Razvoj web aplikacije za upravljanje sadržajem u razvojnom okviru GatsbyJs

Cindori, Matej

SVEUČILIŠTE U ZAGREBU

FAKULTET ORGANIZACIJE I INFORMATIKE

V A R A Ž D I N

Matej Cindori

RAZVOJ WEB APLIKACIJE ZA UPRAVLJANJE SADRŽAJEM U RAZVOJNOM OKVIRU GATSBYJS

ZAVRŠNI RAD

Varaždin, 2022.

SVEUČILIŠTE U ZAGREBU

FAKULTET ORGANIZACIJE I INFORMATIKE

V A R A Ž D I N

Matej Cindori

Matični broj: 2210991301005

Studij: Primjena informacijske tehnologije u poslovanju

RAZVOJ WEB APLIKACIJE ZA UPRAVLJANJE SADRŽAJEM U RAZVOJNOM OKVIRU GATSBYJS

ZAVRŠNI RAD

Mentor:

Prof. dr. sc. Dragutin Kermek

Varaždin, veljača 2022.

Matej Cindori

Izjava o izvornosti

Izjavljujem da je moj završni 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 korištene su etički prikladne i prihvatljive metode i tehnike rada.

Autor/Autorica potvrdio/potvrdila prihvaćanjem odredbi u sustavu FOI-radovi

_____

Sažetak

Cilj rada je obrazložiti perspektivu klijenta i poslužitelja koristeći JavaScript programski jezik. Kroz prvo poglavlje obrazložene su neke promjene koje su promijenile sam JavaScript jezik kroz *ES6 specifikacije* te kako obraditi asinkroni kod u JavaScriptu.

Drugo poglavlje fokusira se na razvoj na strani poslužitelja, na Node i njegov način pisanja modula te korištenje i instalacija modula pomoću npm-a. Također se donosi kratki opis RESTful usluga te kako napraviti poslužiteljske usluge koristeći Express razvojni okvir. Obrada podataka odvija se pomoću *Node* biblioteke *Mongoose-a* koristeći osnovne CRUD operacije nad mongoDB bazom podatka.

Treće poglavlje fokusira se na razvoj na strani klijenta, na *React* i njegove osnovne koncepte, na komponentu, njeno stanje i životni ciklus te pisanje komponenti pomoću funkcija i korištenje *React* metoda koje se mogu nadjačati (eng. *React* hooks). Ukratko su objašnjeni *Gatsby* razvojni okvir, react-spring animacijska biblioteka te PostCss alat za transformaciju CSS-a pomoću JavaScripta.

Zadnje poglavlje razrada je praktičnog dijela rada koje se odnosi na www.lo3go.hr *Gatsby* web stranicu, na mogućnost pisanja, ažuriranja i brisanja blog postova. Razrađena je općenita struktura i arhitektura aplikacija, *Editor* biblioteka oko koje su građene aplikacije, navedene su rute na poslužitelju i kako se privatne rute odvajaju od javnih pomoću JWT-a. Razrađena je komunikacija klijenta s poslužiteljem pomoću Fetch API te obrada tih podataka na dinamičan način pomoću *React*-a i sama integracija *Editor* biblioteke u *Gatsby* razvojni okvir.

Ključne riječi: Web aplikacija, JavaScript, Node.js, Express.js, MongoDB, Mongoose.js, Gatsby.js, React.js, React-spring.js, Postcss, Editor.js

Sadržaj

1. Uvod

Razvoj kompleksnih web aplikacija postaje sve jednostavniji i brži, a u isto vrijeme razvoj zahtjeva sve opširnije znanje alata koje te procese pojednostavljuju. U praksi se razlikuju dvije kategorije programskih poslova za web, a to su: razvoj na strani poslužitelja i razvoj na strani klijent. Obje pozicije zahtijevaju specijalizaciju za tu pojedinu kategoriju posla zbog samog opsega znanja, broja alata i tehnologija potrebnih za obavljanje kvalitetnog posla.

Za ***razvoj na strani klijenta*** u kontekstu web preglednika osnovne tehnologije su JavaScript, HTML i CSS. Poznavanje te tri tehnologije omogućuje inženjerima razvoj modernih web aplikacija, no za ubrzavanje samog procesa razvoja koriste se razni JavaScript razvojni okviri kao što su *React*, *Vue*, *Angular* i dr. te razni dodaci i biblioteke koji ubrzavaju sam taj proces.

Za ***razvoj na strani poslužitelja*** tehnologija se odabire prema kompleksnosti i opsegu same aplikacije, svaka tehnologija ima svoje prednosti i nedostatke za rješavanje specifičnog problema. Uz poslužiteljsku tehnologiju odabire se vrsta baze podataka koja se koristi za trajno čuvanje podataka.

Rad se fokusira na ***JavaScript (JS)*** programski jezik pomoću kojega su napisane i klijent i poslužitelj aplikacije. JavaScript ima jednu od najvećih zajednica otvorenog koda, gdje se mnogi razvojni okviri i biblioteke mogu preuzeti pomoću NPM-a i koristiti u razvoju aplikacija.

Iz perspektive softver inženjera za razvoj na strani klijenta (web preglednik), cilj rada bio je razumjeti i napraviti praktični primjer poslužiteljske usluge za *Gatsby* klijent aplikaciju. Aplikacija se odnosi na sustav a upravljanjem sadržaja (eng. CMS), gdje vlasnik web stranice www.lo3go.hr može pisati, ažurirati i brisati svoj osobni blog za logopedski obrt.

Kroz rad objašnjeni su osnovni koncepti razvoja iz obje perspektive, klijenta i poslužitelja, a kao temelj koristi se JavaScript programski jezik, te su objašnjeni kroz primjere koda neki ES6 koncepti korišteni u praktičnom dijelu rada i kako obratiti asinkrone operacije u JavaScript-u.

Na strani *poslužitelja* korišteno je izvršno okruženje *Node* s mongoDB bazom podataka te razvojni okvir Express za jednostavnije stvaranje RESTful usluga te nekolicina dodataka i biblioteka od kojih je i *Mongoose* pomoću koje se rade jednostavniji upiti na mongoDB bazu podataka. Sigurnosna komunikacija između klijenta i poslužitelja odvija se pomoću JWT-a.

Na strani *klijenta* korišten je *Gatsby* razvojni okvir koji je pak skup alata i procesa, gdje je jedna od temeljnih tehnologija *React*, te su za njega objašnjeni njegovi osnovni koncepti.

Razrađena je i uporaba react-spring biblioteke za animaciju DOM elemenata i kratko objašnjenje PostCss biblioteke za transformaciju CSS-a pomoću JavaScript-a.

U zadnjem poglavlju objašnjeni su opći principi izrade aplikacija, osnovni koncepti iz prva tri poglavlja objašnjeni su kroz praktične primjere aplikacija. Objašnjene su opće strukture aplikacije i njena arhitekture, kratak opis *Editor* biblioteke koja generira čiste podatke u obliku JSON-a oko koje su građene aplikacije. Navedene su krajnje točke, tj. rute na poslužitelju i kako se odvajaju privatne rute od javnih. Obrađen je princip komunikacije klijenta s poslužiteljem pomoću Fetch API te obrada tih dobivenih podatka na dinamičan način pomoću *React*-a i sama integracija *Editor* biblioteke u *Gatsby* razvojni okvir.

2. JavaScript (JS)

JavaScript je lagani, neblokirajući, interpretirani, *objektno-orijentirani jezik* s *funkcijama prve klase*. Baziran na *prototip stilu* objektno-orijentiranog programiranja, on je multi-paradigmi skripti jezik koji je dinamičan i podržava objektno-orijentirani, imperativni i funkcijski programski stil.[1]

Definicija iznad relativno je kompleksna i zbunjujuća i nije potrebno znati sve ove pojmove za početak programiranja u JavaScript-u, ali je korisno. Jeff Delaney u svom uvodnom videu za JavaScript Fireship tečaj objašnjava te pojmove na zanimljiv način:

JavaScript: How It's Made: https://www.youtube.com/watch?v=FSs_JYwnAdI

Prvi oblik JavaScript jezika napravio je Brenden Eich dok je radio za Netscape i sve do sada ažuriran je u skladu s ECMA-262 verzijom 5 i novijim verzijama.[1]

Prema *stackoverflow* izvješćima za 2021 godinu, u proteklih devet godina *JavaScript* najpopularniji je programski jezik među softver inženjerima [31], a razlog popularnosti je njegova korištenost u web preglednicima koji se pretežito koristi za kreiranje dinamičnih i interaktivnih web stranica i aplikacija. JavaScript koristi se ne samo za izradu web stranica, nego i za izradu web poslužitelja, mobilnih i desktop aplikacija, internet stvari (IOT – eng. Internet of Things) te za razvoj video igara.

2.1. ECMAScript6 (ES6)

JavaScript dobio je svoje velike promjene u 2015. godini s izlaskom ECMAScript 2015 (ES6) specifikaciji standarda za JavaScript jezik. Verzija JavaScript-a prije toga dobila je naziv ECMAScript 5 (ES5), gdje su posljednje promjene bile u 2009. godini. Odbor zvan TC39 skup je vodećih web kompanija koji je zadužen za nadzor i razvoj ECMAScript standarda JavaScript jezika koji se još zove i ISO/IEC 16262 ili skraćeno ECMA262.[27]

Dokumentaciji ECMAScript standarda može se pristupiti na web adresi: https://tc39.es/ecma262/.

Mozilla je jedan od članova odbora TC39, a glavni proizvod im je Firefox web preglednik. Kompanija zajedno s zajednicom inženjera prati i dokumentira razvoj web tehnologija i glavnih koncepata kojoj se može pristupiti na web adresi: https://developer.mozilla.org/.

Priloženim primjerima koda za poglavlja ES6 i asinkrono programiranje u JavaScript-u može se pristupiti na web adresi: https://github.com/Cind0/ES6-zavrsni-rad.

2.1.1. Deklaracija varijabli pomoću let i const naredbi i njihov opseg

Prije ES6 promjena u JavaScript-u varijable su se deklarirale putem ključne riječi ***var***, a da bi te varijable ostale lokalne, deklarirale su se unutar funkcije (eng. function scope) [6].

Varijable deklarirane pomoću ***let*** i ***const*** naredbi nalaze se u opsegu unutar blokova, što znači da su sve varijable koje se deklarirane unutar vitičastih zagrada „**{ }**“ dostupne samo u tom blok opsegu, tj. dostupne su lokalno unutar bloka, a ne globalno.[25]

Primjer koda ispod prikazuje *IIFE (eng. Immediately Invoked Function Expression)* funkciju, to je funkcija koja odmah poziva sama sebe te su na taj način varijable lokalne i nalaze se u funkcijskom opsegu:

```
(function () {
  var privatnaVarijabla = true;
})();
console.log(privatnaVarijabla); //ReferenceError
```

Primjer **opsega** unutar vitičastih zagrada ili blok opsega za let i const naredbe prikazana je u primjeru koda ispod:

```
{
  let privatnaLetVarijabla = true;
  const privatnaConstVarijabla = true;
}
console.log(privatnaLetVarijabla); //ReferenceError
console.log(privatnaConstVarijabla); //ReferenceError
```

Glavna razlika između let i const varijabli je u tome da nakon deklaracije **let** varijable *moguće ju je prepisati s drugom vrijednošću* dalje u kodu, a nakon deklaracije **const** varijable *nije ju moguće prepisati dalje u kodu s drugom vrijednošću [25,11]*. Prikazano u primjeru koda ispod:

```
  {
  let mutirajucaVarijabla = true;
  const neMutirajucaVarijabla = true;
  mutirajucaVarijabla = false;
  console.log(mutirajucaVarijabla); // false
  neMutirajucaVarijabla = false; // TypeError
  console.log(neMutirajucaVarijabla);
}
```

2.1.2. Funkcije sa strjelicom

Funkcije sa strjelicom (eng. arrow function) skraćuju i pojednostavljuju uporabu dotadašnje sintakse funkcija. Uporaba funkcija sa strjelicom ima svoja ograničenja i ne može se koristiti u svim situacijama.[3]

Sintaksa funkcija sa strjelicom je:

```
() => {}
```

Te ako se funkcija dodijeliti varijabli:

```
const imeFunkcije = () => {…}
```

U primjeru koda ispod prikazani su tradicionalni načini pisanja funkcijskih izraza s povratnim pozivom u metodi map te funkcija sa strjelicom čija je sintaksa preglednija i kraća.

```
const niz = [1, 2, 3];
const primjerFun = niz.map(function (broj) {
  return broj + 1;
});
console.log(primjerFun); // [2,3,4]
const primjerArrFun = niz.map((broj) => broj + 2);
console.log(primjerArrFun); // [3,4,5]
```

Treba obratiti pozornost na ključnu riječ ***this*** koja se u kontekstu JavaSctipt-a odnosi na način kako je ta funkcija pozvana.[45] U primjeru koda ispod, na klik gumba dodan je događaj koji poziva tradicionalnu funkcijsku izjavu koja ispisuje vrijednost ključne riječi *this,* ona se odnosi na html gumb koji je kliknut. Doduše, kod funkcija sa strjelicom ključna riječ *this* ne postoji, odnosno ne odnosi se na kliknuti gumb, nego na globalni objekt u web pregledniku *window.*

```
const btnFunRef = document.getElementById("btnFun");
btnFunRef.addEventListener("click", function () {
  console.log(this);// [object HTMLButtonElement]
});
const btnArrRef = document.getElementById("btnArrFun");
btnArrRef.addEventListener("click", () => {
  console.log(this); //[object Window]
});
```

Ključna riječ *this* u kontekstu JavaScript objekata i njihovog izvršenja koda kod tradicionalnih funkcijskih izjava i funkcija sa strjelicom prikazana je u primjeru koda ispod:

```
let objekt = {
  vrijednost: 404,
  arrowFun: () => console.log(this.vrijednost, this),
  fun: function () {
    console.log(this.vrijednost, this);
  },
};
objekt.arrowFun(); // undefined, [object Window]
objekt.fun(); // 404, {vrijednost: 404, arrowFun: f, fun: f}
```

2.1.3. Manipulacija teksta

Manipulacija teksta dobila je novu sintaksu u ES6 specifikaciji pod nazivom predložak teksta (eng. Template literals/Template strings). Mogućnost manipulacije višelinijskog teksta te mogućnost interpolacije teksta. Sav tekst koji se manipulira nalazi se unutar kosih jednostrukih navodnika (` `), a dinamični način dodavanja vrijednosti u tekst se deklarira pomoću dolar znaka i vitičastih zagrada (`**${}**`) unutar navodnika.[44]

Primjeri koda ispod uspoređuje manipulaciju teksta prema ES5 i ES6 specifikaciji te oba primjera ispisuju isti rezultat:

```
  const podatci = {
    ime: "John",
    prezime: "Doe",
    br_godina: 23,
    studij: "FOI",
  };
  const ES5tekst ="Osoba" + podatci.ime + " " + podatci.prezime + "ima " +
    podatci.br_godina + " godine.\nStudira na " + podatci.studij +".";
  const ES6tekst =
`Osoba ${podatci.ime} ${podatci.prezime} ima ${podatci.br_godina} godine.
Studira na ${podatci.studij}.`;
   //Osoba John Doe ima 23 godine.
   //Studira na FOI.
```

2.1.4. Sintaksa širenja i ostatka

Sintaksu širenja (eng. spread syntax) reprezentira operator u obliku tri točkice **(…)**, a ona omogućuje jednostavno kopiranje vrijednosti koje se mogu iterirati, kao što su niz, tekst ili objekt pomoću jedne vrijednosti. Ključna oznaka za operator su tri točke te naziv varijable (**…mojNiz**).[43]

Primjer ispod pokazuje korištenje operatora za širenje nad nizom i objektom, kopiranje niza i objekta u novu varijablu koja se ne referencira na istu memorijsku lokaciju kao i proslijeđeni podaci te spajanje niza i objekta u novi niz.

```
    const niz = [1, 2, 3, 4, 5];
    const kopiraniNiz = [...niz];
    console.log(kopiraniNiz === niz, kopiraniNiz); //false, [1, 2, 3, 4, 5]
    const objekt = {
      ime: "John",
      prezime: "Doe",
      br_godina: 23,
      studij: "FOI",
```

```
};
const kopiraniObjekt = { ...objekt };
console.log(kopiraniObjekt); // { ime: "John", prezime: "Doe", …}
const spojeno = [...niz, { ...objekt }];
console.log(spojeno);//[1, 2, 3, 4, 5, {ime: "John", prezime: "Doe", …}]
```

Sintaksa ostatka (eng. rest syntax) koristi iste oznake kao i sintaksa širenja. Može se reći da sintaksa ostatka radi suprotno onome što radi sintaksa širenja. Sintaksa širenja proširuje niz sa svojim elementima, dok sintaksa ostatka prikuplja elemente i dodjeljuje ih u jedan element.[43]

Pod parametrom ostatka (eng. rest parameter) smatra se parametar koji se koristi u nekoj funkciji, što znači da ta funkcija može primiti beskonačno mnogo argumenata, a taj parametar ostatka je niz kroz koji je moguće dalje iterirati.[41]

Primjer koda ispod prikazuje uporabu parametra ostatka gdje su posebno ispisane vrijednosti iz proslijeđenih argumenata kao „prvo, drugo“ te tri ostale vrijednosti u obliku niza kao „ostalo“.

```
const restFun = (prvo, drugo, ...ostalo) => {
      console.log(prvo, drugo); // "jedan", "dva",
      console.log(ostalo); // [3, "cetiri", { pet: 5 }]
};
restFun("jedan", "dva", 3, "cetiri", { pet: 5 });
```

2.1.5.Klase

Klase su modeli po kojima se stvaraju objekti i one su u JavaScript-u temeljene na prototipovima. Klase se deklariraju putem ključne riječi ***class*** [8], prikazano u primjeru koda ispod.

Constructor metoda specijalna je metoda za inicijalizaciju i stvaranje objekata putem klase.[8]

Pomoću ključne riječ ***new*** stvara se objekta od predefinirane funkcije ili klase.

```
class Osoba {
  constructor(ime, prezime) {
    this.ime = ime;
    this.prezime = prezime;
  }
  ispisiOsobu() {
    console.log(`${this.ime} ${this.prezime}`);
  }
}
const Petar = new Osoba("Petar", "Peric");
Petar.ispisiOsobu(); // Petar Peric
```

Primjer koda ispod prikazuje isti princip stvaranja istog objekta putem funkcije kao i s klasama, gdje se metode za taj objekt dodaju u prototip funkcije.

```
function Osoba(ime, prezime) {
  this.ime = ime;
  this.prezime = prezime;
}
Osoba.prototype.ispisiOsobu = function () {
  console.log(`${this.ime} ${this.prezime}`);
};
const Petar = new Osoba("Petar", "Peric");
Petar.ispisiOsobu(); // Petar Peric
```

Problem nastaje kada se želi stvoriti ***podklasa*** definiranog objekta putem funkcije, što je prilično komplicirano u ES5 specifikaciji.[8] U suprotnom, za podklase u ES6 koristi se ključna riječ ***extends***, primjer koda ispod prikazuje *podklasu* od *Osoba* iz prošlog primjera.

```
class Inzinjer extends Osoba {
  constructor(ime, prezime, mjStudiranja) {
    super(ime, prezime);
    this.studij = mjStudiranja;
  }
  ispisiStudij() {
     super.ispisiOsobu(); // Ivan Ivic
     console.log(`${this.studij}`);
  }
}
const Ivan = new Inzinjer("Ivan", "Ivic", "FOI");
Ivan.ispisiStudij(); // FOI
```

U primjeru koda iznad, kod podklase pojavljuje se metoda ***super*** koja poziva odgovarajuće vrijednosti i metode od naslijeđene klase.[8]

2.1.6.Sintaksa destrukturiranja zadatka

Sintaksa destrukturiranja zadatka (eng. destructuring assignment syntax) je JavaScript izjava koja omogućuje da se vrijednosti zapisane u nekim objektima ili nizovima izvuku i pridruže različitim varijablama.[12]

Primjer koda ispod prikazuje ***destrukturiranja niza***:

```
const niz = [1, 2, 3, 4, 5];
const [a, b, ...ostalo] = niz;
console.log(b, a, ostalo); // 2,1,3,4,5
```

Za pridruživanje vrijednosti niza varijablama, varijable se deklariraju unutar vitičastih zagrada **[ ]** za one vrijednosti koje se žele izvaditi iz pridruženog niza.[12] Također se može koristiti sintaksa ostatka, kao što je prikazano u primjeru koda iznad.

Destrukturiranje objekata odvija sa na sličan način kao i kod nizova, no umjesto uglatih zagrada koriste se vitičaste zagrade **{ }.**

Primjer koda ispod prikazuje ***destrukturiranja objekta:***

```
const objekt = { a: 1, b: 2, c: 3, d: 4, e: 5 };
const { a, b, ...ostalo } = objekt;
console.log(b, a, ostalo); // 2, 1, { c: 3, d: 4, e: 5 }
```

2.2. Asinkrono programiranje u JavaScript-u

Asinkrono programiranje drugim riječima znači **ne blokirajući** tip programiranja. U suprotnom, imamo **sinkroni** način programiranja ili **blokirajući**.[30]

Ove pojmove najlakše je objasniti kroz jednostavan primjer koda:

```
console.log("Prije!"); // Prije!
setTimeout(() => {
  console.log("Dohvat podataka!"); // Dohvat podataka!
}, 2000);
console.log("Poslije!"); // Poslije!
```

setTimeout() ugrađena je asinkrona funkcija u JavaScript-u koja prima dva argumenta: funkciju s povratnim pozivom i broj u milisekundama. To je funkcija *koja će izvršiti proslijeđenu funkciju nakon zadanog broja milisekundi*. Ova metoda simulira dohvat podataka s nekog poslužitelja ili dohvat podataka iz baze podataka.

U **sinkronom programiranju** skripta ispisuje podatke slijedećim redom:

1. **> Prije!**
2. *Čeka se dvije sekunde!* **> Dohvat podataka!**
3. **> Poslije!**

Dohvat podataka zaustavio bi cijeli program i čekalo bi se da se sve operacije unutar te funkcije izvrše!

U **asinkronom programiranju** skripta ispisuje podatke slijedećim redom:

1. **> Prije!**
2. **> Poslije!**
3. *Čeka se dvije sekunde!* **> Dohvat podataka!**

U asinkronom načinu programiranja, setTimeout() metoda ***zadaje zadatak*** JavaScript motoru (*eng. engine*) koja će izvršiti proslijeđenu funkciju nakon zadanog vremena i nakon što je zadatak zadan, *program se nastavi dalje izvršavati.*[30]

U JavaScript-u postoje tri načina kako obraditi takav asinkroni kod:

- **Funkcija s povratnim pozivom (eng. callback function)**
- **Promise objekt**
- **Async await**

Kroz slijedeća tri poglavlja objašnjeni su koncepti asinkronog programiranja te se koristi isti primjer s istim rezultatom kako bi se obrazložila ta tri koncepta.

2.2.1. Funkcija s povratnim pozivom

Funkcija s povratnim pozivom (eng. callback function) funkcija je koja je proslijeđena kao argument u neku drugu funkciju koja je inicijalizirana unutar te druge funkcije da se izvrši neka operacija.[7]

Za primjer **funkcije s povratnim pozivom** napravljena je funkcija *dohvatiKorisnika()* koja kao prvi parametar prima ***id***, a kao drugi parametar funkciju ***callback***. Pomoću ovakvog principa mogu se obraditi podaci dobiveni od poslužitelja ili iz baze podataka.

```
const korisnici = [
  { id: 1, ime: "Matej", rod: 2 },
  { id: 2, ime: "Petar", rod: 1 },
];
const dohvatiKorisnika = (id, callback) => {
  setTimeout(() => {
    const pronadjeniKorisnik = korisnici.find(
      (korisnik) => korisnik.id === id
    );
    callback(pronadjeniKorisnik);
  }, 2000);
};
dohvatiKorisnika(1, (korisnik) => {
  console.log(korisnik); // { id: 1, ime: "Matej", rod: 2 }
```

```
      dohvatiKorisnika(korisnik.rod, (korisnik) => {
        console.log(korisnik); // { id: 2, ime: "Petar", rod: 1 }
      });
    });
```

No, ovakav princip donosi svoje komplikacije, od samog načina ***efikasnog rješavanja problema (eng. error handling)***, do ***višestrukog ugnježđivanja funkcija s povratnim pozivom*** gdje se zove više servisa ili više upita na bazu podataka, takozvani pakao funkcija s povratnim pozivom (eng. callback hell). Da bi se izbjegli ti problemi s funkcijama s povratnim pozivom, od publikacije ES6 promjena u JavaScript-u postoji objekt ***Promise*** koji je zadužen za takve asinkrone operacije.[30]

2.2.2.Promise objekt

Promise je JavaScript objekt koji sadržava eventualni rezultat od asinkrone operacije. Kada se ta asinkrona operacija završi, njezin rezultat može biti ili neka vrijednost ili pogreška.[34]

Promise objekt može biti u tri stanja: neriješen (eng. pending), ispunjen (eng. fulfilled) i odbijen (eng. rejected)

- ***Neriješen*** je inicijalno stanje, niti je ispunjena niti odbijena.
- **Ispunjen** znači da se operacija uspješno završila.
- **Odbijen** znači da je operacija neuspješna.[34]

Kroz sljedeći primjer koda prikazan je način obrade asinkrone operacije pomoću *Promise* objekta.

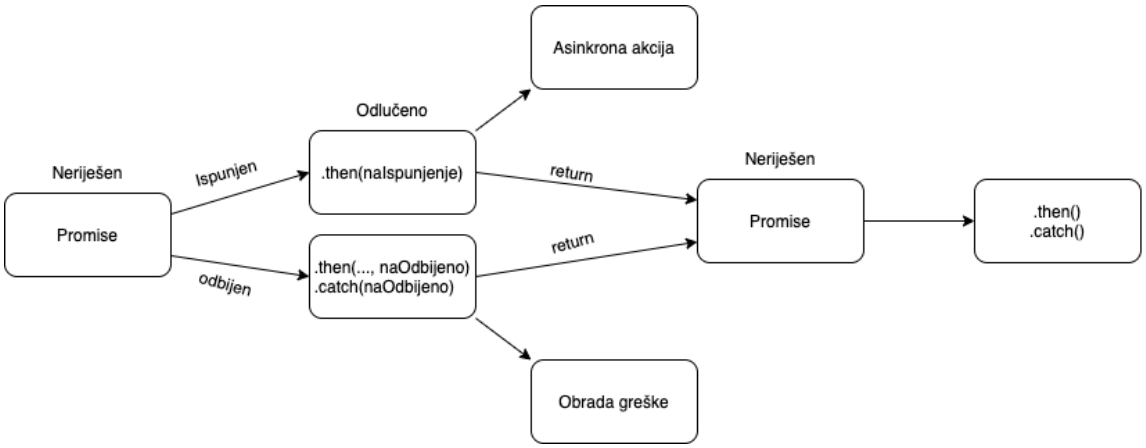
```
const korisnici = […];
const dohvatiKorisnika = (id) => {
  return new Promise((resolve, reject) => {
    setTimeout(() => {
      const pronadjeniKorisnik = korisnici.find(
        (korisnik) => korisnik.id === id
      );
      console.log(pronadjeniKorisnik);
      if (!pronadjeniKorisnik) reject(new Error("Pogreska"));
      resolve(pronadjeniKorisnik);
    }, 2000);
  });};
```

```
dohvatiKorisnika(1)
  .then((korisnik) => {return korisnik;})
  .then((korisnik) => dohvatiKorisnika(korisnik.rod))
  .catch((pogreska) => console.log(pogreska.message));
```

Funkcija *dohvatiKorisnika()* sada vraća novi *Promise* objekt. Taj Promise objekt pri ***deklaraciji*** prima funkciju sa svoja dva argumenta ***resolve*** i ***reject.*** Resolve i reject argumenti su zapravo funkcije s povratnim pozivom, gdje se *resolve* koristi kada je operacija uspješno završena i vraća se neka vrijednost, u konkretnom slučaju kada se dohvate podaci iz baze podataka ili zove neka krajnja točka poslužitelja, a *reject* kada je došlo do pogreške s poslužiteljem ili dohvatom podataka iz baze podataka.

Kada se funkcija *dohvatiKorisnika()* ***inicijalizira,*** stvara se novi *Promise* objekt koji sadržava dvije metode **then** i **catch.** Kada se asinkrona operacija uspješno izvrši, zove se *resolve* funkcije i s *then* metodom se te vrijednosti obrađuju dalje u programu. U slučaju neuspjeha zove se *reject* funkcija, a dalje u programu pogreška se obrađuje putem *catch* metode.

Nadalje, *then* i *catch* metode vraćaju *Promise* objekt koje se može dalje ***povezivati u lanac***: *dohvatiKorisnika().then().then().then().catch().* Koncept je prikazan u primjeru koda iznad pri inicijalizaciji funkcije *dohvatiKorisnika()*, slikovitije na dijagramu ispod.



Slika 1. Povezivanje Promise objekta u lanac (izvor: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise, pristupljeno: 29. 7. 2021.)

Ako se želi **paralelno** izvršiti asinkrone operacije, koristi se ***Promise.all([funkcijaA, funkcijaB])*** metoda koja prima niz asinkronih funkcija.[34]

2.2.3. Async await

Async await dodan je u JavaScript pri publikaciji ES 2017, koji je u suštini *sintaktički šećer* (eng. syntactic sugar) na Promise objektu. *Služi da se asinkrone operacije pišu na sinkroni način.*[30]

Primjer koda ispod prikazuje uporabu ***async*** funkcije i ključne riječi ***await*** gdje su niz *korisnici* i funkcija *dohvatiKorisnika()* isti kao i kod Promise primjera, funkcija vraća novi Promise objekt.

```
const korisnici = […];
const dohvatiKorisnika = (id) => {…};

const obradiPodatke = async () => {
  try {
    const korisnik = await dohvatiKorisnika(1);
    const rod = await dohvatiKorisnika(korisnik.rod);
  } catch (pogreska) {
    console.log(pogreska);
  }
};
obradiPodatke();
```

Svaki puta kada funkcija vraća Promise objekt, u ovom slučaju to je funkcija *dohvatiKorisnika()*, koristiti se ključna riječ ***await.*** Da bi se vrijednost mogla obraditi, pridružuje se nekoj varijabli kao u primjeru iznad, gdje je sintaksa slična sinkronom programiranju.

Svaka ključna riječ *await* ***mora biti unutar funkcije*** koja se dekorira s ključnom riječi **async.**[4]

Kako bi se mogle pogreške obraditi pomoću async await, koristi se ***try catch*** blok koji pokušava obraditi asinkrone operacije unutar *try* bloka te ako jedan od Promise objekata vrati pogrešku, prosljeđuje se u *catch* blok.[46]

3. Razvoj na strani poslužitelja

Pod **poslužiteljem (eng. server)** smatra se ono s čime korisnik nema direktni doticaj, tj. ne vidi pozadinske procese, kao što je komunikacija s bazom podataka i operacije nad njom. Postoje razni programski jezici koji obavljaju te poslužiteljske usluge: php, go, python, java, *JavaScript* i dr. Sukladno tome odabire se neka vrsta baze podataka za trajno čuvanje podataka, gdje se najčešće koristi relacijska (SQL) ili ne-relacijska (NoSQL) baza podataka.

Kroz poglavlja koja slijede objašnjeni su osnovni koncepti ***Node*** izvršnog okruženja i njegov ***modularni sustav*** te instalacija modula pomoću ***npm-a***. Ukratko je objašnjen koncept ***RESTful*** usluga te slijedi uvod u ***Express*** razvojni okvir i ***CRUD*** operacije nad ***mongoDB NoSQL*** bazom podataka koristeći ***Mongoose*** biblioteku i kratki opis ***JWT***-a (eng. Json Web Token) za autentikaciju korisnika.

3.1. Node.js izvršno okruženje

Node je izvršno okruženje (eng. runtime environment) otvorenog koda za izvođenje JavaScript-a izvan web preglednika, često se koristi za izradu poslužiteljskih usluga.[30].

Node je razvio Ryan Dahl u 2009. godini, a prije nastanka *Node-a* JavaScript koristio se samo unutar okruženja web preglednika, svaki web preglednika ima svoj motor (eng. engine) koji pretvara JavaScript jezik u jezik koje računalo razumije. *Node*, iako ima isti JavaScript motor (***V8)*** kao i *Google Chrome*, umjesto pristupa DOM-u ima pristup operacijskom sustavu, datotečnom sustavu ili može slušati na određeni računalni ulaz (eng. port), što se ne može unutar web preglednika. Zbog svoje asinkrone ili neblokirajuće naravi, *Node* je iznimno brz i skalabilan te ga koriste velike korporacije kao Netflix, Uber, Paypal i dr.[30]

3.1.1. Modularni sistem u Node-u

Unutar *Node* okruženja postoje globalni objekti slično kao i u web pregledniku. *JavaScript* u web pregledniku dodaje sve globalne varijable u *window* globalni objekt.

Primjeri takvih globalnih objekata i funkcija u web pregledniku i *Node*-u je:

```
consele.log(), setTimeout(), setInterval()
```

Do tih globalnih funkcija i objekata dolazi se pomoću window objekta u web pregledniku, no ne treba pisati ključnu riječ *window* jer *JavaScript* motor to automatski radi za te metode i funkcije.

```
window.console.log("poruka");
```

Također, ako se deklarira varijabla, a da nije u nikakvom blok opsegu ili funkcijskom opsegu, ta varijabla dodaje se u globalni *window* objekt.

```
let poruka = "pozdrav";
console.log(window.poruka)//pozdrav
```

U *Node*-u ne postoje *window* ni *document* objekt, globalni objekt u *Node*-u je ***global.*** Putem tog objekta dolazi se do globalnih funkcija i metoda:

```
global.console.log("poruka");
```

Ako se napiše varijabla izvan blok opsega ili funkcije u *Node*-u, one se ***ne dodaju u globali objekt, global***. Opseg varijable se zbog modularnog sistema u *Node*-u dodjeljuju na ***razini datoteke***.[30]

Moduli predstavljaju izoliranu cjelinu koda, sve varijable inkapsulirane su unutar modula, nisu dostupne putem globalnog objekta, iz terminologije objektno orijentirano programiranja one su privatne.[30]

Svaka *Node* aplikacija ima barem jedan modul, tj. datoteku u kojoj se piše kod, no inače u arhitekturi aplikacije svaka zasebna cjelina ima svoju datoteku ili modul. Svaka datoteka unutar *Node* izvršnog okruženja učahurena je unutar *IIFE* funkcije koja kao jedan od parametara ima ***objekt module*** koji služi da se ti moduli mogu uključiti i dalje koristiti u ostalim modulima, tj. datotekama.[30]

3.1.1.1. Stvaranje i korištenje modula

Pomoću ***module*** *objekta*, koji se nalazi u svakom modulu unutar *Node* izvršnog okuženja, omogućuje se da su proslijeđeni objekti ili funkcije dostupni van tog modula i da se mogu uključiti u novu datoteku tj. modul.

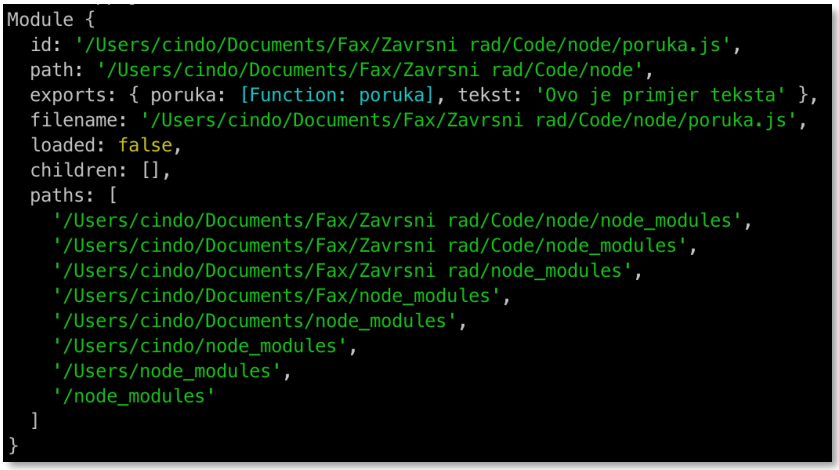
U primjeru koda ispod, koji se nalazi u novoj datoteci *poruka.js*, pomoću *module objekta* omogućuje se pristup tim varijablama i funkcijama van te datoteke koristeći ***module.exports*** objekt:

```
const poruka = (tekst) => {
  console.log(tekst);
};
let tekst = "Ovo je primjer teksta";
module.exports.poruka = poruka;
module.exports.tekst = tekst;
```

U slučaju više varijabli ili funkcija u module.exports objekt mogu se proslijedi vrijednosti kao jedan objekt, prikazano u primjeru koda ispod za funkciju *poruka* i varijablu *tekst*:

```
module.exports = { poruka, tekst };
```

Slika 2 prikazuje ispis objekta *module* unutar komandne linije, gdje se nalazi i objekt *exports* i njegove proslijeđene funkcije i varijable:



Slika 2. Node objekt module (izvor: vlastita izrada)

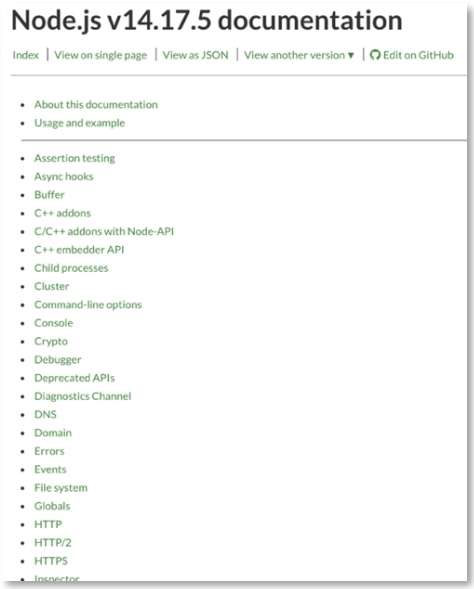
Za uključivanje modula koristi se funkciju ***require(),*** koja je jedan od parametara IIFE funkcije, koja kao argument prima putanju do određene datoteke. Funkciju *poruka()* pridružujemo varijabli pomoću *require()* funkcije:

```
const poruka = require("./poruka");
```

I dalje se ta funkcija koristi u pridruženoj datoteci/modulu:

```
poruka("Primjer funkcije!"); //Primjer funkcije!
```

3.1.1.2. Node ugrađeni moduli



Node ima u sebi ugrađene module pomoću kojih se može manipulirati s operativnim sustavom, datotečnim sustavom, putanjama, događajima, http protokolom i dr. Opširniji pregled ugrađenih modula može se vidjeti na web dokumentaciji: https://nodejs.org/dist/latest-v14.x/docs/api/, od pisanja ovog rada zadnja stabilna verzija je: ***v14.17.5 API*** $_{LTS}$. Klik na neki od tih modula vodi dalje na njihovo detaljnije objašnjenje, opis klasa, metoda i događaja koji se mogu koristiti za gradnju aplikacije.

Slika 3. *Node,js* popis modula i objekata (izvor: https://nodejs.org/dist/latest-v14.x/docs/api/, pristupljeno: 8. 8. 2021.)

- **Events modul**

Treba obratit pozornost na ***Events modul***, jedan od glavnih koncepata u *Node* izvršnom okruženju baziran je na događajima (eng. events), na svaki događaj može se slušati. a može ih se i objaviti.

Primjer događaja je kada klijentska aplikacija napravi zahtjev prema poslužitelju, unutar *Node* izvršnog okruženja objavi se događaj pomoću kojega se dalje vrši dohvat podataka iz baze i/ili šalje dalje klijentu.

Sličnu funkciju ima i web preglednik u svom *window* objektu gdje klijentska aplikacija može slušati na događaje kao što je na klik(*onclick)*:

```
window.addEventListener()
```

Node ima svoj modul *Events* koji omogućuje slušanje događaja, a i njihovu objavu.

Pomoću *require()* funkcije pristupamo modulu te pomoću ključne riječi *new* kreira se objekt *EventEmitter*:

```
const EventEmitter = require("events");
const emiter = new EventEmitter();
```

Uglavnom se koriste dvije metode ovog modula: ***emit() i on().*** Metoda *on()* omogućuje slušanje na objavljeni događaj, prvi argument je naziv događaja, a drugi argument je funkcija koja dalje radi određene akcije:

```
emiter.on("obavjestKorisniku", (podatci) => {
  console.log(podatci.poruka); // Obavijest korisniku!
});
```

Metoda *emit()* služi za objavu događaja, gdje je prvi argument *naziv* događaja, a drugi argument su *podaci* koji se šalju za taj događaj:

```
emiter.emit("obavjestKorisniku", {poruka: "Obavijest korisniku!"});
```

- **HTTP modul**

Pomoću ***HTTP modula*** radi se web servis koji omogućuje slanje podataka preko HTTP protokola. Pomoću require() funkcije pristupa se http modulu i stvara se novi *objekt server* s metodom createServer():

```
const http = require("http");
const server = http.createServer();
```

Pomoću tog *server objekta* poslužitelj sluša na određeni računalni ulaz (eng. port):

```
server.listen(3000)
```

Kada se *Node* aplikacija pokrene pomoću komandne linije: ***node imeDatoteke.js***, u web pregledniku se pomoću URL-a pristupa poslužitelju na lokalnoj adresi: localhost:3000.

Premda je HTTP modul zapravo prošireni Events modul, on može slušati na određene događaje. Primjer, pri posjeti *localhost:3000* adrese, objavi se događaj *"connection"* te nakon posjete te adrese u komandnoj liniji ispiše se: *Nova konekcija!*

```
server.on("connection", () => {
  console.log("Nova konekcija!");
});
```

Za obradu zahtjeva od klijenta, metoda *createServer()* prima funkciju s povratnim pozivom koja sadrži dva objekta ***req i res***. To su skraćenice za engleske riječi request ili **zahtjev** i response ili **odgovor.** Ta dva objekta služe za obradu zahtjeva:

```
const server = http.createServer((req, res) => {
  if (req.url === "/") {
    res.write("Pozdrav!");
    res.end();
  }
});
```

HTTP modul i primjer koda iznad prikazuju primjer najniže razine stvaranja *Node* web poslužitelja, no za stvaranje ruti i obrada zahtjeva za praktični dio projekta koristi se razvojni okvir *Express*.

3.2. Instaliranje paketa pomoću NPM-a

NPM (eng. Node Package Manager) je alat za komandnu liniju, također i registar biblioteka softvera treće strane koje se mogu dodati u *Node* aplikaciju. NPM registar jedan je od najvećih i najbrže rastućih registara besplatnih softvera otvorenog koda.[32]

Pri instalaciji *Node*-a automatski se instalira npm za komandnu liniju. Svaki *Node* projekt ima ***package.json*** datoteku u korijenu aplikacije, to je json datoteka koja sadržava osnovne informacije o projektu kao što je naziv projekta, verzija, autor, git repozitorij i dr. Stvaranje *package.json* datoteke radi se putem naredbe ***npm init.*** Također, unutar te datoteke spremaju se informacije o svim bibliotekama treće strane i njihovim trenutačnim verzijama.

Instalacija biblioteke pomoću npm-a odvija se u komandnoj liniji, gdje se poslije npm init naredbe upiše ***npm install imePaketa.*** Poslije instalacije npr. *Express* paketa u *package.json* datoteci stvara se *dependencies* objekt i svi instalirani paketi nalaze se unutar tog objekta, zajedno s trenutnom verzijom paketa.

```
"dependencies": {"express": "^4.17.1"}
```

Svi instalirani paketi nalaze se u direktoriju ***node_modules*** koji je dodan u korijen projekta. Kako bi se koristio *Express*, na početku datoteke ga se uključi pomoću *require()* funkcije.

```
const express = require("express");
```

Paket se uključuje u projekt bez navigacije u direktorij npr. (“***./express***“), jer ako se doda ./ ili ../, *Node* smatra da je do datoteka koja je u datotečnom sustavu projekta, a ne *node_modules* direktoriju.

Naredba za brisanje paketa je: ***npm uninstall imePaketa.***

3.3. Restful usluge

Danas se klijent – poslužitelj arhitektura koristi u većini web aplikacija. S poslužiteljem se komunicira preko URL-a, tj. krajnjih točaka. Ta konekcija između programa ili računala također se naziva **API** (eng. application programing interface).

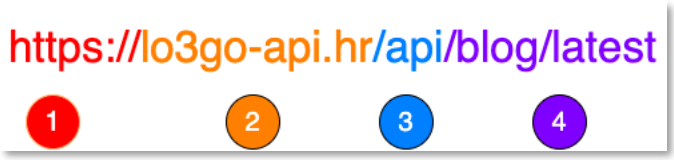
Slika 4. Dijagram klijent-poslužitelj komunikacije (izvor: vlastita izrada)

REST je skraćenica od engleske riječi **Re**presentational **S**tate **T**ransfer, to je zapravo konvencija za gradnju HTTP servisa. RESTful usluga koristi princip HTTP protokola za *stvaranje, ažuriranje, čitanje i brisanje* podataka, a te operacije nazivaju se CRUD operacije od engleskih riječi **C**reate, **R**ead, **U**pdate i **D**elete. Tip HTTP zahtjeva određuje koja će se od tih CRUD operacija izvršiti, svaki HTTP zahtjev ima svoju metodu pomoću koje određujemo tip ili namjeru zahtjeva.[30]

Standardne HTTP metode:

- **Get** – za *čitanje* podataka
- **Post** – za *spremanje* podataka
- **Put** – za *ažuriranje* podataka
- **Delete** – za *brisanje* podataka

Kako bi se komunikacija mogla ostvariti između klijenta i poslužitelja, na poslužitelju se mora omogućiti krajnja točka (eng. end-point), koja se reprezentira kao URL:



Slika 5. Primjer krajnje točke poslužitelja (izvor: vlastita izrada)

1. **https://** – protokol pomoću kojeg se odvija komunikacija, https predstavlja sigurnu vezu
2. **lo3go-api.hr** – predstavlja domenu poslužitelja
3. **/api** – predstavlja konvenciju pisanja API krajnjih točaka, nije pravilo
4. **/blog/latest** – odnosi na se što ta krajnja točka vraća, u kontekstu primjera vraća posljednji objavljeni blog post

3.4. Uvod u Express razvojni okvir

Express je minimalni i fleksibilni *Node* razvojni okvir za razvoj web aplikacije.[13] Uvid u Express API dokumentaciju može se pristupiti na web stranici za verziju 4x:

https://expressjs.com/en/4x/api.html.

U poglavlju 3.1.1.2 ukratko je obrađen HTTP modul pomoću kojega se u *Node*-u stvara web servis, no pisanje koda pomoću tog modula nije prikladno čim je riječ o većoj aplikaciji, tj. nije održivo. Zbog takvih razloga pojavljuju se razvojni okviri kao što je Express koji taj proces pojednostavljuju te olakšava razvoj i održivost.

Primjer koda ispod prikazuje osnovni primjer web servisa pomoću Express razvojnog okvira, Express modul vraća funkciju koja kreira Express objekt sa svojim metodama. Kroz praktični dio rada koristile su se CRUD metode: *get(), post(), put() i delete()*.

```
const express = require("express");
const app = express();
app.get();
app.post();
app.put();
app.delete();
const port = process.env.PORT || 3000;
app.listen(port, () => console.log(`Slusam na port ${port}`));
```

Operacije s bazom podataka obrađene su u slijedećem poglavlju, a zbog lakšeg uvida u ove metode koristit će se statične vrijednosti iz niza s objektima.

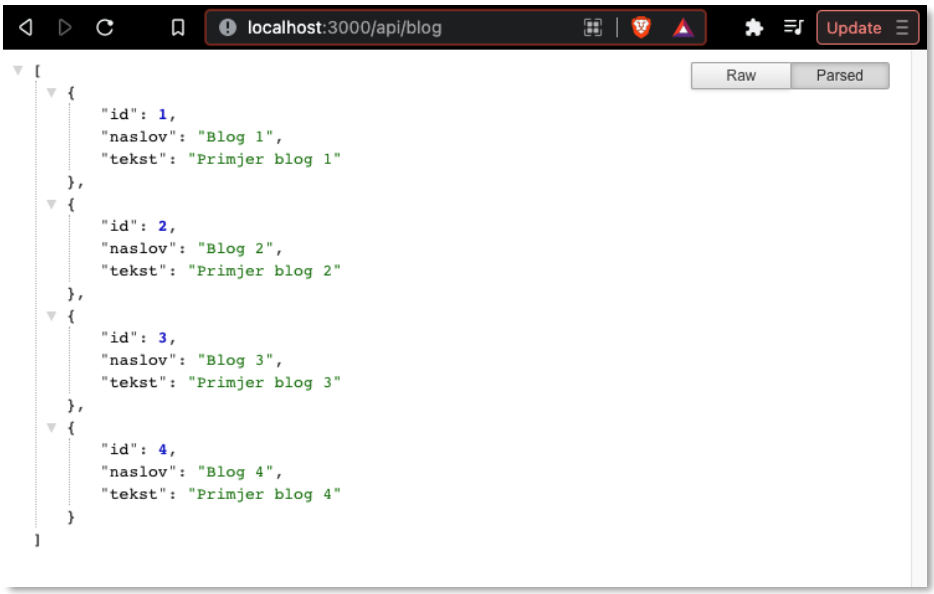
```
const blogPosts = [
  {id: 1, naslov: "Blog 1", tekst: "Primjer blog 1"},
  {id: 2, naslov: "Blog 2", tekst: "Primjer blog 2"},
  {id: 3, naslov: "Blog 3", tekst: "Primjer blog 3"},
  …
];
```

Dohvat podataka putem ***get()*** metode:

```
app.get("/api/blog", (req, res) => {
  res.send(blogPosts);
});
```

Prvi argument koji prima, kao i svaka Express CRUD metoda, je **putanja**, u primjeru koda iznad putanja je "**/api/blog**". Drugi argument je funkcija s povratnim pozivom koja služi za obradu zahtjeva i slanje odgovora, objekti **req i res** u *Express-u* imaju korisna svojstva koje razvojni okvir nudi, u radu je korišteno samo nekolicina njih: *.status(), .json(), .send()*.

Pomoću *res.json()* metode klijentu se šalje niz objekata blog postova u obliku JSON formata, prikazano na Slici 6 ispod.



Slika 6. Primjer JSON datoteke poslanih putem get metode (izvor: vlastita izrada)

Ako se dohvaća samo jedan blog post, unutar putanje *get()* metode dodaje se dinamički parametar pomoću dvotočke: */api/blog/**:id.*** Primjer koda ispod prikazuje način dohvata blog posta po njegovom *id-u*. Do *id-a* iz *URL-a* dolazi se putem **req.params objekta**, /:id je samo primjer imenovanja, parametar može imati bilo koji naziv, npr. */:blogId*.

```
app.get("/api/blog/:id", (req, res) => {
  const post = blogPosts.find((bp) => bp.id === parseInt(req.params.id));
  if (!post) res.status(404).send(`Ne postoji Blog Post`);
  res.send(post);
});
```

Ako blog post ne postoji unutar blogPosts objekta, poslužitelj vraća odgovarajući HTTP status kod pomoću metode *status()*.

Slanje novih podataka s klijent aplikacije na poslužitelj aplikaciju u obliku forme ili novog blog posta odvija se preko ***post()*** metode, slanje podataka odvija se preko iste rute *"/api/blog"*.

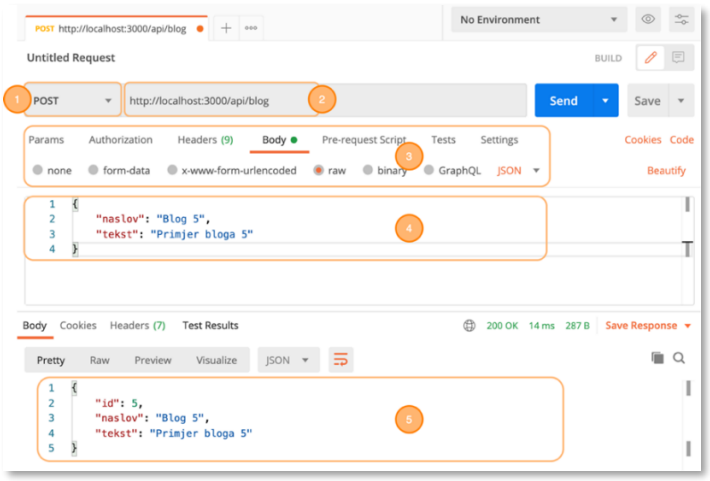
Primjer koda ispod prikazuje obradu pristiglih podataka s klijenta, gdje se pristigli podaci čitaju iz tijela zahtjeva ***req.body.***

Podaci koji se šalju putem tijela zahtjeva su tekstualnog tipa podataka (eng. string). Kako bi se obradili ti pristigli podaci, na početku aplikacije treba se koristit posrednički softver ugrađen u *Express* razvojni okvir za pretvorbu tijela zahtjeva u JSON format ili se može koristiti metoda *JOSN.parse()*.

```
app.use(exoress.json())

app.post("/api/blog", (req, res) => {
  const blogPost = {
    id: blogPosts.length + 1,
    naslov: req.body.naslov,
    tekst: req.body.tekst,
  };
  blogPosts.push(blogPost);
  res.send(blogPost);
});
```

Pomoću *postman* aplikacije (https://www.postman.com/) mogu se slati zahtjevi na neku određenu krajnju točku poslužitelja. Slika 7 ispod prikazuje slanje blog post objekta pomoću postman aplikacije za primjer koda iznad.



Slika 7. Postman aplikacija – primjer slanja podatak pomoću post() metode (izvor: vlastita izrada)

1. Padajući izbornik HTTP zahtjeva, odabrana je post metoda.
2. Krajnja točka poslužitelja tj. njegov URL.
3. Postavljanje **tijela (eng. body)** zahtjeva. Odabere se **row** i iz padajućeg izbornika tip podatka **JSON**.
4. JSON objekt blog posta koji klijent šalje poslužitelju.
5. Odgovor poslužitelja s JSON objektom spremljen u varijablu.

Validacija pristiglih podataka odvija se u trenutku kada su podaci prvi puta dostupni u *req.body* objektu, i ispravnost tih podataka provjerava se prije daljnjih operacija. U praktičnom dijelu ovog rada korištena je popularna *Node* ***Joi*** biblioteka (https://github.com/sideway/joi) koja omogućuje jednostavan i intuitivan način validacije tih podataka.

Ostale dvije metode **put() i delete()** obrađuju se na sličan način, pomoću **req.params** vrijednosti može se proslijediti *id* za neku kolekciju podataka koja se želi **ažurirati ili obrisati**, a pomoću **req.body** objekta šalju se podaci koji se mijenjaju u nekoj kolekciji podataka.

3.4.1. Posrednički softver

Jedan od glavnih koncepata *Express* razvojnog okvira je posrednički softver (eng. middleware), to je zapravo funkcija koja prima *req i res objekt* te vraća odgovor klijentu ili predaje kontrolu drugoj posredničkoj funkciji.[48]

Razlikuje se pet tipova posredničkih softvera, na razini:

- Aplikacije (eng. application-level)
- Usmjerivača (eng. router-level)
- Obrade pogrešaka (eng. error-handling)
- Ugrađeni (eng. built-in)
- Treće strane (eng. third-party) [48]

U prijašnjim primjerima koda već su korištene posredničke funkcije, na razini aplikacije, drugi argument *get() i post()* metode je funkcija koja obrađuje zahtjev, ona se smatra posredničkom funkcijom. Također, spomenuta je *express.json()* posrednička funkcija koja je ugrađena unutar razvojnog okvira Express.

Pomoću metode ***use(),*** posrednička funkcija dodaje se između metoda u aplikaciji. Ako posrednička funkcija ne odgovora klijentu pomoću *res* objekta, funkcija mora zvati ***next()*** funkciju da se proslijedi kontrola slijedećoj posredničkoj funkciji.

Usmjerivač (eng. router) predstavlja objekt koji služi za obradu posredničkih funkcija i usmjerivanje aplikacije na neku rutu.[14] Pomoću ovog objekta opredjeljuju se specifične rute aplikacije, npr. ruta */blog* da bude odvojena od */korisnik*.

Primjer koda ispod prikazuje *router* objekt koji pomoću *get()* metode dohvaća sve blog postove na ruti */privatniBlog*, a prije *get()* metode poziva se funkcija unutar *router.use()* metode koja ispisuje HTTP tip zahtjeva GET u komandnoj liniji te se u glavi *Express* objekt ***app*** dodaje *router* objekt na isti način kao i posrednička funkcija, pomoću *use()* metode.

```
const router = express.Router();
router.use(function (req, res, next) {
  console.log(req.method); //GET
  next();
});
router.get("/", (req, res) => {
  res.json(blogPosts);
});
app.use("/privatniBlog", router);
```

U praktičnom dijelu rada korišten je princip posredničke funkcije kod provjere autentičnosti korisnika pomoću JWT-a.

3.5. CRUD operacije koristeći Mongoose biblioteku

U prošlom poglavlju korištena je varijabla s nizom objekata koja predstavlja JOSN (eng. **J**ava**S**cript **O**bject **N**otation) dokument koja čuva podatke na poslužitelju, no ako poslužitelj prestane s radom, sve vrijednosti se gube jer su pohranjene u radnoj memoriji. Kako se ne bi izgubili podaci, koristi se neki oblik baze podataka. U kontekstu tehnologije JavaScript popularna baza podataka je **MongoDB** koja je NoSQL ili nerelacijski tip baze podataka.

U relacijskim bazama ili SQL bazama podataka arhitektura se mora isplanirati prije početka rada, a kod nerelacijskih ili NoSQL baza, u ovom slučaju mongoDB, podaci se spremaju u kolekcije sa svojim dokumentima koji su u obliku JSON formata i to bez neke predefinirane strukture, pri čemu svaki dokument ima svoj jedinstveni broj ili *id*.[30]

Rad s mongoDB bazom podataka jako je širok pojam, a za potrebe ovog rada korištena je ***Mongoose*** biblioteka koja daje jednostavne metode pomoću kojih se rade operacije nad mongoDB bazom podataka, također daje strukturu JOSN dokumentima pomoću ***shema*** i jednostavne metode za upite nad dokumentima pomoću ***modela***.

Pomoću npm-a instaliramo mongoDB i *Mongoose* unutar komandne linije:

```
nam install mongodb --save
npm install mongoose
```

Unutar aplikacije dodaje se *Mongoose* modul:

```
const mongoose = require('mongoose');
```

Spajanje na bazu se odvija pomoću metode *connect* iz mongoose objekta.

```
mongoose
  .connect("mongodb://localhost/blogs")
  .then(() => console.log("Uspješno spojeno na bazu!"))
.catch((err) => console.log("Pogreška pri spajanju na bazu!", err));
```

Metoda *connect* stvara *Promise* objekt na kojega se može vezati *then()* i *catch()* metode, a kao argument prima konekcijsku putanju u obliku tekstualnog (eng. string) tipa podataka koja započinje s ***mongodb://***, nakon koje slijedi domena poslužitelja, u ovom je slučaju *mongodb* instaliran na lokalnom računalu i adresa je ***localhost***, i zadnja riječ ***/blogs*** naziv je baze podataka. *Mongoose* automatski stvara bazu pri prvom pokretanju web poslužitelja, ako baza već ne postoji.

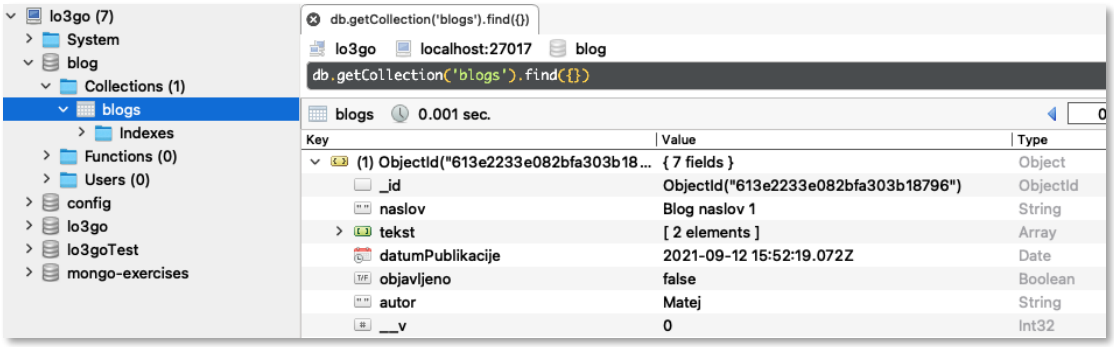
3.5.1. Mongoose shema

Nakon spajanja na bazu i stvaranja kolekcije, dokument se formira pomoću ***Mongoose* sheme**. Pomoću te *sheme* daje se struktura dokumentu, tj. oblikuje se sa specifičnim vrijednostima i tipom podataka koje te vrijednosti poprimaju.[29]

```
const blogShema = new mongoose.Schema({
  naslov: String,
  tekst: { type: Array, required: true },
  autor: { type: String, maxlength: 255 },
  datumPublikacije: Date,
  objavljen: Boolean,
});
```

Primjer koda iznad prikazuje definiranje *mongoDB* sheme dokumenta, gdje sve vrijednosti koje se spremaju imaju svoj definirani tip podataka. Tip podatka može biti definiran samo s tipom podatka, kao što je *naslov* ili *objavljen* te se može postaviti kao objekt, a unutar tog objekta definira se tip (eng. type) i dodatne vrijednosti koje dodaju neko ograničenje, primjer za *tekst* gdje je ključ *required* postavljen na *true,* što znači da *tekst* ne smije biti prazna vrijednost.

Slika 8 ispod prikazuje popunjeni dokument s podatcima definiranim u *Mongoose* shemi iznad.



Slika 8. Prikaz kolekcije i sadržaj dokumenta u mongoDB bazi podataka (izvor: vlastita izrada)

3.5.2.Mongoose model i upiti nad dokumentima

Pomoću definirane sheme dalje se stvara ***Mongoose model***. *Modeli (eng. models)* su tvornički konstruktori sastavljeni od *shema* (eng. schema), a objekt koji vraća model naziva se dokument i on predstavlja dokument koji se sprema u mongoDB bazu podataka [28]. Taj model služi za interakciju s bazom, za stvaranje, čitanje, ažuriranje i brisanje (CRUD) dokumenata.

Stvaranje *Mongoose* modela pomoću *Mongoose* sheme iz prijašnjeg poglavlja:

```
const BlogModel = mongoose.model("blog", blogShema);
```

Popunjavanje modula podacima pomoću JavaScript objekta:

```
const Blog = new BlogModel({
  naslov: "Blog naslov",
  tekst: ["tekst1", "tekst2"],
});
```

Spremanje dokumenta odvija se putem metode ***save()***, to je asinkrona operacija jer spremanje dokumenata zahtjeva neko određeno vrijeme. Ako se funkciji koja sprema dokument doda ključna riječ async, može se sačekati rezultat pomoću *await* ključne riječi nad metodom *save()*. Spremanje se odvija nad novim objektom *Blog* od *BlogModel* modela.

```
async function spremiPodatke(){
   const rezultat = await Blog.save();
};
```

Upiti nad dokumentima, tj. za pronalazak nekog određenog dokumenta iz baze podataka odvija se putem metode **find()** i nekolicine drugih metoda kao što su *findById() ili findOne() i dr*.

```
async function dohvatiPodatke() {
  const blogPodatci = await BlogModel.find();
};
```

Prazna metoda *find()* pronalazi i dohvaća sve blog postove koji su spremljeni u kolekciji. Kao prvi argument može se proslijediti objekt sa svojim ključ-vrijednost parovima kao filter. U primjeru koda ispod, *find()* metoda pronalazi i vraća dokument s naslovom *„Blog naslov 1“*.

```
const blogPodatci = await BlogModel.find({ naslov: "Blog naslov 1" });
```

Taj upit također se može proširiti s više opcija/ograničenja s kojima se dohvaćaju dokumenti.

Sortiranje dokumenata po nekoj vrijednosti provodi se pomoću metode ***sort().*** Pri čemu je argument *sort()* metode objekt s ključ-vrijednost parom ili parovima.

```
const blogPodatci = await BlogModel.find().sort({datumPublikacije: 1});
```

Svi blog postovi sada su sortirani po datumu publikacije, pri čemu broj 1 predstavlja uzlazno sortiranje, a -1 silazno sortiranje.

Dohvat samo određenih podataka iz dokumenta radi se pomoću metode ***select().*** Ta metoda također prima objekt s ključ-vrijednost parovima koji predstavljaju one vrijednosti koje upit vraća.

```
const blogPodatci = await BlogModel.find()
    .sort({datumPublikacije: -1})
    .select({ naslov: 1, datumPublikacije: 1 });
```

Primjer koda iznad prikazuje metodu *select()* koja iz silazno sortiranih dokumenata vadi samo naslov i datum publikacije, pri čemu se ključu dodaje vrijednost 1 koja naznačuje da se ta vrijednost dohvaća iz dokumenta.

Ažuriranje dokumenata radi se pomoću metoda findByIdAndUpdate(), findOneAndUpdate() i updateOne() za jedan dokument, a za više njih putem metode updateMany(). Kod metode **findByIdAndUpdate(),** prvi argument je ***id*** dokumenta, a drugi argument je objekt s onim ključ-vrijednost parovima koji se ažuriraju. U većini slučajeva id se dobiva od klijenta ili se traži iz baze ovisno o uvjetima zahtjeva i dinamički se dodjeljuje u upit.

```
const azuriraniPodatci = await BlogModel.findByIdAndUpdate(
    "613132f31f368057961247d2",
    { naslov: "Ažurirani naslov" }
);
```

Brisanje dokumenta radi se pomoću metoda *findByIdAndDelete(), findOneAndDelete () i deleteOne()* za jedan dokument, a za više njih putem metode *deleteMany().* Metoda **findByIdAndDelete()** kao argument prima id dokumenta, što je potrebno da se dokument obriše iz kolekcije.

```
const obrisaniPodatci = await BlogModel.findByIdAndDelete(
    "613132f31f368057961247d2"
);
```

3.6. Json Web Token (JWT)

JSON Web Token (JWT) je otvoreni standard (EFC 7519) koji definira kompaktni i samostalan način za sigurnosni transport informacija između stranaka kao što je JSON objekt. Ta informacija može se potvrditi i može joj se vjerovati jer je digitalno potpisana. JWT može biti potpisan koristeći tajnu (eng. secret) (s HMAC algoritmom) ili s javnim/privatnim parom ključeva koristeći RSA ili ECDSA [23]

Iako se JWT mogu kriptirati kako bi se osigurala tajnost između stranaka, pozornost je usmjerena na potpisane tokene. Potpisani tokeni mogu provjeriti *integritet* tvrdnji (eng. claims) sadržanih u njemu, dok kriptirani tokeni skrivaju te zahtjeve od drugih stranaka. Kada su tokeni potpisani pomoću parova javnih/privatnih ključeva, potpis također potvrđuje da je to strana koja drži privatni ključ i da ga je ta strana potpisala.[23]

Najčešći scenariji u kojima se koristi JWT su: autorizacija korisnika i razmjena informacija.

3.6.1. Struktura JWT

U svom kompaktnom obliku, JSON web tokeni sastoje se od tri dijela odvojena točkama (.), a to su [23]:

- Zaglavlje (eng. header)
- Opterećenja (eng. payload)
- Potpis (eng. signature)

Gdje JWT tipično izgleda: ***xxxxx.yyyyy.zzzzz***

- **Zaglavlje**

Zaglavlje se obično sastoji od dva dijela: vrste tokena, koji je JWT, i algoritma za potpisivanje koji se koristi, poput HMAC SHA256 ili RSA.[23]

- **Opterećenje**

Drugi dio tokena je *opterećenje* koji sadrži tvrdnje (eng. claims). Tvrdnje su izjave o entitetu (obično korisnik) i dodatni podaci. Postoje tri vrste potraživanja: registrirane, javne i privatne tvrdnje.[23]

- **Potpis**

Potpis se stvara pomoću šifriranog zaglavlja, šifriranog (eng. encrypted) opterećenja i tajne, pomoću definiranog algoritma navedenog u zaglavlju.[23]

Primjer pomoću HMAC SHA256 algoritma:

```
HMACSHA256(base64UrlEncode(zaglavlje) + "." +base64UrlEncode(opterećenje),
tajna)
```

4. Razvoj na strani klijenta

Pod ***klijentom (eng. client)*** smatra se platforma koju koristi korisnik nekog uređaja, a to može biti osobno računalo, mobitel, frižider ili bilo koji pametni uređaj, u suštini sve s čime korisnik upravlja. U razradi ovog poglavlja pod klijentom se smatra web preglednik, gdje su osnovne tehnologije u razvoju aplikacije za web preglednik: ***html, css i JavaScript***.

Kroz poglavlja koja slijede obrađeni su osnovni koncepti ***React*** razvojnog okvira, što su to komponente i kako koristiti njihova stanja i životni ciklus. Ukratko je objašnjen ***Gatsby*** razvojnog okvira koji je građen nad *React-om.* Nadalje, razrađeni su osnovni principi ***react-spring*** biblioteke za animaciju DOM elemenata i ***postCss*** alat za transformaciju CSS sintakse pomoću JavaScript-a.

4.1. React JS

React je razvojni okvir razvijen od strane Facebook korporacije za gradnju aplikacija na jednoj stranici, (eng. SAP – single page app). Temelji je na ideji komponenata, izoliranoj cjelini aplikacije, njenom stanju i životnom ciklusu.

Manipulacija DOM-a u web pregledniku je jako spora, a kako bi ta manipulacija bila što brža, *React* koristi svoj virtualni DOM u pozadini, koji je jako brz u usporedbi s tradicionalnom manipulacijom DOM-a. *React* uspoređuje promjene u stvarnom DOM-u i virtualnom DOM-u i radi samo one promjene koje zapravo treba mijenjati.[24]

4.1.1. Uvod u JSX

JSX predstavlja sintaktičku ekstenziju za JavaScript, stvara *React* „elemente“ s punom snagom JavaScript-a.[22].

Drugim riječima, JSX je jezik koji omogućuje lagano miješanje JavaScript jezika i elemente slične HTML-u kako bi se definirali elementi korisničkog sučelja i njihova funkcionalnost.[24]

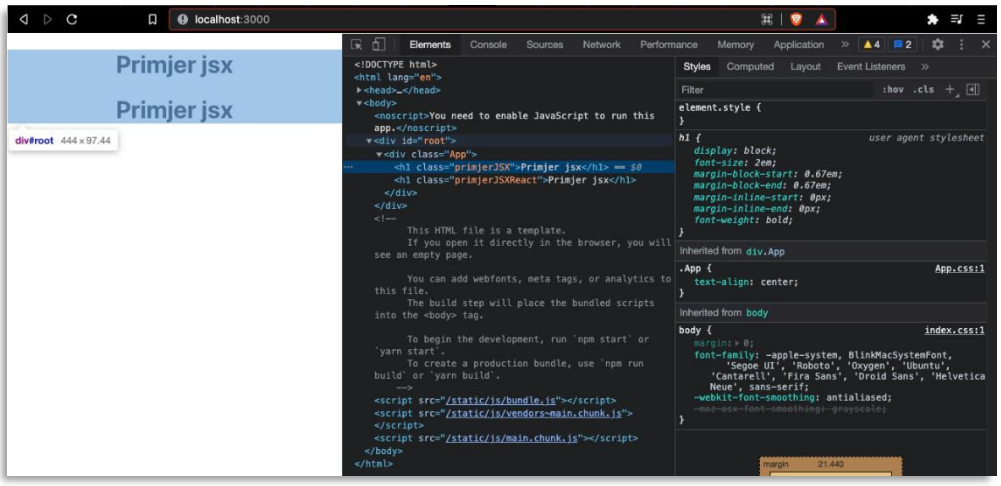
Postoje dva načina kako pretvoriti JSX u JavaScript, jer web preglednik ne prepoznaje JSX sintaksu, nego samo JavaScript i html:

1. Osposobiti razvojno okruženje pomoću *Node*-a i alata koji konvertiraju JSX u JavaScript, nešto kao create-react-app ili *Gatsby*.
2. Prepustiti konverziju JSX u JavaScript web pregledniku za vrijeme izvođenja aplikacije.[24]

Primjer koda ispod pokazuje funkcijsku komponentu s JSX sintaksom, gdje se h1 element pridružuje varijabli te funkcija komponenta vraća JSX vrijednosti unutar vitičastih zagrada:

```
const PrimjerJSX = (props) => {
  const primjerJSX = <h1 className='primjerJSX'>Primjer jsx</h1>;
  const primjerJSX1 = React.createElement(
    "h1",
    { className: "primjerJSXReact" },
    "Primjer jsx"
  );
  return (
    <>
      {primjerJSX}
      {primjerJSX1}
    </>
  );
};
```

JSX je zapravo samo sintaktički šećer za *React.createElement* metodu, Slika 9 ispod prikazuje kod iznad, pretvoren u html za web preglednik.



Slika 9. Primjer JSX elemenata u web pregledniku (izvor: vlastita izrada)

*U primjeru koda iznad prikazana je prazna JSX oznaka: **<></>**, tj. prazna oznaka slična html-u za početak i kraj elementa. Ona predstavlja ***React.Fregment*** komponentu koja daje mogućnost pisanja više JSX elemenata bez da se generira dodatni html element oko proslijeđenih elemenata.

JSX omogućuje lagano manipuliranje dinamičnim sadržajem, u primjeru koda ispod prikazan je takav dinamičan primjer, gdje se iz JavaScript objekta generira JSX element.

```
const podatci = {
  naslov: "Naslov h2 jsx",
  podnaslov: (podnaslov) => {
    return <h3>Naslov {podnaslov} jsx</h3>;
  }};
const naslov = <h2>{podatci.naslov}</h2>;
const podnaslov = podatci.podnaslov("h3");
```

4.1.1.1. Atributi i događaji

Dodjeljivanje atributa ili događaja u JSX sintaksi unutar *React* DOM-a zapisuje se putem *camelCase* načina imenovanja.[22]

Primjer atributa u html-u za klasu:

```
<div class="klasa"></div>
```

Sintaksa JSX-a za html atribut *class* je *className*:

```
<div className="klasa" />
```

Isti način imenovanja *camelCase* dodjeljuje se događajima, primjer *na klik (eng. onclick)* elementa u html-u:

```
<button onclick="pozoviFunkciju()">X</button>
```

Te sintaksa JSX-a *na klik* događaj:

```
<button onClick={pozoviFunkciju}>X</button>
```

Za obradu događaja u *React*-u prosljeđuje se funkcija koja obrađuje zadanu operaciju, može se proslijediti samo referenca funkcije ili se može direktno napisati funkcija sa strjelicom:

```
<button onClick={(e) => console.log(e, "Primjer")}>X</button>
```

4.1.2. Komponente i proslijeđene vrijednosti

React je sagrađen na ideji komponenata (eng. components). Cijela aplikacija koja se gradi zapravo će biti skup komponenata, gdje komponenta predstavljaju jednu izoliranu cjelinu aplikacije/web stranice.

Komponente su poput JavaScript funkcija, prihvaćaju proizvoljne unose (tzv „props“) i vraćaju *React* elemente koji opisuju ono što bi se trebalo pojaviti na ekranu.[10]

Slika 10 prikazuje početnu web stranicu www.lo3go.hr u kojoj su označene komponente brojkama – 1 i 2 su komponente zaglavlja, a od 3 do 6 su komponente unutar sekcije o nama.

Slika 10. Označene React komponente na web stranici www.lo3go.hr (izvor: vlastita izrada)

Komponente se mogu pisati pomoću **funkcija** i pomoću **klasa.** Pri čemu je najjednostavniji primjer komponente obična JavaScript funkcija koja prima vrijednosti, kao jedan objekt ***props.***

Funkcijske komponente mogu se napisati na dva načina, tradicionalni način pisanja funkcija:

```
function Osoba(props) {
  return <p>Moje ime je {props.ime}!</p>;
};
```

I putem funkcija sa strjelicom:

```
const Osoba = (props) => {
  return <p> Moje ime je {props.ime}!</p>;
};
```

Primjer komponente pisane putem klase:

```
class Osoba extends React.Component {
  constructor(props) {
    super(props);
  }
  render() {
    return <p> Moje ime je {this.props.ime}!</p>;
  }
}
export default Osoba;
```

Proslijeđene vrijednosti ***(props)*** predstavljaju vrijednosti koje komponente primaju pri inicijalizaciji, mogu se dinamički dodijeliti i jedna komponenta može se iskoristiti za prikazivanje različitih vrijednosti.

Slično kao i u primjeru JXS, komponenta se može pridružiti varijabli:

```
const element = <Osoba ime="Ivan" />
```

Ili, što je češći slučaj, koristi se ES6 izvoz (eng. export) i uvoz (eng. import) modula (https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/export).

Komponenta se koristi direktno u drugoj datoteci:

```
Import Osoba from "./komponente/Osoba".
<Osoba ime="Petar" />
```

U primjeru koda iznad pod vrijednosti *(props)* predstavlja ***ime*** do kojeg se može doći putem ***props objekta***, a može se proslijediti i više vrijednosti:

```
<Osoba ime="Petar" prezime="Perić" />
```

Gdje se do vrijednosti *prezime* dolazi pomoću: *props.prezime.*

Treba obratiti pozornost na način pisanja komponenti, gdje se *React* komponenta piše s velikim početnim slovom, jer *React* prepoznaje elemente pisane malim slovima kao html elemente, a one s velikim slovom kao *React* komponente.[10]

4.1.3. Stanje i životni ciklus komponente

Stanje (eng. state) je predefinirana varijabla unutar *React* komponente slična *props* objektu koja predstavlja vrijednost ili vrijednosti koje služe za dinamičko mijenjanje ili teksta ili DOM elemenata.[40] Stanje je jedan od okidača koje pokreću životni ciklus komponente. Pri čemu su komponente izolirane cjeline aplikacije i pomoću životnog ciklusa i stanja, dinamički se mijenja samo ta komponenta koja predstavlja jedan dio aplikacije.

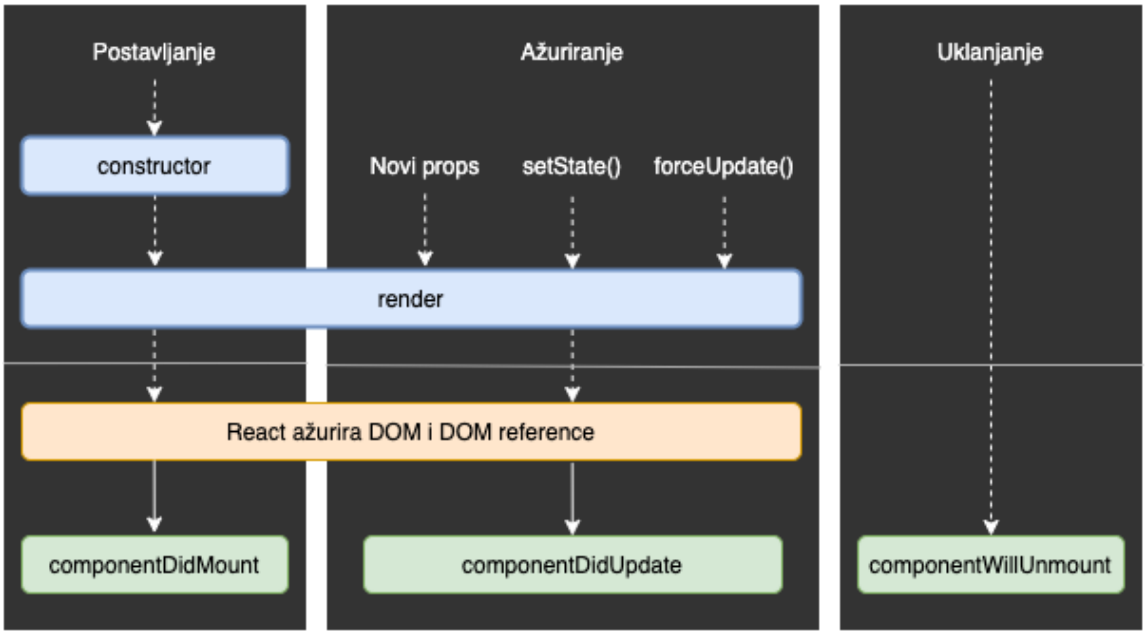
Primjer lokalnog stanja prikazano je ispod u primjeru *React* komponente pisane putem klase. Stanje u komponenti pisanoj pomoću klase deklarira se unutar constructor() metode: ***this.state*** u obliku JavaScript objekta.

```
class StanjeKomponenta extends React.Component {
  constructor(props) {
    super(props);
      this.state = { broj: 0 };
  }
}
```

Promjena stanja unutar komponente pisane pomoću klase odvija se pomoću metode ***setState*** koja kao argument može primiti funkciju ili se direktno može proslijediti objekt.

```
this.setState((state) => ({
   broj: state.broj + 1,
}));
this.setState({broj: this.state.broj + 1});
```

Svaka komponenta prolazi kroz svoj životni ciklus te za svaki ciklus postoji nekoliko predefiniranih metoda u *React* klasnoj komponenti pomoću kojih se može izvršavati kod u određenom trenutku ciklusa. Životni ciklus komponente je: ***postavljanje > ažuriranje > uklanjanje* (eng. *mounting > updating > unmounting*).**



Slika 11. Dijagram životnog ciklusa komponente i najkorištenije metode (izvor: https://projects.wojtekmaj.pl/react-lifecycle-methods-diagram/, pristupljeno: 2. 8. 2021.)

Postavljanje (eng. mounting) predstavlja početak inicijalizacije komponente, inicijalizira se stanje i proslijeđene vrijednosti (*props*) komponente unutar ***constructor*** metode. ***Render*** metoda jedina je metoda koja se mora postaviti unutar komponente, ona mora vraćati nekakvu vrijednost, što je najčešće JSX. *Render* metoda može obavljati logiku prije nego što se ažurira DOM i nije preporučeno postavljati stanje unutar te metode jer najčešće dolazi do pogrešaka. ***ComponentDidMount()*** metoda poziva se odmah nakon što komponenta generira elemente u DOM, a služi za dohvat reference DOM elemenata ili dohvat podataka s poslužitelja.[40]

Ažuriranje (eng. updating) predstavlja dio ciklusa kada je komponenta aktivna u DOM-u. Sve što ponovno pokreće *render* metodu su *proslijeđene vrijednosti (props)*, *postavljanje novog stanje putem setState()* metode i *prisilno ažuriranje stanje putem* ***forceUpdate()*** metode. Odmah nakon što se ažuriranje stanja poziva se metoda ***componentDidUpdate().*** Služi za manipulaciju DOM-a nakon što se dogode promjene, primjerice animacije i tranzicije elemenata.[40]

Uklanjanje (eng. unmounting) predstavlja kraj životnog vijeka komponente i prije nego što se komponenta uništi, tj. makne iz DOM-a, moguće je pozvati metodu ***componentWillUnmount()*** koja služi za čišćenje svih potrebnih operacija te komponente.[40]

4.1.4.React metode koje se mogu nadjačavati

Prije izlaska *React* verzije 16.8, stanje se moglo definirati samo u komponentama pisanim pomoću ES6 klasa. ***React* metode koje se mogu nadjačati (eng. React Hooks)** su funkcije koje su od verzije 16.8 omogućile funkcijskim komponentama da poprime stanje i životni ciklus.[21]

Treba obratiti pozornost samo na par pravila kada se koriste *React* metode koje se mogu nadjačati.

- *Metode trebaju biti samo pisane na početku komponente, prije korištenja*
- *Ne pozivati metode u petljama, uvjetima ili ugniježdenim funkcijama*
- *Metode trebaju biti samo korištene unutar funkcijskih komponenti*[21]

Najkorištenije *metode*: *useState, useEffect i useContext*, ostale metode: *useReducer, useCallback, useMemo, useRef, useImerativeHandle, useLayoutEffect, useDebugValue*.[20]

U nastavku su obrađene metode: ***useState*** koja predstavlja *stanje* komponente i ***useEffect*** pomoću koje se može reprezentirati najkorištenije metode *životnog ciklusa* komponente.

```
Import React, {useState, useEffect} from 'react';
```

4.1.4.1. useState

UseState metoda omogućuje funkcijskoj komponenti da poprimi svoje lokalno stanje. Na početku funkcijske komponente pomoću destrukturiranja niza definiramo stanje:

```
const [broj, postaviBroj] = useState(0);
```

Broj predstavlja vrijednost stanja, a inicijalna vrijednost u ovom slučaju je nula, koja se prosljeđuje kao argument u *useState* metodi. A *postaviBroj* je funkcija koja mijenja to stanje:

```
postaviBroj(broj + 1);
```

Primjer funkcijske komponente koja povećava broj na klik:

```
const UseState = () => {
  const [broj, postaviBroj] = useState(0);
  return (
    <>
      <h1>useState</h1>
      <h2>{broj}</h2>
      <button onClick={() => postaviBroj(broj + 1)}>Povećaj broj</button>
    </>
```

```
    );
};
```

Za definiranje više stanja unutar jedne funkcijske komponente potrebno je pozvati *useState* metoda s drugim nazivom stanja i funkcije.

4.1.4.2. useEffect

Pomoću *useEffect* metoda moguće je reprezentirati najkorištenije metode iz životnog ciklusa komponente pisane putem klase. *React useEffect* metoda prima dva argumenta, prvi argument je funkcija pomoću koje se izvršavaju željene operacije, a drugi argument je niz koji je opcionalan, pomoću kojega se dodjeljuju željene operacije iz životnog ciklusa.

Unutar funkcijske komponente napiše se useEffect funkcija:

```
const primjerKuke = () => {
  useEffect(() => {...}, []);
}
```

Ako se ***ne proslijedi niz,*** *useEffect* metoda pokreće se svaki puta kada se elementi prikažu na DOM-u, poslije svake *render* metode u životnom ciklusu komponente [19].

```
useEffect(() => {...})
```

Ako se ***proslijedi prazan niz,*** useEffect metoda izvrši se samo jednom kada se komponenta prikaže u DOM-u. Isto kao i *componentDidMount()* metoda iz životnog ciklusa komponente.

```
useEffect(() => {...}, [])
```

Ako se ***proslijedi vrijednost unutar niza***, *useEffect* metoda pokreće se jedino onda ako se proslijeđena vrijednost promijenila.[19] Primjer iz prošle cjeline, gdje se klikom na gumb povećava broj, *useEffect* metoda zove se nakon što se stanje komponente promijenilo. Slično je metodi *componentWillUpdate()* iz životnog ciklusa komponente.

```
useEffect(() => {
    ...
}, [broj]);
```

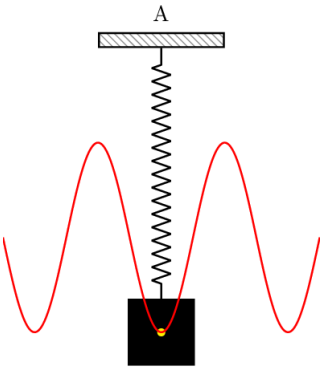
Ako se želi koristiti metoda ***componentWillUnmount()***, unutar *useEffect* metoda napiše se ***return*** ključna riječ koja vraća funkciju te ta funkcija se koristi za željene operacije kada se komponenta makne iz DOM-a.

```
useEffect(() => {
    ...
    return () => {...}
}, [broj])
```

Isto kao i *useState* funkcija, *useEffect* može se više puta napisati unutar jedne komponente.

4.1.5. React-spring

React-spring animacijska je biblioteka bazirana na fizici opruge i predstavlja moderni pristup animaciji. Inspirirana je od biblioteka *animated* i *react-motion.* Od animated biblioteke nasljeđuje moćnu interpolaciju i performansu, a od react-motion jednostavnost uporabe.[37] Princip s kojim se radi je tzv. opruga (eng. spring), ona nema definiranu krivulju ili zadano trajanje. Kada se misli na animacije, gleda se u kontekstu vremena i krivulja, ali to samo po sebi uzrokuje probleme kada se želi postići prirodno micanje elemenata na ekranu jer se ništa u stvarnom svijetu ne kreće na taj način.[37]



Slika 12. Vizualna reprezentacija principa rada react-spring biblioteke
(izvor: https://react-spring.io/, pristupljeno: 5. 8. 2021.)

Kroz praktični dio rada korištene su dva tipa opruga (eng. springs), osnovana opruga ili funkcija useSpring za animaciju elemenata pri pojavi na ekranu te useTrail za animaciju više elemenata u lanac.

Premda biblioteka ima svoje metode koje se mogu nadjačati (eng. hooks), a to su **useSpring**, useSprings, useChain, useTransition i **useTrail**, slično kao i *React* metode koje se mogu nadjačati, *React* komponente kroz projekt pisane su pomoću funkcija i elementi su animirani pomoću react-spring metoda.

Način na koji se animacije manipuliraju je s konfiguracijom same opruge, po njenoj masi, napetosti, trenju i dr.[36]

Dohvat biblioteke:

```
import { useSpring, useTrail, animated } from 'react-spring'
```

animated – biblioteka koja radi animacije van *React*-a zbog performanse. Proširuje DOM elemente koje primaju vrijednosti za animaciju.[35]

4.1.5.1. useSpring

UseSpring funkcija pretvara vrijednosti u animirane vrijednosti.[38] Primjer koda ispod prikazuje uporabu useSpring funkcije koja pri inicijalizaciji *React* komponente pokreće animaciju i mijenja css atribut *opacity* od 0 prema 1.

```
const Opruga = () => {
  const stil = useSpring({ to: { opacity: 1 }, from: { opacity: 0 } });
  return <animated.h1 style={stil}>Primjer fade-in animacije</animated.h1>;
};
```

Animacije se također mogu mijenjati pomoću nekog oblika okidača, klikom na gumb ili kada se element pojavi na ekranu pomoću stanja u *React*-u.

```
const [okidac, postaviOkidac] = useState(false);
const stil = useSpring({ opacity: okidac ? 0 : 1 });
```

4.1.5.2. useTrail

useTrail() funkcija stvara više opruga (eng. springs) s jednom konfiguracijom, svaka opruga slijedi prijašnju.[39]

Primjer koda ispod prikazuje uporabu *useTrail()* funkcije koja pri inicijalizaciji *React* komponente pokreće animaciju za svaki element u nizu, jedan za drugim, od nultog elementa prema n-tom elementu.

```
const Put = () => {
  const elementi = ["jedan", "dva", "tri", "...", "koraka"];
  const trail = useTrail(elementi.length, {
    to: { opacity: 1 },
    from: { opacity: 0 },
  });
  return trail.map((styles, index) => (
    <animated.h1 style={styles}>{elementi[index]}</animated.h1>
  ));
};
```

4.2. Gatsby JS

Gatsby je razvojni okvir otvorenog koda baziran na *React*-u za stvaranje web stranica i aplikacija.[18]

Gatsby alat za komandnu liniju (CLI – eng. command line interface) početna je točka izrade *Gatsby* aplikacija, također za pokretanja razvojnog okruženja i gradnju *Gatsby* aplikacija za produkcijsko okruženje.[9]

Instalacija Gatsby CLI-a dostupna je preko npm-a, gdje se instalira globalno pomoću npm-a:

```
npm install -g gatsby-cli
```

Inicijalizacija novog projekta pomoću ključne riječi ***new***:

```
gatsby new
```

Pokretanje razvojnog okruženja pomoću ključne riječi ***develop***:

```
gatsby develop
```

Gradnja (eng. build) aplikacije za produkcijsko okruženje pomoću ključne riječi ***build***:

```
gatsby build
```

Pokretanje produkcijskog okruženja pomoću ključne riječi ***serve***:

```
gatsby serve
```

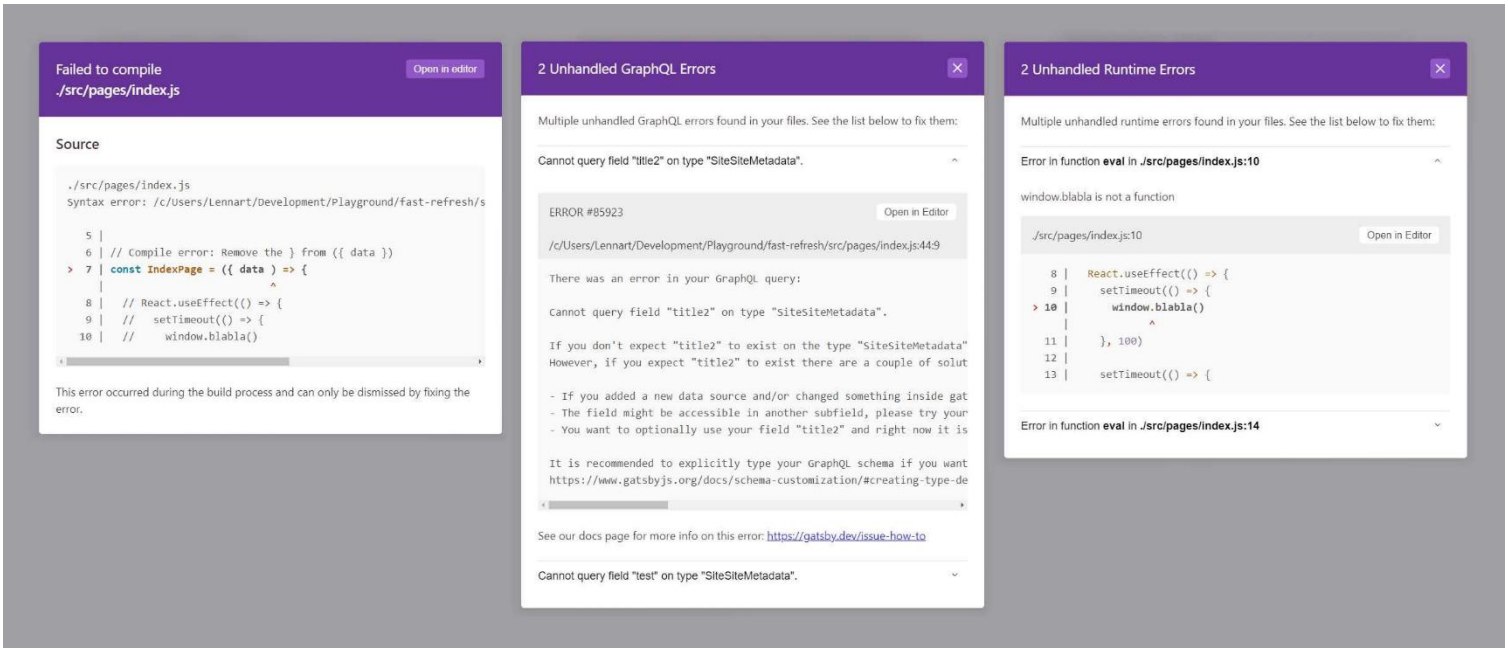
Nakon instalacije novog projekta mogu se vidjeti svi ili neki direktoriji i datoteke: [17]

|-- /.cache
|-- /plugins
|-- /public
|-- /src
  |-- /api
  |-- /pages
  |-- /templates
  |-- html.js
|-- /static
|-- gatsby-config.js
|-- gatsby-node.js
|-- gatsby-ssr.js
|-- gatsby-browser.js

Gatsby programerima nudi mogućnost prilagodbe razvojnog okruženja s njima preferiranim JavaScript alatima, mogućnost konfiguracije Typescipta-a, Webpack-a, ESLint-a, i Prettier-a.[26] Također, razvojno okruženje u *Gatsby*-u ima brzo osvježavanje (eng. fast

refresh) i pri spremanju neke datoteke unutar *Gatsby* aplikacije promjene se dogode unutar jedne sekunde u web pregledniku, bez da *React* komponenta gubi svoje stanje.[15]

Gatsby za indikaciju pogrešaka u kodu nudi svoj prilagođeni sloj koji objašnjava tu pogrešku i kako je riješiti.[15]



Slika 13. *Gatsby* indikacija pogrešaka (izvor: https://www.gatsbyjs.com/docs/reference/local-development/fast-refresh/#error-resilience, pristupljeno: 20. 9. 2021.)

Jedan od razloga zašto su *Gatsby* web stranice tako brze je to što se dosta posla odvija za vrijeme gradnje aplikacije i stranica koja je pokrenuta je statična stranica poslužena pomoću CDN-a. Za vrijeme gradnje aplikacije, *Gatsby* stvara putanje pomoću kojih se dolazi do pojedinih stranica i omogućuje usmjeravanje (eng. routing)[42]

Definirane rute nalaze se u direktoriju: `./src/pages`, gdje *Gatsby* stvara stranice za svaku `.js` datoteku, npr. ruta u web pregledniku je:

```
https://imeDomene/imeDokumenta > https://www.lo3go.hr/pricelist
```

Navigacija između stranica odvije se pomoću *Gatsby Link* komponente:

```
<Link to="/pricelist" />
```

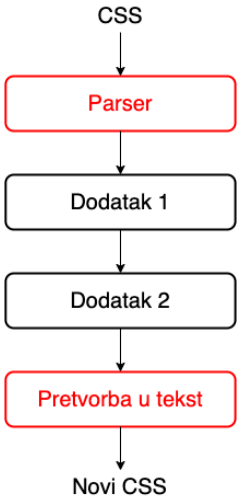
Gatsby ima svoj ugrađen podatkovni sloj koji se odvija pomoću GraphQL tehnologije, gdje se podaci prikupljaju za vrijeme gradnje aplikacije i automatski se dodaju u sheme (eng. schema) koje definiraju upite nad podacima kroz web stranicu. Kada se stranica izradi, pregled podatkovnog sloja može se istražiti na: http://localhost:8000/__graphql.[19]

Gatsby organizacija zajedno s zajednicom inženjera i kompanijama stvara veliki broj *Node* paketa, tj. dodataka (eng. plugins) za *Gatsby* razvojno okruženje koji se mogu uključiti u *Gatsby* aplikaciju.[16]

4.3. PostCss

PostCss je alat koji transformira CSS pomoću JavaScripta.[33] Dozvoljava da se definira prilagođena CSS sintaksa koja je razumljiva i može se transformirati pomoću dodataka (eng. plugins).[2]

Ključ PostCSS-a je u tome što se s njim postupa kao s omogućavateljem, PostCSS nije zamišljen kao izravna zamjena za postojeći predprocesor, pa čak ni za postprocesor, već za njihovu nadopunu. Radi na temelju raščlanjivanja CSS koda, obrađuje ga s dodijeljenim dodacima i prikazuje rezultat:[2]



Slika 14. Pregled rada PostCss-a na visokoj razini (izvor: https://github.com/postcss/postcss/blob/main/docs/architecture.md, preuzeto: 3. 8. 2021.)

Radi na način da analizira sadržaj (CSS) u stablo apstraktne sintakse (ili AST, eng. Abstract Syntax Tree) s nizom čvorova (eng. nodes). Svaki čvor u stablu sadrži simbolički prikaz elementa u kodu. Drugim riječima, za neki uvjet koji ima tri moguća ishoda AST bi imao jedan čvor s tri grane koje predstavljaju moguće ishode.[2]

Nakon toga AST prolazi kroz jedan ili više dodatak (eng. plugins) te pretvara kod u dugački tekstualni niz i obrađuje ga kroz neki dodatak te kao rezultat stvara valjani CSS.[2]

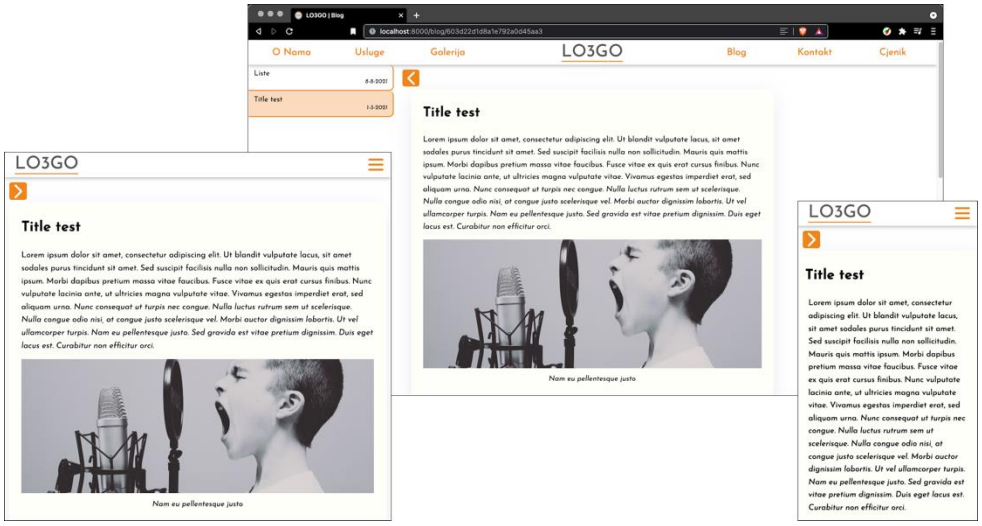
Sam po sebi PostCss ne radi puno stvari, premda je napravljen kao *Node* modul, cijeli ekosistem vrti se oko dodataka (eng. plugins), gdje se bira određeni dodatak koji je poželjan za projekt. Razlika u usporedbi sa stilskim pred-procesorom kao što je Sass ili Less je da se bira dodatak koji se želi priključiti u projekt, a ne treba se dodati cijela biblioteka.[2]

U praktičnom dijelu projekta korištena je nekolicina dodataka koji se mogu pronaći na npm web stranici:

- autoprefixer
- cssnano
- postcss-import
- postcss-nested
- postcss-preset-env
- postcss-pxtor

5. Razrada praktičnog dijela projekta

LO3GO je obrt za logopedske usluge baziran u Našicama te je praktični dio ovog rada fokusiran na web aplikaciji za upravljanjem sadržaja, a u kontekstu web stranice to je mogućnost pisanja, ažuriranje i brisanja bloga. Projekt se sastoji od dvije aplikacije, poslužitelj (eng. server) aplikacije i klijent (eng. client) aplikacije. Projekt s pregledavanjem blog sadržaja dostupan je na web adresi: www.lo3go.hr/blog, podržan je na svim modernim web preglednicima, s responzivnim web dizajnom:



Slika 15. Responzivni web dizajn LO3GO web stranice (izvor: vlastita izrada)

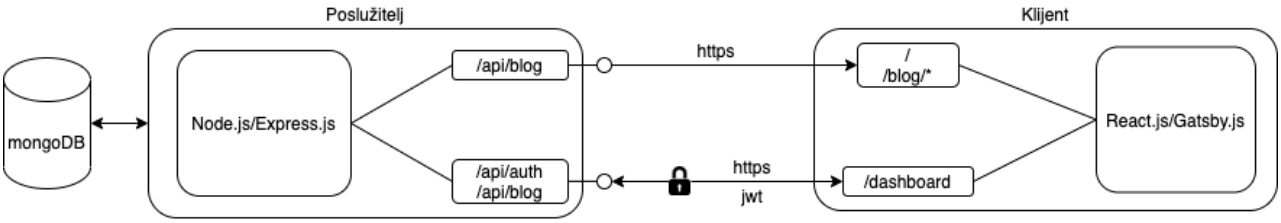
Kroz slijedeća poglavlja objašnjeni su osnovni koncepti web aplikacije za upravljanje sadržaja (eng. CMS – Content Management System). Objašnjena je perspektiva poslužitelja i klijenta, njihova arhitektura i struktura te kratak opis *Editor* biblioteka oko koje je građena aplikacija.

Za poslužitelja ukratko su objašnjene rute koje su dostupne, tj. krajnje točke poslužitelja s kojima klijentska aplikacija komunicira, proces provjere autentičnosti korisnika i odvajanje privatnih ruti od javnih pomoću JWT-a. Razvojno okruženje za poslužitelja je otvorenog koda i može se pristupiti na web adresi: https://github.com/Cind0/lo3go-api/tree/dev

Za klijent aplikaciju ukratko je objašnjen proces dohvata podatak s poslužitelja koristeći *fetch API* te kako obraditi te podatke za blog post pomoću *React* stanja, životnog ciklusa i JSX-a. Te sama integracija *Editor* biblioteke u *Gatsby* razvojni okvir.

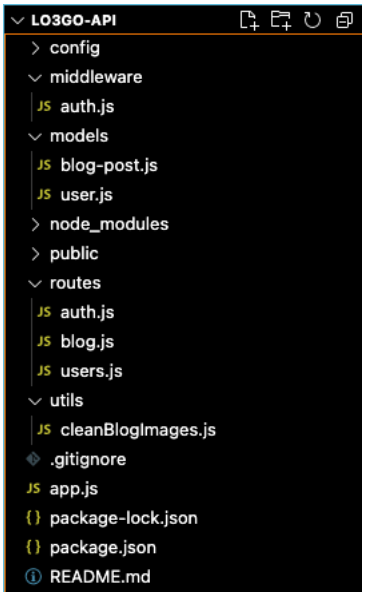
5.1. Opći pogled na strukturu i arhitekturu aplikacija

Slika 16. ispod prikazuje jednostavni dijagram arhitekture aplikacija. Na poslužitelju su odvojene privatne i javne rute, tj. krajnje točke, a komunikacija se odvija preko https protokola i privatne rute odvojene su pomoću posredničke funkcije koristeći JWT. Klijentska aplikacija na svojim rutama zove te krajnje točke poslužitelja i dinamički generira sadržaj.



Slika 16. Dijagram arhitekture aplikacija (izvor: vlastita izrada)

5.1.1. Struktura poslužitelj aplikacije



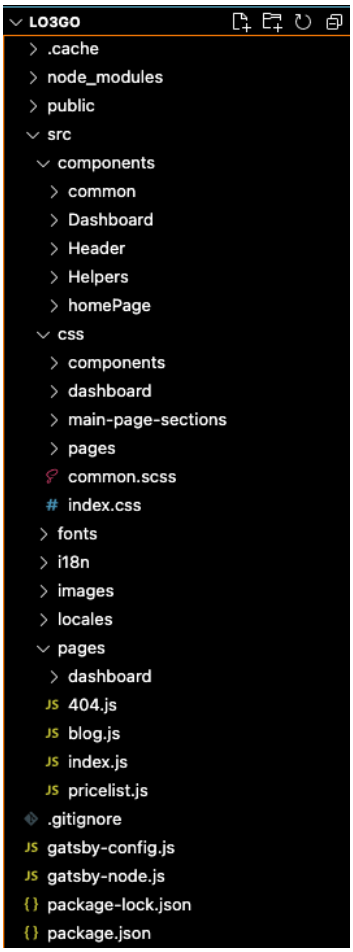
- Početna datoteka je ***app.js*** u koju se uključuju svi potrebni moduli i aplikacija se spaja na bazu podataka.
- Direktorij **models** sadrži datoteke koje predstavljaju *Mongoose* sheme za mongoDB dokumente.
- Direktorij **routs** sa svojim datotekama predstavlja odvojene rute na poslužitelju.
- Direktorij **config** sadrži JSON datoteku koja predstavlja konfiguraciju aplikacije.
- Direktorij **middleware** služi za odvajanje posredničkih funkcije/softver.
- Direktorij **public/images** služi za spremanje blog slika koje su dostupne preko url-a domene poslužitelja.

Slika 17. Struktura poslužiteljske aplikacije (izvor: vlastita izrada)

```
"dependencies": {
  "bcrypt": "^5.0.0",
  "config": "^3.3.2",
  "cors": "^2.8.5",
  "express": "^4.17.1",
  "joi": "^14.3.1",
  "joi-objectid": "^3.0.1",
  "jsonwebtoken": "^8.5.1",
  "lodash": "^4.17.20",
  "mongoose": "^5.11.15",
  "multer": "^1.4.2"
}
```

Slika 18. Dodaci i trenutne verzije dodataka poslužitelj aplikacije (izvor: vlastita izrada)

5.1.2. Struktura klijent aplikacije



- Direktorij **pages** služi za odvajanje pojedinih stranica na *Gatsby* aplikaciji i to su početne komponente za svaku pojedinu stranicu.
- Direktorij **src** sadrži sve ostale dijelove aplikacije: komponente, pomoćne funkcije, css, fontove, slike i prijevode.
- Direktorij **Dashboard** u *compnents* direktoriju sadržava sve ostale komponente i pomoćne funkcije potrebne za */dashboard* rutu pomoću koje korisnih piše i ažurira sadržaj blog posta.
- Css u aplikaciji se odvaja po svakoj stranici ili zasebnoj komponenti, index.css datoteka je ulazna točka za sve ostale odvojene css datoteke.
- **gatsby-confirg.js** datoteka predstavlja konfiguraciju *Gatsby* aplikacije s nekim osnovnim podacima o aplikaciji i konfiguracijom dodataka.

Slika 19. Struktura klijent aplikacije (izvor: vlastita izrada)

Slika 20. Dodaci i trenutne verzije dodataka klijent aplikacije (izvor: vlastita izrada)

5.2. Editor.js

Editor je uređivač teksta baziran na uređivanju blokova za bogate multimedijske priče. *Editor* vraća čiste podatke u obliku JSON-a umjesto teškog HTML-markupa. Najbitnije je da je *Editor* dizajniran na način da se može proširiti pomoću svojih API-a i s dodacima (eng. plugins).[5]

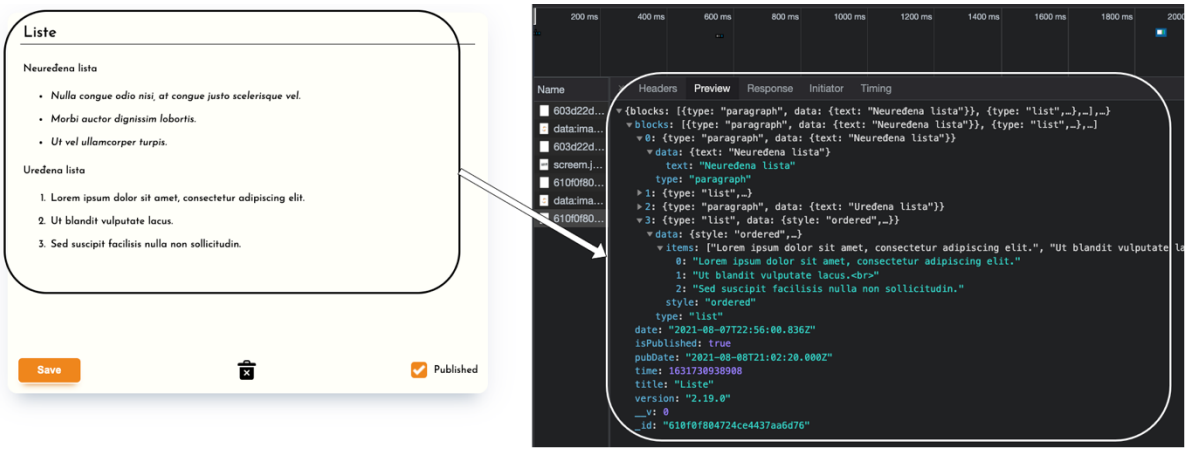
Neka glavna obilježja:

- Vraća čiste podatke
- Proširivo putem API-a
- Otvorenog koda [5]

Dodaci korišteni u projektu za *Editor*:

- "@editorjs/header": "^2.6.1"
- "@editorjs/image": "^2.6.0"
- "@editorjs/list": "^1.6.2"
- "@editorjs/quote": "^2.4.0"

Pri spremanju sadržaja pomoću *Editor*-a, uređivač vraća čiste podatke u obliku JSON-a, gdje se glavni sadržaj uređenog teksta nalazi pod ključem: ***blocks:[{…},{…}]*** koji je niz objekata. Pri čemu jedan objekt predstavlja blok pojedinog tipa podataka, neovisno je li to tekst, lista, naslov, slika ili citat.



Slika 21. Prikaz Editor.js sučelja i JOSN podataka blog posta (izvor: vlastita izrada)

5.3. Krajnje točke na strani poslužitelja

Rute predstavljaju krajnje točke na poslužitelju pomoću kojih klijent može komunicirati s poslužiteljem preko HTTP protokola.

Razlikuju se ***javne i privatne*** rute, gdje javnim rutama može pristupiti svatko, a privatnim samo onaj korisnik koji ima privilegije za to.

5.3.1. Blog - `/api/blog`

Ruta *blog* predstavlja krajnju točku pomoću koje klijentska aplikacija pregledava, stvara, briše i ažurira sadržaj bloga.

Na poslužitelju to predstavlja posredničku funkciju ili usmjerivač (eng. router):

```
app.use("/api/blog", blog);
```

Gdje je puna ruta, za npr. domenu www.lo3go-api.hr:

www.lo3go-api.hr**/api/blog/***all-published

Definiranje i validacija *Mongoose* sheme i modela blog posta:

- https://github.com/Cind0/lo3go-api/blob/dev/models/blog-post.js

Rute i upiti na mongoDB bazu podataka za blog:

https://github.com/Cind0/lo3go-api/blob/dev/routes/blog.js

5.3.1.1. Javne

- `GET` - **`/all-published`**
  - Krajnja točka dohvaća **sve objavljenje blog postove.**
- `GET` - **`/latest`**
  - Krajnja točka dohvaća **posljednji objavljeni blog post**.
- `GET` - **`/find/:id`**
  - Krajnja točka dohvaća **određeni objavljeni blog post**.

5.3.1.2. Privatne

- `GET` - **`/all`**
  - Krajnja točka dohvaća **sve blog postove.**
- `GET` - **`/toedit/:id`**
  - Krajnja točka dohvaća **određeni blog post.**

- `POST` - **`/`**
  - Krajnja točka **sprema novi blog post**.

- `PUT - /:id`
  - Krajnja točka **ažurira određeni blog post.**
- `DELETE - /:id`
  - Krajnja točka **briše određeni blog post.**
- `POST - /uploadImg`
  - Krajnja točka **sprema dodanu sliku blog posta na poslužitelja.**

5.3.2. Korisnici - `/api/users`

Ruta *korisnici* služi za dohvat korisnikovih podataka i registraciju. Obje rute su privatne, može im pristupiti samo korisnik s određenim privilegijama.

U produkcijskom okruženju ***rute će biti onemogućene*** jer potreban je samo jedan korisnik koji će pisati i ažurirati sadržaj bloga.

Na poslužitelju to predstavlja posredničku funkciju ili usmjerivač (eng. router):

```
app.use("/api/users", users);
```

Gdje je puna ruta, za npr. domenu www.lo3go-api.hr:

www.lo3go-api.hr**/api/users/***me

Definiranje i validacija *Mongoose* sheme i modela korisnika:

- https://github.com/Cind0/lo3go-api/blob/dev/models/user.js

Rute za korisnika:

- https://github.com/Cind0/lo3go-api/blob/dev/routes/users.js

- `GET - /me`
  - Krajnja točka **dohvaća podataka o korisniku** koji je prijavljen u sustav.
- `POST - /`
  - Krajnja točka služi za **registraciju novog korisnika**.

Pri registraciji, zaporka (eng. password) se šifrira pomoću *Node* biblioteke ***bcrypt.*** (https://www.npmjs.com/package/bcrypt) Tako da se zaporka ne zapisuje u čistom tekstualnom obliku u bazu podataka, nego u šifriranom, u slučaju da podaci dođu u neželjene ruke. Zaporka **12345** pretvara se u:

$2b$10$xk6HoXSvqYXHAKQe6UoxKOmR/iI7yPDmeuo1NmOC1cgHgNMj8sls2

5.3.3. Provjera autentičnosti - `/api/auth`

Na poslužitelju to predstavlja posredničku funkciju ili usmjerivač (eng. router):

```
app.use("/api/auth", auth);
```

Gdje je puna ruta, za npr. domenu www.lo3go-api.hr:

www.lo3go-api.hr**/api/auth/**

Ruta za provjeru autentičnosti korisnika:

- https://github.com/Cind0/lo3go-api/blob/dev/routes/auth.js

- `POST - /`
  - Krajnja točka služi za ***provjeru autentičnosti korisnika***, tj. prijavu korisnika u sustav.
  - Koraci provjere autentičnosti funkcije:
    - Provjerava jesu li poslani podaci valjani.
    - Pronalazi korisnika pomoću email-a.
    - Provjerava je li zaporka valjana, pomoću *bcrypt* biblioteke uspoređuje se šifriranom zaporka s unesenom zaporkom.
    - Autorizacija korisnika, odbija se pristup sustavu ako korisnik nema administrativna prava.
    - Generira se JWT i šalje se klijentu.

5.4. Privatna rute pomoću JWT-a

Nakon autentikacije korisnika, tj. login korisnika u sustav, poslužitelj šalje JWT klijentu. Klijent u ovoj arhitekturi zadužen je za čuvanje JWT-a, a na JWT može se gledati kao ključ koji otvara vrata za zaštićene krajnje točke. Jedna od sigurnosnih zahtjeva da JWT bude zaštićen je da se komunikacija odvija pomoću *HTTPS* protokola.

JWT na klijentu čuva se u stanju *React* komponente, što znači kada se stranica osvježi stanje se vraća na inicijalno stanje i JWT se briše.

```
this.state = {
  logged_in: false,
  nav: "",
  jwt: "",
  action: {
    setJwt: (newJwt) => this.setState({ jwt: newJwt }),
    setNav: (newNav) => this.setState({ nav: newNav }),
    setLogged_in: (state) => this.setState({ logged_in: state })
  },
};
```

Kada se šalje zahtjev poslužitelju u zaglavlju HTTP protokola šalje se ključ-vrijednost par koji predstavlja JWT, `x-auth-token: "jwt".`

Privatne rute odvojene su pomoću prilagođene posredničke funkcije (eng. middleware) ***auth*** na poslužiteljskoj aplikaciji.

Primjer koda ispod prikazuje */api/blog/all* privatnu rugu s posredničkom funkcijom *auth* koja se izvršava prije *async* funkcije.

```
router.get("/all", auth, async (req, res) => {};
```

Kod ispod prikazuje ***auth.js*** posredničku funkciju:

- JWT se vadi iz zaglavlja zahtjeva.
- Ako JWT ne postoji, poslužitelj odgovara klijentu da je zabranjen pristup krajnjoj točki.
- Ako JWT postoji, uspoređuje se JWT s tajnom iz konfiguracije
- Ako je validan JWT, "raspakirani/dekodirani" objekt se dodaje u req objekt pod ključem *user* i proslijedi se kontrola slijedećoj funkciji.
- Ako JWT nije valjan, poslužitelj odgovara klijentu da JWT nije validan.

```
const jwt = require("jsonwebtoken");
const config = require("config");

module.exports = function auth(req, res, next) {
  const token = req.header("x-auth-token");
  if (!token) return res.status(401).send("Access denied. No token provided!");
  try {
    const decoded = jwt.verify(token, config.get("jwtPrivateKey"));
    req.user = decoded;
    next();
  } catch (error) {
    return res.status(400).send("Invalid token.");
  }};
```

Pri provjeri autentičnosti korisnika poziva se prilagođena metoda *generateAuthToken()* koja potpisuje podatke od korisnika zajedno s tajnom i generira se novi JWT:

```
userSchema.methods.generateAuthToken = function () {
  const token = jwt.sign(
    { _id: this._id, isAdmin: this.isAdmin },
    config.get("jwtPrivateKey")
  );
  return token;
};
```

5.5. Dohvat i slanje podataka na strani klijenta

Komunikacija s poslužiteljem obavlja se pomoću ***Fetch API-a***. *Fetch API* pruža JavaScript sučelje za pristup i manipulaciju dijelovima HTTP cjevovoda (eng. pipeline), kao što su zahtjevi (eng. requests) i odgovori (eng. responses). Također, pruža globalnu *fetch()* metodu koja daje jednostavni, logički način dohvata resursa asinkrono preko mreže. Takva funkcionalnost nekada se postizala koristeći *XMLHttpRequest(XHR)* u AJAX načinu programiranja.[47]

Primjer koda ispod prikazuje asinkronu funkciju koja je zadužena za dohvat pojedinog blog posta pomoću *fetch()* metode te su priloženi uvjeti koji služe za obradu odgovora s poslužitelja:

- *site.siteMetadata.apiHost* je domena poslužitelja koja se nalazi u *gatsby-config.js* datoteci te *fetch()* metoda prima krajnju točku poslužitelja kao argument.

```
const fetchBlogPost = async (id) => {
    try {
      const response = await fetch(
        `${site.siteMetadata.apiHost}blog/find/${id}`
      );
      if (!response.ok) {
        const { error } = await response.json();
        // Obrada pogreške
      } else {
        const data = await response.json();
        // Obrada podataka blog objave
      }
    } catch ({ response }) {
      if (!response) {
        // Poslužitelj nije pronađen
      } else {
        const { error } = response.data;
        // Obrada neočekivane pogreške
      }
    }};
```

Fetch() metoda samo s argumentom krajnje točke prema zadanim postavkama u zaglavlju HTTP protokola dodaje metodu GET. Kako bi se taj ključ-vrijednost par promijenio, kao drugi argument *fetch()* metode dodaje se objekt koji predstavlja konfiguraciju zahtjeva.

Primjer koda ispod prikazuje konfiguracijski objekt za fetch() metodu, korišten u zahtjevu za spremanje novog blog posta:

- Prvi ključ-vrijednost par je HTTP metoda POST, *može biti i PUT i DELETE u drugim HTTP zahtjevima.*
- Drugi ključ-vrijednost par je “headers“ koji predstavlja ulaznu točku za dodatne ključ-vrijednost parove u zaglavlju HTTP zahtjeva.
- Treći ključ-vrijednost par je ***tijelo*** zahtjeva s kojim se šalju podaci na krajnju točku poslužitelja.

```
const config = {
      method: "POST",
      headers: {
        "x-auth-token": jwt,
        "Content-Type": "application/json",
      },
      body: JSON.stringify(data),
};
```

Primjer koda ispod prikazuje uporabu *fetch()* metode s konfiguracijskim objektom za spremanje novog blog posta te je obrada tog zahtjeva slična kao i za GET metodu dohvata pojedinog blog posta:

```
const response = await fetch(`${site.siteMetadata.apiHost}blog`, config);
```

5.6. Obrada podataka na klijentu

Gotovo svaka komponenta u *Gatsby* aplikaciji je *funkcijska komponenta,* što znači da su stanje i životni ciklus obrađeni pomoću *React* metoda koje se mogu nadjačati (eng. React hooks) koristeći *useState()* i *useEffect()* funkcije.

Primjer korištenja *useEffect()* funkcije u *Gatsby* aplikaciji je pri navigaciji na blog stranicu. Gdje se *useEffect()* funkcija poziva svaki puta kada se promjeni aktivni blog id i kada se komponenta inicijalizira, a aktivni blog id čuva se u stanju pomoću *useState()* funkcije.

```
const [activeBlogId, setActiveBlogId] = useState(props.params["*"]);
```

```
useEffect(() => {
      fetchBlogData();
}, [activeBlogId]);
```

Pomoću *props.params["*"]* objekta u *Gatsby*-u vadi se iz URL-a id blog posta, sve što je na ruti poslije `/blog/`***,** a postavljanje dinamičke rute blog posta odvija se pomoću ugrađene funkcije u *Gatsby*-u: *navigate(`/blog$/{id}`).*

FetchBlogData() funkcija zadužena je za dinamički dohvat blog posta pri inicijalizaciji komponente, poziva krajnju točku */blog/all-published* i funkciju *fetchBlogPost(id)* .

Dobiveni podaci s poslužitelja dodjeljuju se u stanje komponente, pomoću kojih se dinamički generiraju html elementi kada se stanje promjeni. U stanje *blogData* spremaju se svi blog podaci u obliku niza objekata, a u stanje *blogDataErr* sprema se tekst s opisom pogreške dobivene sa strane poslužitelja.

```
const [blogData, setBlogData] = useState([]);
const [blogDataErr, setBlogDataErr] = useState("");
```

Svaki poziv krajnjih točaka poslužitelja traje neko određeno vrijeme, to vrijeme čekanja ili učitavanja na korisničkom sučelju reprezentira se u stanju komponente, kao zastavica (eng. flag). Inicijalno stanje zastavice je istina (eng. true), znači sve dok poslužitelj ne odgovori s uspjehom ili pogreškom, komponenta je u stanju učitavanja.

```
const [loadingBlogData, setLoadingBlogData] = useState(true);
```

Pomoću JSX-a i stanja komponente dinamički se mijenjaju html elementi komponente ili učitavaju druge komponente.

Primjer koda ispod prikazuje način kako obraditi stanje učitavanja pomoću ***ternarnog*** operatora i obradu blog podataka iteracijom po JSON-u dobivenom od poslužitelja:

```
return (
     <div className='blog-page'>
        {!loadingBlogData ? (
          blogData && blogData.length !== 0 ? (
            <div className='blog-content'>
               {blogData.blocks.map((data, index) => {
                    return (
                      <div className='blog_content_item' key={index}>
                         {parsBlogData(data)}
                      </div>
                    );})}
             </div>
            ) : (
              <div className='center_screen'>
                <h2>Nema objavljenih blog postova!</h2>
              </div>
            )
          ) : (
```

```
            <div className='center_screen'>
               <Loader />
            </div>
           )}
      </div>);
```

Uz *ternarnog* operator može se koristiti i JavaScript logički operator „i“ **(&&**) (eng. and) za dinamičko generiranje html elemenata, prikazano u primjeru koda ispod za obradu pogreške:

```
{blogDataErr.length !== 0 && <h2>{blogDataErr}</h2>}
```

5.6.1. Integracija Editor.js biblioteke u Gatsby

Ista *React* komponenta koristi se za ažuriranje i za pisanje novog blog posta. Na ruti */dashboard* poslije autentikacije korisnika poziva se funkcijska komponenta *new-blog.js* koja inicijalizira *Editor.js* komponentu, također poziva ju i *edit-blog.js* komponenta za uređivanje blog sadržaja.

Dodavanje *Editor* biblioteke i njenih dodataka u komponentu:

```
const EditorJS = require("@editorjs/editorjs");
const Header = require("@editorjs/header");
const ImageTool = require("@editorjs/image");
const Quote = require("@editorjs/quote");
const List = require("@editorjs/list");
```

Deklarira se prazna varijabla koja služi za spremanje novog *Editor* objekta.

```
let editor = null;
```

Za primjer koda ispod:

- Ključ *holder* s vrijednosti “blog-editor“ referira se na html div element s id=“ blog-editor“ u kojega se inicijalizira *Editor* uređivač.
- Ključ *tools* predstavlja objekt u kojega se definiraju konfiguracije dodataka *Editor* uređivača.
- Ključ *data* predstavlja podatke koje *Editor* uređivač prima za ažuriranje blog posta.

```
const initEditor = () => {
    editor = new EditorJS({
      holder: "blog-editor",
      placeholder: "Start typing...",
      tools: {
        header: {…},
        quote: {…},
        list: {…},
        image: {…},
      },
      data: !props.blogData ? null : props.blogData,
    });};
```

Pomoću *useEffect*() funkcije, kada je komponenta dostupna u DOM-u, poziva se funkcija *initEditor()* te kada se komponenta makne iz DOM-a *editor* varijabla vraća se na početno stanje. Pomoću ovog načina dobije se nova instanca *Editor* biblioteke za novi blog post i ažuriranje postojećeg.

```
useEffect(() => {
  if (!editor) initEditor();
  return () => {
    editor = null;
  };
}, []);
```

Primjer koda ispod prikazuje dio JSX-a koji vraća komponenta Editor.js:

```
return (
     <div className='blog-editor-wrapper'>
        <div id='blog-editor'></div>
        <button className='button' onClick={() => saveEditorData()}>
          Save
        </button>
     </div>
);
```

Spremanje podataka iz *Editor* biblioteke odvija se pomoću metode *editor.save()* koja vraća *Promise* objekt i daljnje operacije obrađuju se na asinkroni način.

```
const saveEditorData = () => {
  editor
    .save()
    .then((outputData) => {
      const data = {
        title,
        isPublished,
        ...outputData,
      };
      props.saveData(data);
    })
    .catch((error) => {
      console.log("Saving failed: ", error);
    });
};
```

Primjer inicijalizacije Editor.js komponente u new-blog.js i edit-blog.js komponentama prikazana je u primjeru koda ispod. Potrebna su samo dvije vrijednosti *saveData* i *jwt* za novi blog post, a svih pet za ažuriranje blog posta.

Vrijednosti u obliku objekta (props) koje komponenta prima su:

- **edit** – služi za logiku odvajanja html elemenata i funkcija između pisanja novog blog posta i ažuriranja postojećeg.
- **jwt** – JWT za pozivanje krajnje točke koja je zadužena za spremanje slike na poslužitelju
- **saveData** – funkcija s povratnim pozivom zadužena je za spremanje novog blog posta ili spremanje postojećeg ažuriranog blog posta.
- **deleteBPost** – funkcija s povratnim pozivom zadužena za brisanje blog posta.
- **blogData** – podaci postojećeg blog posta, za ažuriranje.

```
<Editor
   edit
   jwt={jwt}
   saveData={saveData}
   deleteBPost={deleteBlogPost}
   blogData={blogData}
/>
```

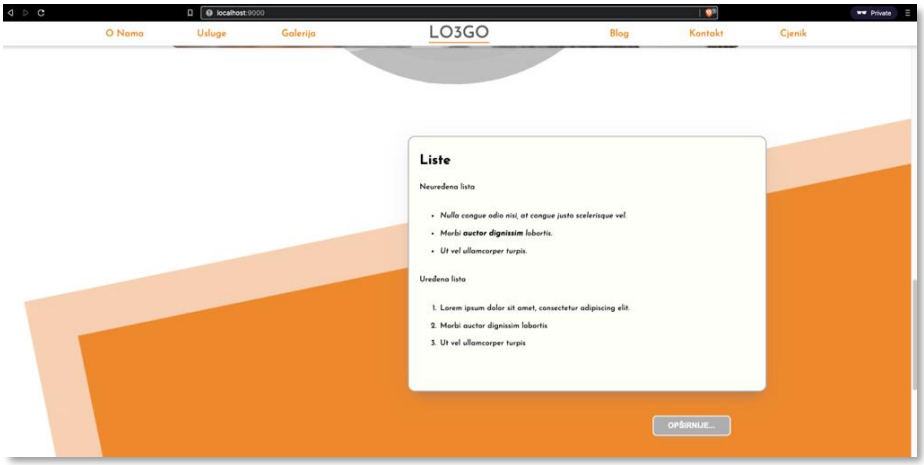
5.7. Pregled sučelja za prikaz, objavu i uređivanje blog sadržaja

Kroz ovo poglavlje priloženi su slike ekrana za prikaz, objavu i uređivanje blog sadržaja i kod koji ih reprezentira. Prvo pod poglavlje predstavlja javno dostupne sekcije za prikaz blog sadržaja, a drugo poglavlje privatne sekcije za objavu i uređivanje blog sadržaja, zaštićene zaporkom.

5.7.1. Komponente za prikaz blog sadržaja

React funkcijske komponente za prikaz blog sadržaja pisane su na sličan način, gdje stanja su deklarirana za sadržaj blog posta, pogrešku blog posta i stanje učitavanja (čekanje odgovora od poslužitelja), za svaku krajnju točku.

- **Blog komponenta na početnoj stranici** - /#blog
  - GET – /api/blog/latest



Slika 22. Prikaz ekrana blog sekcije na početnoj stranici

Primjer koda ispod prikazuje veći dio komponente na početnoj stanici www.lo3go.hr prikazanoj na slici 22 iznad. To je najjednostavnija komponenta koja poziva jednu krajnju točku poslužitelja i pomoću životnog ciklusa komponente generira odgovarajuće podatke na zaslon.

```
const Blog = (props) => {
  const [blogData, setBlogData] = useState([]);
  const [blogDataErr, setBlogDataErr] = useState("");
  const [loadingBlogData, setLoadingBlogData] = useState(true);
  const fetchBlogData = async () => {
    try {
```

```
    const response = await fetch(`${props.apiHost}blog/latest`);
    if (!response.ok) {
      const data = await response.json();
      setBlogDataErr(`${response.status} - ${t(data.error)}`);
      setLoadingBlogData(false);
    } else {
      const data = await response.json();
      setBlogData(data);
      setLoadingBlogData(false);
    }
  } catch (error) {
    if (!error.response) {
      setBlogDataErr(t("server_not_found"));
    } else {
      setBlogDataErr(t(error.response.data.error));
    }
    setLoadingBlogData(false);
  }
};
useEffect(() => {
  fetchBlogData();
}, []);
return (
  <div className='box'>
    {!blogDataErr.length ? (
      <>
        {!loadingBlogData ? (
          blogData && !blogData.length ? (
            <>
              <div className='blog-content'>
                <h1 className='blog_title'>{blogData.title}</h1>
                {blogData.blocks.map((data, index) => {
                  return (
                    <div className='blog_content_item' key={index}>
                      {parsBlogData(data)}
                    </div>
                  );
                })}
              </div>
              <div className='button-wrapper'>
```

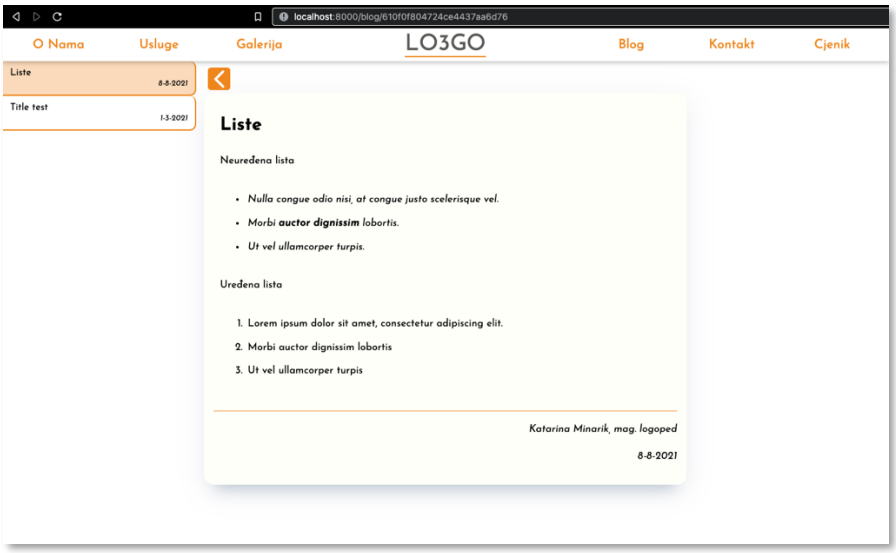
```
                    <button
                      className='button gray'
                      onClick={() => navigate("/blog")}>
                      {t("blog_button_more")}
                    </button>
                  </div>
                </>
              ) : (
                <div className='blog-content'>
                  <div className='blog_center_content'>
                    <h2>{t("blog_no_blog_post_published")}</h2>
                  </div>
                </div>
              )
            ) : (
              <div className='blog_center_content'>
                <Loader />
              </div>
            )}
          </>
        ) : (
          <div className='blog-content'>
            <div className='blog_center_content'>
              <h2>{blogDataErr}</h2>
            </div>
          </div>
        )}
      </div>
    );
  };
```

*Priloženi kod iznad nije najčitljiviji, iz tog razloga ostali primjeri koda kroz poglavlja koja slijede su skraćeni na neke osnovne funkcije, stanja i JSX koje komponente vraćaju.

Dobra praksa za pisanje čitljivijeg koda je smisleno odvajanje JSX-a po cjelinama, primjer za kod iznad može biti jednostavna funkcijska komponenta za prikaz pogreške koja je centrirana unutar roditeljskog div elementa.

Također, funkcije koje zovu direktno krajnje točke poslužitelja treba odvojiti od funkcijskih komponenti u zasebne datoteke i dokumente, i pomoću povezivanja *Promise* objekta u lanca obraditi podatke dobivenih od poslužitelja. Ili se može koristiti biblioteka koja pojednostavljuje taj proces, react-query. (https://react-query.tanstack.com/)

- **Blog stanica** - /blog
  - GET – /api/blog/:id
  - GET – /api/blog/all-published



Slika 23. Prikaz ekrana blog stranice

Primjer koda ispod predstavlja skraćeni oblik funkcijske komponente blog stranica, prikazane na slici 23 iznad, slično kao blog komponenta na početnoj stranici. Komponenta sadrži stanja (*podatci, učitavanje i pogreška*) za sve objavljene blog postove i stanja za trenutno odabranog blog posta.

- fetchBlogData() – služi za dohvat svih objavljenih blog postova
- fetchBlogPost(id) – služi za dohvat pojedinog blog posta pomoću id-a
- <BlogSidebar /> – komponenta koja generira listu svih objavljenih blog postova

```
const BlogPage = (props) => {
  const fetchBlogData = async () => {...};
  const fetchBlogPost = async (id) => {...};
  const blogDate = () => {...};

  return (
    <section id='blog-page'>
      {!loadingAllBlogData && allBlogData.length > 1 ? (
        <BlogSidebar
          items={allBlogData}
          loadingBlogData={loadingBlogData}
          changeBlogPost={handleChangeBlogPost}
          activeBlogId={activeBlogId}
```

```
          />
        ) : null}
        <div className='blog-content'>
          <h1 className='blog_title'>{blogData.title}</h1>
          {blogData.blocks.map((data, index) => {
            return (
              <div className='blog_content_item' key={index}>
                {parsBlogData(data)}
              </div>
            );
          })}
          <div className='blog_info'>
            <p>Katarina Minarik, mag. logoped</p>
            <p>{blogDate()}</p>
          </div>
        </div>
      </section>
    );
  };
```

Glavna funkcija koja obrađuje podatke za prikaz blog sadržaja prikazana je u primjeru koda ispod. Svrha te funkcije je da iz primljenih podataka odvaja pojedini tip sadržaja (tekst, lista, naslov, slika ili citat) i vraća JSX za pojedini tip podatka sa odgovarajućom CSS klasom.

```
const ParsBD = (bData) => {
  let data = null;
  switch (bData.type) {
    case "header": //...
    case "paragraph":
      const checkABI = find_abi(bData.data.text);
      if (checkABI) {
        data = (
          <div
            className='blog_paragraph'
            dangerouslySetInnerHTML={{ __html: bData.data.text }}></div>
        );
      } else {
        data = <div className='blog_paragraph'>{bData.data.text}</div>;
      }
      break;
    case "list": //...
```

```
      case "quote": //...
      case "image": //...
      default:
        data = null;
    }
    return data;
};
```

Svrha pomoćne funkcije *find_abi()* je da u proslijeđenim podatcima blog sadržaja pronađe tekst koji sadrži hipertekst, podebljana slova ili ukošena slova. Dodavanje tih elemenata u DOM radi se pomoću *dangeroulySetInnerHTML.* (https://reactjs.org/docs/dom-elements.html)

```
const find_abi = (text) => {
  const conditions = ["</a>", "</b>", "</i>"];
  const check = conditions.some((el) => text.includes(el));
  return check;
};
```

Primjer CSS koda ispod reprezentira ugniježdene CSS klase koje dodaju stil sadržaju blog komponente. PostCss modul: postcss-nested. (https://github.com/postcss/postcss-nested)

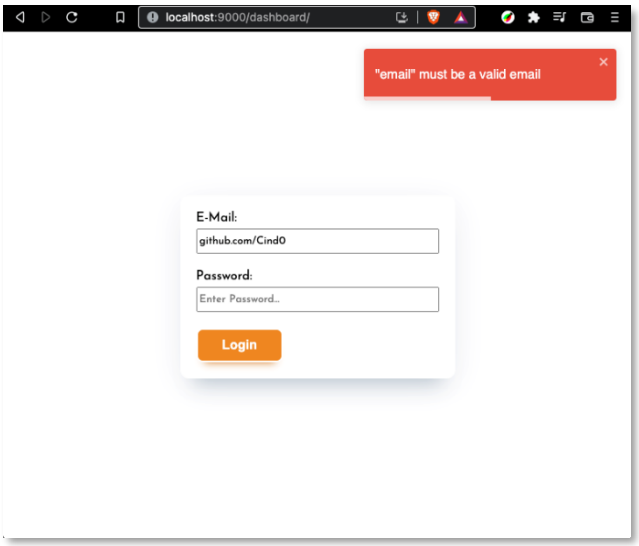
```
.blog-content {
  .blog_title {...}
  .blog_content_item {
    margin: 0 auto;
    max-width: 700px;
    padding: 0.4em;
    li {
      padding: 5.5px 0 5.5px 3px;
    }
  }
  h1 {...}
  h2 {...}
  a {...}
  .blog_quote_wrapper {...}
  .blog_paragraph {...}
  .blog_img_wrapper {...}
  .blog_info {...}
}
```

5.7.2. Komponente za objavu i uređivanje blog sadržaja

Na ruti */dashboard* napravljena je aplikacija na jednoj stranici (SPA – eng. Single-Page Application). Komponente koje upravljaju tom aplikacijom su:

- AdminContext.js – inicijalizira globalno stanje aplikacije
- AdminLayout.js – komponenta koja čuva globalno stanje aplikacije te pomoću nje stanje je dostupno kroz aplikaciju koristeći *useContext* funkciju. (https://reactjs.org/docs/hooks-reference.html#usecontext)
- AppControler.js – komponenta koja je zadužena za mijenjanje komponente pomoću globalnog stanja, a promjena tog stanja radi se pomoću ključa *„nav“.* (eng. *nav*igation)

- **Login komponenta**
  - POST – /api/auth



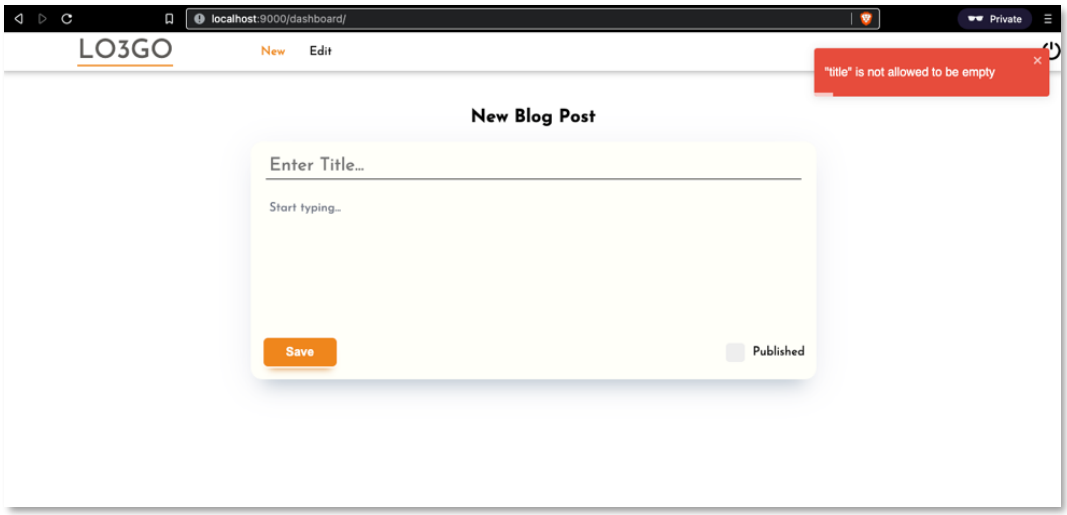
Primjer koda ispod predstavlja jednostavan oblik forme za prijavu korisnika u sustav, prikazanu na slici 24. Podatci uneseni u formu su e-mail i zaporka (*pass*), a ti podatci su spremljeni unutar stanja komponente. Provjera valjanosti podataka odvija se na poslužitelju i prikazuje se korisniku pomoću biblioteke: react-toastify. (https://www.npmjs.com/package/react-toastify)

Slika 24. Prikaz ekrana za prijavu u sustav

```
const Login = () => {
  const [email, setEmail] = useState("");
  const [pass, setPass] = useState("");
  const notifyError = (msg) => toast.error(msg);
  const login = async () => {…};
  return (
    <section id='admin-login'>
      <div className='center_container'>
        <div className='login-wrapper'>
          <form>
```

```
            <div className='input-wrapper'>
              <label htmlFor='email'>E-Mail:</label>
              <input
                type='text'
                onChange={(e) => setEmail(e.target.value)}/>
            </div>
            <div className='input-wrapper'>
              <label htmlFor='password'>Password:</label>
              <input
                type='password'
                onChange={(e) => setPass(e.target.value)}/>
            </div>
          </form>
          <button className='button' onClick={login}>Login</button>
        </div>
      </div>
    </section>
  );
};
```

- **Komponenta novi blog post**
  - POST – /api/blog



Slika 25. Prikaz ekrana za pisanje novog blog sadržaja

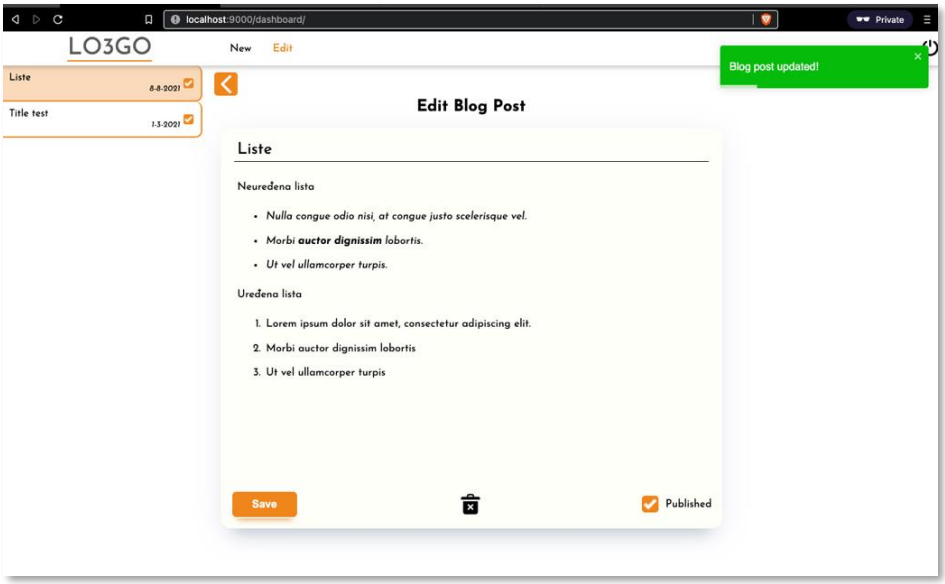
Skraćeni primjer koda ispod predstavlja komponentu za pisanje novog blog posta, kako je prikazano na slici 25 iznad, ona upravlja Editor.js komponentom sa svojim stanjem i životnim ciklusom.

```
const NewBlog = () => {
  const notifySuccess = (msg) => toast.success(msg);
  const notifyError = (msg) => toast.error(msg);
  const { logged_in, jwt, action } = React.useContext(AdminContext);

  useEffect(() => {
    if (!logged_in) action.setNav("login");
  }, []);

  const saveData = async (editorData) => {};
  return (
    <section id='admin-new-blog'>
      <h1>New Blog Post</h1>
      <div className='blog-container'>
        <Editor jwt={jwt} saveData={saveData} />
      </div>
    </section>
  );
};
```

- **Komponenta uredi blog post**
  - GET – /api/blog/:id
  - GET – /api/blog/all
  - DELET – /api/blog/:id
  - PUT – /api/blog/:id



Slika 26. Prikaz ekrana za uređivanje blog sadržaja

Skraćeni primjer koda ispod predstavlja komponentu koja upravlja uređivanjem blog sadržaja, kako je prikazano na slici 26 iznad. Slično kao i komponenta za pisanje novog blog posta, ona upravlja Editor.js komponentom pomoću stanja i životnog ciklusa komponente.

```
const EditBlog = () => {
  const fetchBlogPost = async (blogID) => {...};
  const fetchAllBlogPosts = async () => {...};
  const fetchInitalBlogData = async () => {...};
  useEffect(() => {
    fetchInitalBlogData();
  }, []);
  const handleChangeBlogPost = async (blog) => {...};
  const saveData = async (editorData) => {...};
  const deleteBlogPost = async () => {...};

  return (
      <section id='admin-edit-blog'>
        <h1>Edit Blog Post</h1>
        {!loadingAllBlogData && allBlogData.length ? (
          <BlogSidebar
            items={allBlogData}
            loadingBlogData={loadingBlogData}
            changeBlogPost={handleChangeBlogPost}
            activeBlogId={activeBlogId}
          />
        ) : null}
        <div className='blog-container'>
          <Editor
            edit
            jwt={jwt}
            saveData={saveData}
            deleteBPost={deleteBlogPost}
            blogData={blogData}
          />
        </div>
      </section>
  );
};
```

6. Zaključak

Kroz razvoj i pisanje rada napravljena su dva servisa, tj. *Node* aplikacije, prvi servis je *klijentska* aplikacija (*Gatsby*) kojoj je zadatak posluživanje krajnjeg korisnika sa HTML-om, CSS-om i JavaScrip-om, a drugi servis je ***poslužitelj*** aplikacija (*Express*) koja komunicira s bazom podataka (*MongoDB*) i s klijent aplikacijom preko HTTP protokola, gdje poslužitelj aplikacija prima i šalje odgovarajuće podatke klijent aplikaciji. Ovakva arhitektura omogućava odvajanje klijentskih aplikacija od poslužiteljskih, gdje je moguće napraviti npr. mobilnu aplikaciju i spojiti je sa poslužiteljskom aplikacijom.

Glavni koncept razrađen tijekom razvoja klijent aplikacije je pisanje *React* funkcijskih komponenti umjesto klasnih komponenti te samo razumijevanje *React funkcija koje se mogu nadjačati (eng. Hooks),* a taj funkcijski stil pisanja komponenti može se primijeniti u bilo kojem razvojnom okviru koji koristi *React,* a ne samo *Gatsby*. Razvojem jednostavne poslužiteljske aplikacije razrađeni su koncepti trajnog spremanja podataka i njihova obrada, provjera autentičnosti korisnika i sigurnosna komunikacija između aplikacija te slanje i primanje podataka. Prednost razvoja takve poslužiteljske aplikacije je da su svi podatci u vlasništvu autora aplikacije, tj. podatci su spremljeni na računalu koje pokreće tu aplikaciju, a najveći nedostatak je da razvoj i održavanje zahtijevaju više vremena nego neke usluge dostupne danas na tržištu, primjerice Googleov Firebase.

Problemi koji su zahtijevali najviše vremena za razvoj su sama integracija *Editor.js* biblioteke u razvojni okvir *Gatsby* i *React-ov* životni ciklus. A na razvoju poslužitelj aplikacije najveći problem bio je spremanje blog slike iz *Editor.js* biblioteke u datotečni sustav poslužitelja te proces generiranja JWT-a i razumijevanje koncepta ***posredničke funkcije*** unutar razvojnog okvira *Express*, pomoću koje se provjerava autentičnost korisnika.

JavaScript ekosistem u proteklih deset godina ubrzano raste u samom pogledu broja modula, razvojnih okvira, biblioteka ili platformi na kojoj se razvija aplikacija, bilo da se radi o mobilnoj ili desktop aplikaciji, poslužitelju, IoT-u ili pak nekom web rješenju pomoću jednog programskog jezika (*JavaScript*). Praktični dio rada mogao se riješiti brže i jednostavnije pomoću *baze podataka kao usluga* ili *CMS ako usluga*, ali cilj rada bio je naučiti i razumjeti s niže apstraktne razine kako te procese obraditi i primijeniti u praksi.

JavaScript jezik prvenstveno je napravljen za web preglednik i lakšu pretragu i interakciju sa sadržajem na internetu. Sukladno tome veliki broj problema, rješenja i znanja dostupno je na indeksiranim web stranicama Googlea i pretraživača DuckDuckGo, koje piše i održava JavaScript zajednica otvorenog koda.

Popis literature

[1] „About JavaScript“, https://developer.mozilla.org/, (2021). [Na internetu]. Dostupno: https://developer.mozilla.org/en-US/docs/Web/JavaScript/About_JavaScript [pristupljeno: 25. 7. 2021.]

[2] Alex Libby, (2016). „*Mastering PostCss for Web Design“.* Packt Publishing

[3] „Arorw function expressions“, https://developer.mozilla.org/, (2021). [Na internetu]. Dostupno na: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Functions/Arrow_functions [pristupljeno: 26. 7. 2021.]

[4] „Async function“, https://developer.mozilla.org/, (2021). [Na internetu]. Dostupno na: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/async_function [Pristupano: 29.7.2021]

[5] "Base Concepts", https://editorjs.io/. [Na internetu]. Dostupno na: https://editorjs.io/base-concepts [pristupljeno: 10. 9. 2021.]

[6] „Block“, https://developer.mozilla.org/, (2021). [Na internetu]. Dostupno na: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/block [pristupljeno: 25. 7. 2021.]

[7] „Callback function“, https://developer.mozilla.org/, (2021). [Na internetu]. Dostupno: https://developer.mozilla.org/en-US/docs/Glossary/Callback_function [pristupljeno: 29. 7. 2021.]

[8] „Classes“, https://developer.mozilla.org/, (2021). [Na internetu]. Dostupno na: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Classes [pristupljeno: 23. 7. 2021.]

[9] „Commands (Gatsby CLI)“, https://www.gatsbyjs.com/. [Na internetu]. Dostupno na: https://www.gatsbyjs.com/docs/reference/gatsby-cli/ [pristupljeno: 15. 9. 2021.]

[10] „Components and Props“, https://reactjs.org. [Na internetu]. Dostupno na: https://reactjs.org/docs/components-and-props.html [pristupljeno: 5. 9. 2021.]

[11] „Const“, https://developer.mozilla.org/, (2021). [Na internetu]. Dostupno na: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/const [pristupljeno: 25 .7. 2021.]

[12] „Destructuring assignment“, https://developer.mozilla.org/, (2021). [Na internetu]. Dostupno na: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Destructuring_assignment [pristupljeno: 28. 7. 2021.]

[13] „Express“, https://expressjs.com/. [Na internetu]. Dostupno na: https://expressjs.com/ [pristupljeno: 4. 8. 2021.]
[14] „Express router“, https://expressjs.com/. [Na internetu]. Dostupno na: https://expressjs.com/en/4x/api.html#router [pristupljeno: 4. 8. 2021.]
[15] „Fast Refresh“, https://www.gatsbyjs.com/. [Na internetu]. Dostupno na: https://www.gatsbyjs.com/docs/reference/local-development/fast-refresh/ [pristupljeno: 15. 9. 2021]
[16] „Gatsby Plugin Library“, https://www.gatsbyjs.com/. [Na internetu]. Dostupno na: https://www.gatsbyjs.com/plugins [pristupljeno: 15. 9. 2021.]
[17] „Gatsby Project Structure“, https://www.gatsbyjs.com/. [Na internetu]. Dostupno na: https://www.gatsbyjs.com/docs/reference/gatsby-project-structure/ [pristupljeno: 15. 9. 2021.]
[18] „Gatsby Way of Building“, https://www.gatsbyjs.com/. [Na internetu]. Dostupno na: https://www.gatsbyjs.com/docs/ [pristupljeno: 15. 9. 2021.]
[19] „GraphQL API“, https://www.gatsbyjs.com/. [Na internetu]. Dostupno na: https://www.gatsbyjs.com/docs/reference/graphql-data-layer/graphql-api/ [pristupljeno: 15. 9. 2021.]
[20] „Hooks API Reference“, https://reactjs.org. [Na internetu]. Dostupno na: https://reactjs.org/docs/hooks-reference.html [pristupljeno: 7. 9. 2021.]
[21] „Introducing Hooks“, https://reactjs.org. [Na internetu]. Dostupno na: https://reactjs.org/docs/hooks-intro.html [pristupljeno: 7. 9. 2021.]
[22] „Introducing JSX“, https://reactjs.org. [Na internetu]. Dostupno na: https://reactjs.org/docs/introducing-jsx.html [pristupljeno: 4. 9. 2021.]
[23] „Introduction to JSON Web Tokens“, https://jwt.io/. [Na internetu]. Dostupno na: https://jwt.io/introduction [pristupljeno: 17. 9. 2021.]
[24] Kirupa Chinnathambi. (2018). *Learning Reat, A Hands-On Guide to Building Web Applications Using React and Redux, second edition,* Pearson Education
[25] „Let“, https://developer.mozilla.org/, (2021). [Na internetu]. Dostupno na: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/let [pristupljeno: 25. 7. 2021.]
[26] „Local Development“, https://www.gatsbyjs.com/. [Na internetu]. Dostupno na: https://www.gatsbyjs.com/docs/how-to/local-development/ [pristupljeno: 15. 9. 2021.]
[27] Luis Atencio. (2021). *The Joy of JavaScript. Shelter Island*, NY: Manning Publications
[28] „Mongoose Models“, https://mongoosejs.com/. [Na internetu]. Dostupno na: https://mongoosejs.com/docs/5.x/docs/models.html [pristupljeno: 15. 8. 2021.]
[29] „Mongoose Shematypes“, https://mongoosejs.com/. [Na internetu]. Dostupno na: https://mongoosejs.com/docs/5.x/docs/schematypes.html [pristupljeno: 15. 8. 2021.]

[30] Mosh Hamedani. (2018). Node.js: The Complete Guide to Build RESTful APIs. [Na internetu]. Dostupno na: https://www.udemy.com/course/nodejs-master-class/. [pristupljeno 25. 7. 2021.]

[31] „Most popular technologies“. (2021). https://insights.stackoverflow.com/, [Na internetu]. Dostupno na: https://insights.stackoverflow.com/survey/2021#technology-most-popular-technologies [pristupljeno 25. 9. 2021.]

[32] Node package manager (NPM), https://www.npmjs.com/. [Na internetu]. Dostupno na: https://www.npmjs.com/ [pristupljeno 18. 8. 2021.]

[33] PostCSS, https://postcss.org/. [Na internetu]. Dostupno na: https://postcss.org/ [pristupljeno 17. 9. 2021.]

[34] „Promise“, https://developer.mozilla.org/, (2021). [Na internetu]. Dostupno na: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise [pristupljeno: 29. 7. 2021.]

[35] „React-spring Basics“, https://react-spring.io/. [Na internetu]. Dostupno na: https://react-spring.io/basics [pristupljeno: 10. 9. 2021.]

[36] „React-spring Configs“, https://react-spring.io/. [Na internetu]. Dostupno na: https://react-spring.io/common/configs [pristupljeno: 9. 9. 2021.]

[37] „React-spring introduction“, https://react-spring.io/. [Na internetu]. Dostupno na: https://react-spring.io/ [pristupljeno: 9. 9. 2021.]

[38] „React-spring useSpring“, https://react-spring.io/. [Na internetu]. Dostupno na: https://react-spring.io/hooks/use-spring [pristupljeno: 10. 9. 2021.]

[39] „React-spring useTrail“, https://react-spring.io/. [Na internetu]. Dostupno na: https://react-spring.io/hooks/use-trail [pristupljeno: 10. 9. 2021.]

[40] „React.Component“, https://reactjs.org. [Na internetu]. Dostupno na: https://reactjs.org/docs/react-component.html [pristupljeno: 5. 9. 2021.]

[41] „Rest parameters“, https://developer.mozilla.org/, (2021). [Na internetu]. Dostupno na: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Functions/rest_parameters [pristupljeno: 28. 7. 2021.]

[42] „Routing“, https://www.gatsbyjs.com/. [Na internetu]. Dostupno na: https://www.gatsbyjs.com/docs/reference/routing/creating-routes/ [pristupljeno: 15. 9. 2021.]

[43] „Spred synrax (...)“, https://developer.mozilla.org/, (2021). [Na internetu]. Dostupno na: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Spread_syntax [pristupljeno: 28. 7. 2021.]

[44] „Template literals (Template strings)“, https://developer.mozilla.org/, (2021). [Na internetu]. Dostupno na: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Template_literals [pristupljeno: 27. 7. 2021.]

[45] „This“, https://developer.mozilla.org/, (2021). [Na internetu]. Dostupno na: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/this [pristupljeno: 26. 7. 2021.]

[46] „try…catch“, https://developer.mozilla.org/, (2021). [Na internetu]. Dostupno na: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/try...catch [pristupljeno: 29. 7. 2021.]

[47] „Using Fetch“, https://developer.mozilla.org/, (2021). [Na internetu]. Dostupno na: https://developer.mozilla.org/en-US/docs/Web/API/Fetch_API/Using_Fetch [pristupljeno: 29. 8. 2021.]

[48] „Using middleware“, https://expressjs.com/. [Na internetu]. Dostupno na: https://expressjs.com/en/guide/using-middleware.html [pristupljeno: 10. 8. 2021.]

Popis slika