**SVEUČILIŠTE U ZAGREBU, FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA**

36. Chat

**Izvješće**

Bruna Anđelić

Matija Šekrst

Tea Pović

Josipa Kelava

Luka Kovačić

Karlo Mravunac

**Zagreb, siječanj 2017.**

**React.js**

# Uvod

U ovom će se izvješću obrađivati tema React.js koji je danas sve popularnija i korištenija JavaScript programska knjižnica. Kroz ovo izvješće u idućih nekoliko poglavlja opisat će se ukratko što je to React.js i koja je glavna razlika između njega i ostalih JavaScript programskih knjižnica. Nadalje objasniti će se kratka povijest React.js-a. Kako se uopće došlo do ideje za React.js, tko ga je razvio te kako se i danas ravija. Također budući da je React.js prilično velika knjižnica sa mnoštvom različitih mogućnosti te koncepata koji se koriste u jednom će se poglavlju objasniti one najosnovnije stvari vezane uz programiranje u React.js-u. Također uz React.js veže se i posebni arhitekturni obrazac koji se uz React.js najviše koristi, a to je Flux koji će također biti pobliže objašnjen u jednom poglavlju.

# Što je React.js?

React.js je *open-source* JavaScript programska knjižnica koja se koristi za *render* HTML podataka. Korištenjem React.js-a stvaramo interaktivne komponente sa stanjem(*statefull*) koje se mogu ponovno iskoristiti. Jedna od važnih prednosti React.js-a je činjenica da komponenta vraća izravno kao HTML pa se tako isti kod može pokretati i na poslužitelju (*server-side rendering*) i na klijent(*client-side rendering*), ali i izomorfno kombiniranjem najboljem iz oba pristupa.

Jedna od glavnih karakteristika React.js-a je činjenica da on predstavlja samo *view* sloj MVC arhitekture i to je jedna od glavnih razlika između React.js-a i ostalih JavaScript knjižnica kao što su Angular ili Backbone koji predstavljaju potpunu arhitekturu uključujući i *view* sloj. Iz tog se razloga ne može izgraditi potpuno funkcionalna dinamička aplikacija korištenjem isključivo React.js-a. Iako se iz ove usporedbe može zaključiti da se druge JavaScript knjižnice korisnije i bolje za koristiti to nije tako. Teško ih je uspoređivati direktno jer rade na drugim principima tako da se ne može reći da se za React.js ne može reći da je bolji ili gori od drugih, ali zbog svojih karakteristika koje će se opisati u sljedećim podpoglavljima postaje sve popularnija i sve korištenija JavaScript programska knjižnica.

Osnovna karakteristika React.js-a je jednosmjerni protok podataka. Svojstva(*properties*), koja su skup nepromjenjivih (*immutable*) vrijednosti prenose se između komponenti kao svojstva unutar HTML[[1]](#footnote-2) oznaka. Upravo su svojstva(*properties, props*) zajedno sa stanjem(*state*) jedini način komunikacije među komponentama, a komponente su osnovna jedinica koju React.js koristi. Budući da se upravo na komponentima, svojstvima i stanju temelji React.js te da se ti koncepti najbitniji prilikom programiranja u React.js-u oni su pojedinačno opisani u 4. poglavlju. Najvažnije je za neki opće pregled shvatiti da se sva svojstva(*props*) mogu prenositi samo u jednom smjeru. Komponenta ne može direktno mijenjati niti jednu vrijednost svojstva koja joj je proslijeđena, ali može proslijediti *callback* funkciju koja će modificirati vrijednost. Taj mehanizam koji React.js koristi poznat je pod nazivom „*properties flow down, actions flow up*“ i on možda najslikovitije opisuje mehanizam. Dakle vrijednosti svojstava(*props*) prenose se u jednom smjeru, ali se *callback* funkcijom akcije prenose u smjeru suprotnom smjeru protoka podataka.

Nadalje React.js koristi virtualni DOM[[2]](#footnote-3) koji omogućuje selektivni *render* komponenata baziran na promjenama stanja. React kreira *in-memory* strukturu podataka(*cache*) i određuje razlike između dosadašnjem i novog DOM-a te ažurira(*update*) potrebne razlike i na taj se način DOM učinkovito prikazuje. Ovim pristupom programer može pisati kao da će se cijela stranica prevesti ponovno na svaku promjenu podataka dok React zapravo ažurira u DOM-u samo dijelove komponenti koje se zapravo mijenjaju. Također komponente se u React.js-u pišu u JSX-u. On dozvoljava navođenje HTML-a korištenjem HTML oznaka kako. JSX je statični, objektno orijentirani programski jezik dizajniran kako bi se izvodio u modernim web preglednicima. Bitno je napomenuti i da je osnovna arhitektura React.js aplikacije nadilazi samo prikazivanje HTML-a u web preglednicima te uvodi dinamičnost.

# Povijest React.js-a

React.js razvio je softver inženjer iz Facebooka[[3]](#footnote-4), Jordan Walke. Osnovi utjecaj na njegov rad bio je XHP, PHP razvojni okvir(*framework*) za HTML komponente. U vrijeme kada ga je razvijao, Jordan Walke radio je na dijelu za oglase, ali se prebacio u „*Product Infra team*“ kako bi se mogao *full-time* posvetiti razvoju React.js-a. Prilikom razvoja prva stvar na kojoj je testirao React bila je React verzija za „*lajkanje“* i komentiranje. To je bio prvi korak u prevođenju Facebooka na React.js.

Do te prve uporabe React.js-a Instagram[[4]](#footnote-5) se pripojio Facebook-u te je bivši inženjer Facebook tima za fotografije, Pete Hunt, prešao u Instagram tim te napravio prvu verziju Instagram obavijesti za web koristeći upravo React. Budući da je kod za Instagram bio potpuno odvojen od Facebook koda zatražio je da se React izdvoji iz Facebook koda kako bi ga se moglo koristiti i za razvoj Instagrama. Tako gledajući može se reći da je Instagram prvi „vanjski“ korisnik React.js-a. Taj je potez bio začetnik ideje za „*open-source*“ React.js-a koji je omogućio da ga danas može koristiti cijeli svijet.

Prvo korištenje React.js-a u sklopu Facebooka bilo je 2011.godine dok se prvi puta za Instagram iskoristio 2012.godine. *Open-source* je postao na JSConf US u svibnju 2013.godine.

Danas je React.js jedan od najpopularnijih repozitorija na Git-u te danas ima 876 suradnika(„*contributor*a“)[[5]](#footnote-6). Broj stranica koje koriste React.js povećava se iz dana u dan, a neke od popularnijih su: BBC[[6]](#footnote-7), eBay, Instagram, Facebook, IMDb[[7]](#footnote-8), Imgur[[8]](#footnote-9) i mnoge druge. Popis web stranica napisanih u React.js-u dostupan je putem poveznice „*https://github.com/facebook/react/wiki/Sites-Using-React“*.

Također je važno napomenuti da se iz React.js-a razvio i React Native koji omogućuje stvaranje *nativnih* mobilnih aplikacija korištenjem JavaScripta i React.js-a te tako omogućuje razvoj iOS, Android i UWP[[9]](#footnote-10) aplikacija. React Native prvi puta je predstavljen na „*React.js Conf*“ u veljači 2015. godine, a postao je *open-source* u svibnju iste godine.

# Osnovni dijelovi React.js programskom koda

Kao što je prija napomenuto osnovna jedinica kojom se gradi Reac.js aplikacija naziva se komponenta. Svaka komponenta ima stanje(*state*) i svojstva(*props*). Stanje se postavlja i mijenja korištenjem posebne metode „*setState*“ koja ja okidač(*trigger*) za ažuriranje(*update*) UI-a[[10]](#footnote-11). Kao što smo naglasili DOM se ažurira selektivno ovisno o promjenama u stanju. Svojstva se koriste za definiranje komponenti korištenjem atributa *props* koji su unutar komponenta dostupni uporabom *this.props* i mogu se koristiti u *render* metodi kako bi se prevodio dinamički sadržaj.

Render metoda je jedini dio koji je obavezano dodati kako bi se kreirala nova komponenta. Unutar metode render potrebno je imati *return* koji vraća dio koda koji vizualno predstavlja komponentu kojoj je pridružen. Uz render metodu mogu se koristiti i *lifecycle* metode koji su jako korisne kada se od komponente traži da nešto napravi. Glavne takve metode su : *componentWillMount, componentDidMount, shouldComponentUpdate* i *componentWillUnmount*. Takve metode govore nam kada je komponenta u jednom od stavnja. Primjerice *componentWillMount* poziva se jednom na klijentskoj i serverskoj strani prije nego što će se pozvati metoda render, dok se metoda *componentDidMount* poziva jednom samo na klijentskoj strani nakon što se izvrši metoda render. Nadalje metoda *shouldComponenetUpdate* vraća vrijednost koja govori o tome treba li se ili ne komponenta ažurirati, a metoda *componentWillUnmount* poziva se prije nego što će se komponenta *unmountati.* Korištenjem ovih metoda točno se zna u kojem će se trenutku pozvati kod koji im je pridružen što može biti jako korisno i praktično.

Rukovanje događajima korištenjem React.js-a vrlo je slično rukovanjem događajima nad DOM elementima uz iznimku malo sintaksne razlike. React događaji imenuju se korištenjem *camelCase*, a ne *lowecase* te se pomoću JSX-a funkcije prenose kao *event handler*, a ne kao *string*.

U poglavlju su opisana neka od glavnih dijelova koda, ali React.js ima mnoštvo različitih dijelova i mogućnost pa su i neke od osnovnih koncepata nekom tko nikada nije koristio React.js prilično apstraktne. U poglavlju se samo pokušalo navesti osnovne ideje i dijelove koda. Najbitnije je shvatiti da se komponenta uvijek sastoji u nekom stanju te da se ovisno o stanju ona može mijenjati te da se osim stanja informacije prenose *propsima*. Ostale se karakteristike koda pisanog React.js-om može lako naučiti proučavanjem službene Facebook dokumentacije[[11]](#footnote-12) koja pobliže objašnjava svaku od komponenata koda.

Komponente React.js-a definirane su u *app/src* direktoriju. Pogledajmo kako se *props* atributima prenose podatci iz *ChatContainer.jsx* u komponentu *Chat.jsx:*

**if** (**this**.**state**.messages) {

**return** (

<**div className="container centered chat"**>

<**Chat messages=**{**this**.**state**.messages} **active=**{**this**.**state**.**active**}

**received=**{**this**.**state**.**received**} **open=**{**this**.openConversation} **openPrivate=**{**this**.openPrivate}

**uploadFile=**{**this**.uploadFile}

**private=**{**false**}

**click=**{**this**.clickHandler} **sender=**{**this**.sendMessage} **users=**{**this**.**state**.**users**}/>

</**div**>

);

Sve definirano kao atribut u tagovima <Chat ... /> unutar komponente *ChatContainer* se dohvaća unutar komponente *Chat* kao *propsi* odnosno kao this.props.[imeAtributa]. Jedan primjer dohvata *propsa* dan je u slijedećem odsječku koda.

renderActive() {

**if**(**this**.**props**.**active** && **this**.**props**.**private**){

**return** <**Conversation sender=**{**this**.**props**.**sender**} **active=**{**this**.**props**.**active**} **uploadFile=**{**this**.**props**.uploadFile} **styleName=**{**"darkInput"**} **resize=**{**"fullScreen"**}/>;

}

**if** (**this**.**props**.**active**) {

**return** <**Conversation sender=**{**this**.**props**.**sender**} **active=**{**this**.**props**.**active**} **uploadFile=**{**this**.**props**.uploadFile} />;

}

}

# Što je Flux?

Flux je arhitekturni obrazac koju Facebook koristi prilikom rada s React.js-om. Važno je napomenuti da to nije niti *framework* niti *library*, već arhitekturni obrazac koja se slaže s React.js-om i konceptom jednosmjernog protoka podataka. Flux arhitekturni obrazac može se implementirati na više načina, a Redux[[12]](#footnote-13) implementacija koristi se u Facebook-u te je najkorištenije implementacija tog obrasca.

Glavna ideja je, kao što je napomenuto jednosmjerni protok podataka. Jedan uobičajeni slučaj korištenja bio bi da korisnik pokreće akciju(*action*) koja zatim šalje podatke u *reducer* koji služi kao sabirna točka svih ackija. Svaki *reducer* na osnovu akcije i trenutnog stanja aplikacije stvara novo stanje. Bitno je naglasiti dodatno činjenicu da se stvara novo stanje, dakle nema modifikacije trenutnog stanja. To se novo stanje šalje u spremište(*store*). Spremište je zaduženo za prijenos novog pogleda korisniku i za aktivaciju nove akcije kada je to potrebno.

Usporedimo li Redux sa arhitekturom MVC vidimo da je jednostavnija, osnovno izbjegava neke složene konstrukcije, ali također ima laku predvidivost. Kao što je naznačeno cijeli je fokus stavljen na trenutno stanje te je samo to bitno. Budući da je protok podataka jednosmjeran lako se predviđaju pogreške te se aplikacije na taj način brže razvijaju.

# Zaključak

React.js jedna je od najmlađih programskih knjižnica JavaScripta te se kao takva još uvijek intenzivno mijenja. U ovom su izvješću nabrojane su mnoge karakteristike React.js-a koje ga danas čine sve više popularnim i korištenim u svijetu web developmenta, ali kada se odluči koristiti React.js potrebno je shvatiti da je se on još uvijek dinamički mijenja te da je potrebno pratiti novosti iz React.js svijeta te pratiti bilo kakve promjene u dokumentaciji kako ne bi imali problema prilikom nekih većih promjena kao što su izbacivanje neke od funkcionalnosti i sl. Kod spomenutom React Native-a situacija je još gora jer je on još mlađi. Ali usprkos spomenutim manama sama činjenica da je React.js toliko raširen i korišten govori o tome da se isplati koristiti ga usprkos manama. Upravo smo se zato i mi u ovom projektu odlučili koristiti ga jer se kod projekta kao što je chat treba dobiti velika brzina osvježavanja sadržaja ekrana koji je vrlo dinamičan.

**Node.js**

# Općenito o Node.js-u

Node.js je okruženje koje nam omogućuje pokretanje JavaScript koda izvan web preglednika. JavaScript se većinom koristi za programiranje interaktivnih web sučelja, a Node.js omogućuje iskorištavanje tog programskog jezika u širem kontekstu - najčešće za programiranje web poslužitelja. Korištenje Node.js-a omogućuje da se cijela web aplikacija izradi korištenjem samo jednog programskog jezika – JavaScripta. JavaScript je jedna od najraširenijih tehnologija s obzirom da ga svaki moderni web preglednik podržava, a pregledavanje weba je danas postala norma prilikom korištenja računala. Uz to programiranje klijentske strane i poslužiteljske strane istom tehnologijom, programski kod može biti dijeljen, odnosno povećava se količina koda koja se može ponovno iskoristiti. Koristeći neki drugi programski jezik za programiranje poslužitelja poput PHP-a, Rubya ili Pythona tako nešto ne bi bilo moguće. Koliko je moćno okruženje Node.js pokazuje i skup njegovih korisnika od kojih možemo spomenuti neke svjetske kompanije poput: Microsoft, Netflix, PayPal.

Kao interpreter Node.js-a, koristi se *Google’s V8 JavaScript engine* zbog čega su aplikacije napisane u Node.js-u i nekoliko puta brže od aplikacija napisanih u ostalim skriptnim jezicima. V8 koristi novije tehnologije prevođenja koje često omogućuju višim programskim jezicima poput JavaScripta da se izvode sličnom brzinom kao i niži programski jezici poput C-a. Viši programski jezici često omogućuju brže razvijanje proizvoda, ali je i njihovo izvođenje često sporije. Brzina izvođenja je ključan aspekt Nodea. Node dodatno uključuje apstrakcije koje omogućuju bolje korištenje JavaScripta u kontekstu izvan web preglednika, na primjer razredi za manipuliranje binarnih podataka što je često potrebno u poslužiteljskom dijelu aplikacije.

Programska paradigma programiranja zasnovanog na događajima svojstvena je za programski jezik JavaScript, a Node iskorištava tu programsku paradigmu za ostvarivanje visoko skalabilnih poslužitelja. Arhitektura koju koristi Node zove se *event loop.* Ta arhitektura omogućuje brzo i sigurno razvijanje visoko skalabilnih poslužitelja. To je vrlo bitna značajka ove tehnologije jer je konkurentno programiranje zahtjevno i sklono pogreškama. Node zaobilazi te izazove zadržavajući dobre performanse. Da bi arhitektura programa razvijenog za okruženje Node mogla biti zasnovana na događajima, Node nudi biblioteke koje apstrahiraju operacije poput pristupanja datotekama na disku u operacije koje nazivamo događajima. To znači da program neće biti blokiran odnosno neće čekati da se ta operacija izvrši, već će biti obaviješten kada operacija završi, a u međuvremenu će se moći izvršavati neka druga radnja. Takav način izvršavanja sličan je onom u web pregledniku tijekom događaja poput klika mišem. Programiranje takvih arhitektura poznato je programerima koji nisu nužno koristili Node, ali su koristili JavaScript. Također, takve arhitekture su intuitivne i sigurne od pogrešaka koje se javljaju kod konkuretnog programiranja pa je vrijeme uloženo za učenje i razvijanje takve arhitekture znatno smanjeno.

Node radi na principu asinkronog programiranja. Glavna dretva postavlja poslove u dijeljeni red iz kojeg druge dretve mogu preuzeti i obaviti te poslove. Po završetku posla, pozvat će se odgovarajuća funkcija u glavnoj dretvi. To znači da procesor neće biti blokiran na operacijama poput čitanja i pisanja jer će te operacije biti obavljene u posebnim dretvama. Glavna dretva bit će obavještena po završetku, a do tad će moći trošiti procesorsku snagu na nešto drugo. Problem s operacijama čitanja i pisanja je taj što moraju koristiti fizičke uređaje poput tvrdog diska koji imaju pomične dijelove i često puno sporije obavljaju operacije od procesora. Tijekom obavljanja jedne operacije tvrdog diska, procesor može obaviti ciklus koji uključuje u milijunima operacija. Zato je bitno ne blokirati procesorsku snagu dok se obavljaju takve operacije.

Prednost asinkronog programiranja je što će procesorska snaga biti bolje iskorištena, ali postoji i nedostatak jer će u slučaju CPU zahtjevnih operacija drugi poslovi biti blokirani dok čekaju u redu da ih glavna dretva izvrši. Pokazalo se da aplikacije programirane ovim pristupom mogu podržavati znatno veći broj istodobnih konekcija od onih koji otvaraju dretvu za svaku konekciju ako takva aplikacija ne obavlja operacije koje su procesorski jako zahtjevne. Asinkrono programiranje također olakšava proces razvoja jer je jednostavnije u odnosu na konkuretno programiranje.

# Instalacija Node.js-a

Node je dostupan na gotovo svim operacijskim sustavima poput Windowsa, Linuxa i OSX-a. Node je tehnologija otvorenog koda i moguće ga je preuzeti direktno sa GitHub repozitorija. Taj kod je najnovija verzija Nodea pa vjerojatno sadrži i neke mogućnosti koje nisu dostupne u stabilnoj verziji, ali vjerojatno nisu još dovoljno testirane i sigurne. Sa službene stranice moguće je preuzeti stabilnu verziju. Za razvojno okruženje moguće je koristiti širok spektar alata s obzirom da danas postoji mnogo ekstenzija koje omogućuju razvoj Node aplikacija u alatima inicijalno namijenjenih za druge tehnologije.

Instalacijom Noda instalirava se i sustav za upravljanje i paktima koji se zove npm. Sam Node je veoma koristan, ali često postoje funkcionalnosti koje nisu dio Nodea, a želimo ih koristiti iznova u razvoju različitih aplikacija. Te funkcionalnosti je dobro razviti zasebno i nastaviti ih nadograđivati. Takav kod se distribuira u paketima preko sustava npm. To je način za korištenje programskih kodova drugih programera, a isto tako i način za dijeljenje svojih kodova. Međusobno dijeljenje i razvijanje koda podiže općenitu kvalitetu proizvoda. Npm koristi package.json dokument i u njega upisuje module koje želimo dodati. U polje dependencies upisuju se ime modula i željena verzija tog istog modula.

# Moduli Node.js-a

## 3.1 Express

Jedan od najpopularnijih modula razvijenih za Node.js je Express. Instalacija Expressa uz korištenja npm-a je vrlo jednostavna. Sve što je potrebno napraviti je izvesti naredbu ‘npm install express’. Express služi kao MVC okruženje za Node. Karakteristike Expressa su jednostavnost, minimalizam, fleksibilnost i skalabilnost.  Osnovno korištenje Expressa sastoji se od definiranja putanja (*eng. routes*) i funkcija za obradu zahtjeva poslanih na te putanje. Funkcije za obradu zahtjeva na nekoj putanji definiramo za HTTP metode. To znači da za neku putanju možemo definirati više funkcija za obradu zahtjeva, naprimjer jednu za obradu HTTP GET zahtjeva a drugu za obradu HTTP POST zahtjeva. Te funkcije su zapravo promatrači HTTP zahtjeva. Koristeći Express ne moramo pisati vlastite mehanizme za usmjeravanje zahtjeva iznova za svaku aplikaciju, već iskorištavamo mehanizme koji su već razvijeni i koji se redovito poboljšavaju i nadograđuju. Primjer jedne jednostavne aplikacije uz korištenje Expressa dan je u nastavku:

var express = require('express');  
var app = express();  
app.get('/api', function (req, res) {  
  res.send('Hello World');  
})

app.listen(10001);

U primjeru možemo vidjeti da aplikacija ima jednu putanju /api. Exress metodom get() osluškuje posjećivanje te putanje. Funkcija koja obrađuje zahjtev upućen prema toj putanji ima parametre req i res ondnosno zahtjev (request) i odgovor (response). Funkcija odgovara sa send() metodom koja sama zaključuje koje podatke treba poslati, naprimjer što treba pisati u zaglavlju http odgovora. Metoda listen(port) nam govori na kojim vratima će polužitelj slušati zahtjeve. Željeni modul nije dovoljno samo instalirati pomoću npm-a. Potrebno ga je i uključiti u programski kod. Upravo to nam omogućava funkcija require(‘express’).

U ovoj aplikaciji modul Express je korišten za definiranje putanja i obradu HTTP zahtjeva. U ovoj aplikaciji smo koristili routere express modula za definiranje ruti u smislenim cjelinama. Rute aplikacije su definirane u direktoriju *chat/api/route –* *auth.js* datoteka implementira rute za login i registraciju korisnika, *users.js* implementira rute za dohvaćanje korisnika i njegovog profila, *messages.js* implementira rute za stvaranje novih poruka i dohvaćanje povijesti razgovora, te *index.js* exporta routere iz navedenih datoteka. U slijedećem isječku koda iz *auth.js* prikazan je post zahtjev za login korisnika:

authRouter.post(**'/login'**,

passport.authenticate(**'local'**, { **session**: **false** }),

authMiddleware.generateToken,

authMiddleware.sendResponse,

);

Korisnik se na početku autentificira i zatim mu se generira token i šalje odgovor. Metode za generiranje tokena i slanje odgovora su definirane u datoteci *middleware.js* koja se nalazi u direktoriju *chat/api/auth*. Idući primjer prikazuje rutu za registraciju korisnika:

authRouter.post(**'/register'**, (req, res) => {

**const** db = *getDb*();

db.collection(**'users'**).findOne({**username**: req.**body**.**username**}, (err, user) => {

**if** (err) {

**return** res.status(500).json({**message**: err});

}

**if** (user) {

**return** res.status(403).json({**message**: **'Username already exists.'**});

} **else** {

**const** newUser = { ...req.**body**, **image**: **''**, **password**: hash.generate(req.**body**.**password**)};

db.collection(**'users'**).insertOne(newUser);

**return** res.status(201).json({ **username**: newUser.**username** });

}

});

},

);

Prvo se provjerava postoji li već u bazi korisnik s takvim korisničkim imenom, ako da onda se u odgovoru šalje poruka da takav korisnik već postoji, a ako ne stvara se novi korisnik s parametrima iz zahtjeva i šalje odgovor.

## 3. 2 MongoDB

Još jedan koristan modul je ‘monogodb’. Za njegovu instalaciju potrebno je napraviti sve kao i za Express, odnosno npm install mongodb. Taj modul nam omogućuje korištenje Node.js-a sa nerelacijskom bazom podataka MongoDB. MongoDB koristi skupine kolekcija. Kolekcija sadrži dokumente koji imaju dinamicku strukturu (BSON - Binary JSON).  Osnovno korištenje MongaDB u aplikaciji prikazno je u sljedećim linijama:

var mongodb = require('mongodb');  
var MongoClient = mongodb.MongoClient;  
MongoClient.connect(url, function (err, db) {  
 if (err) {  
   console.log('error’);  
 } else {  
   console.log('Connection to', url);

//operacije prema bazi   
   var users = db.collection('users');  
        db.close();  
  }  
});

Uključivanje *MongoDB native driver*a radi se pomoću funkcije require kao i kod Expressa. Potrebno je napraviti instancu MongoClienta koja je zapravo sučelje preko kojeg ćemo se spajati na bazu podataka.  MonogoClient se spaja na bazu podataka pomoću urla koji mu se zada kao parametar i sadrži *callback* funkciju u kojoj se odvija komuniciranje s bazom (čitanje ili pisanje). Url predstavlja lokaciju na kojoj se nalazi  baza podataka. *Callback* funkcija sadrži varijablu db u kojoj su svi podaci sa željene baze. Ta funkcija u gore navedebom primjeru sarži jedan primjer dohvaćanja podataka s baze.

## 3.3 Dotenv

## Dotenv je modul Node.js-a koji nam omogućuje učitavanje varijabli okruženja iz .env datoteka u process.env. Dotenv modul instaliramo i uključujemo pomoću npm install dotenv te require(‘dotenv’). Potrebno je još samo učitati sve varijable okrženja. to se radi pomoću naredbe require(‘dotenv’).load(). Sada su sve varijable okruženja učitane u process.env i možemo im prisupati preko njega. Ako je naprimjer env datoteka imala liniju DB\_HOST=localhost onda joj u programskom kodu možemo pristupiti kao process.env.DB\_HOST.

## 3.4 ES7, Babel i ES5

ECMAScript(ES) je specifikacija je skriptnih jezika bazirana na JavaScirptu.  Node.js podržava verziju specifikacije ES5. Najnovija verzija specifikacije za JavaScript je ES7. Kako bi Node.js mogao raditi sa verzijom ES7 potrebno je koristiti modul babel koji omogućuje prevođenje verzije ES7 u verziju ES5. Instalacija babela izvodi se pomoću naredbe npm install --save-dev babel-core. Sljedećih par linija programskog koda prikazuje primjenu babela u izradi aplikacija.

var babel = require("babel-core");  
import { transform } from 'babel-core';  
import \* as babel from 'babel-core';

Babel ima različite varijante metode transforme koja omogućuje zadanog programskog koda ili datoteke u odgovarajuću verziju prikladno za Node.js.

## 3.5 Morgan

Jedan koristan modul je i Morgan. On na omogućuje vođenje dnevnika HTTP zahtjeva. Pojednostavljuje cijeli proces zapisivanja zahtjeva. U sljedećem primjeru možemo vidjeti metodu koja se izvršava nakon svakog zahtjeva:

app.use(function(request, response, next) {  
 console.log(request.method + " to " + request.url);  
 next();  
});

Morgan radi uz Express te ovakvu funkciju zamjenjuje sljedećim programskim linijama:

var morgan = require("morgan");

app.use(morgan());

Unutar funkcije morgan() mogu se dodati razni parametri poput ‘dev’ i ‘common’ koji samo defniraju način na koji će se zahtijevi ispisivati.

# Zaključak

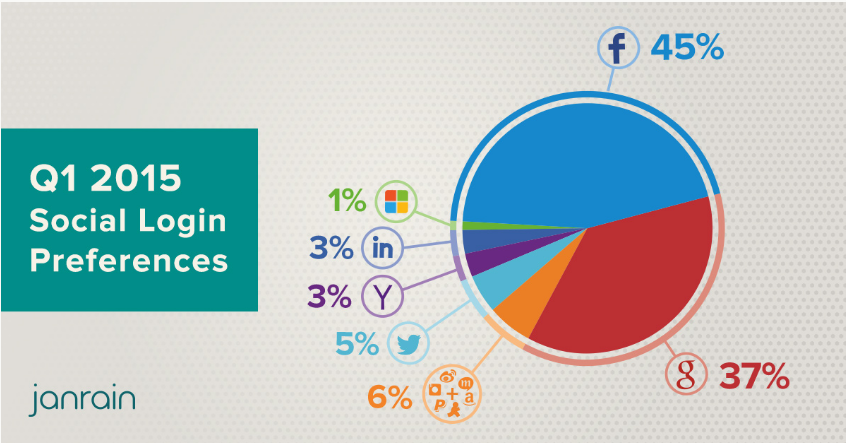
Node. js je okruženje koje nam omogućujue da se cijela web aplikacija izradi korištenjem samo JavaScripta. On radi uz *Google’s V8 JavaScript engine* zbog čega su aplikacije napisane u Node.js dosta brze. Još jedan od glavnih razloga brzine koju postižu aplikacije pisane ovim jezikom je činjenica da Node.js radi asinkrono te se tako izbjegava stalno čekanje sustava na neke rezultate. Node. js ima još puno modula koji ubrzavaju rad, olakšavaju izradu i pomažu pri izradi aplikacija. Jedan modul koji je najkorišteniji i gotovo uvijek dolazi u kompletu s izradom aplikacija u Node.js-u je Express.

**Prijava putem društvenih mreža**

# Uvod

Prijava putem društvenih mreža (*engl. social login*) je oblik prijave u *third party* aplikaciju pri kojem se umjesto stvaranja novog korisničkog računa za prijavu koriste postojeće informacije s društvenih mreža poput Facebooka, Twittera ili Google-a. Osnovna ideja *social logina* je pojednostavljenje registracije i prijave za krajnje korisnike aplikacije. Na ovaj način korisnici se mogu prvi (i svaki sljedeći) put prijaviti u aplikaciju u samo dva klika. Također, korisnici za prijavu koriste već postojeće korisničko ime i lozinku.

Na Slici 1 prikazane su društvene mreže putem kojih su se korisnici najčešće prijavljivali u *third party* aplikacije u prvoj četvrtini 2015. godine.



**Slika 1:** Udio društvenih mreža u prijavama u *third party* aplikacije

u prvoj četvrtini 2015. godine [1]

# Prednosti korištenja prijave putem društvenih mreža

Prednosti korištenje društvenih mreža za prijavu u *third party* aplikacije su:

* jednostavno stvaranje korisničkog računa

*Social login* omogućuje korisnicima stvaranje računa u *third party* aplikaciji bez osmišljavanja i unošenja novog korisničkog imena i lozinke (koje će korisnik, nakon nekog vremena neaktivnosti, vjerojatno zaboraviti).

* dohvat verificirane e-mail adrese

Prilikom stvaranja korisničkog profila na društvenim mrežama korisnik mora verificirati svoju e-mail adresu. Ako je e-mail adresa korisnika nužna za rad aplikacije, aplikacija ne mora sama provesti verifikaciju e-mail adrese.

* pružanje personalizirane usluge

Društvene mreže omogućuju pristup različitim informacijama poput profilne slike, datuma rođenja, rodnog mjesta i trenutne lokacije korisnika.[[13]](#footnote-14) Na temelju tih informacija aplikacija može korisniku pružiti personaliziranu uslugu.

* aplikacija dobiva društvenu komponentu

Korištenjem društvenih mreža aplikacija dobiva društvenu komponentu. Društvene mreže omogućuju dijeljenje, ocjenjivanje i komentiranje aplikacije. Istraživanja su pokazala da će 82% korisnika isprobati aplikaciju koju je netko preporučio na njihovoj društvenoj mreži. [2]

# Prijava u aplikaciju putem društvenih mreža

Prijava u aplikaciju putem društvenih mreža je jednostavan proces, a sastoji se od tri koraka:

1. Korisnik pristupa aplikaciji te odabire društvenu mrežu putem koje se želi prijaviti u aplikaciju.
2. Aplikacija šalje zahtjev za prijavom pružatelju usluge odabrane društvene mreže.
3. Nakon što pružatelj usluge potvrdi identitet korisnika, korisniku se dozvoljava pristup aplikaciji. Ako korisnik aplikaciji pristupa prvi put, najprije će biti registriran kao novi korisnik, a zatim će automatski biti prijavljen u aplikaciju.

U nastavku je prikazano na koji način se s razvojne strane može ostvariti prijava u Android i web-aplikaciju korištenjem društvene mreže Facebook. Sve upute i ispisi su preuzeti iz [3].

## Prijava u aplikaciju korištenjem društvene mreže Facebook

Facebook je najpopularnija društvena mreža današnjice, sa skoro dvije milijarde mjesečno aktivnih korisnika. Stoga i ne čudi da je upravo Facebook najčešća društvena mreža putem koje se korisnici prijavljuju u *third party* aplikacije (Slika 1). Facebook *login* može se koristiti kao dio iOS, Android, Windows Phone, web i desktop aplikacija, ali i na uređajima poput pametnih televizora i IoT[[14]](#footnote-15) objekata.

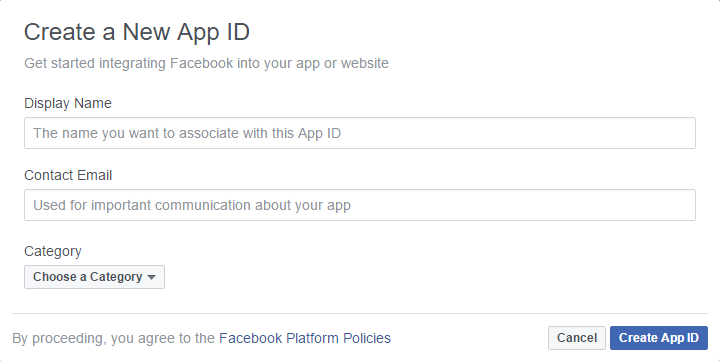
Facebook *login* omogućuje aplikaciji pristup:

* javnom profilu korisnika (puno ime, profilna slika, spol i trenutna lokacija korisnika),
* e-mailu korisnika,
* popisu korisnikovih prijatelja.

Ako su aplikaciji potrebni detaljniji podaci, aplikacija svaki put mora tražiti korisnika dozvolu (*engl. permission*) za pristup tim podacima. Također, takva aplikacija mora bit odobrena od strane Facebooka. Na taj način Facebook garantira svojim korisnicima da su aplikacije koje koriste njihove detaljne podatke sigurne za korištenje.

### Korištenje Facebook logina u Android aplikaciji

Prvi korak za korištenje Facebook *logina* u Android aplikaciji je stvaranje nove Facebook aplikacije na adresi <https://developers.facebook.com/> pri čemu se generiraju jedinstveni App ID i App Secret (Slika 2).



**Slika 2:** Stvaranje nove Facebook aplikacije

Nakon toga potrebno je uvesti Facebook SDK[[15]](#footnote-16) u postojeći Android projekt.

1. Unutar build.gradle datoteke dodati Maven Central Repository prije dependencies. (Ispis 1)

repositories {

mavenCentral()

}

**Ispis 1:** Dodavanje Maven Central Repository

1. Dodati compile 'com.facebook.android:facebook-android-sdk:[4,5)' u build.gradle dependencies.
2. Ponovo izgraditi (*engl. build*) projekt.
3. Uvesti Facebook SDK u projekt. (Ispis 2)

import com.facebook.FacebookSdk;

**Ispis 2:** Uvođenje Facebook SDK u projekt

Zatim je potrebno dodati generirani App ID u Android projekt.

* 1. Unutar strings.xmldatoteke potrebno je dodati novi string s nazivom facebook\_app\_id koji će sadržavati generirani App ID. (Ispis 3)

<string name="facebook\_app\_id">your-app-id</string>

**Ispis 3:** Stvaranje facebook\_app\_id stringa

* 1. Unutar AndroidManifest.xmldatoteke potrebno je dodati meta-data element koji sadrži novi string. (Ispis 4)

<application android:label="@string/app\_name" ...>;

...

<meta-data android:name="com.facebook.sdk.ApplicationId" android:value="@string/facebook\_app\_id"/>

...

</application>

**Ispis 4:** Uključivanje App ID-a u AndroidManifest.xml

Najjednostavniji način za korištenje Facebook *logina* u Android aplikaciji je umetanje LoginButton elementa iz Facebook SDK-a. (Ispis 5)

<com.facebook.login.widget.LoginButton

android:id="@+id/login\_button"

android:layout\_width="wrap\_content"

android:layout\_height="wrap\_content"

android:layout\_gravity="center\_horizontal"

android:layout\_marginTop="30dp"

android:layout\_marginBottom="30dp" />

**Ispis 5:** Stvaranje LoginButton elementa

LoginButton je UI[[16]](#footnote-17) element koji omata funkcionalnost Facebook *logina*, odnosno nakon što korisnik pritisne LoginButton aplikacija će poslati zahtjev za prijavom Facebook poslužitelju. U oncreateView metodi u MainActivity-ju definira se *login* *callback,* odnosno definira se metoda koja će se izvršiti nakon što aplikacija zaprimi odgovor od Facebook poslužitelja.Ako aplikacija za svoj rad treba detaljnije podatke o korisniku (poput datuma rođenja korisnika) ili želi objavljivati sadržaj u ime korisnika tada se u oncreateView metodi definiraju i potrebne dozvole. Prilikom svake prijave korisnika u aplikaciju korisnik će morati potvrditi da se slaže da se njegove informacije koriste u radu aplikacije (*ReadPermissions*) odnosno da aplikacija objavljuje sadržaj na Facebooku u njegovo ime (*WritePermissions*). (Ispis 6)

@Override

public View onCreateView(LayoutInflater inflater, ViewGroup container,

Bundle savedInstanceState) {

View view = inflater.inflate(R.layout.splash, container, false);

loginButton = (LoginButton) view.findViewById(R.id.login\_button;

loginButton.setReadPermissions("email");

// Other app specific specialization

// Calback registration

loginButton.registerCallback(callbackManager, new FacebookCallback<LoginResult>() {

@Override

public void onSuccess(LoginResult loginResult) {

// App code

}

@Override

public void onCancel() {

// App code

}

@Override

public void onError(FacebookException exception) {

// App code

}

});

}

**Ispis 6:** Definiranje *callbacka* i dozvola u oncreateView metodi

### Korištenje Facebook logina u web-aplikaciji

Prvi korak za korištenje Facebook *logina* u web-aplikacijijednak je prvom koraku u Android aplikaciji, odnosno potrebno je na stranici <https://developers.facebook.com/> stvoriti novu Facebook aplikaciju pri čemu se generiraju jedinstveni App ID i App Secret (Slika 2).

Za ostvarivanje Facebook *logina* u web-aplikaciju koristi se JavaScript Facebook SDK. JavaScript Facebook SDK nije potrebno niti skinuti niti instalirati. Umjesto toga potrebno je nakon <body> elementa u HTML dokumentu unijeti odsječak koda prikazan na Ispisu 7 kojim se asinkrono učitava i inicijalizira osnovna verzija SDK-a.

<script>

window.fbAsyncInit = function() {

FB.init({

appId : 'your-app-id',

xfbml : true,

version : 'v2.8'

});

FB.AppEvents.logPageView();

};

(function(d, s, id){

var js, fjs = d.getElementsByTagName(s)[0];

if (d.getElementById(id)) {return;}

js = d.createElement(s); js.id = id;

js.src = "//connect.facebook.net/en\_US/sdk.js";

fjs.parentNode.insertBefore(js, fjs);

}(document, 'script', 'facebook-jssdk'));

</script>

**Ispis 7:** Učitavanje i inicijalizacija osnovne verzije

JavaScript Facebook SDK-a

Kao i u Android aplikaciji najjednostavniji način za korištenje Facebook *logina* je umetanje login-button elementa iz Facebook SDK-a. (Ispis 8)

<fb:login-button scope="public\_profile,email" onlogin="checkLoginState();">

</fb:login-button>

**Ispis 7:** Stvaranje login-button elementa,

definiranje *callbacka* i dozvola

Na Ispisu 7 vidljivo je da se dozvole i *callback* metoda definiraju kao atributi login-button elementa. Dozvole se navode unutar atributa scope, a *callback* metoda kao atribut onlogin. Struktura *callback* metode prikazana je na Ispisu 8.

<script>

function checkLoginState() {

FB.getLoginStatus(function(response) {

statusChangeCallback(response);

});

}

function statusChangeCallback(response) {

if (response.status === 'connected') {

// Logged into your app and Facebook.

// App code

} else if (response.status === 'not\_authorized') {

// The person is logged into Facebook, but not your app.

// App code

} else {

// The person is not logged into Facebook, so we're not sure if

// they are logged into this app or not.

// App code

}

}

</script>

**Ispis 8:** Izgled *callback* metode

U okviru ove aplikacije za login preko Facebook društvene mreže korišten je modul passport. On sadrži različite strategije za autentifikaciju preko različitih društvenih mreža. U datoteci *passport.js* u direktoriju *chat/api/routes/auth* implementirana su metode koje se koriste za login i autentifikaciju preko Facebook-a. Isječak koda iz *passport.js*:

passport.use(**new** FacebookStrategy(options, (accessToken, refreshToken, profile, next) => {

process.nextTick(() => {

**const** db = *getDb*();

**const** newUser = {

**username**: profile.**displayName**,

**image**: profile.**photos**[0].value,

**facebook**: {

**id**: profile.**id**,

**token**: accessToken,

}

};

*// inserts new user or returns existing user -> upsert*

db.collection(**'users'**).findOneAndUpdate({ **username** : newUser.**username** }, newUser,

{ **upsert**: **true** }, (err, user) => {

**if** (err) {

**return** next(err);

}

*// set found user or new user if insert is done*

**const** active = (user || user.active) ? newUser : user.active;

**return** next(**null**, active);

});

});

}));

U varijablu *newUser* se postavljaju podatci korisnika koji se preuzimaju s Facebook-a. Zatim se taj korisnik sprema u bazu podataka ako ne postoji, a ako postoji onda se njegovi podatci ažuriraju s podatcima s Facebook-a. Na kraju se postavlja aktivni korisnik na tog korisnika.

# Zaključak

Osnovna ideja prijave putem društvenih mreža je olakšati sam proces registracije i prijave krajnjim korisnicima aplikacije. Prijava se najčešće odvija u svega dva klika te korisnik ne treba ispunjavati poduže registracijske forme, već se sve informacije o korisniku dohvaćaju s društvenih mreža. Korištenje *social logina*, osim za krajnje korisnike, ima višestruke prednosti i za programere. Ostvarivanje *social logina* je dobro dokumentirano na stranicama društvenih mreža namijenjenih programerima, a najčešće i poprilično jednostavno za ostvariti. Koristeći društvene mreže programeri na vrlo jednostavan način mogu doći do informacija do kojih bi putem registracijskih formi bilo vrlo teško (ako ne i nemoguće) doći. Korištenjem dobivenih informacija aplikacija može pružiti personaliziranu uslugu svakom pojedinom korisniku. Čak će i samo postavljanje korisnikove profilne slike unutar aplikacije doprinijeti korisnikovom dojmu o aplikaciji.

Istraživanja su pokazala da je Facebook najčešća društvena mreža putem koje se korisnici prijavljuju u *third party* aplikacije. Takav rezultat ne čudi s obzirom na to da je Facebook najpopularnija društvena mreža današnjice, sa skoro dvije milijarde mjesečno aktivnih korisnika. Facebook *login* dostupan je na gotovo svim platformama, a u ovom izvješću pokazano je kako na jednostavan način integrirati Facebook *login* u Android i web-aplikaciju. Facebook provjerava i mora odobriti svaku aplikaciju koja pristupa detaljnim informacijama o korisniku. Na taj način Facebook garantira svojim korisnicima da su aplikacije koje koriste njihove detaljne podatke sigurne za korištenje.

**Web priključnice i sigurna komunikacija**

# Uvod

Tema ovog izvješća je su Web priključnice (engl. *Web sockets*), sigurna komunikacija između više priključnica, te mogućnosti ostvarenja takve komunikacije korištenjem *Node.js* platforme.

*Node.js* je platforma koja omogućuje izradu skripti koje se izvode na poslužitelju. Sigurna komunikacija odnosi se na nemogućnost nekog napadača da sazna sadržaj poruka koje razmjenjuju klijent i poslužitelj. Pošto je cilj projekta izrada aplikacije koja će za komunikaciju koristiti gotovo Web priključnice treba se obratiti posebna pažnja sigurnosti web priključnica.

U prvom dijelu bit će objašnjena tehnologija Web priključnica, u drugom dijelu tema je sigurna komunikacija. Treći dio bavi se implementacijom sigurne komunikacija za Node.js platformu.

# Web priključnice

Web priključnice su tehnologija koja omogućuje interaktivno slanje poruka s klijenta na poslužitelj. Interaktivno ovdje znači da je s klijentske strane moguće poslati poruku na poslužitelj korištenjem priključnica, te nastaviti neki drugi posao. Isto tako je moguće da poslužitelj primi poruku, obradi je i da po potrebi pošalje odgovor klijentu koji onda taj odgovor obrađuje. Tehnologija web priključnica je dakle vid asinkrone komunikacije jer klijent ne čeka odgovor nego nakon slanja poruke nastavlja svoj posao. Važno je napomenuti da tehnologija Web priključnica omogućuje i da poslužitelj obavještava klijenta o promjenama. Upravo će se ova ideja upotrijebiti u izradi projekta. Naime klijent će se registrirati kod poslužitelja poruka koji će primati poruke, pohranjivati ih u bazu podataka, te obavještavati klijenta kojemu je namijenjena poruka o novoj poruci. Druga važna napomena je da će se cijela komunikacija odvijati u stvarnom vremenu (engl. *real time*), što znači da će klijent primati poruke i prikazivati ih bez potrebe za osvježavanjem stranice (engl. *refresh*) što donosi manji mrežni promet, ali i ugodnije korištenje korisnicima[[17]](#footnote-18).

Web priključnice se mogu izvesti korištenjem oba transportna protokola: UDP[[18]](#footnote-19) i TCP[[19]](#footnote-20). Korištenje oba protokola ima svojih prednosti i mana pa treba dobro procijeniti zahtjeve svakog projekta i na kraju donijeti ispravnu odluku u vezi odabira protokola.

TCP garantira isporuku poruka, tj. kada se nešto pošalje korištenjem TCP protokola garantira se da će poruka doći na odredište. To se postiže tako da se prvo uspostavlja veza između pošiljatelja i primatelja. Stoga se lako zaključuje da je prednost TCP-a sigurnost isporuke, a nedostatak u odnosu na TCP je potreba za otvaranjem veze i posljedično nešto sporija brzina rada, jer se više opterećuje mreža između primatelja i pošiljatelja slanjem dodatnih zahtjeva.

UDP pak ne garantira isporuku poruka. Kod UDP-a poruke se samo šalju mrežom i ovisi o mreži ako će one doći do primatelja. Ako se neka poruka izgubi pošiljatelj ne može znati da ta poruka nije došla do primatelja. Iz ovoga slijedi da je prednost UDP-a brzina slanja, a mana svakako ne garantirana isporuka. Naravno korištenjem UDP-a se može programski postići da se garantira isporuka, tako da pošiljatelj šalje poruke dok mu primatelj ne vrati potvrdu da je poruku primio te se tek tada šalje iduća poruka.

Prilikom izrade projekta koristit će se TCP komunikacijski protokol jer poruke koje razmjenjuju korisnici moraju garantirano pristići na svoje odredište.

# Sigurna komunikacija

Pojam sigurne komunikacije odnosi se na nemogućnost neke treće strane da sazna o čemu komuniciraju neki entiteti korištenjem Interneta. Laički rečeno to je komunikacija koja ima za zahtjev da se ne može prisluškivati od strane nekog tko u komunikaciji ne sudjeluje. Na Internetu se sigurna komunikacija najčešće postiže enkripcijom prometa između strana koje komuniciraju. Enkripcija znači da se svaka poruka prije slanja šifrira koristeći specijalan algoritam za kriptiranje[[20]](#footnote-21) koji iz originalne poruke i nekog ključa kreira šifriranu poruku. Nakon što primatelj primi takvu poruku on mora moći dekriptirati odnosno dešifrirati primljenu poruku. Osnovni problem kriptiranja je dakle mehanizam kojim samo pošiljatelj i primatelj mogu saznati originalnu poruku, dok ona mora biti nečitljiva drugim, tj. razmjena ključeva kojima se kriptira. Naime zahtjev kriptiranja je da se originalna poruka ne može dešifrirati iz šifrirane poruke niti ako se zna algoritam kojim se šifrira.

Postoje dvije vrste kriptografije: simetrična i asimetrična.

Simetrična kriptografija bazira se na jednom ključu koji imaju i primatelj i pošiljatelj. Simetrična je kriptografija brža i do 1000 puta od asimetrične, ali ima problem kako distribuirati zajednički ključ bez da se njegova tajnost prekrši.

Asimetrična kriptografija bazira se na arhitekturi dva ključa: javnog i privatnog. Privatni ključ posjeduje samo jedna osoba, vlasnik ključa, dok javni ključ mora biti javno dostupan svima. To se postiže najčešće korištenjem poslužitelja javnih ključeva. Kao što je već napomenuto mana asinkrone kriptografije je brzina, ali prednost je lagana razmjena ključeva.

Danas se sigurna komunikacija na Internetu osigurava korištenjem HTTPS[[21]](#footnote-22) protokola. HTTPS protokol zahtjeva da se sav promet između klijenta i poslužitelja kriptira korištenjem SSL[[22]](#footnote-23) ili TLS[[23]](#footnote-24) protokola.

Kod korištenja protokola HTTPS na poslužitelju se mora definirati privatni ključ za dešifriranje, te se treba postaviti i certifikat. Certifikat je potvrda kojim poslužitelj potvrđuje da je upravo onaj koji tvrdi da je, tj. potvrđuje svoju autentičnost. Certifikat koji se koristi dobiva se od specijalne agencije koja se bavi certificiranjem. Riječ je o dokumentu koji sadrži adresu poslužitelja, njegov javni ključ i ostale podatke. Certifikatu autentičnost daje potpis koji ostavlja izdavatelj certifikata i koji garantira da je certifikat valjan koristeći postupak digitalnog potpisa. Postupak izdavanja certifikata na za ovaj projekt bit će opisan u idućem poglavlju.

Kada klijent pristupa poslužitelju on preko poslužiteljeve certifikata zna da je poslužitelj autentičan i da mu može vjerovati. Certifikat automatski pregledava klijentski pretraživač Weba (engl. *browser*).

Protokol korištenja web priključnica dodaje dvije nove URI[[24]](#footnote-25) sheme: *ws* i *wss.* Korištenjem ws sheme zapravo se koristi nekriptirana komunikacija između priključnica (npr. ws://posluzitelj.com), dok se navođenjem wss-a zahtjeva kriptirana komunikacija (npr. wss://posluzitelj.com). Kriptiranje se postiže korištenjem TLS protokola, tj. slanjem poruka preko HTTPS-a. Tako da bi se postigla kriptirana komunikacija između Web priključnica treba također posjedovati certifikat koji će omogućiti kriptiranje i autentičnost poslužitelja.

# Implementacija sigurne komunikacije za Node.js platformu

Node.js platforma omogućuje jednostavan razvoj Web poslužitelja. Sigurna komunikacija se korištenjem Node.js platforme najčešće postiže na dva načina. Prvi način je arhitektura koja se sastoji od jednog poslužitelja koji poslužuje klijente putem HTTPS protokola (HTTPS poslužitelj), a druga arhitektura se sastoji od dva poslužitelja koji u paru osiguravaju sigurnu komunikaciju. Prvi koji služi kao *Proxy* za drugi poslužitelj. Zadaća *Proxyja* je da kriptira sav promet između sebe i klijenta. Ovo kriptiranje obuhvaća ulazni i izlazni promet. Drugi poslužitelj ne komunicira s klijentima nego samo s *Proxy* poslužiteljem koji mu prosljeđuje dekriptirane zahtjeve.

Obje arhitekture aplikacije zato zahtijevaju certifikate kako bi mogli podržati HTTPS protokol. Certifikati se mogu kupiti od tvrtke koja se bavi izdavanjem certifikata ili se mogu koristiti certifikati koji su generirani besplatno. Problem kod nekih besplatnih certifikata je kratko vrijeme trajanja što znači da se oni periodički moraju obnavljati. Drugi problem koji prati besplatne certifikate je to da ih ne priznaju svi preglednici pa često prilikom susreta s takvim certifikatom Web preglednici znaju korisnika upozoriti da je možda riječ o prevari, te tako odvratiti korisnika od korištenja željene aplikacije. Ovaj drugi slučaj se događa i ako je certifikat samostalno izdan.

Za ovaj projekt koristit će se certifikat generiran od strane *Let's Ecrypt* certifikacijskog tijela. Riječ je o besplatnom, automatiziranom i otvorenom certifikacijskom tijelu. *Let's Encrypt* omogućuje jednostavno kreiranje certifikata koje prepoznaje većina današnjih Web preglednika (nije podržan na Android operacijskom sustavu ispod inačice 2.3.6 i na Windows XP operacijskom sustavu). Tako generirani certifikati traju 90 dana, ali *Let's Encypt* podržava automatsko produljenje certifikata, koje omogućuje jednostavno obnavljanje certifikata.

Postupak izdavanja certifikata sastoji se od popunjavanja podataka vezanih uz poslužitelj koji se koristi (domena, operacijski sustav poslužitelja i sl.). Drugi dio je dokazivanje kontrole nad domenom. Kontrola se dokazuje korištenjem programa koji koriste ACME[[25]](#footnote-26) protokol. Takvi programi postaju tako klijeti za poslužitelj i šalju specijalne zahtjeve poslužitelju. Protokol definira kakav odgovor mora vratiti poslužitelj za svaki zahtjev koji primi. Ako poslužitelj dobro odgovori na sve zahtjeve (još se nazivaju engl. *challenges*) onda se izdaje važeći certifikat koji traje 90 dana od trenutka izdavanja. Za izdavanje certifikata će se koristiti *certbot[[26]](#footnote-27)* koji lako izdaje *Let's Encypt* certifikate.

Nakon što se generira certifikat može će prionuti razvoju aplikacije koja koristi sigurnu komunikaciju. U ovom projektu sigurna komunikacija će se postići korištenjem arhitekture s *Proxy* poslužiteljem. Razlog je taj da se takvim razvojem dobiva raspodijeljeni sustav koji je bolje rješenje od jednog centraliziranog poslužitelja.

Za izradu web priključnica koristi će se modul *Socket.io* koji je najpoznatiji modul za izradu priključnica korištenjem Node.js platforme. Prednosti modula su u činjenici da je Modul također sadrži podršku za kriptiranu razmjenu poruka (wss), podršku za razmjenu datoteka, kao i podršku za mobilne uređaje s operacijskim sustavima Android i iOS. Druga važna prednost *Socket.io* modula je svakako mogućnost da u slučaju preopterećenja prijeđe na komunikaciju razmjenama poruka kako bi oslobodio dio resursa na poslužitelju.

Poslužitelj i proxy poslužitelj implementirani su u datoteci *www.js* u direktoriju *chat/api*. Slijedeći isječak koda prikazuje stvaranje HTTP i HTTPS poslužitelja:

// HTTP Server

**const** server = http.createServer((req, res) => {

res.writeHead(301, { **"Location"**: **"https://"** + req.**headers**.host + **":"** + sPort + req.**url** });

res.**end**();

});

**const** sServer = https.createServer(options, app);

Prilikom stvaranja HTTPS servera u varijabli *options* se predaju privatni ključ za šifriranje podataka i certifikat. Sav promet između klijenta i HTTPS servera je šifriran. Certifikat i privatni ključ su generirani na poslužitelju koristeći alat *certbot*. Poslužitelj osluškuje zahtjeve klijenata koji se spajaju pomoću web priključnica:

*// Sockets*

**const** userSockets = [];

**const** io = socketio.listen(sServer);

io.**on**(**'connection'**, socket => {

socket.**on**(**'user'**, user => {

**console**.log(**'Connected: '** + user.**username**);

**const** newUserSocket = {

**user**: user.**username**,

**socket**: socket,

**id**: socket.**handshake**.**address**

};

userSockets.push(newUserSocket);

});

socket.**on**(**'userAndroid'**, user => {

**console**.log(**'Connected: '** + user);

**const** newUserSocket = {

**user**: user,

**socket**: socket,

**id**: socket.**handshake**.**address**,

**mobile**: **true**

};

userSockets.push(newUserSocket);

});

socket.**on**(**'disconnect'**, () => {

**for** (**let** i = 0; i < userSockets.**length**; i++) {

**if** (userSockets[i].**socket** == socket) {

**console**.log(**"Disconnected: "** + userSockets[i].user);

userSockets.splice(i, 1);

}

}

});

});

Prilikom svakog spajanja klijenata na server stvara se web priključnica korisnika i sprema u polje *userSockets*. Kada klijent završava komunikaciju s poslužiteljem iz tog polja se web priključnica izbacuje.

# Zaključak

Prilikom izrade ovog projekta koristit će se Web priključnice kako bi se omogućila komunikacija sudionika u realnom vremenu. Protokol koji će se koristit za prijenos poruka je TCP jer garantira isporuku svake poruke na odredište. Sigurnost komunikacije je bitna stavka ovog projekta, pa će se svi zahtjevi za poslužitelj koristiti HTTPS protokol. Komunikacija između Web priključnica će također biti kriptirana korištenjem wss-a koji koristi TLS protokol. Da bi se ta sigurnost postigla morati će se generirati certifikat za poslužitelj. Za generiranje certifikata koristit će se *certbot.*

**Direct browser communication in the same network**

# A very short history of WebRTC

One of the last major challenges for the web is to enable human communication via voice and video: Real Time Communication, RTC for short. RTC should be as natural in a web application as entering text in a text input. Without it, we're limited in our ability to innovate and develop new ways for people to interact.

Historically, RTC has been corporate and complex, requiring expensive audio and video technologies to be licensed or developed in house. Integrating RTC technology with existing content, data and services has been difficult and time consuming, particularly on the web.

Gmail video chat became popular in 2008, and in 2011 Google introduced Hangouts, which use the Google Talk service (as does Gmail). Google bought GIPS, a company which had developed many components required for RTC, such as codecs and echo cancellation techniques. Google open sourced the technologies developed by GIPS and engaged with relevant standards bodies at the IETF and W3C to ensure industry consensus. In May 2011, Ericsson built [the first implementation of WebRTC](https://labs.ericsson.com/developer-community/blog/beyond-html5-peer-peer-conversational-video).

WebRTC has now implemented open standards for real-time, plugin-free video, audio and data communication. The need is real:

1. Many web services already use RTC, but need downloads, native apps or plugins. These includes Skype, Facebook (which uses Skype) and Google Hangouts (which use the Google Talk plugin).
2. Downloading, installing and updating plugins can be complex, error prone and annoying.
3. Plugins can be difficult to deploy, debug, troubleshoot, test and maintain—and may require licensing and integration with complex, expensive technology. It's often difficult to persuade people to install plugins in the first place!

The guiding principles of the WebRTC project are that its APIs should be open source, free, standardized, built into web browsers and more efficient than existing technologies.

# Where are we now?

WebRTC is used in various apps like WhatsApp, Facebook Messenger, appear.in and platforms such as TokBox. There is even an experimental WebRTC enabled iOS Browser named Bowser. WebRTC has also been integrated with WebKitGTK+ and Qtnative apps.

Microsoft added MediaCapture and Stream APIs to Edge.

WebRTC implements three APIs:

1. MediaStream (aka getUserMedia)
2. RTCPeerConnection
3. RTCDataChannel

**getUserMedia** is available in Chrome, Opera, Firefox and Edge.

**RTCPeerConnection** is in Chrome (on desktop and for Android), Opera (on desktop and in the latest Android Beta) and in Firefox. A word of explanation about the name: after several iterations, RTCPeerConnection is currently implemented by Chrome and Opera as webkitRTCPeerConnection and by Firefox as mozRTCPeerConnection. Other names and implementations have been deprecated. When the standards process has stabilized, the prefixes will be removed.

**RTCDataChannel** is supported by Chrome, Opera and Firefox.

# A word of warning

Be skeptical of reports that a platform 'supports WebRTC'. Often this actually just means that getUserMedia is supported, but not any of the other RTC components.

# Signaling: session control, network and media information

WebRTC uses RTCPeerConnection to communicate streaming data between browsers (aka peers), but also needs a mechanism to coordinate communication and to send control messages, a process known as signaling. Signaling methods and protocols are not specified by WebRTC: signaling is not part of the RTCPeerConnection API.

Instead, WebRTC app developers can choose whatever messaging protocol they prefer, such as SIP or XMPP, and any appropriate duplex (two-way) communication channel. The [apprtc.appspot.com](https://apprtc.appspot.com/) example uses XHR and the Channel API as the signaling mechanism. The [codelab](http://www.bitbucket.org/webrtc/codelab) we built uses [Socket.io](http://socket.io/) running on a [Node server](http://nodejs.org/).

Signaling is used to exchange three types of information:

* Session control messages: to initialize or close communication and report errors.
* Network configuration: to the outside world, what's my computer's IP address and port?
* Media capabilities: what codecs and resolutions can be handled by my browser and the browser it wants to communicate with?

The exchange of information via signaling must have completed successfully before peer-to-peer streaming can begin.

For example, imagine Alice wants to communicate with Bob. First up, Alice and Bob exchange network information. (The expression 'finding candidates' refers to the process of finding network interfaces and ports using the [ICE framework](https://www.html5rocks.com/en/tutorials/webrtc/basics/" \l "ice).)

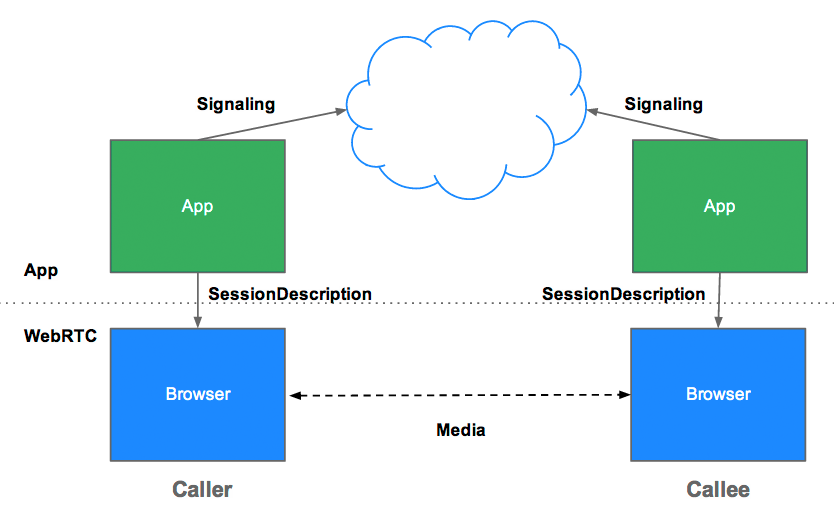
1. Alice creates an RTCPeerConnection object with an onicecandidate handler.
2. The handler is run when network candidates become available.
3. Alice sends serialized candidate data to Bob, via whatever signaling channel they are using: WebSocket or some other mechanism.
4. When Bob gets a candidate message from Alice, he calls addIceCandidate, to add the candidate to the remote peer description.

WebRTC clients (known as **peers**, aka Alice and Bob) also need to ascertain and exchange local and remote audio and video media information, such as resolution and codec capabilities. Signaling to exchange media configuration information proceeds by exchanging an offer and an answer using the Session Description Protocol (SDP):

1. Alice runs the RTCPeerConnection createOffer() method. The callback argument of this is passed an RTCSessionDescription: Alice's local session description.
2. In the callback, Alice sets the local description using setLocalDescription()and then sends this session description to Bob via their signaling channel. Note that RTCPeerConnection won't start gathering candidates until setLocalDescription() is called: this is codified in [JSEP IETF draft](http://tools.ietf.org/html/draft-ietf-rtcweb-jsep-03" \l "section-4.2.4).
3. Bob sets the description Alice sent him as the remote description using setRemoteDescription().
4. Bob runs the RTCPeerConnection createAnswer() method, passing it the remote description he got from Alice, so a local session can be generated that is compatible with hers. The createAnswer() callback is passed an RTCSessionDescription: Bob sets that as the local description and sends it to Alice.
5. When Alice gets Bob's session description, she sets that as the remote description with setRemoteDescription.
6. Ping!

The acquisition and exchange of network and media information can be done simultaneously, but both processes must have completed before audio and video streaming between peers can begin.

The offer/answer architecture described above is called [JSEP](http://tools.ietf.org/html/draft-ietf-rtcweb-jsep-00), JavaScript Session Establishment Protocol. (There's an excellent animation explaining the process of signaling and streaming in [Ericsson's demo video](http://www.ericsson.com/research-blog/context-aware-communication/beyond-html5-peer-peer-conversational-video/) for its first WebRTC implementation.)

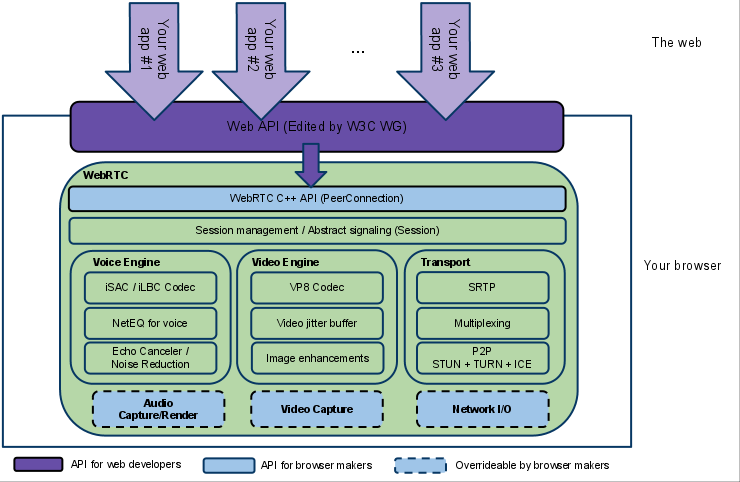
JSEP architecture

Once the signaling process has completed successfully, data can be streamed directly peer to peer, between the caller and callee—or if that fails, via an intermediary relay server (more about that below). Streaming is the job of RTCPeerConnection.

# RTCPeerConnection

RTCPeerConnection is the WebRTC component that handles stable and efficient communication of streaming data between peers.

Below is a WebRTC architecture diagram showing the role of RTCPeerConnection. As you will notice, the green parts are complex!

WebRTC architecture (from [webrtc.org](http://www.webrtc.org/reference/architecture))

From a JavaScript perspective, the main thing to understand from this diagram is that RTCPeerConnection shields web developers from the myriad complexities that lurk beneath. The codecs and protocols used by WebRTC do a huge amount of work to make real-time communication possible, even over unreliable networks:

1. packet loss concealment
2. echo cancellation
3. bandwidth adaptivity
4. dynamic jitter buffering
5. automatic gain control
6. noise reduction and suppression
7. image 'cleaning'.

The W3C code above shows a simplified example of WebRTC from a signaling perspective. Below are walkthroughs of two working WebRTC applications: the first is a simple example to demonstrate RTCPeerConnection; the second is a fully operational video chat client.

# RTCPeerConnection without servers

The code below is taken from the 'single page' WebRTC demo at [https://webrtc.github.io/samples/src/content/peerconnection/pc1](https://webrtc.github.io/samples/src/content/peerconnection/pc1/), which has local andremote RTCPeerConnection (and local and remote video) on one web page. This doesn't constitute anything very useful—caller and callee are on the same page—but it does make the workings of the RTCPeerConnection API a little clearer, since the RTCPeerConnection objects on the page can exchange data and messages directly without having to use intermediary signaling mechanisms.

One gotcha: the optional second 'constraints' parameter of the RTCPeerConnection()constructor is different from the constraints type used by getUserMedia(): see [w3.org/TR/webrtc/#constraints](http://www.w3.org/TR/webrtc/" \l "constraints) for more information.

In this example, pc1 represents the local peer (caller) and pc2 represents the remote peer (callee).

### Caller

1. Create a new RTCPeerConnection and add the stream from getUserMedia():
2. *// servers is an optional config file (see TURN and STUN discussion below)*
3. pc1 = new webkitRTCPeerConnection(servers);
4. *// ...*

pc1.addStream(localStream);

1. Create an offer and set it as the local description for pc1 and as the remote description for pc2. This can be done directly in the code without using signaling, because both caller and callee are on the same page:
2. pc1.createOffer(gotDescription1);
3. *//...*
4. function gotDescription1(desc){
5. pc1.setLocalDescription(desc);
6. trace("Offer from pc1 \n" + desc.sdp);
7. pc2.setRemoteDescription(desc);
8. pc2.createAnswer(gotDescription2);

### Callee

1. Create pc2 and, when the stream from pc1 is added, display it in a video element:
2. pc2 = new webkitRTCPeerConnection(servers);
3. pc2.onaddstream = gotRemoteStream;
4. *//...*
5. function gotRemoteStream(e){
6. vid2.src = URL.createObjectURL(e.stream);

# RTCPeerConnection plus servers

In the real world, WebRTC needs servers, however simple, so the following can happen:

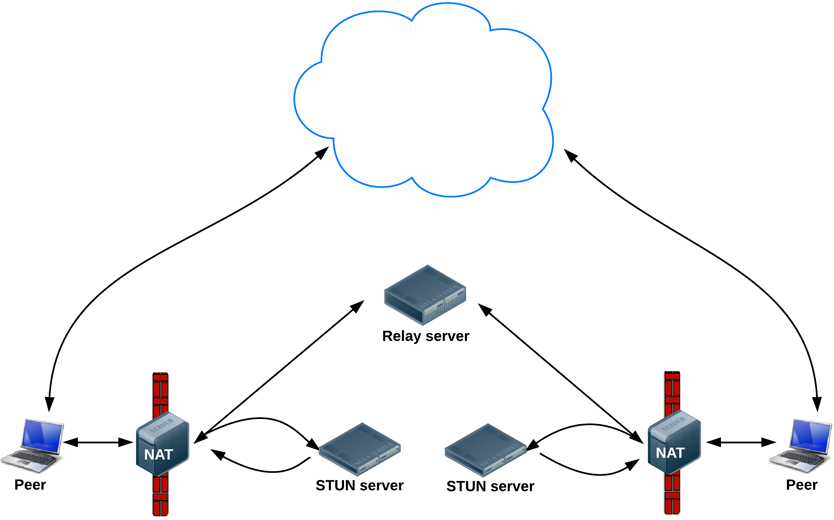
* Users discover each other and exchange 'real world' details such as names.
* WebRTC client applications (peers) exchange network information.
* Peers exchange data about media such as video format and resolution.
* WebRTC client applications traverse [NAT gateways](http://en.wikipedia.org/wiki/NAT_traversal) and firewalls.

In other words, WebRTC needs four types of server-side functionality:

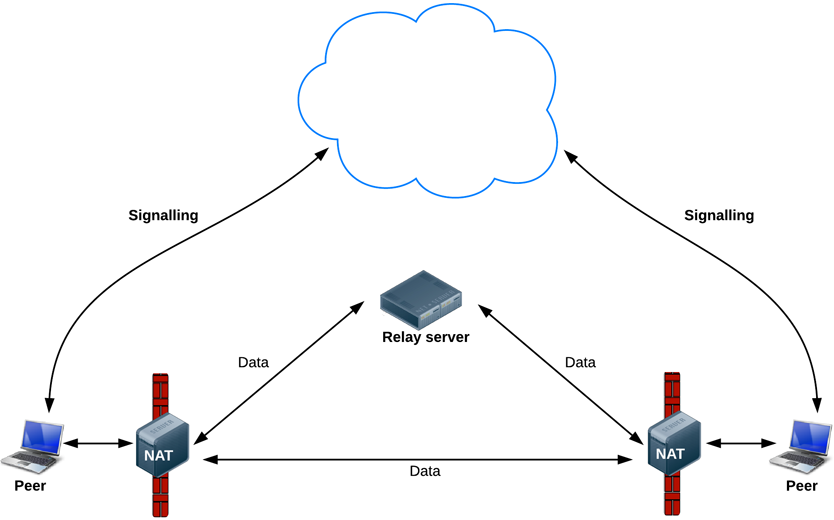
* User discovery and communication.
* Signaling.
* NAT/firewall traversal.
* Relay servers in case peer-to-peer communication fails.

NAT traversal, peer-to-peer networking, and the requirements for building a server app for user discovery and signaling, are beyond the scope of this article. Suffice to say that the [STUN](http://en.wikipedia.org/wiki/STUN) protocol and its extension [TURN](http://en.wikipedia.org/wiki/Traversal_Using_Relay_NAT) are used by the [ICE](http://en.wikipedia.org/wiki/Interactive_Connectivity_Establishment) framework to enable RTCPeerConnection to cope with NAT traversal and other network vagaries.

ICE is a framework for connecting peers, such as two video chat clients. Initially, ICE tries to connect peers directly, with the lowest possible latency, via UDP. In this process, STUN servers have a single task: to enable a peer behind a NAT to find out its public address and port. (Google has a couple of STUN severs, one of which is used in the apprtc.appspot.com example.)

Finding connection candidates

If UDP fails, ICE tries TCP: first HTTP, then HTTPS. If direct connection fails—in particular, because of enterprise NAT traversal and firewalls—ICE uses an intermediary (relay) TURN server. In other words, ICE will first use STUN with UDP to directly connect peers and, if that fails, will fall back to a TURN relay server. The expression 'finding candidates' refers to the process of finding network interfaces and ports.

WebRTC data pathways

WebRTC engineer Justin Uberti provides more information about ICE, STUN and TURN in the 2013 Google I/O WebRTC presentation.

# Security

There are several ways a real-time communication application or plugin might compromise security. For example:

* Unencrypted media or data might be intercepted en route between browsers, or between a browser and a server.
* An application might record and distribute video or audio without the user knowing.
* Malware or viruses might be installed alongside an apparently innocuous plugin or application.

WebRTC has several features to avoid these problems:

* WebRTC implementations use secure protocols such as [DTLS](http://en.wikipedia.org/wiki/Datagram_Transport_Layer_Security) and [SRTP](http://en.wikipedia.org/wiki/Secure_Real-time_Transport_Protocol).
* Encryption is mandatory for all WebRTC components, including signaling mechanisms.
* WebRTC is not a plugin: its components run in the browser sandbox and not in a separate process, components do not require separate installation, and are updated whenever the browser is updated.
* Camera and microphone access must be granted explicitly and, when the camera or microphone are running, this is clearly shown by the user interface.

# In conclusion

The APIs and standards of WebRTC can democratize and decentralize tools for content creation and communication—for telephony, gaming, video production, music making, news gathering and many other applications.

Technology doesn't get much more disruptive than this.

*We look forward to what JavaScript developers make of WebRTC as it becomes widely implemented. As blogger Phil Edholm*[*put it*](http://www.nojitter.com/post/232901042/webrtc-is-it-a-game-changer)*, 'Potentially, WebRTC and HTML5 could enable the same transformation for real-time communications that the original browser did for information.'*

### **MongoDB**

# Uvod

U ovom izvješću ću pisati o MongoDB bazi podataka, MongoDB je besplatna, višeplatformska baza podataka otvorenog koda i danas jedna od najpoznatijih i najkorištenijih NoSQL baza podataka.

Obzirom da je MongoDB NoSQL baza podataka prvo poglavlje se bavi NoSQL bazama podataka, njihovim općenitim karakteristikama i najbitnijim tipovima. Drugo poglavlje detaljno obrađuje MongoDB bazu podataka koju ćemo koristiti u sklopu projekta. U trećem poglavlju je opisano kako koristiti bazu podataka koristeći podržane driver-e ili biblioteku Mongoose.

# NoSQL baze podataka

NoSQL ili nerelacijske baze podataka razvile su se krajem 2000.-ih godina jer su do tada standardne relacijske baze podataka imale problema sa skalabilnošću, raspodijeljenošću podataka, ubrzanim razvojem i velikim brojem korisnika. Osnovne značajke nerelacijskih bazi podataka su raspodijeljenost, otvoreni kod, horizontalna skalabilnost, rad s velikom količinom podataka, nepostojanje sheme ili dinamička shema, jednostavan podatkovni model, pristup podatcima preko jednostavnih aplikacijskih sučelja. Neke od NoSQL baza podataka "žrtvuju" konzistentnost podataka u zamjenu za visoku dostupnost i performanse. Često se kod NoSQL baza podataka ne zahtijeva ispunjavanje ACID svojstva, već one mogu zadovoljiti manje rigorozna svojstva BASE:

* *Basically Available* - podatci su dostupni praktički uvijek
* *Soft-state* - sustav ne mora uvijek biti konzistentan
* *Eventual consistency -* u konačnici izmjene će se propagirati i svi će ih vidjeti

Koja će od ovih svojstava biti ispunjena ovisi o konkretnoj implementaciji baze podataka, ali i o konkretnoj primjeni, u nekim primjenama je bitna visoka konzistentnost dok kod drugih performanse i dostupnost mogu biti presudne.

Kod NoSQL baza podataka postoje 4 glavna tipa modela podataka:

* Ključ-vrijednost (Key Value)
* Dokument (Document)
* *Column family*
* Graf (Graph)

Ključ-vrijednost model podataka je najjednostavniji model gdje se svaki podatak pohranjuje u bazu kao ključ (npr. Ime atributa) i vrijednost tog podatka. Neki od primjera takvih baza podataka su Riak i Dynamo.

Dokument baze podataka su slične ključ-vrijednost modelu samo je vrijednost u ovom slučaju dokument. Dokument može imati kompleksnu strukturu koja sadrži složene objekte ili čak ugniježđene dokumente. Dokumenti su tipično JSON, BSON, XML, YAML ili neki drugi strukturirani ili polustrukturirani formati podataka. Primjer dokument baze podataka je MongoDB o kojemu će biti više pisano u nastavku.

*Column family* baze podataka pohranjuje podatke u obliku dvorazinskih mapa, postoji ključ retka i ključ stupca. Svaki stupac je član neke familije stupaca u kojima su pohranjeni podatci. Primjeri ovakvih baza podataka su Apache HBase i Cassandra.

Graf baze podataka služe za pohranu jako povezanih podataka i polustrukturiranih podataka poput prijateljstva u društvenim mrežama. Model podataka se sastoji od čvorova, bridova i svojstva koji oni imaju. Primjeri su Neo4j i Giraph.

# MongoDB baza podataka

MongoDB je besplatna, *open-source*, višeplatformska dokument baza podataka koja pruža visoke performanse, visoku dostupnost i horizontalnu skalabilnost.

Podatci u MongoDB bazi podataka se spremaju u binarnom obliku nazvanom BSON (*Binary JSON)*. BSON je proširenje standardnog JSON-a, odnosno dozvoljava nove tipove podataka poput *int, long, date* i *float*. BSON dokumenti sadrže 1 ili više zapisa, a polja sadrže vrijednost nekog tipa koji može biti i polje ili dokument ugniježđen unutar postojećega. Za pristup podatcima unutar dokumenta koristi se objektna notacija (*dot notation*). Ovakva struktura dokumenata slična je objektima u programskim jezicima, pa se lako obavlja preslikavanje iz aplikacije u bazu podataka. Tipično ne postoji sloj koji preslikava objekte aplikacije u zapise u bazi podataka, već je odnos objekata i zapisa "jedan na jedan". Fleksibilnost koji pružaju dokumenti omogućava brzo i lagano inkrementalno mijenjanje podatkovnog modela aplikacije.

Dokumenti koji imaju istu ili sličnu strukturu su organizirani u kolekcije, što odgovara tablicama u relacijskim bazama podataka. Povezani podatci su agregirani u jedan dokument dok su takvi podatci kod relacijskih baza podataka često razlomljeni u više tablica. Ovakvi agregati podataka smanjuju broj spajanja i čitanja u bazi podataka što povećava performanse i skalabilnost sustava. Struktura dokumenata u MongoDB bazi podataka nije predodređena, već se može dinamički mijenjati, odnosno dokumenti u kolekciji ne moraju imati iste atribute. Struktura dokumenta se ne mora definirati eksplicitno (iako može), već prvi dokument koji se upisuje definira strukturu tog tipa.

MongoDB omogućava validaciju unutar baze podataka, korisnici mogu provoditi validaciju nad strukturom dokumenta, tipovima podataka, prisutnošću određenih polja i dozvoljenim rasponima.

Osnovne značajke MongoDB baze podataka su:

* visoke performanse
* bogat upitni jezik
* visoka dostupnost
* horizontalna skalabilnost

Visoke performanse se ostvaruju korištenjem različitih vrsta indeksa. Indeksi se mogu graditi nad bilo kojim elementom dokumenta, čak i nad onima koji se nalaze unutar nekog polja. Također podržava indekse za tekstualna pretraživanja i rad s tekstom, te indekse za optimizaciju upita koji koriste geoprostorne podatke. Prije izvođenja samog upita MongoDB provodi optimizaciju upita koristeći indekse i statističke podatke o upitima koje periodički osvježava.

MongoDB podržava različite tipove upita kojima se dohvaćaju različiti podatci. Standardni upiti koriste *CRUD (Create, Read, Update, Delete)* operacije za čitanje, pisanje, ažuriranje i brisanje dokumenata u bazi podataka. Također su još podržane agregacije nad skupom podataka, upiti za pretraživanje teksta, upiti za rad s geoprostornim podatcima i *MapReduce* upiti za složeniju obradu podataka.

Visoka dostupnost se ostvaruje replikacijom podataka. U MongoDB bazi podataka replikacijski skup se sastoji od glavne (primarne) replike i ostalih (sekundarnih) replika. MongoDB pretpostavlja strogu konzistentnost, čitanje i pisanje se provodi nad glavnom replikom. Opcionalno aplikacije mogu čitati podatke iz sekundarnih replika gdje su podatci u konačnici konzistentni (*eventual consistency)*. Ukoliko glavna replika prestane raditi zbog nekog razloga sekundarne replike se automatski dogovaraju koja od njih će postati nova primarna replika, tako da nije potrebna dodatna administracija baze podataka.

MongoDB automatski raspodjeljuje (*sharding)* podatke na više različitih fizičkih particija (*shards)* na transparentan način*.* Kako količina podataka raste ili pada MongoDB automatski balansira njihovu raspodjelu u particijama.Na ovaj način se rješavaju ograničenja sklopovlja na jednom poslužitelju i omogučava dobra horizontalna skalabilnost i balansiranje opterećenja na poslužiteljima.

Osim gore navedenih najbitnijih značajki MongoDB osigurava konzistentnost i sigurnost podataka.

MongoDB podržava ACID svojstva na razini dokumenta. U jedno ili više polja u dokumentu upisivanje se može provesti u jednoj operaciji što omogućava dobru konzistentnost podataka. Ukoliko se pojavi greška prilikom upisivanja u dokument baza će vratiti dokument u prethodno konzistentno stanje. Kako bi se povećala izdržljivost baze korisnik može zadati da se upis obavlja na nekom minimalnom broju replika.

Sigurnost podataka u MongoDB bazi podataka osigurava se autentifikacijom, autorizacijom, nadgledanjem i enkripcijom. MongoDB nudi integraciju s vanjskim sigurnosnim mehanizmima poput certifikata. Podatci u MongoDB bazi podataka mogu se šifrirati na disku ili prije slanja u mrežu. Šifriranje podataka na disku osim što povećava sigurnost, osigurava dobre performanse jer se podatci ne moraju šifrirati svaki put prije slanja u mrežu. Administrator može definirati ili ograničiti pristup podatcima pojedinim korisnicima i aplikacijama.

# Korištenje MongoDB-a

Komunikacija između aplikacije i MongoDB baze podataka odvija se putem programskih biblioteka (*drivers*) jezika u kojem je aplikacija pisana. MongoDB podržava driver-e za većinu današnjih popularnih programskih jezika koji se koriste. Osim službenih driver-a postoji veliki broj driver-a koji su razvijani i održavani od strane velike zajednice korisnika MongoDB-a. Driver-i prilagođavaju manipulaciju bazom podataka programskom jeziku koji se koristi i olakšava korištenje baze podataka korisniku. U ovom projektu ćemo koristit Node.js okruženje za koje se osim podržanih driver-a može koristiti i biblioteka za modeliranje objekata Mongoose. Mongoose olakšava pisanje validacije i poslovne logike, osigurava provjeru strukturu podataka koji se upisuju u bazu i zadržava fleksibilnost koju MongoDB pruža. Pomoću Mongoose biblioteke omogućava se modeliranje sheme baze podataka i dodavanje validacije podataka na razini njihovih tipova. Slijedi primjer kako se spojiti na bazu podataka preko Mongoose-a, definirati jednostavnu shemu i validacije nad njom.

Spajanje na bazu podataka pomoću Mongoose-a je vrlo jednostavno, potreban je samo URL baze podataka:

var mongoose = require('mongoose');

var uristring = 'mongodb://localhost/HelloMongoose'

mongoose.connect(uristring, function(err, res) {

if(err) {

console.log('Error connecting to: ' + uristring + '. ' + err);

} else {

console.log('Succesful connection to: ' + uristring);

}

});

Nakon što smo se spojili na bazu podataka, možemo jednostavno definirati model podataka. Primjerice možemo modelirati ovakvu jednostavnu shemu za korisnika:

var userSchema = new mongoose.Schema({

name:{

first: String,

last: { type: String, trim: true }

},

age:{ type: Number, min: 0 }

});

Ovakva jednostavna shema definira ime korisnika koje se sastoji od imena i prezimena te dob korisnika. Za dob je definirana validacija da mora biti minimalno 0, a prezime ne smije imati bjelina na početku i na kraju. Kada se stvaraju novi objekti ove validacije se provjeravaju i ukoliko nisu zadovoljene dojavit će se pogreška. Na ovaj način se definira jasniji model podataka koji se koristi i ne dozvoljava se unos neispravnih podataka odnosno podataka s nedozvoljenim vrijednostima u bazu podataka.

S obzirom da ova aplikacija barata s malo podataka o korisnicima i porukama koje razmjenjuju odlučili smo se koristiti *mongodb* driver za nodejs. U datoteci *mongo.js* u direktoriju *chat/api* implementirana je konekcija na bazu:

**const** mongoURL = process.**env**.MONGO || **'mongodb://localhost:27017/chat'**;

**let** dbHandler;

**export const** *connectDatabase* = () => {

MongoClient.connect(mongoURL, (err, db) => {

**if** (err) {

**console**.log(**'Error while connecting to the database!'**);

}

dbHandler = db;

});

};

**export const** *getDb* = **function** () {

**return** dbHandler;

};

Metoda *connectDatabase* se spaja URL baze koji je definiran u varijabli *mongoURL*, i u varijablu *dbHandler* sprema konekciju. Metoda *getDb* se exporta i koristi se u aplikaciji za dohvat konekcije na bazu podataka, npr (iz *auth.js*) za spremanje novog korisnika u bazu:

**import** { *getDb* } **from '../mongo'**;

...

**const** db = *getDb*();

**const** newUser = { ...req.**body**, **image**: **''**, **password**: hash.generate(req.**body**.**password**)};

db.collection(**'users'**).insertOne(newUser);

# Zaključak

Za pohranu podataka o korisnicima i njihovim porukama u ovom projektu će se koristiti MongoDB baza podataka. Način na koji se podatci pohranjuju u MongoDB bazu podataka je jednostavan i vrlo je sličan objektima u programskim jezicima pa je korištenje MongoDB-a jednostavno. Dobre značajke MongoDB baze podataka čine ju pogodnom za korištenje u raspodijeljenoj okolini u kojoj će se aplikacija nalaziti. Korištenje MongoDB baze podataka je intuitivno i jednostavno koristeći standarne driver-e ili biblioteku Mongoose. Iz svega prije navedenog proizlazi da je MongoDB kvalitetna baza podataka visokih performansi, horizontalno skalabilna, pouzdana i jednostavna za korištenje.

1. **HTML** – HyperText Markup Language [↑](#footnote-ref-2)
2. **DOM** - The Docuemnt Object Model [↑](#footnote-ref-3)
3. www.facebook.com [↑](#footnote-ref-4)
4. www.instagram.com [↑](#footnote-ref-5)
5. Podatak preuzet sa službene git stranice : ***https://github.com/facebook/react*** [↑](#footnote-ref-6)
6. http://www.bbc.com/ [↑](#footnote-ref-7)
7. http://www.imdb.com/ [↑](#footnote-ref-8)
8. http://imgur.com/ [↑](#footnote-ref-9)
9. Universal Windows Platform [↑](#footnote-ref-10)
10. User interface [↑](#footnote-ref-11)
11. https://facebook.github.io/react/docs/ hello-world.html [↑](#footnote-ref-12)
12. https://github.com/reactjs/redux [↑](#footnote-ref-13)
13. Sve navedeno odnosi se na društvenu mrežu Facebook; informacije kojima aplikacija ima pristup ovise o društvenoj mreži koja se koristi za *social login* [↑](#footnote-ref-14)
14. Internet stvari, *engl. Internet of Things* [↑](#footnote-ref-15)
15. *engl. Software Development Kit* [↑](#footnote-ref-16)
16. Kosrisničko sučelje, *engl. User Interface* [↑](#footnote-ref-17)
17. Bolji UX – ***U****ser* ***E****xperience*  [↑](#footnote-ref-18)
18. ***U****ser* ***D****atagram* ***P****rotocol* – nespojni transportni protokol [↑](#footnote-ref-19)
19. ***T****ransmission* ***C****ontrol* ***P****rotocol* – spojni transportni protokol [↑](#footnote-ref-20)
20. Primjerice: *DES, Triple DES, RSA, Blowfish, AES* [↑](#footnote-ref-21)
21. ***H****yper-* ***T****ext****T****ransfer****P****rotocol****S****ecure* – protokol koji signalizira da se koristi kriptirana komunikacija [↑](#footnote-ref-22)
22. ***S****ecure* ***S****ocket* ***L****ayer* – kriptografski protokol [↑](#footnote-ref-23)
23. ***T****ransfer* ***L****ayer* ***S****ecurity* – kriptografski protokol, nasljednik SSL-a [↑](#footnote-ref-24)
24. ***U****niform* ***R****esource* ***I****dentifier –* identifikator resursa [↑](#footnote-ref-25)
25. ***A****utomatic* ***C****ertificate* ***M****anagement* **E**nvironment – protokol korišten za izdavanje certifikata [↑](#footnote-ref-26)
26. Automatiziran klijent koji služi za dohvaćanje i instalaciju certifikata na poslužitelj [↑](#footnote-ref-27)