript funkcija na klijentskoj strani može otvoriti veza između klijenta koji u svom HTML dokumentu ima oznaku *iFrame*, te poslužitelja koji je naveden kao izvor sadržaja za tu oznaku. Takva veza omogućuje poslužitelju da poziva JavaScript funkcije klijenta koje su posebno pripremljene prema onome što želimo da aplikacija radi i prema tome kakve podatke očekujemo. Ukoliko znamo da poslužitelj često dobiva nove podatke, možemo ga konfigurirati tako da svaki put kada mu stignu novi podaci da on odmah pozove JavaScript funkciju na klijentskoj strani sa novim podacima kao argumentima te funkcije i na taj način ih proslijedi klijentu. Na taj način dobiva se komunikacija u realnom vremenu sa komponentama koje za to nikad nisu bile namijenjene (Gravelle, 2017).

Jedno od svojstava HTML-a 1.1 je i takozvano enkodiranje komada (eng. *chunk encoding*), tj. slanje odgovora na zahtjev u dijelovima kada još ne znamo konačnu duljinu samog odgovora. To omogućuje često ažuriranje klijenta novim podacima bez dugog čekanja. Web preglednici su tada samo morali iznova osvježiti nove podatke kako bi se novi podaci redovito prikazivali i Web aplikacija u realnom vremenu proradila. Prednost te veze je također to da je ona perzistentna, tj. ne bazira se na porukama preko *socket*-a kao što to rade polling i dugi polling, već je potreban samo jedan zahtjev da se otvori veza između klijenta i poslužitelja koja zatim ostaje otvorena i spremna za korištenje. Nedostatak je taj da je ta perzistentna veza jednosmjerna te prolazi samo od poslužitelja prema klijentu koji koristi takozvani poslužiteljski *push* (eng. *server push*) da klijentu šalje podatke bez njegovog eksplicitnog zahtjeva. Ukoliko klijent želi poslati podatke poslužitelju, on to ne može napraviti preko veze koju *iFrame* drži otvorenom, već mora poslati novi zahtjev prema poslužitelju na standardni način koristeći AJAX zahtjev. Ovakav način rada može biti dobar ukoliko se radi o Web aplikaciji koja ne prikuplja podatke od klijenata nego se podaci često mijenjaju na poslužitelju jer tada poslužitelj promijenjene podatke može vrlo lako slati svim klijentima. S druge strane, Forever Frame kao komunikacijska tehnika je vrlo neprikladan za npr. *chat* aplikacije ili za bilo koju aplikaciju u kojoj je potrebna česta komunikacija od klijenta prema poslužitelju.

Forever Frame je moguće implementirati na bilo kojem klijentu koji podržava oznaku *iFrame* što uglavnom čini većina Web preglednika, no svaka aplikacija koja nije zasnovana na HTML-u neće moći koristiti Forever Frame. Ona je također inferiorna naspram kasnije razvijenih komunikacijskih tehnika poput WebSocket protokola te se uglavnom drži na životu zbog raširenosti Internet Explorer-a kao Web preglednika. Biblioteke poput SignalR imale su podršku za Forever Frame, no dolaskom Microsoft Edge-a kao zamjene za Internet Explorer Forever Frame se više ne koristi kod razvoja Web aplikacija u realnom vremenu. Osim ovisnosti o *iFrame* oznaci, nedostatak Forever Frame-a je i neefikasno korištenje memorije. Svi podaci koji se preko otvorene veze šalju prema klijentu ostaju spremljeni u memoriju klijenta, a i sama veza troši nešto memorije, pa je potrebno razviti posebne načine čišćenja memorije kako se ona ne bi zapunila i prouzročila probleme, a pošto je Forever Frame kombinacija više postojećih komponenti ona nema ugrađeni način da to napravi. Već spomenuto ograničenje kod slanja zahtjeva od klijenta prema poslužitelju preko perzistentne veze još je jedan nedostatak koji je pridonio kratkom životu ove komunikacijske tehnike na bilo kojem klijentu osim Internet Explorer-a. Ova komunikacijska tehnika pružala je veliku korist razvojnim inženjerima jer daje puno bolje performanse od polling-a i dugog polling-a na Internet Explorer-u, no njeni brojni nedostaci i ovisnost o puno komponenata i tehnologija koje su se mijenjale nezavisno jedna o drugoj značilo je da Forever Frame nije zaživio poput nekih drugih komunikacijskih tehnika napravljenih za korištenje u Web aplikacijama u realnom vremenu.

4.3. Server Sent Events

Server sent events (skraćeno SSE) je komunikacijska tehnika koja bi se mogla prevesti kao događaji poslani od strane poslužitelja. Ideja SSE-a je podržati perzistentnu, jednosmjernu vezu od poslužitelja do klijenta. Za istu svrhu koristi se i prije opisani Forever Frame, ali za razliku od njega SSE je komunikacijska tehnika koja je zamišljena sa tom svrhom kao svojim glavnim zadatkom te se zbog toga ne sastoji od raznih dijelova koji su zamišljeni za nešto sasvim drugo pa stoga ne troše resurse za funkcije tih dijelova koje se gotovo nikada ne koriste. Nadalje, SSE je kao tehnologija prihvaćena i standardizirana od strane W3C konzorcija kao dio tehnologija koje sastavljaju HTML5 (W3C (SSE dokumentacija), 2015). Zbog toga SSE podržava velika većina Web preglednika, a od onih koji ne podržavaju ističu se Internet Explorer i Microsoft Edge. Ova tehnologija smatra se pomakom od tehnologija koje zahtijevaju stalne zahtjeve da dobiju nove podatke do tehnologija koje koriste sustave bazirane na okidanju i rukovanju događajima kojima se podaci prenose u Web aplikaciji u realnom vremenu. Za razliku od Forever Frame komunikacijske tehnike, SSE ne poziva JavaScript metode direktno kod klijenta, već šalje događaje određenog tipa kojima klijentska strana SSE komunikacijske tehnike zatim rukuje na način na koji je to odredio razvojni inženjer. Komunikacija između klijenta i poslužitelja kada se koristi SSE vidljiva je na sljedećoj slici:



Slika 9: Model komunikacije koja koristi SSE

Na klijentskoj strani, SSE se reprezentira objektom koji implementira sučelje naziva Event Source, ili izvor događaja. Kada klijent pošalje prvi zahtjev prema poslužitelju, on mu vraća odgovor koji sadrži tip sadržaja (eng. *content type*) *text/event-stream*. Kada klijent primi takav odgovor, na klijentskoj strani se kreira novi Event Source koji počne slušati novostvorenu vezu između klijenta i poslužitelja. Ukoliko poslužitelj dobije nove podatke, on ih može poslati prema klijentu koji će zatim rukovati tim podacima na dogovorene načine. Sam Event Source objekt ima tri svojstva koji opisuju njegovo ponašanje. Prvo je *readyState* svojstvo koje opisuje u kakvom je stanju veza između klijenta i poslužitelja, a može imati vrijednost 0 ukoliko se veza upravo stvara, 1 ukoliko je ona već stvorena te 2 ukoliko je ona zatvorena. Drugo svojstvo je *url* u kojem je zapisan URL izvora, tj. poslužitelja. Zadnje svojstvo, *withCredentials*, je tipa *boolean* te opisuje kako Event Source objekt radi sa CORS mehanizmom. CORS, skraćeno

od *Cross-Origin Resource Sharing*, je mehanizam koji zbog sigurnosnih razloga upravlja time da li aplikacija može primiti podatke sa neke lokacije koja je drugačija od njezine, tj. koja ima drugačiju domenu, protokol ili broj porta (MDN web docs (CORS dokumentacija), 2019). Ukoliko to svojstvo kod Event Source-a postavimo na *true,* on neće prihvaćati zahtjeve sa druge lokacije, no ako ga postavimo na *false*, što je automatski postavljena vrijednost, on će normalno raditi sa takvim zahtjevima. Event Source objekt ima jednu metodu, a to je metoda *close()*. Ako se ona pozove, veza između klijenta i poslužitelja se zatvara i vrijednost *readyState* svojstva se postavlja na 2, ukoliko veza već nije bila zatvorena.

Glavni dio Event Source sučelja koje nas ovdje zanima su događaji kojih ima tri vrste. Prvi događaj koji poslužitelj šalje klijentu je događaj *open* koji se rukuje posebnom metodom koja je povezana sa rukovoditeljem naziva *onopen*. Sljedeći događaja koji je ujedno i najkorišteniji u pravilnoj aplikaciji je događaj *message*, kojim rukuje rukovoditelj naziva *onmessage*. Taj događaj se pokreće svaki put kada klijent od poslužitelja primi validnu poruku. Zadnji događaj je *error*, sa rukovoditeljem *onerror*, koji se pokreće svaki put kada poslužitelj klijentu šalje poruku o grešci koja se dogodila tijekom rada aplikacije. Svi ovi događaji od poslužitelja prema klijentu stižu u već spomenutom *text/event-stream* obliku. Taj oblik podataka ima nekoliko polja (eng. *field*) kojima se poruka bolje opisuje. Prvo polje je polje *event* kojim se opisuje koji tipa događaja poruka koja se šalje okida na klijentu. Osim navedene tri vrste koje su opisane, ovdje se porukama mogu dodavati i druga imena za događaje. Klijent može slušati vezu za te događaje dodavanjem novog rukovoditelja sa *addEventListener()* JavaScript metodom, a ukoliko stigne poruka koja nema rukovoditelja tada se smatra da je događaj poruke tipa *message* te se pokreće odgovarajući rukovoditelj. Nadalje, poruka ima podatke unutar *data* polja koji se šalju u obliku teksta, a najčešće je to tekst u obliku validnog JSON stringa koji se na Web-u najčešće koristi kod prijenosa podataka. Sljedeće polje je *id* koji predstavlja jedinstvenu identifikacijsku oznaku zadnjeg događaja koji je stigao do klijenta, a služi tome da Event Source zna ukoliko je propustio neki događaj. Zadnje polje je *retry* koje predstavlja broj milisekunda koje moraju proći nakon gubitka veze do ponovnog slanja podataka klijentu ukoliko prvo slanje nije uspjelo. Ukoliko ovo polje nije popunjeno, ili unesena vrijednost nije prirodni broj, ono se ignorira, a funkcionalnost ponovnog slanja zahtjeva se ne koristi kod slanja događaja između klijenta i poslužitelja. Svako od ovih polja može se i ne mora koristiti. Poslužitelj klijentu može poslati i praznu poruku sa ciljem održavanja veze ukoliko ona ima postavljeno vrijeme nakon kojeg se veza automatski zatvara ako se poruke ne razmjenjuju. ID događaja nije potrebno pratiti ukoliko nam nije važan vremenski redoslijed zaprimanja događaja, pa ga možemo izostaviti u tom slučaju, a *retry* polje koristi se kada je nužno dobiti neku poruku pa se ona opet šalje ukoliko prvo slanje nije uspjelo (MDN web docs (EventSource dokumentacija), 2019).

Ono što je potrebno napomenuti je da SSE ima vrlo slične nedostatke sa Forever Frame komunikacijskom tehnikom. Ova komunikacijska tehnika također otvara jednosmjernu vezu od poslužitelja prema klijentu što je dovoljno za velik broj aplikacija, no SSE ne pomaže u radu aplikacijama koje zahtijevaju da klijenti šalju velik broj poruka prema poslužitelju jer se u tom slučaju mora otvarati zaseban AJAX zahtjev za svako slanje podataka od klijenta prema poslužitelju. Takve aplikacije okreću se drugim tehnologijama, no SSE svejedno ima vrlo veliku upotrebu u velikom broju Web aplikacija u realnom vremenu. Zbog svojih prednosti SSE je tehnologija koja je dostupna u obliku nekoliko okvira za razvoj aplikacija, a koristi je i SignalR. Kod SignalR biblioteke SSE je često korištena komunikacijska tehnika koja se koristi kada nijedna bolja tehnologija nije dostupna, a ondje je to samo WebSocket protokol. Zbog toga što se mnoge aplikacije vrte na starijim Web preglednicima gdje WebSocket protokol nije dostupan, poput starijih verzija Mozilla Firefox ili Google Chrome Web preglednika, SSE je još danas u čestoj upotrebi kao ozbiljna nadogradnja naspram dugog pollinga. Za ovaj primjer iskoristio sam istu *sample* aplikaciju kao i kod primjera za dugi polling, a kod izgleda ovako:

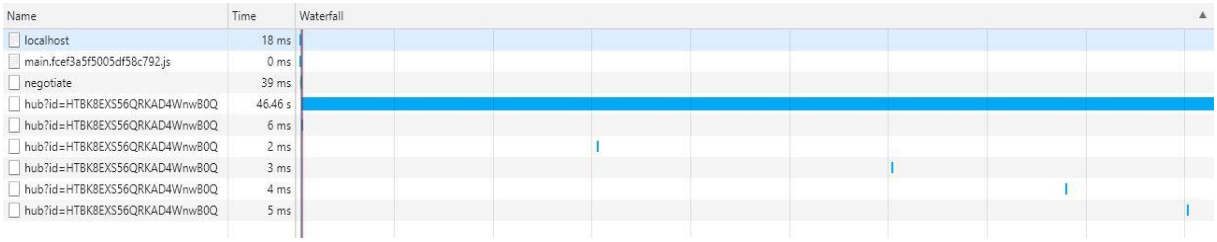
```
app.UseSignalR(configure =>
{
      var desiredTransports = HttpTransportType.LongPolling |
                              HttpTransportType.ServerSentEvents;
      configure.MapHub<ChatHub>("/hub", (options) =>
      {
            options.Transports = desiredTransports;
            options.LongPolling.PollTimeout = TimeSpan.FromSeconds(2);
      });
});
```

SignalR dopušta da se navede više vrsta komunikacijskih tehnika kao željeni tip komunikacije, tako da sam ovdje spremio i dugi polling i SSE u varijablu *desiredTransports*, te ih povezao logičkim „ili“ operatorom (predstavljen znakom „|“). Ako se ne navede nijedna komunikacijska tehnika, SignalR odabere sve tehnike te koristi najbolju koju klijent dopušta. Dugi polling je najslabija komunikacijska tehnika po prioritetu jer je dopuštaju gotovo sve vrste klijenata, pa je stoga dugi polling uvijek sigurna opcija ako sve ostalo ne uspije. Ovdje sam mogao staviti samo SSE kao komunikacijsku tehniku, no smatra se dobrom praksom staviti i dugi polling za slučaj da klijent ili ne podržava SSE kao komunikacijsku tehniku, ili se Event Source objekt zbog određenih problema ne može instancirati. U tom je slučaju dobro imati dugi polling kao *backup* opciju. Ovdje je to samo dodatna mjera opreza jer, kako sam i spomenuo, koristim najnoviji Google Chrome Web preglednik koji već dugo podržava SSE tako da sumnjam da će doći do situacije u kojoj moram koristiti dugi polling. Kod pokretanja aplikacije, u *Console* kartici alata za razvojne inženjere moguće je vidjeti poruku koja izgleda ovako:

```
[2019-06-09T19:11:28.916Z] Information: Normalizing '/hub' to 'https://localhost:44371/hub'.
[2019-06-09T19:11:28.996Z] Information: SSE connected to https://localhost:44371/hub?id=gtPx2fmueDKnn9Fn1DuFvA
```

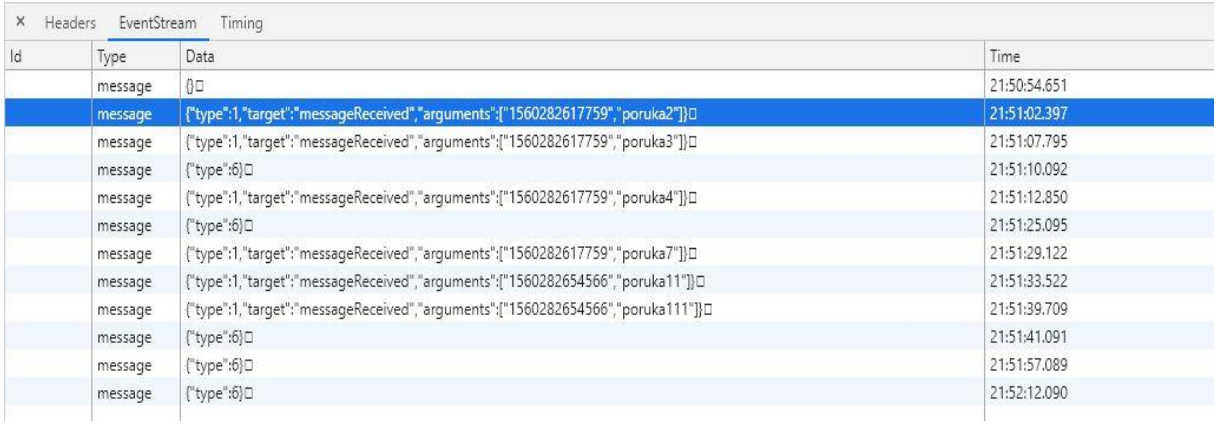
Slika 10: Poruka o uspostavljanju SSE veze

Prva poruka je standardna SignalR poruka koja javlja da je njegov *hub* započeo sa radom, no više ću o tome govoriti kasnije u radu. Ovdje je važna druga poruka koja javlja da se Event Source objekt kreirao te na koju adresu je on spojen. U popisu veza sa kojima Web preglednik i Web aplikacija raspolažu možemo vidjeti jednu vezu koja je aktivna gotovo cijelo vrijeme, što možemo uočiti na vodopadnom modelu komunikacije na sljedećoj slici:



Slika 11: Snimka komunikacije koja koristi SSE

Ta veza, koja je aktivna cijelo vrijeme, predstavlja perzistentnu vezu između klijenta i poslužitelja za koju je zaslužan Event Source. Ostali zahtjevi predstavljaju obične AJAX zahtjeve koje klijent šalje prema poslužitelju. Ukoliko pogledamo zaglavlje (eng. *headers*) poruka koje se kreću tom vezom, vidjet ćemo tip podataka *text/event-stream* koji se koristi kod SSE komunikacijske tehnike, a i sam Event Source objekt te sve poruke koje stižu do njega i kojima on upravlja. U ovom primjeru tip svih događaja je *message*, a svi podaci zapisani su u JSON obliku. Vidljive su i poruke koje nemaju nikakve podatke te služe samo za održavanje veze između klijenta i poslužitelja, a razmak između tih poruka je gotovo uvijek petnaest sekundi zbog unutarnje logike SignalR biblioteke koja upravlja SSE vezom. Sve navedene poruke vidljive su na sljedećoj slici:



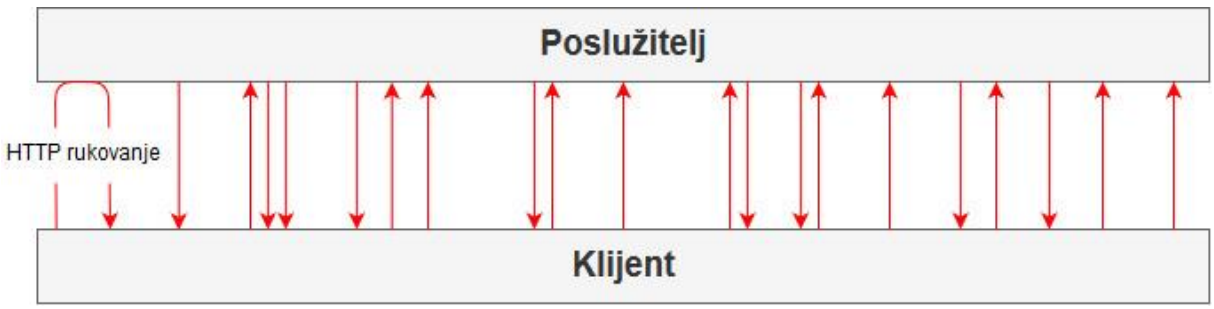
Id	Type	Data	Time
	message	{}□	21:50:54.651
	message	{"type":1,"target":"messageReceived","arguments":["1560282617759","poruka2"]}□	21:51:02.397
	message	{"type":1,"target":"messageReceived","arguments":["1560282617759","poruka3"]}□	21:51:07.795
	message	{"type":6}□	21:51:10.092
	message	{"type":1,"target":"messageReceived","arguments":["1560282617759","poruka4"]}□	21:51:12.850
	message	{"type":6}□	21:51:25.095
	message	{"type":1,"target":"messageReceived","arguments":["1560282617759","poruka7"]}□	21:51:29.122
	message	{"type":1,"target":"messageReceived","arguments":["1560282654566","poruka11"]}□	21:51:33.522
	message	{"type":1,"target":"messageReceived","arguments":["1560282654566","poruka111"]}□	21:51:39.709
	message	{"type":6}□	21:51:41.091
	message	{"type":6}□	21:51:57.089
	message	{"type":6}□	21:52:12.090

Slika 12: Primjer poruka koje se šalju SSE vezom

4.4. WebSocket protokol

WebSocket protokol je po mnogima najbolja tehnologija i komunikacijska tehnika dostupna za korištenje u svrhu izgradnje Web aplikacija u realnom vremenu. Sve što su prije navedene komunikacijske tehnike željele postići, WebSocket protokol sadržava i nadograđuje. Ono što taj protokol u suštini daje je perzistentna, asinkrona, dvosmjerna komunikacija (eng. *full duplex*) preko jedne TCP veze kojom je moguće slati neograničen broj zahtjeva tako dugo dok je veza otvorena i to sa vrlo malim troškom u resursima aplikacije. WebSocket protokol kao tehnologija je standardizirana od strane W3C konzorcija 2011. godine sa pripadnim WebSocket okvirom za razvoj (eng. *API*), a danas ga podržava velika većina modernih Web preglednika te je brzo postao vrlo popularna tehnologija koju su razvojni inženjeri željno iščekivali te vrlo brzo prihvatili kao standard (EDUCBA, 2018). Zbog toga što je to relativno nova tehnologija koju su Web preglednici vrlo brzo implementirali, postoji još mnogo problema sa kojima se razvojni inženjeri susreću kod implementacije WebSocket protokola, no većina popularnih razvojnih okvira i biblioteka koje nude mogućnost njegovog korištenja te probleme rješavaju i dopuštaju da se razvojni inženjeri mogu fokusirati na izradu samih aplikacija.

Sve prije nabrojane i objašnjene komunikacijske tehnike bile su zasnovane na AJAX zahtjevima koji pak su zasnovani na HTTP zahtjevima, no WebSocket protokol ih ne koristi već postoji kao njihova alternativa. Veza koja se otvara WebSocket protokolom je takve prirode da dopušta i klijentu i poslužitelju da, neovisno jedan o drugome, šalju poruke jedan drugome, čak i u isto vrijeme. Također, zbog malog trošenja resursa poput radne memorije, poslužitelj može u isto vrijeme imati otvoren velik broj veza prema više klijenata koristeći WebSocket protokol što omogućuje izradu skalabilnih Web aplikacija u realnom vremenu, pogotovo ako očekujemo da aplikaciju koristi velik broj korisnika u isto vrijeme. Dvosmjerna komunikacija velik je korak u tehnologijama za izradu aplikacija u realnom vremenu jer eliminira potrebu slanja nepotrebnih i skupih AJAX i HTTP zahtjeva. Model komunikacije unutar aplikacije koja koristi WebSocket protokol može se vidjeti na sljedećoj slici:



Slika 13: Model komunikacije koja koristi WebSocket protokol

WebSocket API je nešto širi od SSE API-a jer podržava više slučajeva i sadrži više funkcionalnosti, ali poput SSE-a, na strani klijenta, što je najčešće neki Web preglednik, WebSocket predstavlja objekt istoimenog tipa. Konstruktor tog tipa kao argumente prima URL na koji se želimo spojiti, te opcionalno podprotokol kojim će se vršiti komunikacija. Drugi parametar nije obavezan, ali može olakšati komunikaciju ukoliko želimo da ona koristi točno određeni protokol poput JSON-a, WAMP-a ili SOAP-a. Dogovor oko podprotokola koji će se koristiti u komunikaciji između klijenta i poslužitelja odvija se nakon uspostavljanja WebSocket veze, a navedeni podprotokol se koristi samo ukoliko se obje strane, tj. i klijent i poslužitelj, slože oko korištenja određenog podprotokola, a korisno je za lakše interpretiranje podataka koji se razmjenjuju tijekom komunikacije. Konstruktor može javiti grešku ukoliko je port koji želimo koristiti zauzet, no to se rijetko događa jer WebSocket protokol koristi port 80, ili 443 ukoliko se radi o TLS (eng. *Transport Layer Security*) vezi, koje WebSocket rezervira za svoje korištenje. WebSocket objekt također ima definiranu enumeraciju koja predstavlja status veze koju koristi, a njegovo svojstvo *readyState* je tipa te enumeracije. Vrijednosti koje može poprimiti su 0 ako se veza trenutno otvara, 1 ako je ona otvorena, 2 ako se veza trenutno zatvara te 3 ukoliko je veza već zatvorena. Veza se zatvara prekidom rada bilo koje strane u aplikaciji, najčešće kada se zatvori Web preglednik u kojem se Web aplikacija u realnom vremenu koristi, a to se radi kako bi se sačuvali resursi poslužitelja. Od drugih svojstava vrijedi spomenuti svojstvo *binaryType*, koje predstavlja tip binarnih podataka koji se šalju preko otvorene veze te može poprimiti vrijednosti USVString koja predstavlja tekstualni string, ArrayBuffer koji podatke šalje u obliku niza, Blob kojim se uglavnom šalju objekti te ArrayBufferView koji predstavlja JavaScript niz. Svojstvo *bufferedAmount* može se samo čitati, a predstavlja količinu podataka koja se želi poslati preko veze na drugu stranu, ali koja još nije poslana. Ukoliko vezu zatvorimo ili zbog određenog razloga prestanemo slati poruke te se one spremaju u svojevrsni red čekanja, ova vrijednost će rasti tako dugo dok se slanje poruka opet ne dozvoli. Kada se sve poruke pošalju, vrijednost ovog svojstva past će na nulu. Svojstvo *extensions* predstavlja niz protokola koji se koriste za modifikaciju poruka prije slanja preko veze. Te modifikacije mogu uključivati npr. kompresiju poruke, a sve vrste tih ekstenzija odabire poslužitelj.

Sva ostala svojstva su rukovoditelji događaja koji odgovaraju pripadnim događajima kojih ima više nego kod SSE-a. Možemo postaviti slušača za te događaje na standardni način korištenjem *addEventListener()* JavaScript metode. Prvi događaj je događaj *open*, koji javlja da se WebSocket veza otvorila. S druge strane imamo događaj *close* koji se okida kada se veza zatvorila na pravilan način. Ukoliko se WebSocket veza zatvorila zbog neke greške, okida se *error* događaj. Najčešće korišten događaj je *message* događaj koji se okida svaki put kada preko WebSocket veze do WebSocket objekta stigne poruka. Poruke se šalju prvom od dvije

metode koje su dostupne u WebSocket API-u, a to je metoda *send*. Ona kao argument prima podatke koji se šalju, a koji mogu biti bilo kojeg tipa koje svojstvo *binaryType* opisuje, no to su velikom većinom tekstualni podaci, tj. USVString tip koji koristi standardni UTF-8 format. Svaki put kada se poruka pošalje, vrijednost svojstva *bufferedAmount* se povećava jer samo pozivanje metode *send* ne znači da će se poruka poslati odmah. Ona odlazi u red čekanja jer je moguće da postoje druge poruke koje još nisu poslane, a koje moraju biti poslane prije nje. Količina spremljena u *bufferedAmount* može se prekoračiti nakon čega dolazi do greške, no do toga ne bi smjelo doći ukoliko se porukama pravilno rukuje. Ukoliko se poruka zbog neke greške ne može poslati, veza se zatvara i okida se *error* događaj. Metoda *close* koristi se za zatvaranje veze, a prima dva opcionalna argumenta. Prvi argument je kod koji predstavlja razlog zašto je veza zatvorena. WebSocket API nudi listu kodova (MDN web docs (CloseEvent dokumentacija), 2019) koji predstavljaju sve od normalnog zatvaranja veze predstavljene korištenjem koda 1000, do koda 1009 koji predstavlja situaciju u kojoj se pokušala poslati prevelika poruka. Ukoliko specifičan kod nije naveden, automatski se šalje kod 1005 koji predstavlja poruku o tome da poseban status nije predan metodi za zatvaranje veze. Sljedeći argument, *reason*, predstavlja razlog zatvaranja veze upisan u kratku string vrijednost. Ovaj argument upisuje se kao dodatna informacija koja prati statusni kod iz prijašnjeg argumenta kako bi korisnik više znao o tome zašto je veza prekinuta (MDN web docs (WebSocket dokumentacija), 2019).

SignalR koristi WebSocket protokol kao preferiranu komunikacijsku tehniku koja se koristi kad god je to moguće. Ovdje je vidljiva prednost koju nude SignalR i slične biblioteke jer skriva gotovo sva prije objašnjena svojstva, metode i događaje te nam nudi samo one opcije koje su nama zanimljive, dok ostalo rješavaju same. WebSocket protokol moguće je odabrati kao preferiranu komunikacijsku tehniku, no moguće je podesiti i nekoliko dodatnih opcija. Prva opcija je *CloseTimeout*, tj. svojevrsno vrijeme ispada veze. To je vrijednost koja predstavlja vrijeme koje mora proći između odluke o zatvaranju veze od strane poslužitelja do odluke o zatvaranju veze od strane klijenta prije nego se veza automatski zatvori, a ako se ne navede ona iznosi pet sekundi. U trenutku kada poslužitelj želi zatvoriti vezu klijent možda radi nešto sasvim drugo te ne može odmah odgovoriti zatvaranjem veze, pa ova opcija omogućava zatvaranje veze i oslobađanje resursa na strani poslužitelja bez nepotrebnog čekanja. Druga opcija je *SubProtocolSelector* koja predstavlja delegata za izbor podprotokola koji će se koristiti kod komunikacije sa klijentom. Klijent kod prvog zahtjeva kao dio rukovanja sa poslužiteljem predaje listu podprotokola koje on želi koristiti. Spomenuti delegat uzima tu listu te sadrži logiku kojom se odabire jedan ili više podprotokola koji će se koristiti kod daljnje komunikacije. Ukoliko ova opcija nije popunjena, ona ima vrijednosti *null*, što znači da ne

postoji delegat koji odlučuje o tome koji će se podprotokol koristiti pa se koristi onaj kojeg odredi klijent koji je poslao zahtjev za otvaranjem veze.

Za kratki prikaz korištenja WebSocket protokola iskoristio sam već prije korištenu *sample* SignalR *chat* aplikaciju koju sam modificirao tako da se kod odabira komunikacijske tehnike koja će se koristiti kod rada Web aplikacije u realnom vremenu uzima u obzir i WebSocket protokol. Isječak koda kod kojeg se to događa izgleda ovako:

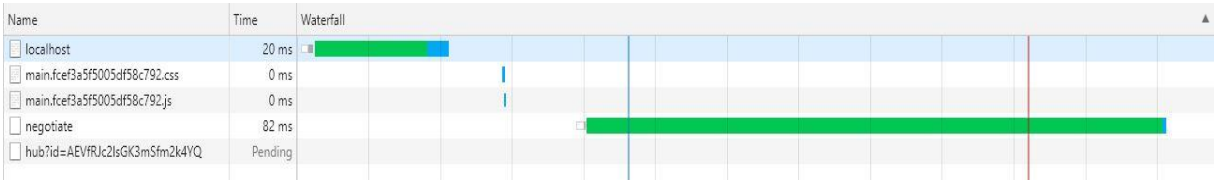
```
app.UseSignalR(configure =>
{
      var desiredTransports = HttpTransportType.LongPolling |
                              HttpTransportType.ServerSentEvents |
                              HttpTransportType.WebSockets;
      configure.MapHub<ChatHub>("/hub", (options) =>
      {
            options.Transports = desiredTransports;
            options.LongPolling.PollTimeout = TimeSpan.FromSeconds(2);
            options.WebSockets.CloseTimeout = TimeSpan.FromSeconds(2);
            options.WebSockets.SubProtocolSelector = requestedProtocols =>
                  {
                        return requestedProtocols.Count > 0 ?
                               requestedProtocols[0] : null;
                  };
      });
});
```

Ovaj puta sam u varijablu *desiredTransports* stavio sve komunikacijske tehnike koje SignalR trenutno podržava, što znači da sam zapravo unio vrijednost koja se koristi automatski, tj. pošto potencijalno koristim sve komunikacijske tehnike, nisam morao navesti nijednu. Nadalje, dodao sam vrijednosti za opisane opcije kod korištenja WebSocket protokola. Kao *CloseTimeout* dao sam vrijednost od dvije sekunde koristeći *TimeSpan*, identično kao i kod *PollTimeout* opcije za dugi polling. Vrijednost pridružena varijabli *SubProtocolSelector* je delegat koji, ukoliko postoji barem jedan podprotokol u zaglavlju zahtjeva koji stiže od strane klijenta, odabire prvi podprotokol u listi. Ovo je vrlo jednostavna logika za demonstraciju, no ova opcija se uglavnom koristi ukoliko poslužitelj ima preferiran način rukovanja podacima pa želi koristiti podprotokol koji mu odgovara, a tu logiku zatim izvršava delegat. Izvršavanjem aplikacije i pogledom u *Console* karticu alata za razvojne inženjere dostupne unutar Web preglednika možemo vidjeti da se, nakon što se inicijalizirao SignalR *hub,* pokrenuo i WebSocket te možemo vidjeti lokaciju na koju se on povezao:

```
[2019-06-10T19:19:41.190Z] Information: Normalizing '/hub' to 'https://localhost:44371/hub'.
[2019-06-10T19:19:41.359Z] Information: WebSocket connected to wss://localhost:44371/hub?id=RZRAuTR6eYBXnYVISPLYaA.
```

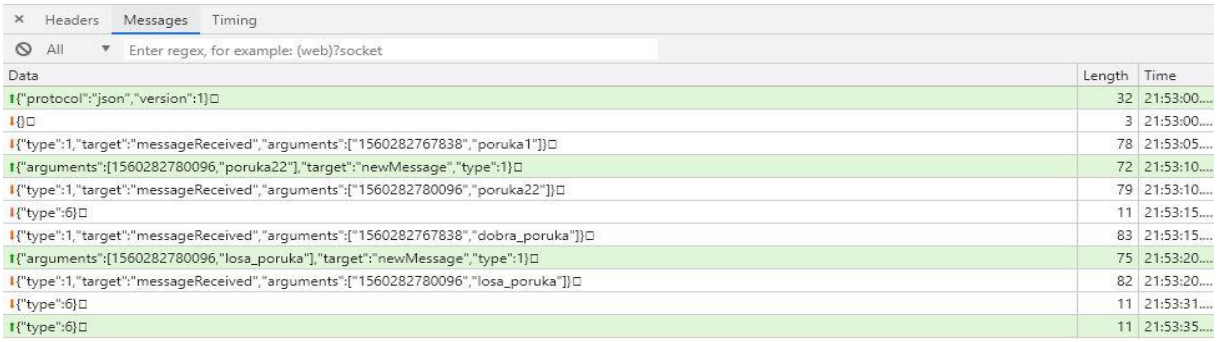
Slika 14: Poruka o uspostavljanju veze WebSocket protokolom

Za testiranje ove aplikacije i dalje koristim Google Chrome Web preglednik koji podržava WebSocket protokol tako da ova aplikacija može iskoristiti tu tehnologiju u svojem radu, pa je SignalR automatski odabire. No ako ovu istu aplikaciju pokrenem koristeći Internet Explorer Web preglednik, do te poruke neće doći. Neće se pojaviti ni poruka o inicijalizaciji Event Source objekta što bi značilo da se koristi SSE, a razlog tome je to da Internet Explorer ne podržava nijednu od te dvije tehnologije. Kod korištenja Internet Explorer-a SignalR će automatski preskočiti WebSocket protokol i SSE te odabrati dugi polling kao jedinu prikladnu komunikacijsku tehniku. Ukoliko se vratimo u Google Chrome Web preglednik i pogledamo što se nalazi unutar *Network* kartice, zateći ćemo sljedeću situaciju:



Slika 15: Snimka komunikacije koja koristi WebSocket protokol

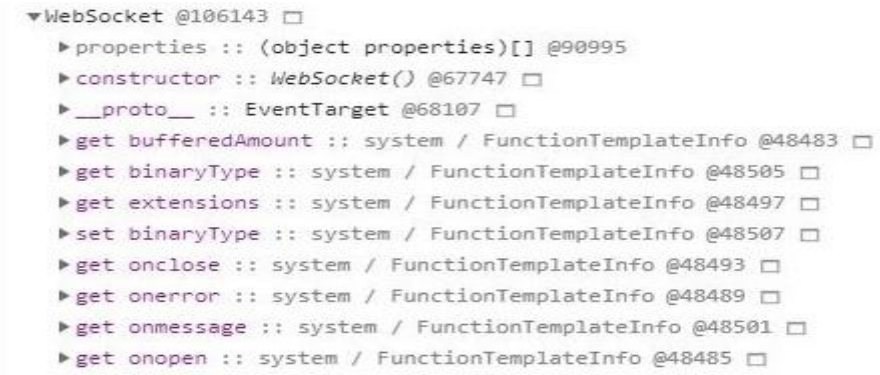
Zahtjev koji je zadnji po redu, te kojem na mjestu gdje bi mu moralo biti vrijeme piše *pending*, predstavlja vezu koju je otvorio WebSocket. Ona je perzistentna i nije ograničena pojedinačnim zahtjevima tako da na ovoj razini nije moguće vidjeti ni poruke koje prolaze tom vezom ni njihov redoslijed. Klikom na taj zahtjev možemo vidjeti poruke koje prolaze ovom vezom:



Slika 16: Primjer poruka koje se šalju preko WebSocket protokola

Na ovoj slici vidi se niz poruka koji je stigao do klijenta aplikacije čiji sam promet snimio uz pomoć alata unutar Web preglednika. Svaka poruka ima svoj tip i „metu“ (eng. *target*) koja opisuje što zapravo poruka predstavlja, npr. da li je klijent poslao ili primio poruku. Unutar argumenata same poruke možemo vidjeti podatke koji se porukom šalju. Zanimljivo je pogledati prvu poruku kojom klijent šalje podprotokol te kao vrijednost navodi JSON koji se često koristi unutar Web aplikacija te se zbog toga u SignalR biblioteci koristi kao svojevrsni

standard. Sadržaj svih poruka koje ovaj klijent prima ili šalje je u dogovorenom JSON obliku jer sam unutar poslužitelju postavio da se koristi prvi navedeni komunikacijski podprotokol. Također se mogu vidjeti i periodičke prazne poruke koje su se mogle vidjeti i kod SSE komunikacijske tehnike, a koje se koriste da se veza održi živom te zbog internih razloga SignalR biblioteke koji nas u ovom kontekstu ne zanimaju. Ukoliko napravimo trenutnu sliku memorije Web aplikacije (eng. *snapshot*), WebSocket se može vidjeti unutar memorije sa svim svojstvima i događajima opisanima prije.



Slika 17: WebSocket objekt kakav je vidljiv u snapshot-u memorije Web aplikacije

4.5. Ostale komunikacijske tehnike

Danas sve više Web aplikacija prelazi na korištenje WebSocket protokola, no postoji nekoliko alternativa koje se mogu koristiti osim navedenih i opisanih komunikacijskih tehnika. Postoji cijeli niz tehnika koje su zapravo modificirane komunikacijske tehnike koje su već objašnjene unutar ovog rada. To se uglavnom odnosi na dugi polling sa nekom posebnom logikom, npr. da nekoliko objekata zasebno šalju zahtjeve prema poslužitelju, da se zahtjevi šalju periodički, ali i kao rezultat nekih programiranih događaja na klijentskoj strani, itd. Forever Frame također ima dobar dio modifikacija koje se koriste što i ne čudi s obzirom na prirodu te komunikacijske tehnike kao kombinacijom mnogih drugih tehnologija. Protokol koji se također često koristi kod izrade aplikacija u realnom vremenu je WebRTC protokol. Poput WebSocket protokola, on je također prihvaćen i standardiziran od strane W3C konzorcija te je dostupan u obliku jednostavnih sučelja za pomoć u izradi Web aplikacija (W3C (WebRTC dokumentacija), 2017). Razlog zbog kojeg ga se često ne može naći u raspravama o Web aplikacijama u realnom vremenu je ta da se on uglavnom koristi u druge svrhe, primarno kod mobilnih aplikacija i IoT rješenja zbog čega je WebSocket protokol, koji je više fokusiran na Web aplikacije, preuzeo prednost u domeni Web aplikacija u realnom vremenu. Nadalje, WebRTC implementacije često kombiniraju WebRTC protokol sa ovdje opisanim tehnologijama poput WebSocket protokola čime se te tehnologije još više šire. WebRTC je podržan od strane velike većine modernih Web preglednika tako da će se njegovo korištenje sigurno samo povećavati.

5. Okviri i biblioteke za razvoj

Prije nego počnem sa uvodom u SignalR i popratnim programskim primjerom, želio bih stvoriti malo širu sliku o razvoju Web aplikacija u realnom vremenu nadovezujući se na obrađenu teoriju. Web i aplikacije na Web-u od samih početaka imaju vrlo velik broj entuzijasta koji rade na unaprjeđenju, standardizaciji i prihvaćanju tehnologija koje bi mogle poboljšati Web. Situacija nije drugačija ni kod Web aplikacija u realnom vremenu, pa tako postoji veliki broj ljudi koji vide potencijal u ovakvim tehnologijama i žele raditi sa njima. Na kraju krajeva, ovaj je rad napisan upravo zbog entuzijazma prema ovakvim tehnologijama i željom da kroz ovaj rad pobliže proučim i opišem rad sa takvom vrstom aplikacija. Stoga ne čudi da danas postoji velik broj biblioteka, modula i razvojnih okvira kojima je cilj olakšati izradu ovakvih aplikacija (Hempel, 2016), te sam odabrao nekoliko njih koje želim ukratko opisati i predstaviti. Kao što sam već i prije spominjao, WebSocket protokol drži se kao najbolja tehnologija kod izrade Web aplikacija u realnom vremenu, tako da veliki broj biblioteka zapravo samo olakšava rad sa tim protokolom. Te biblioteke izrađene su od strane razvojnih inženjera sa raznolikim obrazovanjem, pozadinom i ciljevima, tako da se neke fokusiraju na olakšavanje određene funkcionalnosti dok druge samo žele dovesti i olakšati korištenje WebSocket protokola u drugim programskih jezicima. Kod njih će biti vrlo lako uočiti sličnosti i razlike jer uglavnom samo pružaju još jednu razinu apstrakcije nad WebSocket protokolom koji je sam po sebi dosta intuitivan za korištenje. Mnoga svojstva i metode objekata koje se mogu naći u ovim bibliotekama mogu se vrlo lako povezati sa svojstvima i metodama opisanima u dijelu rada koji proučava WebSocket protokol.

Za izradu primjera odabrao sa biblioteke SockJS, Socket.IO i WebRTC koje rade u JavaScript programskom jeziku, te biblioteke za korištenje WebSocket protokola u aplikacijama izrađenima u programskim jezicima Python i Java. Pokušao sam odabrati nekoliko često korištenih programskih jezika i pripadnih biblioteka koje se nude kao svojevrsna alternativa SignalR biblioteci, no izbjegavao sam okvire za razvoj aplikacija poput Meteor okvira za koje sam prosudio da se previše fokusiraju na druge aspekte razvoja Web aplikacija poput ponovne iskoristivosti komponenti ili baza podataka ugrađenih u Web stranicu, a premalo na funkcionalnosti koje rade u realnom vremenu pa se ne bi previše uklapali u ovaj rad. Svaki primjer predstavlja istu aplikaciju, a to je jednostavna aplikacija za čavrljanje, tj. *chat* aplikacija, u kojoj korisnik unosi korisničko ime i poruku koju zatim šalje drugim korisnicima. Svaki primjer ima isto korisničko sučelje jer ono ovdje uopće nije bitno, već sam se želio fokusirati na sličnosti i razlike između implementacija ovih biblioteka da se mogu vidjeti prednosti i mane korištenja svake od njih u slučaju u kojem se one koriste sa istim ciljem.

Korisničko sučelje ove aplikacije je jednostavna HTML stranica koju dijele svi uskoro objašnjeni primjeri. Svaka od njih ima pripadnu CSS datoteku sa vrlo jednostavnim stilskim uputama koje čine korisničko sučelje lakšim za korištenje u svrhu demonstracije biblioteka. Jedine razlike između HTML stranica kod različitih primjera su JavaScript biblioteke koje se koriste u primjeru, a jedina biblioteka koju dijele je jQuery biblioteka koju koristim za pristup zasebnim elementima HTML stranice. HTML samog korisničkog sučelja, tj. sadržaj *body* oznake HTML stranice isti je kod svih primjera, a izgleda ovako:

```
<textarea readonly id="chat" class="chat-area"></textarea><br/>
Nadimak: <input id="username" class="username-area" type="text"
                maxlength="20" placeholder="Unesite nadimak"/><br/>
Poruka:<br/><textarea id="message" class="message-area"
                    placeholder="Unesite poruku"></textarea><br/>
<input type="button" value="Posalji poruku" onclick="sendMessage()"/>
```

Korisničko sučelje sastoji se od područja za prikaz poruka sa ID-em *chat*, dva polja za unos podataka, a to su korisničko sučelje i sama poruka, sa ID-evima *username* i *message* te gumbom koji služi sa slanje same poruke, a klikom na koji se pokreće JavaScript metoda *sendMessage()* koja je posebno opisana u svakom primjeru. Takvo korisničko sučelje na kraju izgleda ovako:

Slika 18: Korisničko sučelje koje se koristi kod programskih primjera

Svaki od primjera pridodan je kao dodatak ovom radu, a primjeri su sadržani u mapi sa imenom biblioteke ili programskog jezika koji se koristi. Svaki od primjera u svom korijenskom direktoriju ima datoteku sa nazivom *UPUTE* u kojem se nalaze upute za pokretanje svakog programskog primjera te koraci koje je potrebno poduzeti prije nego se oni mogu pokrenuti.

5.1. SockJS

SockJS je JavaScript biblioteka koja omogućuje instanciranje WebSocket objekta koji zatim koristimo za komunikaciju između klijenta i poslužitelja. SockJS je jedna od onih biblioteka koje pružaju dodatnu razinu apstrakcije kada radimo sa WebSocket protokolom za olakšavanje njegovog korištenja, no nudi i neke druge mogućnosti. Slično kao što to radi i SignalR, SockJS nudi mogućnost odabira drugačije komunikacijske tehnike ukoliko WebSocket protokol nije dostupan zbog ograničenja Web preglednika. Unatoč tome što najbolje radi sa Web preglednicima koji podržavaju WebSocket protokol, može koristiti i dugi polling i neke druge komunikacijske tehnike ukoliko se radi sa starijim verzijama Web preglednika ili ako se dogodi greška kod korištenja željene komunikacijske tehnike. Prednost ove biblioteke je ta da razvojni inženjer o tome ne treba razmišljati, već se sve događa kroz objekt koji predstavlja WebSocket, bez obzira da li se uistinu koristi WebSocket protokol ili nešto sasvim drugo (*sockjs* (GitHub korisnik), 2019).

Primjer je izrađen uz pomoć raznih uputa i gotovih primjera koji su izradili korisnici ove biblioteke, no uglavnom je baziran na uputama i popratnoj aplikaciji pronađenoj na TruondTx blogu (Trường, 2014). Ovaj primjer sam, kao i sve ostale primjere, razdvojio na klijentski i poslužiteljski dio aplikacije. Poslužiteljski dio aplikacije sastoji se od samo jedne datoteke, a to je *server.js*. To je JavaScript datoteka koja sadrži logiku za primanje poruka od klijenata i slanje poruka prema klijentima. Ono što je važno spomenuti je da SockJS koristi Node.js biblioteku kod pokretanja poslužitelja. Node.js je biblioteka koja omogućava korištenje JavaScript skripti na poslužiteljskoj strani, te tako omogućuje izradu i pokretanje poslužitelja koristeći isključivo JavaScript, barem što se tiče razvojnog inženjera (Patel, 2018). Za pokretanje poslužitelja koristi se sljedeći isječak koda:

```
var http = require('http');
var sockjs = require('sockjs');
var sockJsServer = sockjs.createServer();
var port = 7000;
var server = http.createServer();
sockJsServer.installHandlers(server, {prefix:'/sockjsserver'});
server.listen(port, 'localhost');
```

Prve dvije linije koda samo su dohvaćanje potrebnih modula i njihovo instanciranje, a onaj koji nas najviše zanima je *sockjs* modul koji spremamo u istoimenu varijablu. Uz pomoć njega kreiramo objekt koji predstavlja SockJS poslužitelj njegovom metodom *createServer()*. Tom poslužitelju potreban je HTTP poslužitelj koji ima mogućnosti primati HTTP zahtjeve i slati pripadajuće odgovore, pa njega kreiramo sa *http* modulom i spremamo ga u istoimenu

varijablu. Taj HTTP poslužitelj zatim se kod prije kreiranog SockJS poslužitelja registrira kao rukovoditelj spomenutih događaja, te mu se kao jedino svojstvo kao drugi argument dodjeljuje prefiks koji će nam koristiti kod spajanja na njega sa klijentske strane. HTTP poslužitelj zatim se pokreće na zadanoj adresi i portu. U svakom programskom primjeru kao adresu ću koristiti *localhost* ili pripadnu IPv4 adresu, a kao port ću koristiti port 7000 ukoliko ću biti u mogućnosti sam zadati port koji želim koristiti. Poslužitelj zatim počinje sa slušanjem za zahtjeve.

Ovakav poslužitelj sada je pokrenut, no sam od sebe ne radi ništa, već moramo dodati logiku na događaje koji stižu na WebSocket kako bi aplikacija za čavrljanje funkcionirala. Ono što želim je da ako jedan klijent koji je spojen na poslužitelj pošalje poruku prema poslužitelju, da poslužitelj tada tu poruku proslijedi svim klijentima koji su trenutno na njega spojeni. To možemo postići tako da registriramo logiku za rukovođenje događaja koja će se pokrenuti kada se novi klijent spoji na poslužitelj, te u slučaju kada od nekog klijenta stigne poruka. Logiku za rukovođenje tim događajima možemo vidjeti u sljedećem isječku koda:

```
sockJsServer.on('connection', function(connection) {
	clients[connection.id] = connection;

	connection.on('data', function(message) {
		broadcastMessage(message);
	});

	connection.on('close', function() {
		delete clients[connection.id];
	});
});
```

Prvi događaj koji hvatamo je događaj *connection*, koji se pokreće kada se novi korisnik spoji na poslužitelj. U tom slučaju klijenta, koji je reprezentiran kroz *connection* varijablu, spremamo u prije pripremljenu kolekciju tako da mu možemo pristupiti u bilo kojem trenutku. Zatim tom klijentu registriramo logiku za rukovođenje događaja u slučaju kada do njega dođe neka poruka, tj. ako se dogodi događaj *data*. Treba napomenuti da se prvi spomenuti događaj odnosi na događaj SockJS poslužitelja, dok sada govorimo o događaju klijenta. Ukoliko do poslužitelja stigne poruka, on tu poruku prosljeđuje svim drugim klijentima korištenjem funkcije *broadcastMessage()* čiji kod izgleda ovako:

```
function broadcastMessage(message) {
	for (var client in clients) {
		clients[client].write(message);
	}
}
```

Ova metoda samo iterira kroz sve klijente u kolekciji klijenata, te nad njima poziva metodu *write()* koja klijentu šalje poruku koju zadamo u argumentu te metode. O tome kako tu poruku pripremiti za čitanje ili za prikaz na HTML stranici poslužitelj ne brine, već o tome brigu vodi klijentski dio aplikacije. Klijentski dio aplikacije puno je jednostavniji od poslužiteljskog, a nalazi se datoteci *index.js* koja zajedno sa pripadnim HTML i CSS datotekama čini klijentski dio aplikacije. Njezin kod izgleda ovako:

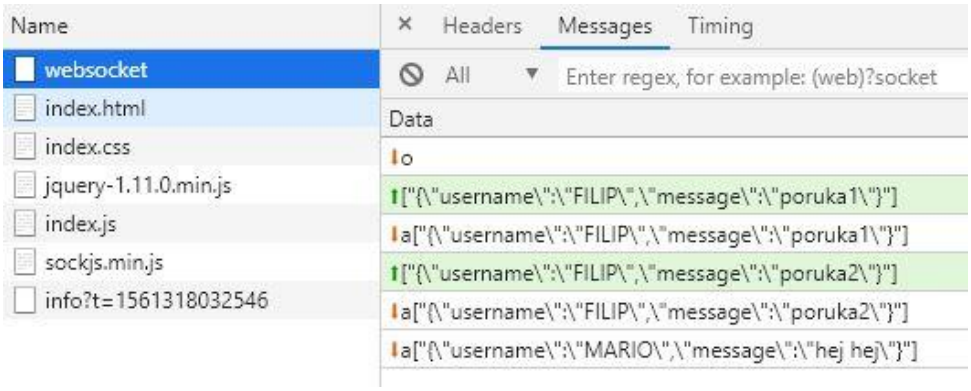
```
var websocket = new SockJS('http://localhost:7000/sockjsserver');

websocket.onmessage = function(message) {
	var content = JSON.parse(message.data);

	$('#chat').val(function(i, text) {
		return text + content.username + ': ' + content.message + '\n';
	});
};

function sendMessage() {
	var username = $('#username').val().substring(0, 20).toUpperCase();
	var message = $('#message').val();

	var chatMessage = {
		username: username,
		message: message
	};

	websocket.send(JSON.stringify(chatMessage));
	$('#message').val('');
}
```

Kreiranje veze prema poslužitelju preko WebSocket protokola ili koje god komunikacijske tehnike koja se trenutno koristi radi se instanciranjem objekta tipa SockJS, kojem kao argument dajemo adresu poslužitelja u obliku kombinacije osnovne adrese poslužitelja, porta koji se koristi i prefiksa koji smo definirali na poslužiteljskoj strani aplikacije. Varijabla *socket* zatim se može koristiti kao WebSocket bez obzira da li na kraju krajeva on to predstavlja ili ne. Funkcija *sendMessage()* služi nam za slanje poruka prema poslužitelju, a ona radi tako da sa HTML stranice uz pomoć jQuery biblioteke preuzme vrijednosti polja u kojima se nalaze korisničko ime i poruka koju želimo poslati, te ih zatim zapakiram u posebnu strukturu. Odlučio sam kao format poruke odabrati JSON kako bih si olakšao posao oko slanja poruke i njihove interpretacije kod primanja, tako da kod rada koristim klasu JSON i njezine

dvije statičke metode. Prva takva metoda je *stringify()* koja JSON objekt ili sličnu strukturu, kao ovakvu kakvu ovdje koristim, pretvara u string koji se lakše može slati kao poruka. Druga metoda je *parse()* koja uzima string te ga pretvara u objekt JSON formata koji se zatim može interpretirati kao JSON. Ovdje navedenu strukturu pretvaram u string te njega kao poruku šaljem uz pomoć metode *send()* SockJS objekta. Nakon toga element na korisničkom sučelju u koji se upisivala poruka ispraznim kako bi se ubrzao unos druge poruke.

Za primanje poruke registriram rukovoditelja događaja *onmessage* koji je opisan u SockJS klasi. U slučaju kada do kreiranog WebSocket-a stigne poruka, iz nje će se izvaditi podaci sadržani u *data* svojstvu te uz pomoć *parse()* metode pretvoriti u JSON objekt. Korisničko ime i poruka sadržani u tom objektu zatim će se uz pomoć jQuery biblioteke ispisati na korisničko sučelje u za to predviđeno područje. Korisničko ime pretvaram u velika slova te ga skraćujem na dvadeset znakova u slučaju vrlo dugog korisničkog imena, te korisničko ime i poruku odvajam dvotočkom i praznim znakom radi čitljivosti nakon čega se prebacujem u novi red zbog zaprimanja sljedeće poruke. Logiku za primanje i slanje poruka ovdje sam malo pobliže opisao jer sam već u početku imao plan koristiti JSON format kod slanja i primanja poruka te sam ih zamislio prikazivati na korisničkom sučelju na isti način, pa stoga svaki primjer koristi barem dobar dio prikazanog koda koji već neću prikazivati. Ono što hoću prikazati i naglasiti su načini kreiranja veze prema poslužitelju i same metode za slanje i primanje podataka pošto su to jedine razlike između biblioteka koje sam ovdje odlučio demonstrirati kroz kratke primjere.

Ono što još mogu prikazati je kako se ova aplikacija ponaša kada radi. Nakon pokretanja poslužitelja, možemo otvoriti nekoliko instanci klijenta pokretanjem *index.html* datoteke. Ukoliko pošaljemo nekoliko poruka koristeći aplikaciju te zatim pogledamo *Network* karticu alata za razvojne inženjere unutar Web preglednika, možemo vidjeti ne samo WebSocket vezu koja je tamo prikazana, već i poruke koje su se razmjenjivale između klijenta i poslužitelja. Svaka poruka ima sadržaj JSON oblika, te ako je klijent kojeg gledamo poslao određenu poruku, vidljivo je i slanje i primanje te poruke. Sve ovo je vidljivo na sljedećoj slici:



Slika 19: Razmjena poruka preko WebSocket protokola koristeći SockJS biblioteku

5.2. Socket.IO

Socket.IO još je jedna biblioteka koja se koristi u JavaScript programskom jeziku te uživa veliku popularnost kod današnjih razvojnih inženjera zbog lakoće korištenja i mogućnosti koje nudi. Odabrao sam ovu biblioteku kao prikaz sličnosti između mnogih biblioteka na ovom području, jer ova biblioteka, poput već prikazane SockJS biblioteke, također nudi dodatnu razinu apstrakcije iznad samog WebSocket protokola te samim time olakšava njegovo korištenje. Poput SockJS-a, i ova biblioteka nudi dodatne opcije u slučaju da WebSocket protokol zbog nekog razloga ne radi ili Web preglednik ne podržava njegovo korištenje, a u ovu svrhu uglavnom se koristi dugi polling. Ona također koristi Node.js kao pomoć na poslužiteljskoj strani, tako da je i ovdje poslužitelj napisan isključivo u JavaScript jeziku, barem što se tiče razvojnog inženjera koji koristi biblioteku. Razlika u odnosu na SockJS je lakše korištenje te svojevrsna standardizacija i veća prihvaćenost (Kelleher, 2014).

Ovaj primjer također se temelji na originalnoj ideji jednostavne aplikacije za čavrljanje te je ostvaren kroz proučavanje raznih uputa i primjera dostupnih na Web-u, no uglavnom se temelji na službenoj dokumentaciji Socket.IO biblioteke (Socket.IO dokumentacija, 2019). Korištenje ove biblioteke počinje izradom *package.json* datoteke u koju navodimo biblioteke koje želimo koristiti, te pozivanjem naredbe za korištenje *npm* registra programskih rješenja, jednog od najvećih i najčešće korištenih registara takve vrste u svijetu koja se često koristi u razvoju Web aplikacija za dijeljenje i preuzimanje potrebnih biblioteka. *npm* se koristi uz pomoć istoimene naredbe koja se poziva u CMD-u (skraćeno od eng. *command prompt*, tj. konzole Windows operacijskog sustava), a dolazi sa instalacijom Node.js biblioteke tako da je već imamo spremnu od prethodnog primjera. Izvršavanje te naredbe također priprema mapu u kojoj je naredba izvršena za to da bude korijenski direktorij projekta, te ćemo je koristiti kao takvu.

Logika će i ovog puta biti smještena u datoteku *server.js*, no klijentski dio aplikacije bit će razdvojen. HTML datoteka koja sadrži korisničko sučelje aplikacije bit će smještena u istom direktoriju kao i *server.js* datoteka, no zbog korištenja modula *express* kod poslužiteljske strane aplikacije, pripadne datoteke nije moguće smjestiti u taj direktorij. Potrebno je neku mapu postaviti takvom da poslužitelj drži sve datoteke sadržane u njoj kao statične, pa pristup tim datotekama nije zabranjen i one se mogu koristiti. Stoga sam napravio posebnu mapu naziv *public* u kojoj držim sve datoteke koje koristi *index.html*, tj. njezine pripadne CSS i JS datoteke. Za razliku od prethodnog primjera, ovdje se instanca klijenta ne otvara otvaranjem same HTML datoteke, već poslužitelj servira datoteku na zadanu adresu te je instancu klijenta moguće otvoriti posjetom te adrese. Implementacija poslužitelja ovaj puta je nešto malo jednostavnija, a može se vidjeti u sljedećem isječku koda:

```
var express = require('express');
var handler = new express();
var http = require('http').Server(handler);
var io = require('socket.io')(http);
var port = 7000;

handler.use(express.static('public'));

handler.get('/', function(request, response) {
	response.sendFile(__dirname + '/index.html');
});

io.on('connection', function(connection) {
	connection.on('message', function(message) {
		io.emit('message', message);
	});
});

http.listen(port, function() {
	console.log('Posluzitelj pokrenut na portu ' + port);
});
```

Kao i kod prethodnog primjera, implementacija poslužitelja započinje instanciranjem raznih modula i biblioteka potrebnih za rad. Ovdje je razlika modul *express*, višenamjenska biblioteka koju koristi Node.js kod serviranja datoteka na poslužitelj i kao rukovoditelj kod nekih događaja (TutorialsPoint, 2014). Ovdje se *express* klasa koristi kod instanciranja HTTP servera, koji se zatim koristi kod instanciranja *io* objekta kojim ćemo stvoriti vezu između klijenta i poslužitelja. Nakon instanciranja potrebnih objekata mapu *public* proglasimo statičnom da bi se datoteke unutar nje mogle koristiti od strane *index.html* datoteke koju serviramo na poslužitelj uz pomoć iste *handler* varijable koja je tipa *express*, no to je već detaljnije bilo pojašnjeno. Sama razmjena poruka riješena je uvođenjem rukovoditelja za *connection* događaj koji smo već vidjeli u prethodnom primjeru. Ovdje klijente nije potrebno spremati u posebnu kolekciju kao što je to bilo potrebno napraviti kada smo koristili SockJS biblioteku, već možemo samo registrirati rukovoditelja za događaj kada klijent pošalje poruku. U tom slučaju dobivena poruka se šalje svim klijentima uz pomoć metode *emit()* koja je definirana u Socket.IO objektu. Nakon definiranja sve logike koja nam je u ovom primjeru potrebna, poslužitelj možemo pokrenuti. To se radi uz pomoć HTTP poslužitelja, tj. njegovom metodom *listen()*, kojoj je potrebno proslijediti definirani broj porta koji želimo koristiti, te

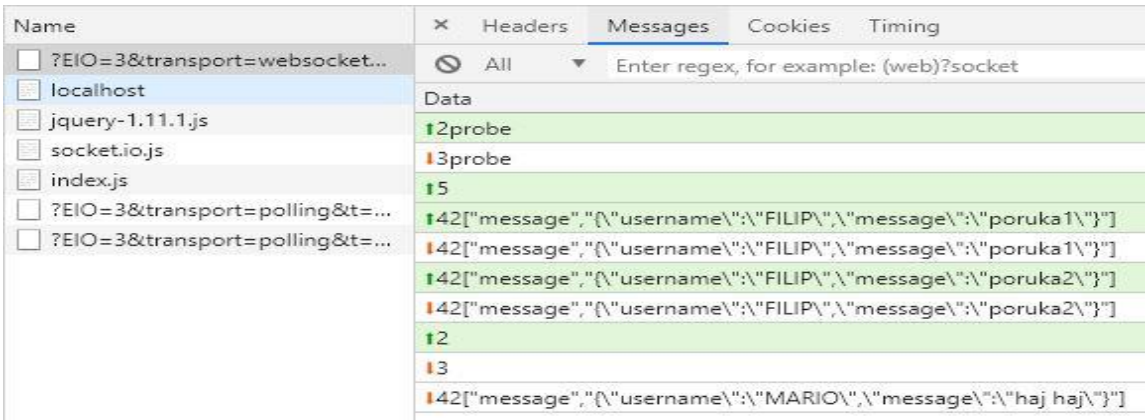
eventualne akcije koje želimo izvršiti kada poslužitelj počne slušati pristigle zahtjeve. Ovdje sam samo postavio ispis poruke o početku rada poslužitelja kao indikator da je poslužitelj pravilno započeo sa radom.

Što se tiče logike kod klijenta, ovdje nema prevelikih promjena s obzirom na prethodni primjer, što služi kao dokaz o sličnostima između ovih vrsta biblioteka. Prva promjena koja se može uočiti je u prvoj liniji koda sadržanoj u datoteci *index.js*, a glasi ovako:

```
var websocket = io();
```

Ova linija koda stvara objekt koji je zaslužan za povezivanje sa poslužiteljem. Za razliku od SockJS biblioteke gdje je potrebno navesti adresu, port i prefiks kako bismo stvorili vezu između klijenta i poslužitelja, zbog serviranja klijenta to nije potrebno te je samo instanciranje ovog objekta sasvim dovoljno. Sljedeća razlika nalazi se kod slanja poruka prema poslužitelju gdje se ne koristi metoda *send()*, već metoda *emit()*, gdje je osim same poruke potrebno navesti i tip poruke koja se šalje, što je u našem slučaju *message*. Ovakav tip poruke već je naveden na poslužiteljskoj strani te se dolazak takve poruke hvata kao dio logike ove aplikacije. Zadnja razlika je u tome da se, kod primanja poruke, ne hvata događaj *onmessage* već se za registraciju događaja koristi svojstvo *on* varijable *socket* za koji možemo povezati tip poruke *message* te na taj način primati poruke. Ostatak koda koji se odnosi na interpretaciju poruke i njezino prikazivanje na korisničkom sučelju, te preuzimanje poruke sa korisničkog sučelja i njezino „pakiranje“ prije slanja prema poslužitelju je potpuno jednako kodu prethodnog primjera.

Kao i kod primjera izrađenog uz pomoć SockJS biblioteke, i ovdje se na klijentskoj strani instancira WebSocket objekt ukoliko mu to Web preglednik omogućava, te je moguće pregledati poruke koje se razmjenjuju između klijenta i poslužitelja. Pogledom na *Network* karticu alata za razvojne inženjere Web preglednika moguće je vidjeti upravo te poruke, kao i sve trenutno aktivne veze koje aplikacija koristi, uključujući i WebSocket. Sve navedeno vidljivo je na sljedećoj slici:



Slika 20: Razmjena poruka preko WebSocket protokola koristeći Socket.IO biblioteku

5.3. WebRTC

WebRTC je jedna vrlo zanimljiva biblioteka o čijoj bi se primjeni mogao napisati cijeli rad poput ovog i koja svakim danom raste u njezinim mogućnostima i popularnosti, no ovdje sam je odlučio demonstrirati kroz kratki primjer jer smatram da nijedna rasprava o Web aplikacijama u realnom vremenu ne bi bila potpuna bez spomena ove biblioteke. Samo ime biblioteke je kratica za *Web Real Time Communication*, tj. komunikacija u realnom vremenu preko Web-a. WebRTC nudi vlastiti protokol za komunikaciju u realnom vremenu ne samo za Web aplikacije koje koriste Web preglednike, već i za nativne mobilne aplikacije kroz mnogo biblioteka i raznih implementacija u raznim programskim jezicima. WebRTC tehnologija već je provjerena i standardizirana od strane W3C konzorcija te je u širokoj upotrebi u Web aplikacijama i IoT rješenjima. WebRTC je popularna biblioteka kod aplikacija koje u realnom vremenu prenose multimedijski sadržaj poput zvuka ili video materijala pa je postala popularan izbor kod aplikacija za video konferencije ili za pomoć na daljinu (eng. *remote assistance*) (BloogGeek, 2017). Ovdje neću preduboko ulaziti u mogućnosti korištenja WebRTC biblioteke jer ona nije fokus ovog rada, već predstaviti jednu od njenih JavaScript implementacija.

U izradi ovog primjera odlučio sam koristiti SimpleWebRTC, JavaScript biblioteku koja olakšava rad sa WebRTC protokolom u JavaScript-u na sličan način na koji prethodne dvije opisane biblioteke olakšavaju rad sa WebSocket protokolom. Primjer je napravljen uz pomoć raznih uputa i primjera nađenih na Web-u, no uglavnom je baziran prema uputama SitePoint Web stranice (Wanyoike, 2018). Ovdje također počinjemo stvaranjem *package.json* datoteke te pokretanjem naredbe iz CMD-a kojim dohvaćamo sve potrebne biblioteke i stvaramo novi projekt u odabranom direktoriju. Nakon toga možemo stvoriti datoteku *server.js* koja će služiti kao poslužitelj kojeg ćemo pokretati uz pomoć biblioteke Node.js, a kod izgleda ovako:

```
var express = require('express');
var handler = express();
var port = 7000;

handler.use(express.static('client'));

handler.use('/scripts', express.static(`${__dirname}/node_modules/`));

handler.use((request, response) =>
      res.sendFile(`${__dirname}/client/index.html`));

handler.listen(port, () => {
      console.info('Posluzitelj pokrenut na portu ' + port);
});
```

Započinjemo dohvaćanjem potrebnih modula i njihovim spremanjem u varijable, kao i odabirom željenog porta. Nakon toga radimo ono što se radilo i u prethodnom primjeru, a to je označavanje svih datoteka određenog direktorija kao statičnima da bi se oni mogli učitati na stranici koju serviramo na poslužitelju. Ovdje sam taj direktorij nazvao *client* jer se u njemu sada nalazi i HTML datoteka zajedno sa svojim popratnim CSS i JS datotekama što predstavlja cjelinu implementacije klijenta. Ukoliko kod pogledamo dalje, uporabom istog *express* rukovoditelja određujemo direktorij na kojem se nalaze skripte kako bi se one mogle koristiti na klijentskoj strani. Nadalje, preusmjeravamo sav promet prema klijentu predstavljenom *index.html* datotekom. Zadnji korak je pokretanje poslužitelja kojem dajemo željeni port, te akcije koje želimo da se izvrše kada se poslužitelj pokrene, što je u ovom slučaju samo slanje poruke o tome da je poslužitelj uspješno započeo s radom.

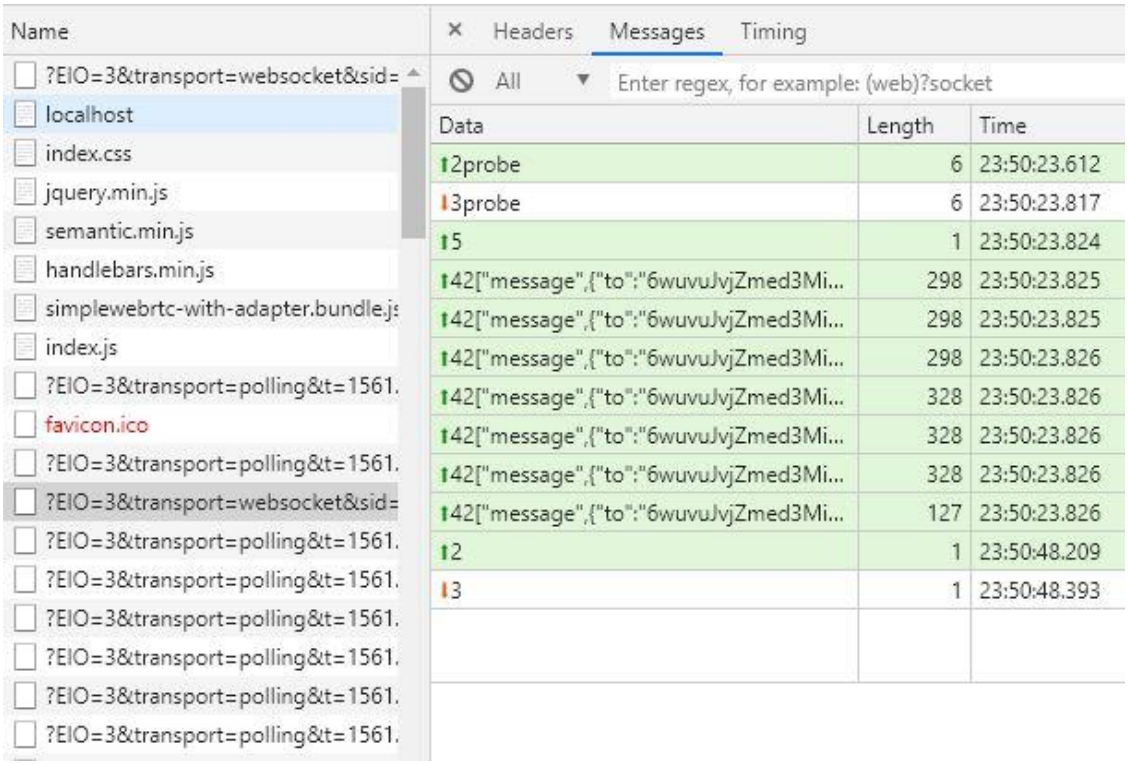
U implementaciji poslužitelja možemo povući mnogo paralela sa implementacijama poslužitelja u prethodnim primjerima što vuče na zaključak da je i ova biblioteka vrlo slična prethodnima, no velike razlike vidljive su tek na klijentskoj strani. Logika koja upravlja ponašanjem klijenta u ovoj aplikaciji nalazi se u *index.js* datoteci, a razlike se mogu vidjeti već kod instanciranja objekta koji otvara vezu između klijenta i poslužitelja, a izgleda ovako:

```
var webrtc = new SimpleWebRTC({ });
webrtc.joinRoom('default');
```

Ovdje instanciramo objekt tipa SimpleWebRTC te ga spremamo u kreiranu varijablu. Kao argument mu dajemo praznu strukturu u koju bi inače spremali dodatne opcije poput kanala za prijenos video i audio sadržaja, no te opcije u našem primjeru nisu potrebne pa struktura ostaje prazna. Ako klijenta želimo spojiti sa drugim klijentima i omogućiti njihovu međusobnu komunikaciju, oni se moraju nalaziti u istoj sobi. Sobe je moguće kreirati, brisati i premještati klijenta iz sobe u sobu po želji, no u ovom primjeru želim omogućiti čavrljanje između svih spojenih klijenata pa kreiram sobu imena *default*, koja se kreira samo ako ona već ne postoji, te odmah kao klijent ulazim u tu sobu. Time je veza stvorena te ostvareni svi preduvjeti za komunikaciju između klijenta i poslužitelja. Ostatak implementacije klijenta vrlo je sličan onome što je već viđeno u prethodna dva programska primjera, no ima razlika koje je potrebno naglasiti. Za slanje poruka preko kreiranog SimpleWebRTC objekta koristim metodu *sendToAll()*. Ona kao argument prima tip poruke kao i kod prethodnog primjera, te objekt koji šaljem koji je u ovom slučaju JSON u obliku stringa kao i u prethodnim primjerima. Još jedna razlika u odnosu na prethodne primjere je ta da poslužitelj neće klijentu koji je poslao poruku poslati tu poruku natrag, već je potrebno samostalno dodati vlastitu poruku na korisničko sučelje jer bi u suprotnom bile vidljive samo poruke koje nam šalju drugi klijenti. Korištena metoda *sendToAll()* samo je jedna od mnogih metoda te je stoga moguće poruku poslati npr. samo određenom korisniku. Primanje poruke odrađeno je na vrlo sličan način kao i kod

prethodnih primjera, kroz registriranje rukovoditelja događaja. Ovdje se radi o *message* događaju kojem pridodajemo rukovoditelja događaja preko *connection.on* svojstva SimpleWebRTC klase. Ovdje je potrebno pažljivo pregledati tip poruke koji stiže do klijenta kako bismo prikazali samo poruke koje želimo, a čiji smo tip odredili kod slanja poruke. Aplikaciju pokrećemo tako da se pozicioniramo u direktorij u kojem se nalazi *server.js* datoteka te u CMD-u pokrenemo naredbu „*npm start*“. Zasebne instance klijenta zatim je moguće pokrenuti odlaskom na pripadnu Web adresu.

WebRTC kao tehnologija koristi razne druge tehnologije i komunikacijske tehnike u svojem radu osim WebRTC protokola, a biblioteke kojima je cilj olakšati rad sa WebRTC protokolom mogu ih koristiti još više. To rezultira time da, ukoliko bacimo pogled na *Network* karticu u alatu za razvojne inženjere u Web pregledniku, nećemo vidjeti jednu WebSocket vezu koja se koristi za komunikaciju kao što je to bio slučaj kod prijašnjih programskih primjera. Ono što ćemo vidjeti je velik broj veza koje koriste sve od dugog pollinga do WebSocket protokola čije je pokretanje, implementacija i svrha skrivena od razvojnog inženjera kroz biblioteke poput SimpleWebRTC. Kao jedna od tehnologija koje se bave prijenosom multimedijskog sadržaja u realnom vremenu preko različitih platformi, te koja je u prošlosti imala svojih problema sa performansama i sigurnošću (*gwarser* (GitHub korisnik), 2019),velik broj veza odrađuje velik broj zadataka kojima se razvojni inženjer ne mora brinuti, no njihov otisak je vidljiv u pregledu rada aplikacije posebnim alatima. Primjer snimke prometa u ovoj aplikaciji u slučaju komunikacije između dva klijenta vidljiv je na sljedećoj slici:



Slika 21: Razmjena poruka u aplikaciji koja koristi SimpleWebRTC biblioteku

5.4. Python WebSocket implementacija

Zbog toga što su prethodni primjeri svi pretežno koristili JavaScript i na klijentskoj i poslužiteljskoj strani, za sljedeća dva primjera odlučio sam uzeti dva vrlo poznata programska jezika koji se često koriste u izradi Web aplikacija. Zbog popularnosti i mogućnosti WebSocket protokola vrlo su brzo kreirane biblioteke za njihovo korištenje u velikom broju programskih jezika, pa sam ovdje odlučio predstaviti dva takva primjera. Prvi primjer izrađen je u programskom jeziku Python. Python je programski jezik koji je na vrlo dobrom glasu zbog jednostavnosti korištenja i održavanja koda kao i blagoj krivulji učenja, što ga čini idealnim za mnoge aspekte Web programiranja sada kada performanse nisu više toliko važne koliko nadogradnja i održavanje različitih dijelova i aspekata Web aplikacija. On se često koristi za pisanje različitih modula i podzadataka na poslužitelju koji je kod većih Web aplikacija uglavnom napisan u nekom programskom jeziku sa boljim performansama, no i Python se može koristiti za pokretanje poslužitelja te sam ga ovdje tako i iskoristio.

Python uživa veliku popularnost pa stoga postoji mnogo različitih biblioteka koje se bave WebSocket protokolom te velik broj uputa i primjera kako ih koristiti koji se mogu naći diljem Web-a, a koje sam koristio u izgradnji ovog primjera, no on se uglavnom temelji na primjeru GitHub korisnika *tuchfarber* (*tuchfarber* (GitHub korisnik), 2017). Ovaj primjer uglavnom se fokusira na poslužiteljsku datoteku te biblioteke koje se koriste na poslužiteljskoj strani, dok se na klijentskoj strani u JavaScript datoteci koristi nativni WebSocket protokol bez ikakve biblioteke koja bi dodavala dodatne razine apstrakcije na njega. Programski kod klijentskog dijela aplikacije zato se praktički ne razlikuje od onog prikazanog u prvom primjeru jer se slanje i primanje poruka odrađuje na identičan način uz pomoć istih metoda. Jedina razlika je u tome što se veza prema poslužitelju ne otvara uz pomoć SockJS objekta kojeg ovdje ne koristimo, već uz pomoć WebSocket objekta koji kao argument prima samo lokaciju na kojoj je WebSocket otvoren na poslužiteljskoj strani aplikacije, a koja se sastoji od adrese i porta koji koristi poslužitelj. Instanciranje takvog objekta prikazano je na sljedećem isječku koda:

```
var websocket = new WebSocket('ws://localhost:7000');
```

Poslužiteljska strana aplikacije koristi nekoliko biblioteka. Biblioteke *os* i *sys* su nativne Python biblioteke koje odrađuju dio posla koji nema veze sa pokretanjem poslužitelja. Biblioteka *asyncio* je važna za spomenuti jer je ona zaslužna za omogućavanje asinkronog načina rada aplikacije kakav nam treba ukoliko želimo raditi sa više korisnika u isto vrijeme, a koristi dobro poznatu *async/await* sintaksu kakva se koristi kod većine biblioteka ove vrste (Python dokumentacija, 2019). Nadalje, koristim *http* biblioteku čija je svrha identična istoimenoj JavaScript biblioteci, a to je uspostavljanje HTTP poslužitelja koji može primati

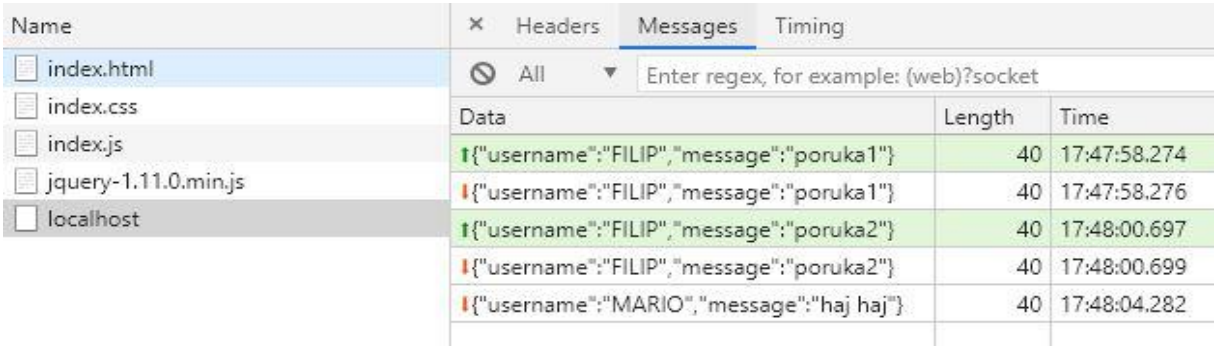
HTTP zahtjeve i slati pripadajuće odgovore. Zadnja biblioteka koju koristim je biblioteka *websocket*s koja omogućuje korištenje WebSocket protokola, a oslanja se na već spomenutu *asyncio* biblioteku (websockets (Python dokumentacija), 2018). Programski kod poslužitelja, izuzevši dohvaćanje biblioteka, izgleda ovako:

```
async def communication_handler(websocket, path):
    global clients
    clients.add(websocket)
    while True:
        try:
            message = await websocket.recv()
            for client in clients:
                await client.send(message)
        except:
            clients.remove(websocket)

sys.stderr = open(os.devnull, 'w')
if __name__ == '__main__':
    print('Posluzitelj pokrenut na portu', port)
    clients = set()
    start_server = websockets.serve(communication_handler, 'localhost',
                                    port)
    asyncio.get_event_loop().run_until_complete(start_server)
    asyncio.get_event_loop().run_forever()
```

Funkcija *communication_handler* obavlja velik dio posla u ovom primjeru jer predstavlja rad poslužitelja. To je asinkrona funkcija koja koristi mogućnosti *asyncio* biblioteke, a počinje dodavanjem klijenta, kojeg predstavlja njegova WebSocket veza, u kolekciju kreiranu za tu svrhu. Ključna riječ *global* ovdje se koristi da bi se mogla koristiti kolekcija koja je inicijalizirana izvan ove funkcije, tj. radi se o globalnoj varijabli. Beskonačna petlja zatim predstavlja rad poslužitelja koji uz pomoć ključne riječi *await* čeka na bilo kakvu poruku bilo kojeg klijenta. Zatim iterira kroz sve korisnike spremljene u kolekciju i šalje im primljenu poruku. Ukoliko neki od korisnika zatvori aplikaciju, tj. jedan klijent se odspoji od poslužitelja i njegova veza nestane, aplikacija generira iznimku koja se hvata i u tom se slučaju taj klijent izbacuje sa liste klijenata. Sljedeća linija koda je ovdje samo da se poruka o toj pogrešci ne ispisuje jer je hvatamo i znamo o čemu je riječ. Sljedeći dio koda predstavlja kod koji se izvršava kada datoteku pokrenemo. Nakon što se korisniku ispiše poruka da je poslužitelj uspješno započeo sa radom, inicijalizira se lista klijenata te se instancira poslužitelj koji koristi WebSocket protokol kao komunikacijsku tehniku, a zadaje mu se opisana *communication_handler* funkcija kao rukovoditelj događaja te adresa i port koji će poslužitelj koristiti u radu. Zadnje dvije linije koda

predstavljaju *asyncio* funkcionalnosti koje omogućuju da će poslužitelj neprestano raditi, tj. barem do trenutka kada ga mi sami ne prekinemo prekidanjem same aplikacije.

Nakon otvaranja nekoliko klijenata i kratkim korištenjem aplikacije, bacio sam pogled na *Network* karticu u alatu za razvojne inženjere u Web pregledniku kako bih se uvjerio da se zaista koristi WebSocket protokol te vidio kako radi sa ovom implementacijom. Prikaz snimljenog prometa prikazuje vezu naziva *localhost* koja predstavlja WebSocket, a to znam jer u zaglavlju te veze mogu pronaći nekoliko spomena WebSocket protokola, no najviše me zanima tip poslužitelja za koji piše da je „*Python/3.7 websockets/7.0*“, što mi odmah pokazuje i to da se koristi trenutno najnovija verzija Python programskog jezika te WebSocket protokola. Pogledom u poruke koje su se prenosile preko te veze mogu vidjeti sve poruke koje je slao klijent kod kojeg pratim mrežni promet, te sve poruke koje je on primio tijekom svog rada. Sve opisano može se vidjeti na sljedećoj slici:



Slika 22: Razmjena poruka WebSocket protokolom kroz njegovu implementaciju u programskom jeziku Python

Ovo je samo jedna od mnogih Python biblioteka koje se mogu koristiti sa istim ciljem. Ovu implementaciju i primjer sam odabrao jer je bila najjednostavnija, a ujedno je prikazala način na koji se WebSocket protokol može efektivno koristiti u izgradnji Web aplikacija u realnom vremenu u programskom jeziku Python. Isprobao sam velik broj biblioteka koje sam odlučio ne prikazati jer se, poput mnogih JavaScript biblioteka koje sam također odlučio ne iskoristiti za potrebe ovog rada, previše fokusiraju na druge aspekte izrade Web aplikacija, a ne toliko na funkcionalnosti u realnom vremenu, no ipak vrijedi ih spomenuti. Autobahn je biblioteka koja je implementirana i u drugim programskim jezicima, no čest je temelj za mnoge biblioteke za izradu aplikacija u realnom vremenu napisanima u Python programskom jeziku. Biblioteke koje sam već obradio u ovom dijelu rada, poput Socket.IO biblioteke, također imaju svoje Python implementacije koje se često koriste (Grinberg, 2018). Nadalje, WebRTC protokol također ima nekoliko popularnih Python implementacija te postoji mnogo uputa o tome kako s njima raditi. Velika većina ovih biblioteka su projekti otvorenog koda te se nadograđuju i poboljšavaju svakim danom, a samim time što su uz to besplatne za korištenje daje im veliku popularnost i jedan su od razloga zašto je Python danas sve prisutniji na Web-u.

5.5. Java WebSocket implementacija

Za zadnji programski primjer prije većeg primjera napravljenog sa SignalR bibliotekom odlučio sam iskoristiti programski jezik Java. Java je jedan od najpopularnijih programskih jezika koji se koristi za sve moguće svrhe, pa tako i za izgradnju Web aplikacija u realnom vremenu, no ujedno je i najveća konkurencija C# programskom jeziku u kojem je napisana SignalR biblioteka. Java je kao objektno orijentirani programski jezik dosta sličan C# programskom jeziku te će se definitivno vidjeti neke sličnosti između ovog primjera i korištenja SignalR biblioteke. Java je jezik koji uživa veliku popularnost u svijetu te se svakodnevno usavršava na vrlo profesionalan način uz mnogo kvalitetne dokumentacije. Unatoč tome što postoji mnogo biblioteka koje omogućuju rad sa WebSocket protokolom i drugim spominjanim komunikacijskim tehnikama, odlučio sam iskoristiti *websocket* biblioteku koja je službeni dio Java Runtime Environment paketa te je vrlo lagana i intuitivna za korištenje. Java je meni osobno vrlo drag programski jezik koji sam susreo mnogo puta te ga koristio i za potrebe obrazovanja i izvan fakulteta, a i već sam ga koristio u suradnji sa WebSocket protokolom tako da primjer nije temeljen na nikakvim već postojećim projektima. Unatoč tome, uvijek se dobro konzultirati sa službenom dokumentacijom biblioteke koja se koristi tako da sam i ja pregledao dokumentaciju *javax.websocket* paketa u svrhu lakše izrade demonstracijskog primjera (WebSocket API (Java dokumentacija), 2011). Ovdje ću, kao i kod Python primjera, također započeti objašnjenjem klijentske strane jer su razlike minimalne, no valja prvo objasniti način izrade projekta. Za izradu sam koristio NetBeans razvojno okruženje verzije 8.2 sa Java-om verzije 8, te sam u tom okruženju kreirao prazan projekt za Web aplikaciju te na temelju tog predloška nastavio s radom. Klijent se i ovog puta sastoji od *index.html* sa popratnim CSS i JS datotekama, a JavaScript datoteka se od prijašnjeg primjera razlikuje jedino u instanciranju objekta koji predstavlja vezu prema poslužitelju, što je vidljivo u sljedećem isječku koda:

```
var uri = 'ws:// ' + document.location.host + document.location.pathname
                   + 'chatendpoint';
var websocket = new WebSocket(uri);
```

Kao i kod prethodnog primjera, ne učitavamo nikakvu zasebnu biblioteku na klijentsku stranu kojom bi iskoristili WebSocket protokol, već se koristi nativna JavaScript biblioteka. Adresa se ovog puta mora dohvatiti na malo drugačiji način jer se aplikacija servira uz pomoć Apache Tomcat poslužitelja verzije 8 te ne možemo uvijek znati kako je poslužitelj konfiguriran, niti nas to velikom većinom vremena ne zanima, barem ne u ovom jednostavnom primjeru. Ono što je potrebno dodati je oznaka *ws* na početku adrese koja pokazuje na to da se radi o WebSocket protokolu, te naziv *endpoint*-a kojeg definiramo na poslužiteljskom dijelu aplikacije. Sav kod koji upravlja slanjem i primanjem poruka je identičan prethodnom primjeru.

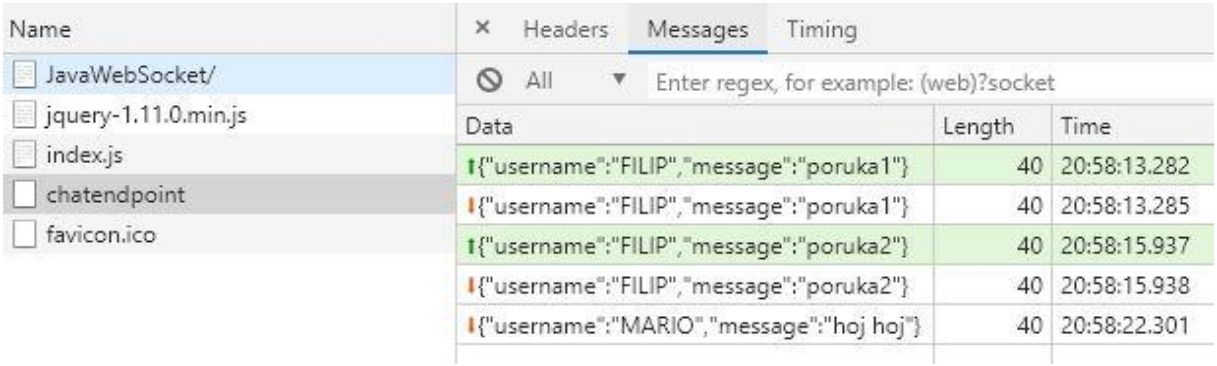
Što se tiče poslužiteljskog dijela aplikacije, već je napomenuto da se koristi poslužitelj Apache Tomcat koji servira našu aplikaciju kako bi se ona mogla koristiti. Ovaj poslužitelj jedan je od mnogih koje sam mogao iskoristiti, no on dolazi zapakiran u instalaciju NetBeans razvojnog okruženja pa sam ga zbog toga, kao i prijašnjeg iskustva rada sa tim poslužiteljem, odlučio ovdje iskoristiti. Logika oko inicijalizacije veze prema klijentu preko WebSocket protokola, te upravljanja slanjem i primanjem poruka, smještena je u jednu klasu naziva *Chat* čiji je kod, izuzev učitavanja paketa i biblioteka, prikazan u sljedećem isječku koda:

```
@ServerEndpoint("/chatendpoint")
public class Chat {
    private static Set<Session> peers = Collections.synchronizedSet(
                                                    new HashSet<Session>());
    @OnOpen
    public void onOpen (Session peer) {
        peers.add(peer);
    }

    @OnClose
    public void onClose (Session peer) {
        peers.remove(peer);
    }

    @OnMessage
    public void onMessage(String message) throws IOException {
        for (Session peer : peers) {
            peer.getBasicRemote().sendText(message);
        }
    }
}
```

Klasa započinjem anotacijom koja je označava kao WebSocket *endpoint* za poslužitelj, tj. klasa koja sadrži logiku koju WebSocket veza mora pratiti. Klasu sam kreirao kroz opcije u razvojno okruženju te kao tip klase odabrao WebSocket *endpoint* čime je kostur klase zajedno sa anotacijom automatski kreiran. *Endpoint* sadrži ime čime se on identificira i uz pomoć kojeg se klijenti na njega mogu spajati, što je i navedeno kod opisa klijentske aplikacije gdje se ovaj naziv pridodavao na kraju adrese WebSocket objekta. Klasa sadrži jedno svojstvo, *peers*, koja predstavlja kolekciju u koju spremamo klijente. Metode *onOpen()* i *onClose()* kao argument primaju sesiju, tj. objekt tipa *Session* koji predstavlja klijenta, te ga spremaju ili izbacuju iz kolekcije ovisno o metodi. Metode same po sebi ne bi radile ništa vezano uz WebSocket ukoliko ne bi bile označene sa *OnOpen* i *OnClose* anotacijama koje govore poslužitelju koji

dio koda treba izvršiti kao odgovor na određeni događaj na WebSocket-u. Zadnja metoda je metoda *onMessage()* koja je označena istoimenom anotacijom te koja predstavlja kod koji će se izvršiti u slučaju *onMessage* događaja na WebSocket-u, tj. svaki put kada poslužitelj primi poruku od klijenta. U tom slučaju poslužitelj će iterirati kroz sve klijente u kolekciji te im poslati poruku koja je stigla od nekog od klijenata.

Ovo je zapravo sve što je potrebno za jednostavnu aplikaciju za čavrljanje, pa sam u opcijama za pokretanje u NetBeans razvojnom okruženju odabrao Google Chrome kao željeni Web preglednik te pokrenuo aplikaciju. NetBeans zatim sam pokreće poslužitelj, servira aplikaciju te pokreće prvog klijenta u odabranom Web pregledniku. Drugi klijenti zatim se mogu otvoriti kopiranjem adrese otvorenog klijenta i otvaranjem novog klijenta u novoj kartici sa tom adresom. Ukoliko bi aplikaciju ponovno pokrenuli preko NetBeans razvojnog okruženja, on bi kreirao novu instancu poslužitelja pa bi se činilo da aplikacija ne radi jer zapravo pokrećemo dva poslužitelja koji međusobno ne komuniciraju u ovakvoj aplikaciji. Nakon pokretanja nekoliko klijenata i kratkog testiranja aplikacije, možemo otvoriti *Network* karticu u odabranom Web pregledniku te pogledati kako je Java implementacija WebSocket protokola ostvarila komunikaciju između klijenta i poslužitelja. Možemo vidjeti vezu istog naziva kao i WebSocket *endpoint* koji smo koristili, a to je *„chatendpoint“*, te otvaranjem te veze vidimo sve poruke koje su od trenutnog klijenta odlazile prema poslužitelju, te koje je poslužitelj slao tom klijentu. Sve navedeno vidljivo je na sljedećoj slici:



Slika 23: Razmjena poruka WebSocket protokolom kroz njegovu implementaciju u programskom jeziku Java

Java ima, kao popularan i često korišten programski jezik, vrlo puno raznih biblioteka i nadogradnji koje omogućuju razvoj Web aplikacija u realnom vremenu, no unatoč tome, ovdje predstavljena biblioteka je najčešće korištena u ovu svrhu. Razlog zašto Python ima puno više popularnih biblioteka u ovom području, dok Java-om dominira jedna je u tome što je ova biblioteka službeno podržana i dokumentirana. Valja napomenuti da postoji mnoštvo drugih Java biblioteka koje se bave izgradnjom Web aplikacija u realnom vremenu, a ne baziraju se na WebSocket protokolu. Ovdje najviše iskaču razne biblioteke koje koriste WebRTC protokol čiji je API napisan u Java-i pa ne treba posebno naglašavati njihovu kompatibilnost.

6. SignalR razvojna biblioteka

SignalR jedna je od najčešće korištenih biblioteka za razvoj Web aplikacija u realnom vremenu. Izrađena je 2011. godine od strane Davida Fowlera i Damiana Edwardsa, no potencijal ove biblioteke ubrzo je prepoznat sa strane Microsoft-ovih razvojnih inženjera. To je bilo vrijeme kada je WebSocket protokol tek standardiziran, a Web preglednici su tek počeli razmišljati o tome kako i kada ga uvesti u svoja rješenja, pa je biblioteka kao što je SignalR, koja radi i sa WebSocket protokolom i sa drugim komunikacijskim tehnikama, bila najbolji izbor za izradu aplikacija u realnom vremenu. SignalR je ubrzo postao standard kod izrade aplikacija u realnom vremenu Microsoft-ovim tehnologijama, a iako još uvijek nije bio službeni dio Microsoft-ovih rješenja, Microsoft-ova službena dokumentacija detaljno je objašnjavala rad sa SignalR bibliotekom. Autori ove biblioteke su nakon nekoliko godina suradnje sa Microsoft-om i službeno postali dio tima koji je radio na novoj verziji ASP.NET razvojnog okvira za izradu Web aplikacija, a samim time SignalR je ubrzo postao dio tog okvira. Prije toga SignalR je postojao kao zasebni NuGet paket kojeg se moglo preuzeti u ASP.NET projektu i koristio se na taj način, dok su za klijente postojali posebni paketi. Trenutno najčešće korištena verzija SignalR biblioteke, i ona koju ću ja koristiti, je ona koja dolazi kao dio ASP.NET Core razvojnog okvira verzije 2.1.

Ova verzija SignalR-a nešto je drugačija od biblioteke od prije nekoliko godina, no promjene su uglavnom olakšale rad sa bibliotekom i popravile neke njezine nedostatke. Najveća razlika je ta da ni poslužiteljski ni klijentski dijelovi aplikacija više ne koriste jQuery biblioteku. To je vrlo često korištena biblioteka koju sam i ja koristio u prije opisanim primjerima, no većina modernih razvojnih okvira za izradu Web aplikacija poput Angular-a više se ne oslanjaju na jQuery koji se smatra zastarjelom tehnologijom. Izbacivanjem ovisnosti o jQuery biblioteci omogućio se rad SignalR biblioteke sa tim programskim okvirima te olakšava proširivanje biblioteke u budućnosti. Druga važna promjena u radu nove verzije SignalR biblioteke je da ponovno povezivanje (eng. *reconnection*) i spremanje poruka u slučaju ponovnog povezivanja više nisu omogućeni. Ova funkcionalnost pokazala se previše kompleksnom i nepouzdanom za implementaciju, pogotovo kada se u obzir uzme vrlo mali broj slučaja korištenja u kojem bi se ova funkcionalnost mogla biti korisna. Druge promjene tiču se protokola za komunikaciju zasnovanih na JSON-u koji nisu kompatibilni sa starijim verzijama SignalR biblioteke, korištenjem injektiranja ovisnosti (eng. *dependency injection*) kod dohvaćanja pojedinih struktura sa kojima SignalR radi, te podrška za rad sa mnogim drugim tehnologijama. Pojavio se i veći broj SignalR biblioteka za korištenje sa klijentske strane osim JavaScript biblioteke, čime je korisnost ove biblioteke porasla u svijetu u kojem se aplikacije sve češće izrađuju uz pomoć razvojnih okvira poput Angular-a i React-a (Chu, 2018).

U završnom primjeru ovog diplomskog rada koristit ću upravo SignalR biblioteku koja je dio ASP.NET Core razvojnog okvira jer sadrži i više nego dovoljno funkcionalnosti koje su mi potrebne da izradim primjer kakav sam zamislio. Poslužiteljski dio aplikacije je standardni ASP.NET projekt izrađen u Visual Studio 2017 razvojnoj okolini, dok je klijentski dio izrađen uz pomoć razvojnog okvira Angular i SignalR biblioteke koju je moguće dodati u Angular projekt. Klijentski dio aplikacije izrađen je u Visual Studio Code razvojnoj okolini, a testiran u Google Chrome Web pregledniku. Poslužiteljski i klijentski dio aplikacije objasnit ću zasebno i tim redoslijedom radi lakšeg praćenja. Kod izrade primjera uglavnom sam koristio službenu dokumentaciju SignalR biblioteke (SignalR dokumentacija (uvod), 2018).

Aplikacija koju sam zamislio je jednostavna aplikacija za pregled sportskih, u ovom slučaju nogometnih, rezultata. Aplikacija ima korisničko sučelje sa različitim opcijama s obzirom na to da li je korisnik administrator ili nije, a to ovdje ovisi samo o nazivu korisnika koji on unosi kod pokretanja aplikacije. Pošto ovaj primjer služi za demonstraciju mogućnosti SignalR biblioteke kao jedne kvalitetne biblioteke za razvoj Web aplikacija u realnom vremenu, odlučio sam ostale dijelove aplikacije, koji se ne bave komunikacijom u realnom vremenu, većim dijelom simulirati u svrhu lakšeg testiranja i demonstracije. Npr., podatke ću spremati u samu aplikaciju te će se uneseni podaci izgubiti nakon izlaska iz aplikacije, a neću koristiti bazu podataka jer sama SignalR biblioteka sa njom nema nikakve veze. Cilj ovog primjera je pokazati kako koristiti SignalR biblioteku te demonstrirati korištenje raznih funkcionalnosti koje ona nudi. Korisničko sučelje izrađeno je uz pomoć Angular razvojnog okvira zbog njegove funkcionalnosti povezivanja vrijednosti na HTML stranici i TypeScript klasi što znači da ne moram pisati logiku za osvježavanje sadržaja kada on stigne sa poslužitelja do klijenta, već će klase uglavnom sadržavati logiku koja se direktno tiče SignalR biblioteke. Također, Angular mi omogućuje korištenje biblioteke Angular Material kojom mogu napraviti korisničko sučelje koje će biti pregledno, pouzdano i lako za korištenje te će vrlo dobro nadopunjavati ovaj programski primjer.

Aplikacija će omogućavati pregled utakmica koje se trenutno igraju, te praćenje događaja odabrane utakmice. Svi korisnici koji prate određenu utakmicu moći će komunicirati jedni sa drugima. Nadalje, bit će omogućen pregled svih već odigranih utakmica te pisanje komentara na te utakmice koji će biti vidljivi svim korisnicima aplikacije. Početni podaci bit će nasumično generirani na poslužiteljskoj strani, no administratori sustava moći će dodavati nove utakmice i nove događaje tim utakmicama preko korisničkog sučelja na klijentskoj strani. Ovaj programski primjer može se naći u direktoriju *SportResultViewer* u direktoriju sa SignalR primjerima koji su pridruženi ovom radu. Taj direktorij sadrži i poslužiteljski i klijentski dio aplikacije koje je moguće raspoznati prema nazivu direktorija, a kod njih se nalazi i datoteka sa uputama o pokretanju primjera.

6.1. Poslužiteljski dio aplikacije

Poslužiteljski dio aplikacije napisan je u C# programskom jeziku u Visual Studio 2017 razvojnom okruženju. Unatoč tome što se u njima znam snaći, potražio sam upute za kreiranje praznog projekta pogodnog za daljnje korištenje SignalR biblioteke zbog promjena i na njoj, kao i na spomenutom programskom okruženju kroz zadnjih nekoliko godina. Projekt tako započinje kao prazan Web API projekt koristeći .NET Core razvojni okvir, a prvi korak nakon toga je podešavanje postavki za pokretanje aplikacije (Spasojevic, 2018). Iste upute koristio sam i kod postavljanja klijentske aplikacije. SignalR projekt čije sam dijelove prikazivao kod demonstracije komunikacijskih tehnika ujedno pokreće i poslužiteljski i klijentski dio aplikacije. Poslužiteljski dio aplikacije pokretao se u pozadini, dok se klijentski dio aplikacije servirao na IIS (eng. *Internet Information Services*) poslužitelju kojeg koriste Web aplikacije izrađene Microsoft-ovim tehnologijama. Zbog toga što je u ovom primjeru klijentski dio potpuno odvojen od poslužiteljskog dijela aplikacije, postavke za pokretanje aplikacije mijenjam u datoteci *launchSettings.json* tako da se poslužitelj pokreće kao obična aplikacija u komandnoj liniji kao što je slučaj bio i kod poslužitelja kod ostalih programskih primjera. Ovdje je moguće promijeniti i broj porta koji poslužitelj koristi kao i neke druge postavke koje su dobre takve kakve jesu, pa ih ne mijenjam.

Rad nastavljam na klasi *Startup.cs*, automatski generiranoj klasi koja sadrži logiku za konfiguriranje i pokretanje poslužitelja. Klasa sadrži dvije metode, *ConfigureServices()* i *Configure()*, od kojih prva konfigurira usluge ili servise koje poslužitelj koristi, dok druga služi za konfiguraciju samog poslužitelja. Prvi korak je postaviti politiku korištenja CORS-a koji je spominjan već prije, a to je potrebno zbog toga što će klijentski dio aplikacije biti smješten na drugom portu te će se njegovi zahtjevi tretirati kao da dolaze sa druge lokacije. Ovdje želimo dopustiti takve zahtjeve, a to postavljamo sljedećim isječkom koda kojeg stavljam u prvu metodu klase:

```
services.AddCors(options =>
{
      options.AddPolicy("CorsPolicy",
      builder => builder.WithOrigins("http://localhost:4200")
                        .AllowAnyMethod()
                        .AllowAnyHeader()
                        .AllowCredentials());
});
```

U ovom isječku koda u servise dodaje novu politiku korištenja CORS-a koja dopušta bilo koju metodu, zaglavlje i uvjerenja koji dolaze u zahtjevu sa navedene lokacije sa portom 4200 na kojem će se nalaziti klijentski dio aplikacije. Ovaj broj porta je standardan za Angular aplikacije te ga nisam mijenjao. Ovime sam dodao tu politiku korištenja CORS-a kao mogućnost, no ona se još mora zadati. To se radi u drugoj spomenutoj metodi sljedećom linijom koda:

```
app.UseCors("CorsPolicy");
```

Sljedeći korak je implementacija SignalR *hub*-a, tj. središta, no pošto se u struci uglavnom koristi izraz *hub*, nadalje ću koristiti njega. *Hub* je SignalR API koji omogućava komunikaciju sa klijentom na način da on poziva određenu metodu *hub*-a koji zatim odradi dio posla koji je naveden unutar metode što uglavnom uključuje i slanje poruka prema klijentima (SignalR dokumentaciju (Hub), 2018). *Hub* se ubacuje u aplikaciju kao obična klasa koja nasljeđuje klasu *Hub* koja je definirana u SignalR biblioteci. Kao što je bilo spomenuto, SignalR je danas dio ASP.NET Core razvojnog okvira pa ga nije potrebno posebno dodavati u aplikaciju, no potrebno je dodati liniju koda koja označava da koristimo imenski prostor *SignalR*. Moj *hub* nazvan je *SportViewerHub* te sadrži sve metode potrebne za komunikaciju prema klijentu. Te metode uvijek se definiraju na način da ne vraćaju ništa kao rezultat metode, tj. tip im je *void*, jer uglavnom šalju poruke klijentima koji nisu dio ove aplikacije. Klijenti ove metode mogu izravno pozivati putem njezinog imena tako da je važno da je ime dovoljno deskriptivno u odnosu na njezinu zadaću i da se dovoljno razlikuje od ostalih metoda *hub*-a. Kod poziva metoda klijenti mogu slati parametre koje je potrebno opisati kod metoda u *hub*-u, a broj parametara nije ničim ograničen, no pošto se radi o slanju i primanju poruka poželjno je da je parametara što manje. Pošto većina mog primjera obuhvaća ili primanje određenih podataka od poslužitelja, poput utakmice ili *chat* poruke, te slanje podataka o npr. novoj utakmici od klijenta prema poslužitelju, metode uglavnom imaju samo jedan parametar. Kod dohvata podataka to je uglavnom identifikacijski broj utakmice, dok kod slanja podataka na poslužitelj šaljem cijelu strukturu u JSON formatu. Podaci se mogu poslati i na druge načine jer se poruka zapravo šalje u obliku stringa, no JSON format je vrlo prikladan kod slanja i interpretiranja podataka na mreži zbog čega se i smatra *de facto* standardom u Web aplikacijama. Podaci se prema klijentu također šalju u JSON obliku. ASP.NET Core i Angular imaju svoje klase koje se bave serijalizacijom i deserijalizacijom JSON struktura pa sam njih koristio u tu svrhu. Neke metode klijentu samo šalju podatke koje on zatraži, druge obrađuju podatke koje dobiju od klijenata, npr. dodaju dobivene podatke poput nove utakmice te ih spremaju i zatim ponovno šalju prema klijentu, a neke metode klijentima ne šalju nikakav odgovor, ukoliko on nije potreban. Mnoge metode koje koristim imaju sličnu funkciju, npr. slanje podataka prema klijentu, tako da sa odabrao nekoliko važnijih kako bih ih ovdje objasnio.

Prva metoda koju želim objasniti je metoda *SendGame()*. Ona služi za slanje podataka o utakmici prema klijentu koji je te podatke zatražio, a kao parametar prima identifikacijski broj utakmice. Ta metoda izgleda ovako:

```
public void SendGame(string gameId)
{
      Clients.Caller.SendAsync("gameInfo", JsonConvert.SerializeObject(
            GlobalInfoSingleton.GetInstance().AllGames.Where(
                  _ => _.ID == int.Parse(gameId)).SingleOrDefault()));
}
```

Metoda pristupa svim klijentima preko svojstva *Clients*. To je svojstvo klase *Hub* koju moja klasa nasljeđuje, a sadrži sve klijente trenutno spojene na poslužitelj. SignalR automatski obrađuje spajanje i odspajanje klijenata na poslužitelj što je jedna od prednosti ove biblioteke i dio razloga zašto sam je odlučio koristiti. Preko tog svojstva možemo pristupiti njezinom svojstvu *Caller* koje predstavlja klijenta koji je pozvao metodu. Njemu zatim, koristeći metodu *SendAsync()*, asinkrono šaljem poruku tipa *gameInfo* prema čemu će klijent moći raspoznati koja metoda joj je poslala odgovor u slučaju dobivanja velikog broja poruka. Drugi parametar te metode je sama poruka. Ovdje koristim klasu *JsonConvert* koja mi služi za serijalizaciju i deserijalizaciju JSON struktura. Poruka koju šaljem je instanca klase *Game* koja predstavlja utakmicu, a koju uzimam iz liste svih utakmica smještenoj u klasi *GlobalInfoSingleton*. Kao što sam već spomenuo, odlučio sam pojednostaviti dio aplikacije koja nema veze sa temom ovog rada, tako da spomenuta klasa generira i sprema sve podatke koje koristim u aplikaciji. Modeli koje koristim predstavljeni su instancama modela, a predstavljaju utakmicu, događaj utakmice, *chat* poruku ili komentar. Vrlo slično metodi *SendGame()* rade i metode *hub*-a koje se bave slanjem svih utakmica koje su u tijeku, svih *chat* poruka za određenu utakmicu, svih završenih utakmica, svih komentara određene završene utakmice i svih događaja određene utakmice. Sve one primaju identifikacijski broj utakmice kao parametar te klijentu koji poziva metodu šalju tražene podatke u JSON obliku. Jedina razlika nalazi se u metodama koje vraćaju *chat* poruke i komentare određene utakmice. Npr., kod pristupa stranici za određenu utakmicu koja još traje korisnik dobije mogućnost prisustvovanja funkcionalnosti za čavrljanje o toj utakmici. Pošto želim da se *chat* poruke šalju samo klijentima koji gledaju tu utakmicu, njih je potrebno dodati u posebnu grupu preko koje onda mogu pristupiti svim klijentima koji tu utakmicu gledaju. Kod pristupa stranici pojedine utakmice zato se poziva metoda koja vraća sve *chat* poruke koje su već napisane kroz trajanje utakmice, a usput korisnika kojem te poruke šaljem ubacujem u grupu koja te poruke razmjenjuje. Svakom korisniku pridodana je jedinstvena identifikacijska oznaka kojom ga dodajemo u grupu, a to radimo sljedećom metodom:

```
public void SendChatMessagesForGame(string gameId)
{
      Groups.AddToGroupAsync(Context.ConnectionId, _chatGroup + gameId);
      Clients.Caller.SendAsync("chatMessages", JsonConvert.SerializeObject(
            GlobalInfoSingleton.GetInstance().AllChatMessages.Where(
                  _ => _.GameID == int.Parse(gameId))));
}
```

Ova metoda prvo pristupa svojstvu *Groups* koje je, kao i svojstvo *Clients*, dio klase *Hub* koju smo naslijedili. Metodom *AddToGroupAsync()* zatim dodajem korisnika željenoj grupi. Prvi parametar koji prosljeđujem metodi je jedinstveni identifikacijski broj klijenta sadržan u *ConnectionId* svojstvu spremljenom u svojstvu *Context*, koje je također dio klase *Hub*. Drugi parametar je ime grupe, koje se u mom slučaju sastoji od prefiksa spremljenog u varijablu *_chatGroup* radi raspoznavanja grupa za *chat* poruke i grupa za komentare, te identifikacijskog broj utakmice. Time postižem to da se svaki klijent pridruži točno onoj grupi koja mu je potrebna. Ukoliko je klijent prvi koji se grupi pridružuje, ona se automatski kreira bez potrebne samostalnog kreiranja grupe od strane razvojnog inženjera. Sljedeći dio metode bavi se slanjem svim dosadašnjih *chat* poruka povezanih sa željenom utakmicom. Sada, kada znamo da se veći broj korisnika može nalaziti u određenoj grupi, možemo vrlo jednostavno poslati poruku svim pripadnicima grupe, što je vidljivo u sljedećem isječku koda iz metode za slanje *chat* poruke od klijenta prema poslužitelju:

```
Clients.Group(_chatGroup + gameId).SendAsync("newChatMessage",
                        JsonConvert.SerializeObject(newMessage));
```

Ovdje svim klijentima određene grupe pristupam tako da pozovem metodu *Group* svojstva *Clients* te joj predam naziv grupe. Pošto znam da se metoda bavi razmjenom *chat* poruka, dajem prikladan prefiks nazivu grupe zajedno sa identifikacijskim brojem utakmice koju sam dobio preko parametra. U ovom slučaju kao parametar metode nisam slao samo identifikacijski broj utakmice već cijelu strukturu koja predstavlja *chat* poruku. Nju sam deserijalizirao *JsonConvert* klasom te iz nje izvadio potrebne podatke kao što su identifikacijski broj utakmice i podaci koji čine novu poruku sadržanu u varijabli *newMessage* koju zatim spremam na poslužiteljsku stranu te šaljem svim pripadnicima grupe kao što je ranije bilo i opisano. Zadnji korak u radu sa grupama je izbacivanje korisnika iz određene grupe. To radim zbog toga što ne želim da klijent prima poruke iz grupa za utakmice koje već aktivno ne gleda čime se šalju nepotrebni podaci i nepotrebno opterećuje mreža. Izbacivanje korisnika iz grupe radim sljedećom metodom:

```
public void RemoveFromChatGroup(string gameId)
{
      Groups.RemoveFromGroupAsync(Context.ConnectionId, _chatGroup +
                                                          gameId);
}
```

Metoda se pokreće u slučaju kada se korisnik nalazi na dijelu aplikacije koji prati određenu utakmicu u trajanju, što znači da prati i čavrljanje vezano uz tu utakmicu, te prijeđe na neki drugi dio aplikacije. Slanje *chat* poruka više nije potrebno jer one korisniku neće biti prikazane, a ukoliko se odluči vratiti na praćenje te iste utakmice sve *chat* poruke bit će mu poslane prije objašnjenom metodom *SendChatMessagesForGame()*. Izbacivanje korisnika iz grupe radi se metodom *RemoveFromGroupAsync()* svojstva *Groups* kojoj predajemo iste parametre kao i kod dodavanja korisnika u grupu, a to su njegova jedinstvena identifikacijska oznaka te naziv grupe. Ova metoda korisniku ne vraća nikakvu poruku jer to nije potrebno. Još jedna funkcionalnost koju koristim je slanje poruka svim klijentima. Jedan od slučajeva kada to radim je kada administrator sustava dodaje novu utakmicu u aplikaciju, a pošto su sve utakmice u trajanju uvijek vidljive svim klijentima, to se radi metodom koja dobije podatke o novoj utakmici, sprema je na poslužiteljsku stranu te je zatim šalje svim klijentima koristeći sljedeću liniju koda:

```
Clients.All.SendAsync("newGame", JsonConvert.SerializeObject(newGame));
```

Svim klijentima pristupa se na vrlo sličan način kao što je to bio slučaj i kod slanja poruke jednom klijentu ili svim klijentima određene grupe, osim što se ovdje pristupa svojstvu *All*. Sve metode koje su dio mog *SportViewerHub*-a rade na načine objašnjene u napisanim isječcima koda, osim što šalju ili primaju druge podatke. Način slanja i primanja poruka koristeći SignalR biblioteku ostaje isti, tako da neću detaljno objašnjavati ostale metode. Osim navedenih načina za slanje poruka, moguće je koristiti i druge koji meni nisu bili potrebni. Ovdje se uglavnom radi o svojstvima i metodama koji su vrlo slični onima koje sam već objasnio. Npr., moguće je poslati poruku svim klijentima osim onog koji metodu poziva, svim klijentima određene grupe osim onima koje posebno navedemo unutar parametra, svim klijentima određene grupe osim onom klijentu koji je pozvao metodu, itd. Ove mogućnosti omogućuju lakše i brže korištenje biblioteke, a ujedno povećavaju čitljivost koda i lakoću održavanja poslužitelja što je jedan od razlog zašto je SignalR i stekao toliku popularnost. Ono što je još potrebno spomenuti u vezi *Hub*-a je to da mu je moguće dodati ime u obliku anotacije. Ime *Hub*-a dodatno ga opisuje što može biti korisno ukoliko se koristi nekoliko različitih *hub*-ova, no to nije toliko čest slučaj pa tu mogućnost ovdje nisam iskoristio. Ukoliko se ime ne navede, koristi se ime same klase ili ime koje navedemo kod pokretanja poslužitelja.

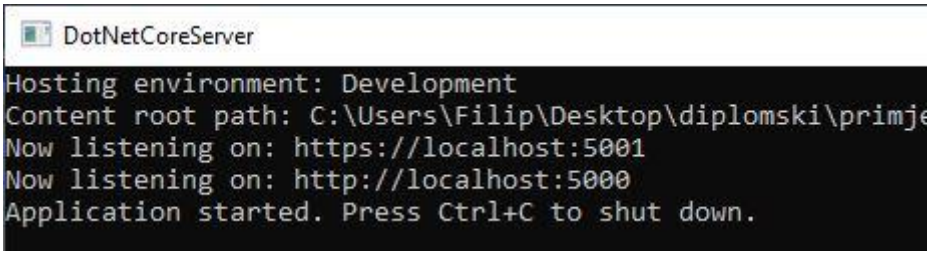
Zadnji korak kod izrade poslužitelja za ovu aplikaciju je definiranje izrađenog *SportViewerHub*-a kao *hub*-a koji će se koristiti kod rada aplikacije, te definiranje postavki koje aplikaciji govore da će se koristiti SignalR biblioteka. Prvi korak je dodavanje SignalR funkcionalnosti u *ConfigureServices()* metodu u *Startup.cs* klasi, što se radi umetanjem SignalR biblioteke kao servisa korištenjem sljedeće linije koda:

```
services.AddSignalR();
```

Sljedeći korak je reći aplikaciji da mora koristiti *hub* koji smo izradili, a usput dodajemo i putanju do željenog *hub*-a kako bi mogli pristupiti njegovim metodama sa klijentskog dijela aplikacije. Ovaj dio koda određuje kamo će se slati zahtjevi koji dolaze od strane klijenata i tko će ih obrađivati, što je u ovom slučaju *SportViewerHub* koji je mapiran na poslužitelju. To se definira u *Configure()* metodi *Startup.cs* klase korištenjem sljedeće linije koda:

```
app.UseSignalR(routes =>
{
    routes.MapHub<SportViewerHub>("/sportviewer");
});
```

Poslužitelj je sada spreman za korištenje te ga možemo pokrenuti kao i svaku drugu .NET Core aplikaciju. Nakon pokretanja, otvara se komandna linija u kojoj je poslužitelj pokrenut i javlja početne informacije o svom radu što izgleda ovako:



Slika 24: Poslužiteljski dio SignalR aplikacije

Osim osnovnih informacija o okolini u kojoj je poslužitelj pokrenut te putanji na kojoj se nalazi, poslužitelj javlja i portove na kojima on radi i čeka da stignu poruke klijenata. Oba porta su standardni za ASP.NET Core aplikacije, a pošto se ne preklapaju sa brojem porta koji koristi Angular klijent, nisam ih mijenjao, no ta opcija uvijek postoji. Zadnja poruka koju poslužitelj šalje je da on započinje sa radom. Ono što još ovdje želim spomenuti je to da nisam mijenjao opcije kojima se definira komunikacijska tehnika koju će aplikacija koristiti kao što sam to radio kod manjih primjera u dijelu rada o komunikacijskim tehnikama. SignalR može sam zaključiti koja komunikacijska tehnika je najbolja u određenom trenutku te nemam potrebe dodatno mijenjati te opcije u ovom programskom primjeru.

6.2. Klijentski dio aplikacije

Klijentski dio aplikacije izrađen je uz pomoć Angular razvojnog okvira. Angular je vrlo popularan JavaScript razvojni okvir koji se koristi kod izrade dinamičnih Web stranica (Aldwin, 2019). Angular sam odabrao zbog nekoliko razloga, između ostalog i zbog mog dosadašnjeg vrlo pozitivnog iskustva u radu sa ovim okvirom zbog kojeg sam zaključio da bi bio vrlo prikladan za korištenje u ovom programskom primjeru. Angular omogućuje povezivanje (eng. *binding*) vrijednosti sa korisničkog sučelja sa vrijednosti u programskom kodu na način da ako se ta vrijednost promijeni, vrijednost prikazana na korisničkom sučelju će se također promijeniti. Kao što je već bilo spomenuto, u ovom programskom primjeru želio sam se fokusirati na mogućnosti SignalR biblioteke pa bi mnogo koda koji se bavi samo osvježavanjem vrijednosti na korisničkom sučelju zakompliciralo primjer i smanjilo njegovu čitljivost. Nadalje, Angular ima svoju implementaciju SignalR biblioteke za klijente koja je laka za korištenje i radi vrlo dobro. Zadnji razlog za odabir Angular razvojnog okvira je biblioteka Angular Material koja sadrži vrlo velik broj gotovih elemenata korisničkog sučelja sa vrlo kvalitetnom dokumentacijom (Angular Material dokumentacija, 2019) što mi uvelike olakšava posao izrade korisničkog sučelja koje će biti i funkcionalno i ugodno za oči.

Postavljanje početnog Angular projekta radi se preko definiranih Angular naredbi u komandnoj liniji. Ovdje nisam koristio CMD već terminal dostupan unutar razvojnog okruženja Visual Studio Code. To je vrlo kvalitetno razvojno okruženje za projekte pisane u raznim programskim jezicima te se vrlo često koristi kod izrade Angular aplikacija. U izradi aplikacije koristio sam programski jezik TypeScript, jezik koji uvelike olakšava pisanje Web aplikacija zbog svoje objektno orijentirane prirode, a kod kompilacije koda se pretvara u JavaScript kod koji Web preglednici mogu pokretati. Nakon izrade kostura projekta, dodao sam biblioteke SignalR i Angular Material koristeći naredbu *npm install*. Kao i kod poslužitelja, u kojem se veći dio logike oko SignalR komunikacije nalazio u jednoj klasi koja se tamo zvala *SportViewerHub*, i ovdje je većina logike smještena u jednu klasu ili servis koji ovdje nazivam *SignalRService*. Angular radi na način da se klasa može referencirati i koristiti u drugoj klasi kroz injektiranje ovisnosti, a ovu klasu također navodim kao servis koji je dostupan u korijenskom, tj. *root* dijelu aplikacije. To mi omogućava da svi dijelovi aplikacije koriste istu instancu servisa pa mogu jednom otvoriti vezu prema poslužitelju koja zatim ostaje otvorena i spremna za korištenje. U toj se klasi nalazi metoda za spajanje na poslužitelj koju pozivam na korijenskoj komponenti aplikacije, klasi *AppComponent*, koja u sebi sadrži sve ostale komponente aplikacije pa u nju spremam i listu svih utakmica koje trenutno traju jer ona uvijek mora biti dostupna svim komponentama.

Rad sa podacima također se nalazi u jednom klasi, no ovdje je to korijenski servis naziva *DataService* koji ne samo da sprema podatke, već omogućuje drugim komponentama da se pretplate na određena svojstva ovog servisa u koje se podaci spremaju. Ukoliko poslužitelj pošalje nove podatke koji se zatim spreme u ta svojstva, servis će pretplaćenim komponentama javiti da je došlo do promjene podataka da one mogu uzeti najnovije podatke. Primjer kako radi spremanje utakmica i obavještavanje komponenata o promjeni vidljiv je u sljedećem isječku koda:

```
private allLiveGames = new BehaviorSubject<Array<Game>>(new Array<Game>());

currentAllLiveGames = this.allLiveGames.asObservable();


public changeLiveGames(liveGames: string) {
    var jsonGames = JSON.parse(liveGames);
    var games: Array<Game> = [];
    for(var i: number = 0; i < jsonGames.length; i++) {
        games.push(this.ObjectToGame(jsonGames[i]));
    }
    this.allLiveGames.next(games);
}
```

Prva linija koda predstavlja stvaranje instance klase *BehaviorSubject* koja u sebi sadrži polje utakmica. BehaviorSubject klasa koristi se kod dijeljenja podataka sa drugim komponentama, a to je ono što želimo postići. Sljedeća linija koda sprema tu *BehaviorSubject* instancu kao *Observable* što omogućuje da se komponente pretplate na promjene u vrijednosti koju *BehaviorSubject* sadrži. Metoda *changeLiveGames()* je metoda koju poziva SignalR servis koji poziva metodu poslužitelja i od njega dobiva nove podatke, u ovom slučaju listu svih utakmica u trajanju, koje zatim prosljeđuje ovoj metodi. Ona te podatke izvadi iz JSON strukture te ih spremi u novo polje sa utakmicama u trajanju. To se polje zatim predaje kao nova vrijednost *BehaviorSubject*-u metodom *next()*. Komponenta koja se želi pretplatiti na određene podatke to radi preko *DataService* servisa. Komponenta mora injektirati ovisnost o tom servisu u svom konstruktoru, a servis dijeli sa drugim komponentama aplikacije. Na podatke, u ovom slučaju utakmice u trajanju, se zatim pretplaćuje sljedećom linijom koda:

```
this.dataService.currentAllLiveGames.subscribe(games =>
                    this.liveGames = games);
```

Ova linija koda predstavlja pretplatu na sve utakmice u trajanju koje se nalaze u *DataService* servisu. Komponenta koja sadrži ovaj isječak koda ima vlastitu varijablu u koju sprema utakmice koje trenutno prikazuje. Ukoliko dođe do promjene u podacima koji se nalaze u *DataService* servisu, on odmah obavještava ovu komponentu koja zatim u svoju varijablu sprema nove podatke, tj. novu listu utakmica u trajanju. Za svaki podatak koji se obrađuje tijekom rada aplikacije postoji skup varijabli i metoda u *DataService* servisu koje se bave njihovim spremanjem, promjenom i distribucijom po komponentama kojima su potrebni. Sadrži i nekoliko metoda koje se bave pretvaranjem podataka iz JSON formata u instance klasa koje predstavljaju modele koje koristim u aplikaciji, a koji se podudaraju onima sa poslužitelja. Da ponovim, radi se o klasama za utakmicu, događaj utakmice, *chat* poruku i komentar utakmice. Od svega potrebnog za postavljanje aplikacije vrijedi spomenuti i definiranje putanja ili ruta različitih komponenti u *AppModule* klasi koja predstavlja osnovni dio aplikacije.

Glavni dio logike oko komunikacije sa poslužiteljem nalazi se u spomenutom *SignalRService* servisu. On u sebi ima varijablu u koju se sprema veza prema *hub*-u poslužitelja, metodu za spajanje na poslužitelj te metode za komunikaciju sa poslužiteljem. Za početak je potrebno objasniti dio koda kojim se spajamo na poslužitelj, a on izgleda ovako:

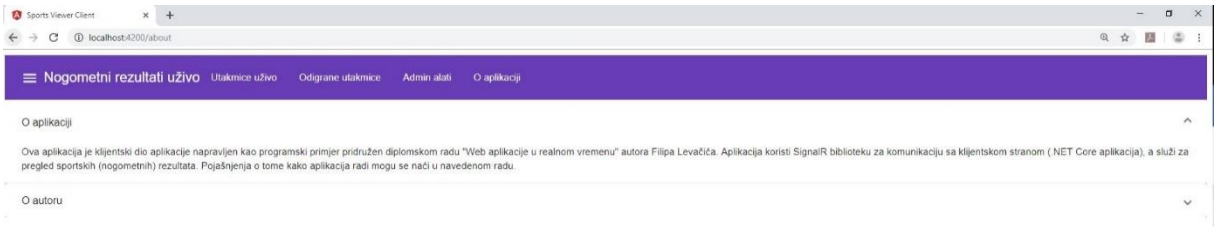
```
public startConnection() {

    this.hubConnection = new signalR.HubConnectionBuilder()
                            .configureLogging(signalR.LogLevel.Debug)
                            .withUrl('https://localhost:5001/sportviewer')
                            .build();


    this.hubConnection.on('liveGames', (message: string) => {
            this.dataService.changeCurrentGame(message)
            });
      // ostali slusaci


    this.hubConnection
      .start()
      .then(() => this.getAllLiveGames())
      .catch(err => console.log('Error while starting connection: ' + err))
}
```

Metoda *startConnection()* poziva se u osnovnoj komponenti aplikacije, a počinje stvaranjem SignalR veze prema poslužitelju. Ovdje konfiguriram razinu izvještavanja (eng. *logging*) koju želim, što je u ovom slučaju *Debug* jer je najkorisniji kod razvoja aplikacije. Definiram i URL poslužitelja koji se sastoji od njegove adrese, porta te naziva *hub*-a na koji se želimo spojiti. Metodom *build()* veza se izgradi te je spremamo u za to pripremljenu varijablu. Nadalje, definiramo slušače događaja koji stižu sa poslužitelja. U prijašnjem dijelu rada, u kojem sam objašnjavao poslužiteljski dio ove aplikacije, objasnio sam da se kod slanja poruke prema klijentu mora navesti tip te poruke. Ovdje definiramo slušače koji čekaju na dolazak određenih tipova poruka koje smo sami definirali, te zatim možemo iz poruke izvaditi njezin sadržaj, ovdje predstavljen vrijednošću *message*, sa kojim onda radimo što trebamo. U ovaj isječak stavio sam samo jedan slušač jer svi ostali izgledaju jednako, a razlikuju se jedino po tipu poruke čiji dolazak čekaju. Postoji slušač za svaku metodu *hub*-a. Ovdje pozivam metodu *DataService* servisa koja će spremiti novu vrijednost utakmica u trajanju te o tome obavijestiti sve komponente koje su pretplaćene na taj podatak. Zadnji dio isječka programskog koda pokreće vezu prema *hub*-u. Nakon spajanja on dohvaća sve utakmice u trajanju jer je to podatak koji uvijek mora biti prisutan u aplikaciji, a to radi pozivom metode *getAllLiveGames()* definiranoj u ovom servisu. U slučaju da kod povezivanja sa poslužiteljem dođe do greške, nju hvatamo te ispisujemo u konzolu kojoj možemo pristupiti preko njezine kartice u sklopu alata za razvojne inženjere dostupnih u Web pregledniku. Metoda *getAllLiveGames()* jedna je u nizu sličnih metoda koje se podudaraju sa metodama *hub*-a koje sam već objašnjavao. Razlog podudaranju je taj što svaka ta metoda poziva određenu metodu *hub*-a. Spomenuta metoda poziva metodu *SendAllLiveGames()* koja klijentu zatim vraća sve utakmice u trajanju. Ovdje je vidljiva lakoća komuniciranja između klijenta i poslužitelja gdje oni međusobno mogu pozivati metode drugoga. Metoda *getAllLiveGames()* zapravo poziva metodu *callServerMethod()* koju sam napisao radi lakše čitljivosti koda, a ona izgleda ovako:

```
private callServerMethod(type: string) {
     this.hubConnection
    .invoke(type)
    .catch(err => console.error(err));
}
```

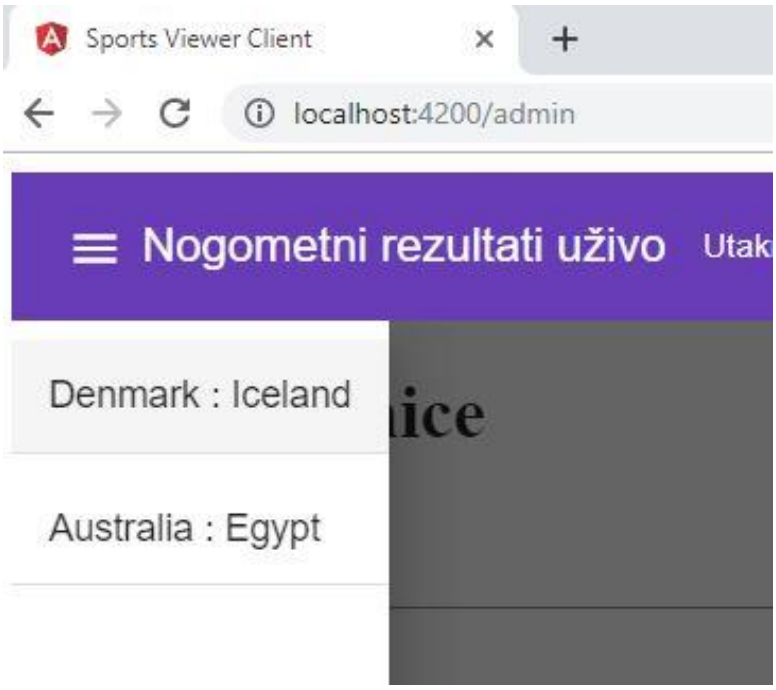
Ovdje se poziva metoda *invoke* veze prema *hub*-u koja poziva metodu čije je ime ovdje spremljeno u parametar *type*. Kao što je bilo spomenuto kod objašnjenja poslužitelja, tim metodama moguće je proslijediti i parametre što se radi dodavanjem parametara nakon naziva metode u pozivu metode *invoke*, a može se upisati proizvoljan broj parametara, tako dugo dok se oni podudaraju sa onima u metodi koja se nalazi na poslužiteljskoj strani aplikacije.

Pokretanje aplikacije vrši se naredbom *ng serve* koja se upisuje u terminal, a ona servira aplikaciju na adresu koju smo naveli kod postavljanja aplikacije. Ovdje nisam mijenjao vrijednosti te se aplikacija servira na adresu *localhost* i port 4200 što je standard kod razvoja Angular aplikacija. Nakon što serviranje aplikacije završi, nju je moguće pokrenuti preko njezine Web adrese unutar Web poslužitelja. Početna stranica sastoji se od komponente *About* koja sadrži osnovne informacije o aplikaciji i njezinom autoru, a izgleda ovako:



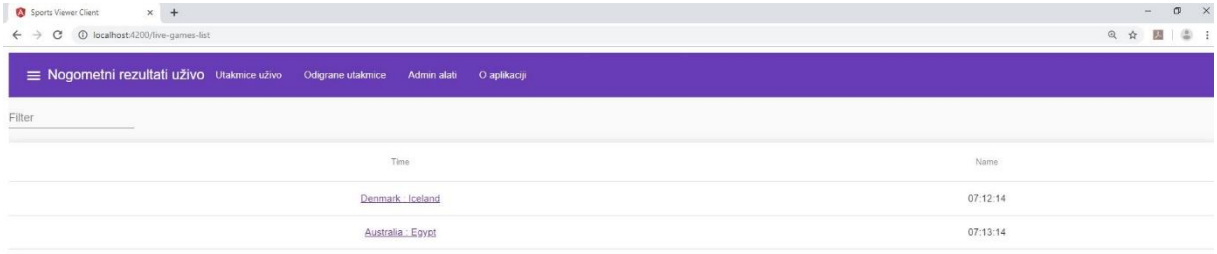
Slika 25: About *komponenta aplikacije*

Kod učitavanja ove komponente učitava se i osnovna komponenta aplikacije koja ostvaruje vezu prema poslužitelju i dohvaća podatke o svim utakmicama koje trenutno traju. Pokreće se i dijalog u koji je potrebno upisati korisničko ime, a ukoliko ono započinje sa riječi *admin*, na alatnoj traci se dodaje i poveznica na komponentu sa administratorskim alatima aplikacije. Na alatnoj traci aplikacije možemo vidjeti navigacijski izbornik čije opcije vode do drugih komponenti aplikacije, a klikom na gumb u gornjem lijevom kutu možemo vidjeti još jedan izbornik koji sadrži popis svih utakmica u trajanju, što je vidljivo na sljedećoj slici:



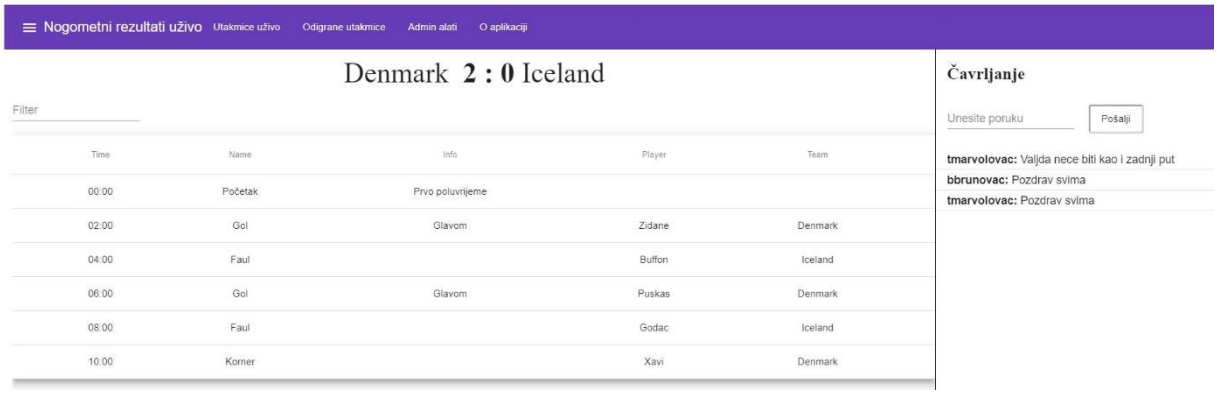
Slika 26: Izbornik aplikacije koji prikazuje sve utakmice u trajanju

Prva komponenta po redu je komponenta *LiveGamesList* koja prikazuje popis svih utakmica u trajanju. Ona se kod inicijalizacije pretplaćuje na popis svih utakmica u trajanju, a ukoliko se dogodi do promjene u tim podacima, ona ih primi ih spremi u izvor podataka (eng. *datasource*) tablice koja prikazuje utakmice. Ta komponenta izgleda ovako:



Slika 27: LiveGamesList *komponenta aplikacije*

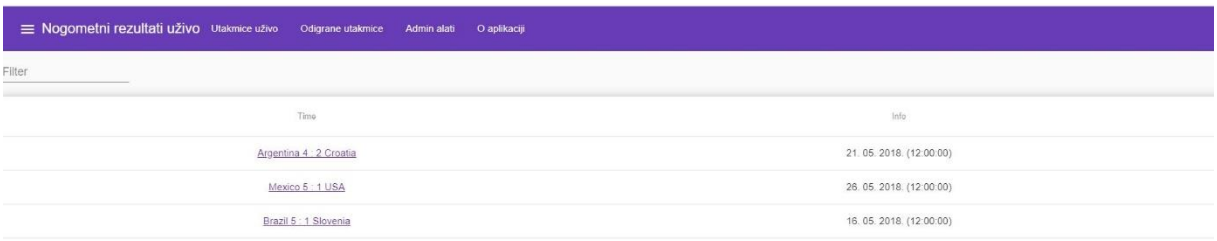
Na početku rada aplikacije, čak i prije razmjene poruka između klijenta i poslužitelja iniciranih od strane nas kao korisnika, poslužitelj generira određen broj utakmica i drugih podataka koje služe za razvoj, testiranje i demonstraciju aplikacije, a takve su i ovdje vidljive utakmice. Vidljivo je vrijeme početka utakmice kao i ime utakmice koje se sastoji od imena ekipa koje igraju, a to ime je ujedno i poveznica do sljedeće komponente aplikacije u kojoj je moguće praćenje događaja utakmice u trajanju te čavrljanje o toj utakmici. Korisničko sučelje te komponente vidljivo je na sljedećoj slici:



Slika 28: LiveGame *komponenta aplikacije*

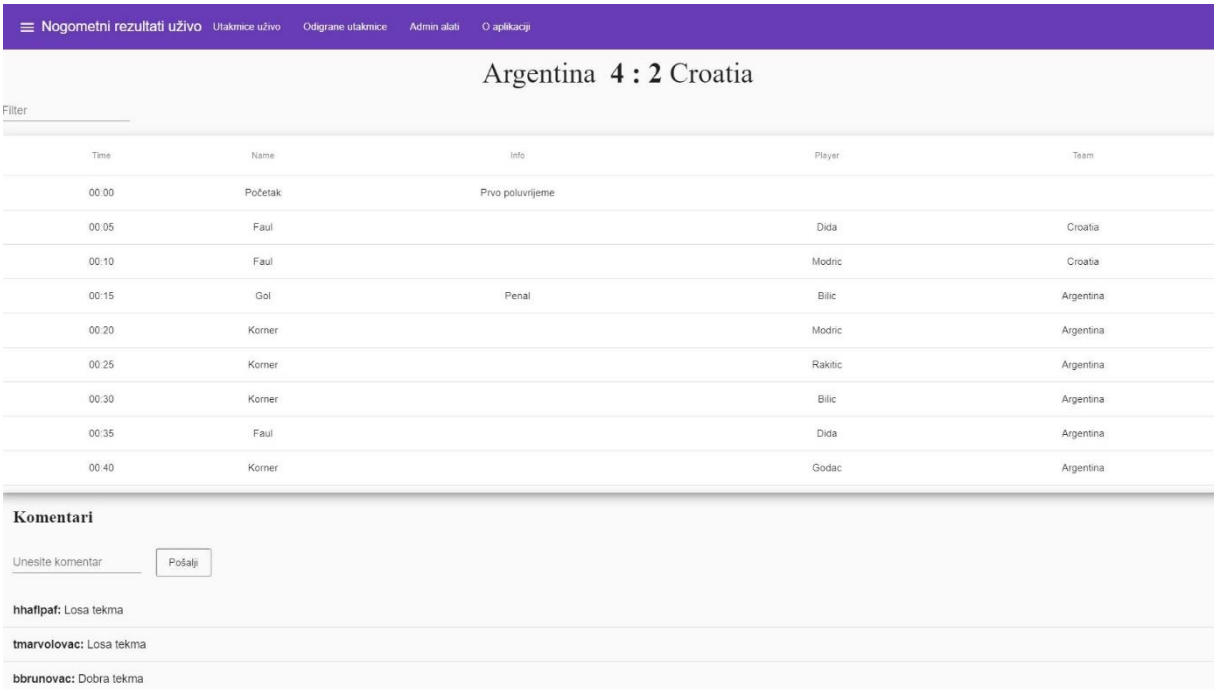
Ova komponenta pretplaćena je na događaje utakmice koju smo odabrali, a oni su prikazani u tablici. Kada se dogodi novi događaj na utakmici, on se pojavi na dnu tablice, a u njoj se prikazuju važni podaci o događajima poput tipa događaja, dodatnih informacija, vremena događaja, te igrača i ekipe koji su zaslužni za događaj. Ukoliko se radi o događaju „Gol“, osvježava se i rezultat utakmice, ovisno o timu koji je zabio gol, kako bi odgovarao stvarnom stanju, a vidljiv je na vrhu komponente. Ukoliko se radu o događaju „Kraj“ te dodatnoj informaciji „Drugo poluvrijeme“, znači da je utakmici došao kraj te se ona više ne smatra utakmicom u trajanju, no klijenti mogu još ostati u ovoj komponenti i čavrljati o utakmici.

Čavrljanje se nalazi na desnoj strani komponente. Kada se komponenta inicijalizira, klijenta se stavlja u grupu za čavrljanje za ovu utakmicu tako da on prima samo *chat* poruke vezane uz ovu utakmicu, a i on sam može slati poruke drugim klijentima. Nakon što klijent otiđe na neku drugu komponentu, njega se izbacuje iz grupe za čavrljanje o trenutnoj utakmici. Sljedeća komponenta predstavlja listu odigranih utakmica, a izgleda ovako:



Slika 29: PastGamesList *komponenta aplikacije*

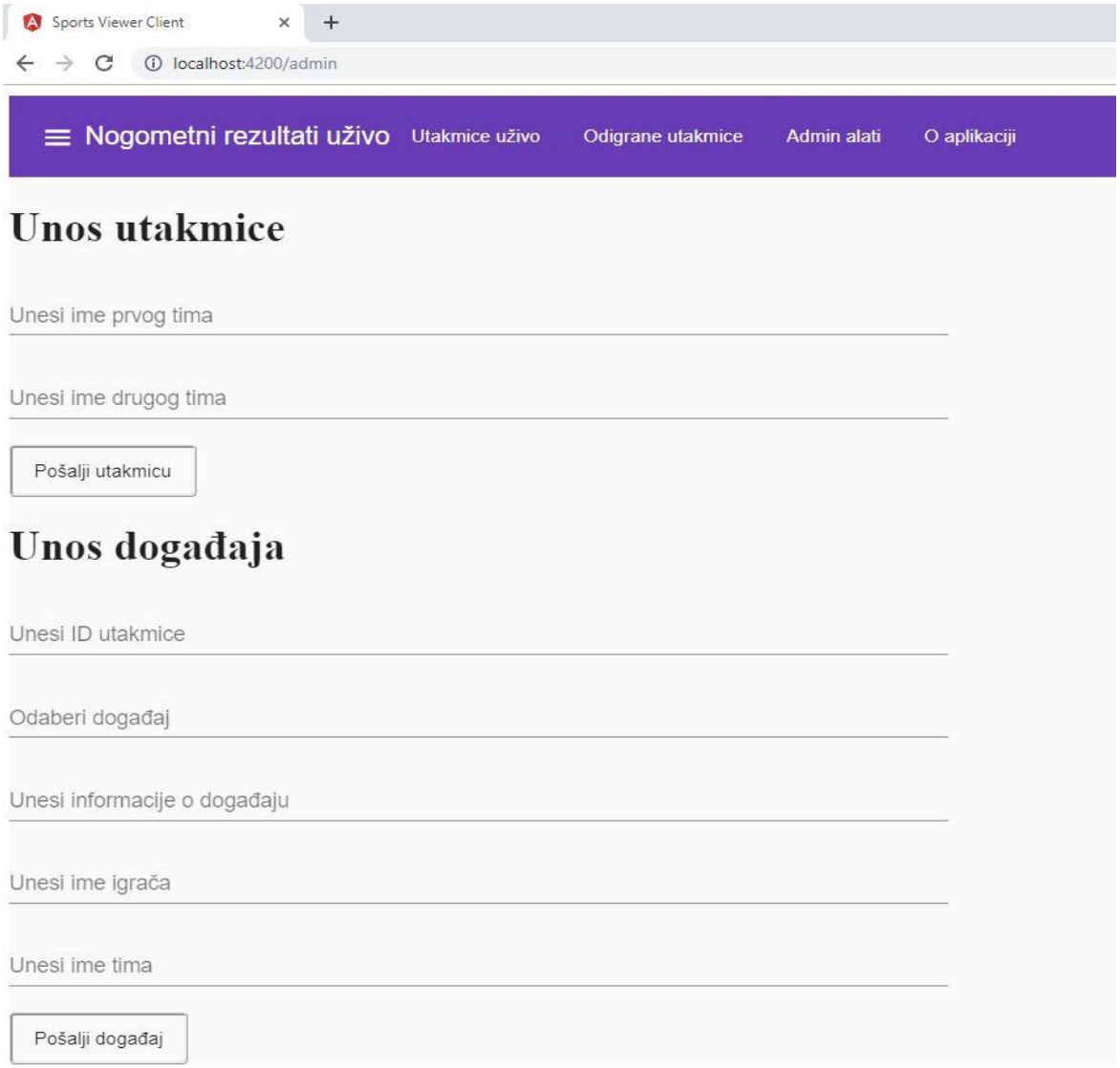
Ova komponenta vrlo je slična komponenti koja prikazuje popis svih utakmica u trajanju, no prikazuje odigrane utakmice. Kada utakmica završi, ona se odmah prikazuje u ovoj listi, a moguće je pristupiti i određenoj odigranoj utakmici i pregledati njezine događaje u *PastGame* komponenti. Korisničko sučelje te komponente izgleda ovako:



Slika 30: PastGame *komponenta aplikacije*

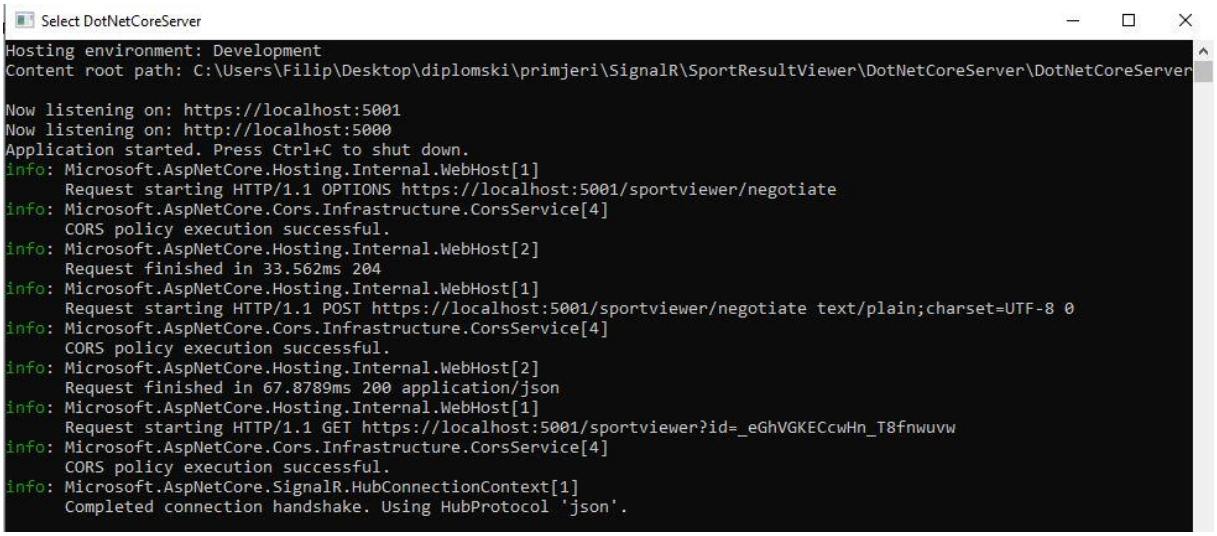
Ova komponenta, osim pregleda rezultata i događaja odigrane utakmice, omogućava i pisanje i pregled komentara na utakmicu. Ova funkcionalnost radi na jednak način kao i *chat* funkcionalnost na komponenti za pregled utakmice u trajanju, no napravio sam je kao primjer stvaranja različitih grupa u kojima klijent može biti član. Kada se klijent prebaci na neku drugu komponentu aplikacije, on će biti izbačen iz grupe za komentare trenutne utakmice.

Zadnja komponenta aplikacije je komponenta sa administratorskim alatima. Ovdje je moguće dodavanje novih utakmica te dodavanje novih događaja za bilo koju utakmicu u trajanju. U stvarnoj, komercijalnoj aplikaciji, ova funkcionalnost najvjerojatnije bi se nalazila zasebno od klijentske aplikacije te bi možda čak automatski generirala te događaje na temelju određenih Web servisa, no za potrebe demonstracije ova komponenta je vrlo korisna jer je moguće prikazati način komunikacije klijenta sa poslužiteljem kada se pojavi nova utakmica ili događaj utakmice. Svaki put kada dodamo novu utakmicu, zahtjev se šalje poslužitelju koji je dodaje u svoju listu utakmica u trajanju te svim klijentima vraća popis utakmica u trajanju pa je ona odmah vidljiva u listi utakmica u trajanju, a svaki novi događaj također je obrađen na poslužitelju, dodan pripadnoj utakmici te poslan klijentima koji prate tu utakmicu. Ponavljam, poveznica na alatnoj traci do ovoj dijela aplikacije vidljiva je samo ako korisničko ime klijenta započinje sa riječi *admin*. Korisničko sučelje ove komponente izgleda ovako:



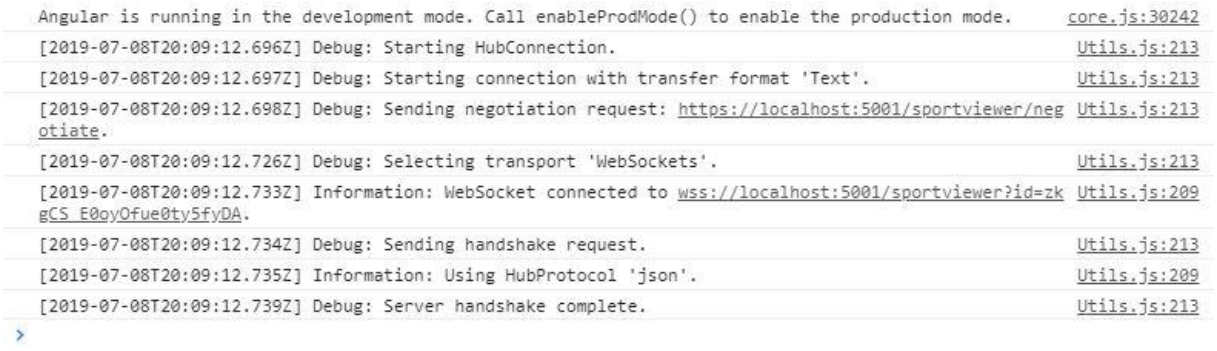
Slika 31: Admin *komponenta aplikacije*

Ono što je još zanimljivo pogledati su poruke koje su generirane i na poslužiteljskoj i klijentskoj strani u slučaju spajanja klijenta na poslužitelj. Te poruke prikazuju način ostvarivanja veze i dogovaranja načina komunikacije koji je skriven od korisnika aplikacije. Kada se novi klijent spoji na poslužitelj, kod poslužiteljske aplikacije možemo vidjeti sljedeće:



Slika 32: Poruke o spajanju klijenta na poslužitelj na poslužiteljskoj strani aplikacije

Na slici možemo vidjeti tijek događaja koji su doveli do uspješnog ostvarivanja veze. Klijent prema poslužitelju šalje zahtjev o pregovoru oko ostvarivanja veze, a zatim se zadaje CORS politika korištenja koja propušta klijentski zahtjev do poslužitelja. Nakon toga se razmjenjuju podaci o formatu poruka koje će se razmjenjivati, što je u ovom slučaju običan tekst u kodnom zapisu UTF-8. Zatim je vidljiv završetak rukovanja te poruka o tome da je JSON format dogovoren za prijenos poruka. Ta komunikacija, sa još nekoliko dodatnih informacija, na klijentskoj je strani vidljiva u *Console* kartici unutar alata za razvojne inženjere Web preglednika. Na klijentskoj strani vidljiva je i odabrana komunikacijska tehnika, u ovom slučaju WebSocket protokol. Ta komunikacija vidljiva je na sljedećoj slici:



Slika 33: Poruke o spajanju klijenta na poslužitelj na klijentskoj strani aplikacije

7. Zaključak

Ovaj rad služi kao svojevrsni pregled razvoja Web aplikacija u realnom vremenu od početka do kraja u svakom pogledu. Prošao sam kroz povijest Web-a i Interneta i objasnio razloge nastanka i razvoja Web aplikacija u realnom vremenu, naveo njihove korijene u aplikacijama u realnom vremenu koje su postojale prije nastanka Web-a, te pokušao predstaviti entuzijazam koji je prisutan kod velikog broja razvojnih inženjera koji rade sa takvim aplikacijama. Objasnio sam osnovnu arhitekturu Web aplikacija koje su prisutne i kod Web aplikacija u realnom vremenu, te naveo niz primjera korištenja takvih aplikacija, od osnovnih i često navođenih primjera poput aplikacija za čavrljanje do aplikacija za aukcije koje polako počinju ugrađivati Web aplikacije u realnom vremenu u svoj način rada. Rad sam završio predstavljanjem programskih primjera u raznim bibliotekama, od onih koje su entuzijasti izradili u svoje slobodno vrijeme do službenih biblioteka razvijanih od strane velikih poduzeća za komercijalnu upotrebu. Završnim primjerom nastojao sam prikazati jedan mogući slučaj korištenja ovakvih aplikacija izrađen uz pomoć komplementarnih tehnologija među kojima se ističe SignalR, jedna od najkvalitetnijih biblioteka za izradu Web aplikacija u realnom vremenu u današnjem svijetu.

Web se razvija iz dana u dan, a Web aplikacije već su dugo dio naše svakodnevice. Responzivne i kvalitetne aplikacije definitivno su tome pridonijele, a nastavak na izgradnji infrastrukture Interneta znači samo veće mogućnosti u izradi Web aplikacija. U vremenu koje se često naziva dobom podataka Web aplikacije u realnom vremenu dobivaju nove potencijale slučajeve korištenja svakim danom, a aplikacije koje su nekad postojale kao *desktop* ili mobilne aplikacije prebacuju se na model Web aplikacija. Društvene mreže vrlo su dobar primjer brzog rasta onih koji su iskoristili prednosti koje pružaju komunikacijske tehnike objašnjene u ovom radu te na taj način izradili Web aplikacije u realnom vremenu koje su brzo u popularnosti nadišle one društvene mreže koje su odlučile ignorirati ove mogućnosti. Standardizacijom tehnologija poput WebRTC ili WebSocket protokola više nema previše prepreka u izgradnji Web aplikacija u realnom vremenu što se može lako uočiti u broju takvih aplikacija koje je moguće vidjeti na Web-u.

Ovim radom nastojao sam prenijeti dio entuzijazma koji dijelim zajedno sa velikim brojem razvojnih inženjera diljem svijeta, a tiče se razvoja Web aplikacija kakve su opisane u ovom radu. Razvojem biblioteka poput onih opisanih u ovom radu te razvojem ostalih aspekata Web-a i Interneta više nema previše razloga zašto Web aplikacije koje izrađujemo ne bi iskoristile neke od mnogih mogućnosti Web aplikacija u realnom vremenu te tako višestruko poboljšale korisničko iskustvo svojih korisnika, povećale svoj uspjeh na tržištu, te još više potaknule razvoj ovakvih vrsta aplikacija i Web-a općenito.

Popis literature

I. Microsoft Docs (Azure). (2017). N-tier architecture style - Azure Application Architecture Guide | Microsoft Docs. Preuzeto 26. lipanj 2019., od https://docs.microsoft.com/en-us/azure/architecture/guide/architecture-styles/n-tier

II. Mark Richards. (2015). 1. Layered Architecture - Software Architecture Patterns [Book]. Preuzeto 12. lipanj 2019., od https://www.oreilly.com/library/view/software-architecture-patterns/9781491971437/ch01.html

III. Ivankov, A. (bez dat.). Thick Client vs. Thin Client: Advantages and Disadvantages | Profolus. Preuzeto 26. lipanj 2019., od https://www.profolus.com/topics/thick-client-vs-thin-client-advantages-and-disadvantages/

IV. Shin, K. G., & Ramanathan, P. (1994). Real-Time Computing: A New Discipline of Computer Science and Engineering. *Proceedings of the IEEE*, *82*(1), 6–24.

V. Miller, R. B. (1968). *Response time in man-computer conversational transactions INTRODUCTION AND MAJOR CONCEPTS*.

VI. C. Holmberg, S. Hakansson, G. E. (2015). *RFC7478: Web Real-Time Communication Use Cases and Requirements This*. 1–29.

VII. Carter, M. (2019). Iconic Brixham Fish Market „shout“ auction takes place for the final time - Devon Live. Preuzeto 26. lipanj 2019., od https://www.devonlive.com/news/devon-news/iconic-brixham-fish-market-shout-2934558

VIII. Robie, J. (1998). What is the Document Object Model? Preuzeto 26. lipanj 2019., od https://www.w3.org/TR/WD-DOM/introduction.html

IX. W3C (XMLHttpRequest dokumentacija). (2016). XMLHttpRequest Level 1. Preuzeto 26. lipanj 2019., od https://www.w3.org/TR/XMLHttpRequest/

X. Garrett, J. J. (2005). Ajax: A New Approach to Web Applications by. *Seminars in respiratory infections*, *1*(2), 1–5.

XI. Realtime API. (2019). HTTP Long-polling - Realtime API Hub. Preuzeto 26. lipanj 2019., od https://realtimeapi.io/hub/http-long-polling/

XII. Microsoft Docs (SignalR). (2019). Use ASP.NET Core SignalR with TypeScript and Webpack | Microsoft Docs. Preuzeto 27. lipanj 2019., od https://docs.microsoft.com/en-us/aspnet/core/tutorials/signalr-typescript-webpack?view=aspnetcore-2.2&tabs=visual-studio

XIII. Roth, G. (2008). Asynchronous HTTP Comet architectures | JavaWorld. Preuzeto 26. lipanj 2019., od https://www.javaworld.com/article/2077843/java-se-asynchronous-http-comet-architectures.html

XIV. Kevin Griffin. (2015). SignalR Transports Explained | Kevin Griffin | Developer, Trainer, Entrepreneur. Preuzeto 12. lipanj 2019., od https://kevgriffin.com/signalr-transports-explained/

XV. Gravelle, R. (2017). Comet Programming: the Hidden IFrame Technique | WebReference. Preuzeto 26. lipanj 2019., od http://webreference.com/programming/javascript/rg30/index-2.html

XVI. W3C (SSE dokumentacija). (2015). Server-Sent Events. Preuzeto 26. lipanj 2019., od https://www.w3.org/TR/eventsource/

XVII. MDN web docs (CORS dokumentacija). (2019). Cross-Origin Resource Sharing (CORS) - HTTP | MDN. Preuzeto 12. lipanj 2019., od https://developer.mozilla.org/en-US/docs/Web/HTTP/CORS

XVIII. MDN web docs (EventSource dokumentacija). (2019). EventSource - Web APIs | MDN. Preuzeto 26. lipanj 2019., od https://developer.mozilla.org/en-US/docs/Web/API/EventSource

XIX. EDUCBA. (2018). WebSocket vs REST | Learn The 8 Important Differences. Preuzeto 12. lipanj 2019., od https://www.educba.com/websocket-vs-rest/

XX. MDN web docs (CloseEvent dokumentacija). (2019). CloseEvent - Web APIs | MDN. Preuzeto 12. lipanj 2019., od https://developer.mozilla.org/en-US/docs/Web/API/CloseEvent#Status_codes

XXI. MDN web docs (WebSocket dokumentacija). (2019). WebSocket - Web APIs | MDN. Preuzeto 26. lipanj 2019., od https://developer.mozilla.org/en-US/docs/Web/API/WebSocket

XXII. W3C (WebRTC dokumentacija). (2017). WebRTC 1.0: Real-time Communication Between Browsers. Preuzeto 26. lipanj 2019., od https://www.w3.org/TR/2017/CR-webrtc-20171102/

XXIII. Hempel, W. (2016). An overview of realtime libraries and frameworks. Preuzeto 27. lipanj 2019., od https://deepstreamhub.com/blog/realtime-framework-overview/

XXIV. *sockjs* (GitHub korisnik). (2019). GitHub - sockjs/sockjs-client: WebSocket emulation - Javascript client. Preuzeto 26. lipanj 2019., od https://github.com/sockjs/sockjs-client

XXV. Trường, T. X. (2014). Simple chat application using SockJS. Preuzeto 26. lipanj 2019., od https://truongtx.me/2014/06/07/simple-chat-application-using-sockjs

XXVI. Patel, P. (2018). What exactly is Node.js? Preuzeto 26. lipanj 2019., od https://www.freecodecamp.org/news/what-exactly-is-node-js-ae36e97449f5/

XXVII. Kelleher, F. (2014). Understanding Socket.IO - NodeSource. Preuzeto 26. lipanj 2019., od https://nodesource.com/blog/understanding-socketio/

XXVIII. Socket.IO dokumentacija. (2019). Socket.IO — Chat | Socket.IO. Preuzeto 26. lipanj 2019., od https://socket.io/get-started/chat

XXIX. TutorialsPoint. (2014). Node.js Express Framework. Preuzeto 26. lipanj 2019., od https://www.tutorialspoint.com/nodejs/nodejs_express_framework.htm

XXX. BloogGeek. (2017). What is WebRTC and What is it Good For? • BlogGeek.me. Preuzeto 26. lipanj 2019., od https://bloggeek.me/what-is-webrtc/

XXXI. Wanyoike, M. (2018). Building a WebRTC Video Chat Application with SimpleWebRTC — SitePoint. Preuzeto 26. lipanj 2019., od https://www.sitepoint.com/webrtc-video-chat-application-simplewebrtc/

XXXII. *gwarser* (GitHub korisnik). (2019). Prevent WebRTC from leaking local IP address · gorhill/uBlock Wiki · GitHub. Preuzeto 26. lipanj 2019., od https://github.com/gorhill/uBlock/wiki/Prevent-WebRTC-from-leaking-local-IP-address

XXXIII. *tuchfarber* (GitHub korisnik). (2017). GitHub - tuchfarber/websocket_chat: Simple proof of concept websocket chat application. Preuzeto 26. lipanj 2019., od https://github.com/tuchfarber/websocket_chat

XXXIV. Python dokumentacija. (2019). asyncio — Asynchronous I/O — Python 3.7.4rc1 documentation. Preuzeto 26. lipanj 2019., od https://docs.python.org/3/library/asyncio.html

XXXV. websockets (Python dokumentacija). (2018). WebSockets — websockets 7.0 documentation. Preuzeto 26. lipanj 2019., od https://websockets.readthedocs.io/en/stable/

XXXVI. Grinberg, M. (2018). python-socketio — python-socketio documentation. Preuzeto 26. lipanj 2019., od https://python-socketio.readthedocs.io/en/latest/

XXXVII. WebSocket API (Java dokumentacija). (2011). Using the WebSocket API in a Web Application. Preuzeto 26. lipanj 2019., od https://netbeans.org/kb/docs/javaee/maven-websocketapi.html

XXXVIII. Chu, A. (2018). Differences Between ASP.NET SignalR and ASP.NET Core SignalR. Preuzeto 08. srpanj 2019., od https://www.codemag.com/Article/1807061/Build-Real-time-Applications-with-ASP.NET-Core-SignalR

XXXIX. SignalR dokumentacija (uvod). (2018). Introduction to ASP.NET Core SignalR | Microsoft Docs. Preuzeto 08. srpanj 2019., od https://docs.microsoft.com/en-us/aspnet/core/signalr/introduction?view=aspnetcore-2.2

XL. Spasojevic, M. (bez dat.). How to Use SignalR with .NET Core and Angular - Real-Time Charts. Preuzeto 08. srpanj 2019., od 2018 website: https://code-maze.com/netcore-signalr-angular/

XLI. SignalR dokumentaciju (Hub). (bez dat.). Use hubs in ASP.NET Core SignalR | Microsoft Docs. Preuzeto 08. srpanj 2019., od 2018 website: https://docs.microsoft.com/en-us/aspnet/core/signalr/hubs?view=aspnetcore-2.2

XLII. Aldwin N. (2019). What is Angular: All You Need to Know About the Popular JS Framework. Preuzeto 08. srpanj 2019., od https://www.hostinger.com/tutorials/what-is-angular

XLIII. Angular Material dokumentacija. (2019). Components | Angular Material. Preuzeto 08. srpanj 2019., od https://material.angular.io/components/categories

Popis slika