Usporedba alata za automatizaciju procesa programiranja na strani klijenta

Pikl, Mateo

SVEUČILIŠTE U ZAGREBU

FAKULTET ORGANIZACIJE I INFORMATIKE

V A R A Ž D I N

Mateo Pikl

Usporedba alata za automatizaciju procesa programiranja na strani klijenta

ZAVRŠNI RAD

Varaždin, 2024.

SVEUČILIŠTE U ZAGREBU

FAKULTET ORGANIZACIJE I INFORMATIKE

V A R A Ž D I N

Mateo Pikl

JMBAG: 0016151626

Studij: Informacijski i poslovni sustavi

Usporedba alata za automatizaciju procesa programiranja na strani klijenta

ZAVRŠNI RAD

Mentor:

doc. dr. sc. Matija Novak

Varaždin, rujan 2024.

Mateo Pikl

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 su korištene etički prikladne i prihvatljive metode i tehnike rada.

Autor potvrdio prihvaćanjem odredbi u sustavu FOI-radovi

Sažetak

Kako složenost web aplikacija stalno raste, tako raste i potreba za učinkovitim rukovanjem resursima. U ovom radu istražit će se vrste tehnologija ili alata za automatizaciju procesa programiranja na strani klijenta. Alati koji će biti obrađeni su Webpack, Grunt i Gulp. Ovi alati spadaju u tri najkorištenija alata za automatizaciju zadaća kao što su grupiranje (eng. *Bundling*), smanjivanje (eng. *minification*) i upravljanje ovisnostima (eng. *dependency management*). Kroz rad će se obradit svaki alat na način da će se prikazati kratak uvod u alat, instalacija ili integracija, preporuka kada i zašto odabrati jedan alat prije drugog ovisno o problematici projekta, koje su prednosti svakog alata, ali naravno i njegovi nedostatci.

Ključne riječi: automatizacija, webpack, grunt, gulp, web aplikacija, agilni razvoj, klijentska strana

Sadržaj

1. Uvod

Tema ovog završnog rada je „*Usporedba alata za automatizaciju procesa programiranja na strani klijenta*“. U ovom radu se kroz teoretski i praktični dio prikazuje konfiguracija i primjena alata za automatizaciju procesa u razvoju modernih web aplikacija.

Predmet ovog završnog rada podrazumijeva važnost za automatizacijom procesa na strani klijenta koja je neizostavan dio svakog razvoja moderne web aplikacije. Automatizacija procesa ističe potrebu za učinkovitim upravljanjem različitim vrstama resursa, uključujući razne vrste JavaScript odnosno TypeScript datoteka, CSS odnosno Sass datoteka, fontova, slika i slično. U radu će biti analizirana tri najkorištenija alata za automatizaciju zadaća kao što su grupiranje (eng. *Bundling*), smanjivanje (eng. *minification*) i upravljanje ovisnostima (eng. *dependency management*), a to su: Webpack, Grunt i Gulp. Kako složenost web aplikacija stalno raste, tako raste i potreba za učinkovitim rukovanjem resursima. Cilj ovog završnog rada je objasniti pojam automatizacije procesa na strani klijenta te na stvarnom primjeru kroz razvoj vlastite web aplikacije objasniti i prikazati pojam i važnost automatizacije.

Prilikom izrade praktičnog dijela rada korištene su razne tehnologije na klijentskoj i poslužiteljskoj strani. Na strani klijenta korištene su tehnologije poput TypeScript, React.js i Redux Toolkit, te Webpack alat za automatizaciju procesa. Na poslužiteljskoj strani korištene su tehnologije poput Node.js i Express za kreiranje REST API sučelja za komunikaciju s bazom podataka. Za bazu podataka odabrana je PostgreSQL baza podataka i pgAdmin4 sučelje za upravljanje bazom podataka.

Rad je podijeljen na 6 poglavlja uključujući uvod i zaključak. Rad započinje pojmovnim određenjem automatizacije procesa i značajem automatizacije u modernom web razvoju. Zatim se navodi pregled tehnologija za automatizaciju procesa gdje se daje uvod u nadolazeća poglavlja. Sljedeće poglavlje se odnosi na alat Grunt. U okviru ovog poglavlja spomenute su i objašnjene prednosti i nedostatci s kojima svaki od alata dolazi, te je dan uvid u proces instalacije i konfiguracije alata na novom projektu. Četvrto poglavlje posvećeno je usporedbi između obrađenih alata. Usporedba alata provedena je kroz nekoliko kriterija: jednostavnost korištenja, brzina izvršavanja i popularnost. U petom poglavlju rada prikazana je implementacija vlastite web aplikacije koristeći odabrani alat za automatizaciju. Poglavlje započinje opisom aplikacije, te se nastavlja objašnjenjem korištenih tehnologija, arhitekture aplikacije i na kraju implementacije. Na kraju rada, u zadnjem poglavlju nalazi se zaključak.

2. Automatizacija procesa u web razvoju

U tradicionalnom razvoju web aplikacija, postupci instalacije i održavanja paketa i okvira trećih strana bili su često složeni i dugotrajni procesi. Programeri bi ručno preuzimali pakete, bez pouzdanih alata za praćenje dostupnih ažuriranja, što ih je prisiljavalo da redovito istražuju i pronalaze najnovije verzije te ručno zamjenjuju zastarjele [1]. Ovaj pristup postajao je posebno zamoran kada je bilo potrebno upravljati velikim brojem paketa, jer se svaki paket morao ažurirati pojedinačno. Takav način rada ne samo da je usporavao cijeli razvojni proces, već je također povećavao rizik od grešaka i problema u aplikaciji.

Većina web aplikacija oslanja se na određene ovisnosti, poput biblioteka ili okvira na klijentskoj strani, kao što su jQuery, Vue, React ili Angular. Ručno preuzimanje tih biblioteka za svaki pojedinačni projekt često postaje zamorno i zahtjeva mnogo vremena. Tradicionalni proces dodavanja novih ovisnosti podrazumijevao je najprije pregledavanje službene dokumentacije biblioteke kako bi se preuzeo izvorni kod. Taj kod bi se potom morao ručno kopirati u direktorij s resursima, a zatim dodati odgovarajuće poveznice na JavaScript ili CSS datoteke unutar HTML datoteke [1]. Ažuriranje postojećih ovisnosti dodatno bi kompliciralo proces, budući da je programer morao ručno provjeravati dostupnost novih verzija i uklanjati stare verzije biblioteke, što je cijeli postupak činilo sporim i sklonim greškama.

S razvojem modernih alata za upravljanje paketima i automatizaciju, ovi problemi su gotovo pa potpuno nestali. Alati za automatizaciju omogućuju da se vrijeme i napori usmjere na razvoj aplikacija, umjesto repetitivnih i zamornih zadatka. Automatizacija smanjuje vrijeme potrebno za obavljanje ponavljajućih poslova, čime se povećava učinkovitost i konzistentnost u razvoju. Također, svi članovi razvojnog tima koriste iste verzije jezika i alata, što uvelike smanjuje rizik od pogrešaka.

Automatizacija poboljšava skalabilnost projekata, omogućujući lakše upravljanje velikim projektima. Omogućuje razvojnim timovima da efikasno odgovore na zahtjeve korisnika i tržišta, povećavajući ukupnu kvalitetu i održivost proizvoda. Uvođenje Node.js i Node Package Manager (npm) kao standarda u razvoju aplikacija na strani klijenta donijelo je značajne promjene. Umjesto kompleksnih zastarjelih alata, fokus se prebacio na male pakete i dodatke koji rješavaju specifične probleme s visokim stupnjem učinkovitosti [1], [2].

2.1. Node.js u razvoju na strani klijenta

Razvoj softvera danas je daleko jednostavniji i brži zahvaljujući raznovrsnim alatima i tehnologijama koje automatiziraju ponavljajuće zadatke i omogućuju kreiranje složenih

aplikacija s većom efikasnošću. Prema [1], [3] jedna od ključnih tehnologija koja je transformirala ovaj proces je Node.js. Node.js je donio promjene u načinu na koji programeri razvijaju web aplikacije, omogućujući im da koriste JavaScript, jezik koji je izvorno bio namijenjen za kreiranje interaktivnih elemenata na web stranicama, za izradu klijentskog i poslužiteljskog dijela aplikacija. Ova mogućnost korištenja istog jezika na oba kraja razvoja značajno pojednostavljuje proces izgradnje i održavanja aplikacija, jer programeri mogu pisati kompletan kod bez potrebe za prelaskom na drugi programski jezik.

Node.js, u kombinaciji s npm-om, postao je nezamjenjiv alat u razvoju modernih web aplikacija, posebno na klijentskoj strani. Kako ističu autori u radovima [1], [3], Node.js omogućuje programerima da brzo i učinkovito kreiraju interaktivna i dinamična korisnička sučelja, zahvaljujući bogatom ekosustavu paketa dostupnih putem npm-a. Ovi paketi pokrivaju širok spektar funkcionalnosti, od upravljanja stanjem aplikacije, optimizacije performansi, do specifičnih alata za testiranje, automatizaciju i sigurnost. Time se značajno ubrzava razvojni proces, jer programeri mogu lako integrirati gotove module u svoje projekte, što smanjuje potrebu za pisanjem ponavljajućeg koda od nule. Tako, ne samo da se skraćuje vrijeme potrebno za razvoj, već se poboljšava i kvaliteta koda, čineći ga robusnijim, efikasnijim i lakšim za održavanje.

Node.js posjeduje bogat ekosustav i kontinuirano se razvija zahvaljujući velikoj i aktivnoj zajednici programera koji doprinose njegovom razvoju i održavanju. Njegova modularna priroda omogućuje programerima da lako dodaju nove funkcionalnosti u svoje aplikacije. Node.js posjeduje ogroman skup paketa dostupnih preko npm-a, poznatih kao moduli. Ovi moduli omogućuju programerima da prošire osnovne mogućnosti Node.js-a. Paket u Node.js-u predstavlja direktorij koji sadrži sve potrebne datoteke i konfiguracije, čime se pojednostavljuje integracija novih značajki u aplikacije [1], [3].

Standardizirani alati i biblioteke unutar ovog alata omogućuju konzistentnost u razvoju, što olakšava razmjenu koda između članova tima i osigurava da svi rade s istim verzijama alata i modula. Prema [1] ovakav pristup smanjuje rizik od pogrešaka u razvoju i poboljšava ukupnu efikasnost tima.

2.2. Raznolikost klijentskog koda u agilnom razvoju

Raznolikost klijentskog koda u agilnom razvoju omogućuje fleksibilnost i široku prilagodljivost potrebama različitih tipova korisnika i tržišta [1]. Raznolikost često se postiže upotrebom širokog raspona tehnologija, omogućujući razvojnim timovima da softverska rješenja prilagode specifičnim zahtjevima korisnika.

U modernom web razvoju, raznolikost različitih vrsta koda igra ulogu u razvoju dinamičkih, vizualno primamljivih i funkcionalnih web aplikacija. Svaka vrsta koda ili resursa ima svoju ulogu koja unapređuje korisničko iskustvo. U takve vrste koda spadaju:

- Stilovi za definiranje dizajna i osjećaja kojeg korisnik ima prilikom dolaska na web aplikaciju. Stilovi omogućuju kontrolu nad rasporedom, bojama, fontovima te kompletnom vizualnom prezentacijom stranica [4].
- JavaScript - omogućuje manipulaciju podacima i izgradnju interaktivnosti na web stranicama. JavaScript također omogućuje izvršavanje zadataka poput validacije podataka, implementacije složenih animacija, asinkronog dohvaćanja podataka i slanja zahtjeva, te manipulacije stilovima i ostalim resursima [3].
- Fontovi - igraju ključnu ulogu u tipografiji, čitljivosti i brendingu web aplikacija. Prema [5], korištenje ispravnih fontova osigurava jedinstveno i dosljedno tipografsko iskustvo na svim vrstama uređaja. Korištenje prilagođenih fontova moguće je ostvariti putem stilova, raznih usluga kao što je Google Fonts ili uključivanjem vlastitih fontova u projekt.
- Slike - jedan od najvažnijih vizualnih elemenata bilo koje web stranice ili aplikacije te mogu imati značajan utjecaj na vizualnu estetiku i performanse. Kako bi se postigla željena vizualnu estetiku bez gubljenja na performansama, potrebno je obratiti pažnju na veličinu i format slika. U radu [6] autori govore kako korištenje odgovarajućih formata kao što su JPG, PNG, SVG, WebP te njihova optimizacija omogućuju poboljšanja vezana za brzinu, učitavanje i responzivnost stranica. Tehnike poput sporog učitavanja (eng. *lazy loading*) omogućavaju učitavanje slika samo kada su potrebne, čime se poboljšava brzina stranice i korisničko iskustvo [6].

Ove vrste resursa povećavaju upotrebljivost aplikacije i angažman korisnika. U modernom web razvoju. Okviri i biblioteke poput React.js, Angular i Vue.js-a dodatno proširuju mogućnosti JavaScript-a, omogućavajući brži i lakši razvoj kvalitetnih, robusnih i intuitivnih web aplikacija.

3. Pregled tehnologija za automatizaciju procesa na klijentskoj strani

Kako složenost razvoja web aplikacija raste, pojavljuju se različiti izazovi posebno u smislu upravljanja projektima i optimizaciji. Alati za automatizaciju kao što su Webpack, Gulp i Grunt rješavaju većinu problema s učinkovitošću u razvojnom procesu i poboljšavaju korisničko iskustvo [7]. U ovom dijelu rada obradit će se istraživanje tehnologija za automatiziranje procesa programiranja na strani klijenta. Alati koji će biti istraženi su Grunt.js, Gulp.js i Webpack, te će se dati uvid u prednosti i nedostatke, instalaciju i implementaciju pojedinog alata.

3.1. Grunt

Grunt je JavaScript pokretač zadataka (eng. *task runner*) koji omogućuje automatizaciju različitih repetitivnih razvojnih zadataka. Njegova struktura, temeljena na konceptu dodataka (eng. *plugins*), omogućuje visoku razinu prilagodbe. Grunt sadrži bogat ekosustav paketa i dodataka koji omogućuju prilagodbu gotovo svakom projektu. Grunt i njegovi dodaci instaliraju i upravljaju se putem npm-a, što olakšava integraciju unutar Node.js okruženja [8].

Zadaci koji se mogu i često obavljaju pomoću Grunta uključuju [9]: prefiksiranje CSS-a, kompilaciju Sass-a u CSS, minifikaciju JavaScript datoteka te povezivanje datoteka spajanjem JavaScript i CSS datoteka kako bi se smanjili zahtjevi prema serveru.

3.1.1. Prednosti i nedostatci

Prednosti korištenja alata Grunt.js uključuju njegovu jednostavnost korištenja i relativno brzu krivulju učenja. Grunt se sadrži veliki ekosustav dodataka (eng. *plugins*), što znači da gotovo uvijek postoji dodatak koji može riješiti specifične potrebe i izazove [10], [11], [12]. Iako Grunt sadrži i dosta nedostatka ove prednosti ga čine posebno korisnim za brzu automatizaciju raznih jednostavnih zadataka bez potrebe za kompleksnim prilagodbama.

Nedostatci korištenja Grunt.js alata su da s vremenom, alat može postati pretrpan (eng. *bloated*), što može otežati održavanje i konfiguraciju, posebno u većim projektima. Također, nedostatak fleksibilnosti u konfiguraciji može predstavljati ograničenje u usporedbi s modernijim alatima [10], [11] [12]. Posljednje ažuriranje Grunta dogodilo se krajem 2023. godine, što može značiti da neki od dodataka možda imaju sigurnosne propuste ili su zastarjeli (eng. *deprecated*). Navedene nedostatke i faktori se moraju uzeti u obzir prilikom odabira alata

i mogu utjecati na sigurnost i dugoročnu održivost projekta korištenja Grunta u modernim web aplikacijama.

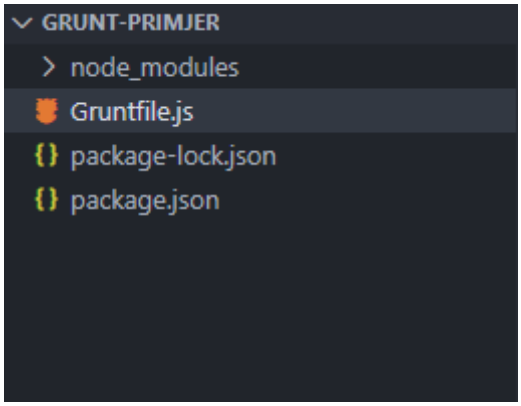
3.1.2. Instalacija Grunt.js

Grunt i Grunt dodaci se instaliraju i upravljaju putem npm-a, upravitelja paketa Node.js. Nakon instalacije Node.js i npm-a, sljedeći korak je globalna instalacija Grunt sučelja naredbenog retka (eng. *Command Line Interface*) pomoću naredbe npm install grunt-cli -g. Grunt sučelje naredbenog retka radi tako da traži lokalno instalirani Grunt. Ako je Grunt pronađen, sučelje naredbenog retka učitava lokalnu instalaciju Grunt biblioteke, primjenjuje konfiguraciju iz datoteke Gruntfile.js, i izvršava sve zatražene zadatke [8].

Sljedeći korak je lokalna instalacija Grunt alata. Grunt je moguće instalirati u razvojne ovisnosti projekta koristeći naredbu: `npm install grunt --save-dev` [8]. Tako, Grunt će biti dostupan samo u razvojnom okruženju jer u produkciji nije potreban.

Slika 1. Grunt dodan u package.json popis razvojnih ovisnosti

Sljedeći korak instalacije je kreiranje `Gruntfile.js` datoteke. `Gruntfile.js` datoteka sadrži sve zadatke, konfiguracije i potrebne dodatke (eng. *plugins*) koje Grunt treba izvršiti.



Slika 2. Izgled projekta nakon kreiranje Gruntfile.js datoteke.

3.1.3. Primjer konfiguriranja i korištenja Gruntfile.js datoteke

U ovom poglavlju prikazat će se konfiguracija Grunt.js alata koja će podržavati automatizirano pokretanje zadataka kao što su: povezivanje JavaScript i CSS datoteka, pretvaranje Sass, odnosno Scss datoteka u CSS datoteke i minifikaciju JavaScript i CSS datoteka.

Prvi korak konfiguracije je kreiranje `Gruntfile.js` datoteke. Kreiranu datoteku potrebno je pohraniti u korijen projekta, te zatim definirati funkciju s `grunt` argumentom za inicijalizaciju konfiguracije. Metoda `initConfig` uzima objekt u kojem su definirani zadaci (eng. *task*) s dodatnim konfiguracijama za pojedini zadatak.

```
module.exports = function (grunt) {
      grunt.initConfig({ ... });
};
```

Sljedeći korak je definiranje zadataka koji će se automatizirati. Zadatak koji će se automatizirati je povezivanje (eng. *concat*) JavaScript i CSS datoteka. Zadatak povezivanja se automatizira kreiranjem `concat` objekta koji prima `js` i `css` objekte. Navedeni objekti sadrže putanju izvorišnih datoteka koje se definiraju u svojstvu `src`, te datoteku odredišta koja se definira svojstvom `dest`. Rezultati zadatka se spremaju u datoteku odredišta.

```
concat: {
      js: { src: ["...", "..."], dest: "...", },
      css: { src: ["...", "..."], dest: "...", },
},
```

Sljedeći zadatak koji će se automatizirati odnosi se na pretvaranje Sass odnosno Scss datoteka u CSS datoteke. Automatizacija će se postići kreiranjem novog objekta za zadatak pod nazivom `sass`, unutar kojeg se nalazi objekt `build` koji sadrži niz `files`. Niz `files` sadrži putanje do izvorišne datoteke `src` na kojoj će se izvršiti zadatak, te putanju do odredišne datoteke `dest` u kojoj će rezultati biti pohranjeni.

```
sass: {
      build: {
        files: [ { src: "...", dest: "..." }, ],
      },
},
```

Sljedeći korak konfiguracije je definiranje promatrača zadataka (eng. *watch task monitors*). Zadatak promatrača je praćenje promjena u definiranim datotekama, te na događaj promjena automatizirano pokretanje jednog ili više od definiranih zadataka.

```
watch: {
      js: { files: ["js/**/*.js"], tasks: ["..."], },
      css: { files: ["css/**/*.css"], tasks: ["..."], },
},
```

Nakon definiranja zadataka i promatrača potrebno je uključiti dodatke (eng. *plugins*). Dodaci omogućuju rad s definiranim zadacima. Dodaci se uključuju pomoću metode `loadNpmTasks`.

```
grunt.loadNpmTasks("grunt-contrib-concat");
```

Za kraj konfiguracije potrebno je sve zadatke registrirati pomoću metode `registerTask`.

```
grunt.registerTask("concat-js", ["conctat:js"]);
```

Nakon kreirane konfiguracije definirani zadatci se pokreću pomoću naredbe `grunt <ime_zadatka>`.

Konačna konfiguracija bila izgleda ovako:

```
module.exports = function (grunt) {
  grunt.initConfig({
    concat: {
      js: { src: ["...", "..."], dest: "...", },
      css: { src: ["...", "..."], dest: "...", },
    },
    sass: {
      build: {
        files: [ { src: "...", dest: "...", }, ],
      },
    },
    uglify: {
      build: {
        files: [ { src: "...", dest: "...", }, ],
      },
    },
    watch: {
      js: { files: ["..."], tasks: ["..."], },
      css: { files: ["..."], tasks: ["..."], },
    },
  });
```

```
  grunt.loadNpmTasks("...");
  grunt.registerTask("...", ["..."]);
};
```

3.2. Gulp

Gulp je pokretač zadataka (eng. *task runner*) koji također poput Grunt.js automatizira repetitivne poslove. Neovisno o platformi, integracije su ugrađene u sva glavna radna okruženja (eng. *integrated development environment*) i programeri koriste gulp s PHP-om, .NET-om, Node.js-om, Java-om i drugim platformama. Gulp posjeduje snažan ekosustav s više od 3000 mogućih dodataka za automatizaciju poslova [13], [14].

Neke od zadaća koje se automatiziraju pomoću Gulpa uključuju [14]: minifikaciju JavaScript i CSS datoteka, povezivanje datoteka, cache busting (tehniku koja se koristi za osiguravanje da preglednik učita najnoviju verziju datoteka s poslužitelja), razne vrste automatiziranih testova poput unit, integration, i acceptance testova, automatiziranu optimizaciju, linterske operacije, te pokretanje razvojnog servera.

3.2.1. Prednosti i nedostatci

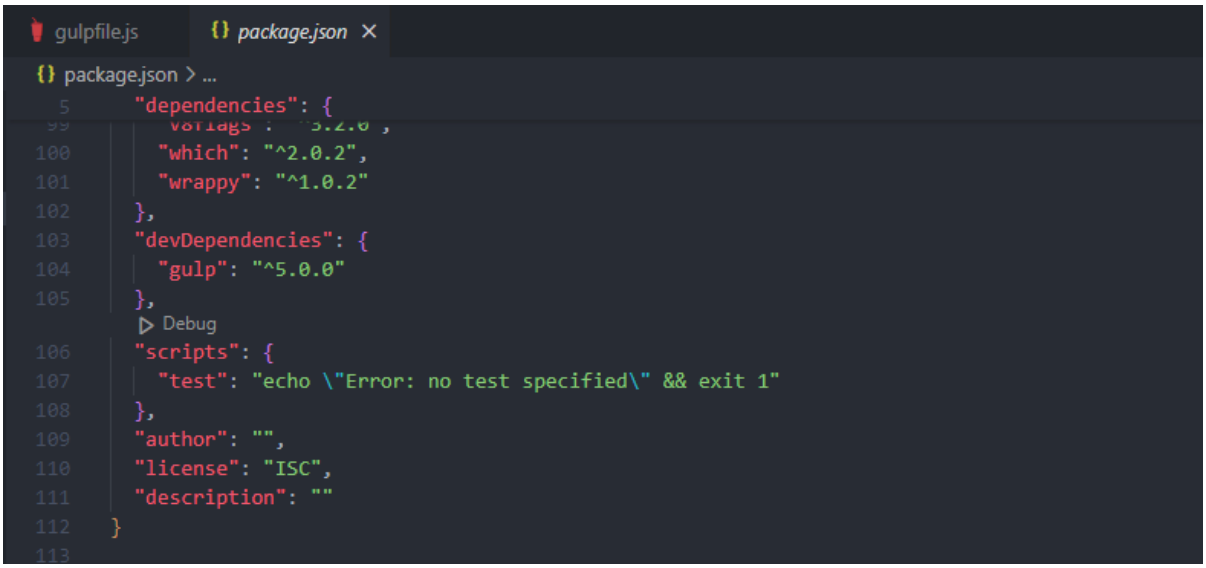
Korištenje alata poput Gulp.js nudi brojne prednosti, među kojima su najuočljivije jednostavna sintaksa i intuitivnost. Gulp.js koristi Node tokove (eng. *Node streams*) tijekom izvršavanja, što omogućuje upotrebu cjevovoda za spajanje i paralelno izvršavanje zadataka. Takav način obrade značajno povećava brzinu izvršavanja zadataka. Node tokovi optimiziraju korištenje memorije i omogućuju direktno spremanje rezultata u izlazne datoteke. Osim toga, Gulp.js je alat otvorenog koda, što znači da je dostupan za besplatnu upotrebu i ima širok spektar dodataka koji omogućuju prilagodbu za bilo koju vrstu zadatka [10], [11], [12].

Iako Gulp.js nudi mnoge prednosti, postoje i određeni nedostatci prilikom korištenja ovog alata. Jedan od glavnih problema je složenost node tokova, koji iako značajno ubrzavaju vrijeme izvršavanja, mogu biti prilično složeni za početnike [10], [11], [12]. Nadalje, korištenje takvog načina obrade i Promise-a može predstavljati veliki izazov za one koji nisu upoznati s ovim konceptima. Upravljanje Promise-ima zahtjeva dublje razumijevanje JavaScript jezika i može povećati krivulju učenja za nove korisnike.

3.2.2. Instalacija Gulp.js

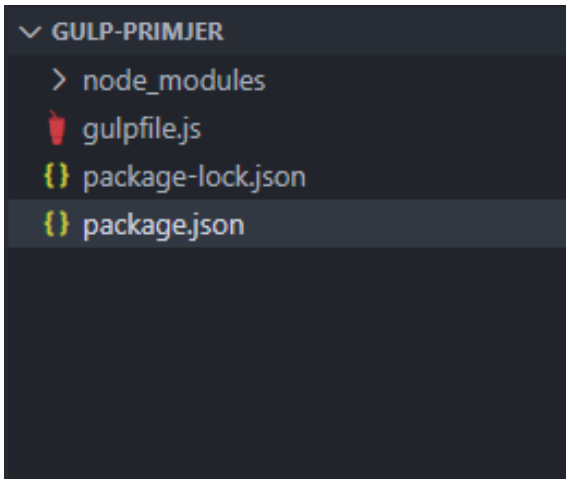
Za instalaciju Gulp.js potrebno je prvo instalirati Node.js i npm. Nakon instalacije Node.js-a i npm-a, Gulp se instalira globalno pomoću naredbe `npm install -g gulp`.

Kako bi se Gulp instalirao samo u razvojne ovisnosti (eng. *development dependencies*), koristi se naredba `npm install --save-dev gulp`.



Slika 3. Prikaz Gulp.js u razvojnim ovisnostima

Nakon što se Gulp instalira u projekt, potrebno je u korijenu projekta kreirati konfiguracijsku datoteku `gulpfile.js` koja će sadržavati konfiguraciju.



Slika 4. Prikaz gulpfile.js u korijenu projekta

3.2.3. Primjer konfiguriranja i korištenja gulpfile.js datoteke

U ovom poglavlju prikazat će se konfiguracija `gulpfile.js` datoteke koja će podržavati automatizirano pokretanje zadataka kao što su: povezivanje JavaScript i CSS datoteka, pretvaranje Scss odnosno Sass datoteka u CSS datoteke i minifikaciju JavaScript i CSS datoteka.

Prvi korak konfiguracije je kreiranje `gulpfile.js` datoteke. Datoteku je zatim potrebno pohraniti u korijen projekta. Nakon kreiranje datotekom, potrebno je uključiti gulp koristeći ključnu riječ require.

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

Kreirana gulp varijabla će omogućiti rad s nizom različitih poslova poput: definiranja zadataka, definiranje putanja izvorišnih i odredišnih datoteka, korištenje dodataka, automatsko praćenje promjena, upravljanje uključivanjem (eng. *import*) i izvozom (eng. *export*) i slično.

Prije kreiranja zadataka, potrebno je uključiti dodatke koji omogućuju automatizaciju poslova ključnom riječi `require`. Dodaci se uključuju na sljedeći način: `const <ime_varijable_dodatka> = require("<dodatak>");`.

Zadaci koji će se automatizirati, poput povezivanja i minifikacije (eng. *concat*) JavaScript i CSS datoteka kreiraju se pomoću metode `task`. Task metoda prima kao argumente ime zadatka i funkciju koja obavlja razne operacije poput obrade, minifikacije, povezivanja i slično. Unutar funkcije, potrebno je definirat i izvornu datoteku `src`. Dalje, potrebno je povezati korištene dodatke (eng. *plugins*) koristeći `pipe`. Metoda `pipe` prima `concat` za spajanje datoteka u `main.js` datoteku i `uglify` za minifikaciju. Na kraju, potrebno je spremiti obrađene datoteke u direktorij `dist/js`.

```
gulp.task("concat", () => {
  gulp
    .src("src/js/*.js")
    .pipe(concat("main.js"))
    .pipe(uglify())
    .pipe(gulp.dest("dist/js"));
});
```

Zadatak praćenja promjena, također se definira pomoću metode task, pridružujući zadatku naziv `watch`. Unutar funkcije, potrebno je koristiti metodu `watch`. Metoda prima dva argumenta, prvi argument predstavlja izvorišnu putanju do datoteka koje treba pratiti, dok drugi argument predstavlja naziv zadatka koji će se automatizirano pokretati na promjene u navedenim datotekama.

```
gulp.task("watch", () => {
     gulp.watch("src/js/*.js", ["concat"]);
     gulp.watch("src/sass/*.scss", ["sass"]);
});
```

Definirane zadatke moguće je pokrenuti putem naredbenog retka naredbom `gulp <ime_zadatka>`. Konačna konfiguracija gulpfile.js datoteke izgleda ovako:

```
const gulp = require("gulp");
const uglify = require("gulp-uglify");
const concat = require("gulp-concat");

gulp.task("sass", () => {
  gulp
    .src("src/sass/*.scss")
    .pipe(sass().on("error", sass.logError))
    .pipe(gulp.dest("dist/css"));
});

gulp.task("concat", () => {
  gulp.src("src/js/*.js")
    .pipe(concat("main.js"))
    .pipe(uglify())
    .pipe(gulp.dest("dist/js"));
});

gulp.task("watch", () => {
  gulp.watch("src/js/*.js", ["concat"]);
});
```

3.3. Webpack

Webpack, osim što je pokretač zadataka (eng. *task runner*), također sakupljač statičkih modula, poznat kao „*module bundler*", te je moćniji alat u odnosu na prethodno obrađene Gulp.js i Grunt.js. Prilikom obrade, Webpack gradi grafikon ovisnosti (eng. *dependency graph*) iz jedne ili više ulaznih točaka, te zatim kombinira svaki modul koji je projektu potreban u jedan ili više paketa, koji postaju statička sredstva za posluživanje sadržaja. Webpack podržava sve preglednike koji su usklađeni s ES5. Webpack koristi Promise za `import()` i `require.ensure()`. Ako je neophodno da projekt podržava starije preglednike, Webpack će zahtijevati učitavanje polifila prije njegove upotrebe [15], [16].

Neki od zadataka koje Webpack omogućuje su [16]: obrada CSS datoteka, uključujući učitavanje i ugrađivanje CSS datoteka unutar JavaScript datoteka. Također, pretvaranje Sass odnosno Scss datoteka u izvorne CSS datoteke. Webpack također omogućuje upravljanje statičkim datotekama kao što su slike i fontovi, i omogućuje minifikaciju JavaScript i CSS datoteka. Omogućuje i dijeljenje koda (eng. *code splitting*), razdvajanje

koda u manje pakete s ciljem poboljšanja performansi aplikacije. Podržane su i mogućnosti generiranja HTML datoteka, definiranja varijabla okruženja (eng. *environment variables*) i mnogo više.

3.3.1. Prednosti i nedostatci

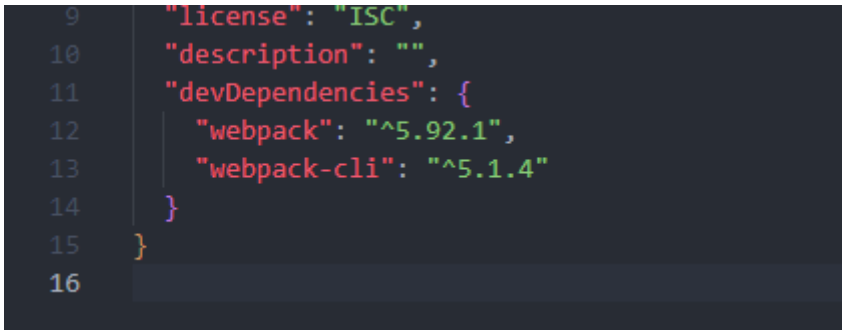
Za razliku od prethodno obrađenih alata kao što su Grunt.js i Gulp.js, Webpack nije samo pokretač zadataka već je sakupljač statičkih modula što nudi širi spektar funkcionalnosti i time pruža brojne dodatne prednosti. Jedna od ključnih prednosti Webpacka je korištenje učitavača (eng. *loaders*). Učitavači omogućuju Webpacku pretvaranje različitih vrsta datoteka u format koji je kompatibilan s preglednicima [15], [16]. Na primjer, Webpack može pretvoriti ES6 JavaScript datoteke u ES5 datoteke prikladne za starije preglednike. Još jedna značajna prednost Webpacka je dijeljenje koda (eng. *code splitting*). Ova funkcionalnost omogućuje razdvajanje koda na manje dijelove koji se mogu asinkrono učitavati. Time se značajno ubrzava vrijeme učitavanja stranica, jer se samo potrebni dijelovi koda učitavaju odmah. Funkcionalnosti koje nisu dostupne u jednostavnijim alatima poput Grunta i Gulpa uključuju automatsko učitavanje (eng. *hot reload*). Automatsko učitavanje omogućuje uvid u promjene u kodu odmah nakon što se naprave, bez potrebe za ponovnim učitavanjem stranice. Nadalje, Webpack omogućuje pokretanje lokalnog razvojnog servera korištenjem paketa `webpack-dev-server`. Kao Grunt i Gulp, Webpack ima veliku zajednicu korisnika i bogat ekosistem koji se redovito održava i poboljšava.

Iako Webpack nudi brojne prednosti, također dolazi i s određenim nedostatcima, posebno u usporedbi s jednostavnijim alatima poput Grunta i Gulpa. Glavne slabosti Webpacka uključuju: visoku krivulju učenja, kompleksnu konfiguraciju i zahtjevan inicijalni proces postavljanja alata za nove projekte. Zbog svoje kompleksnosti i brojnih funkcionalnosti, Webpack može postati prilično kompliciran za početnike. Kompleksnost konfiguracije Webpacka je također jedan od izazova. Za razliku od jednostavnijih alata koji koriste lako razumljive konfiguracijske datoteke, Webpack konfiguracijske datoteke mogu biti složene i teške za razumijevanje [15], [16]. Postavljanje konfiguracije koja će pokrivati sve problematike novog projekta može zahtijevati detaljno proučavanje dokumentacije i razumijevanje različitih opcija i postavki koje Webpack nudi. Inicijalno postavljanje Webpacka za novi projekt može biti i vremenski intenzivno. Proces postavljanja svih potrebnih učitavača, dodataka i konfiguracijskih postavki može oduzeti puno vremena. Kod velikih projekata, brzina izvršavanja Webpacka može biti značajno sporija u odnosu na alata poput Grunt i Gulp s obzirom na kompleksnost procesa koje Webpack obavlja prilikom izvršavanja i primjenjivanja promjena.

3.3.2.Instalacija Webpacka

Prvi korak instalacije Webpack alata je instalacija Node.js-a i npm-a. Nakon instalacije Webpack je moguće instalirati globalno pomoću naredbe `npm install -g webpack`.

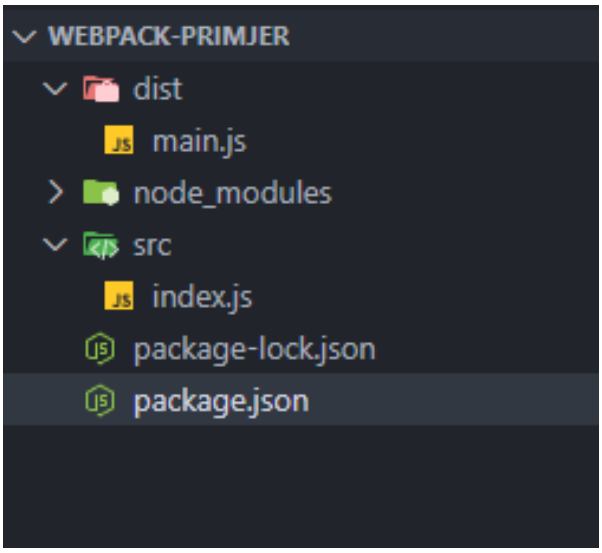
Webpack se instalirava lokalno u projekt u razvojne ovisnosti naredbom `npm install –save-dev webpack` za najnoviju stabilnu verziju, no može se instalirati i jedna od ranijih verzija naredbom `npm install –save-dev webpack@<verzija>`. Instalacija starijih verzija nije preporučljiva jer mogu sadržavati sigurnosne propuste. Sljedeći korak instalacije je instalacija webpack-cli, pomoću koje će se raditi s Webpackom. webpack-cli se instalira lokalno u razvojne ovisnosti koristeći naredbu `npm install --save-dev webpack-cli`.



Slika 5. Prikaz webpack i webpack-cli u razvojnim ovisnostima

Webpack se nakon instalacije pokreće putem naredbene linije unosom npm skripta. Tvornički, Webpack dolazi sa zadanom konfiguracijom koja se može pokrenuti kreiranjem skripte u package.json datoteci: `build: webpack`. Prije pokretanja potrebno je kreirati potrebnu strukturu projekta, naime Webpack zahtjeva određenu strukturu projekta kako bi zadana konfiguracija mogla raditi. Struktura projekta koju Webpack zahtjeva sadrži `src` direktorij s `index.js` datotekom koja služi kao ulaz u korijenu projekta. Kreirana `build` skripta se zatim može pokrenuti naredbom `npm run build`.

Slika 6. Prikaz build skripte u package.json datoteci



Slika 7. Prikaz strukture projekta

Vlastitu webpack konfiguraciju sa zadacima specifičnim za projekt, moguće je stvoriti kreiranjem i konfiguriranjem `webpack.config.js` datoteke u korijen (eng. *root*) projekta.

3.3.3. Glavni koncepti

Razumijevanje Webpack-ovih glavnih koncepata, ključno je za efikasno upravljanje i optimiziranje našeg koda i projekta u fazama razvoja. Ovaj dio rada osvrnut ću se na glavne koncepte unutar Webpack alata i njihovo korištenje.

3.3.3.1. Ulazna točka

Ulazna točka pokazuje koji modul webpack treba koristiti za početak izgradnje svog internog grafikona ovisnosti. Webpack će otkriti o kojim drugim modulima i bibliotekama ta ulazna točka ovisi (izravno i neizravno) [17].

Inicijalna vrijednost ulazne točke je `./src/index.js`, što znači da je u korijenu projekta potrebno kreirati direktorij `/src` te unutar direktorija `index.js` datoteku koja se želi optimizirati. Ulazna točka definira se u `webpack.config.js` datoteci u svojstvu `entry`. Svojstvo `entry` definira putanju do ulazne točke.

```
module.exports = {
  entry: "<putanja_do_ulazne_točke>",
}
```

3.3.3.2. Izlaz

Svojstvo izlaza govori webpacku gdje spremiti pakete koje stvara i kako imenovati te datoteke. Zadana je `./dist/optimizirana_datoteka.js` za glavnu izlaznu datoteku i direktorij `./dist` za bilo koju drugu generiranu datoteku [17].

Generirane datoteke sadrže sve optimizacije definirane u datoteci `webpack.config.js`. Definirane optimizacije ili drugačija putanju može se prilagoditi pomoću svojstva `output` koji je objekt s različitim svojstvima koja možemo iskoristiti, primjerice: `path` i `filename` [17]. Praktičan primjer za promjenu putanje izlaza je konfigurirati Webpack da generira sve izlazne datoteke izravno u direktorij servera što osigurava da, nakon integracije našeg klijentskog projekta s lokalnim serverom, nećemo morati ručno kopirati datoteke svaki put kada napravimo promjene.

```
module.exports = {
  entry: "./src/index.js",
  output: {
    path: path.resolve(__dirname, "dist"),
    filename: "optimizirana_datoteka.js",
  },
};
```

3.3.3.3. Učitavači

Webpack inicijalno nakon instalacije razumije samo JavaScript i JSON datoteke. Učitavači (eng. *loaders*) omogućuju webpacku da obradi druge vrste datoteka i pretvori ih u valjane module koje aplikacija može iskoristiti i dodati u graf ovisnosti [17].

Na visokoj razini, učitavači (eng. *loaders*) imaju dva svojstva u konfiguraciji webpacka: `test` svojstvo identificira koju datoteku ili datoteke treba transformirati. Svojstvo `use` označava koji bi se učitavač trebao koristiti za transformaciju [17].

Učitavačima su definirane vrste datoteka koje se žele obraditi i na koji ih je način potrebno optimizirati pomoću Webpacka. U konfiguraciji je u svojstvu `module` potrebno dodati svojstvo `rules` koje predstavlja niz objekata u obliku pravila, svako pravili sadrži svojstvo `test` unutar kojeg je definirana vrsta datoteke i svojstvo `use` unutar kojeg je priključen učitavač (eng. *loader*). Primjer samo nekih od datoteka koje se često optimiziraju putem Webpack alata su: .ts, .tsx, .js, .jsx, .scss, .css, .txt, .svg, .jpg, .png, .gif, .webp.

```
module: {
    rules: [
      {  test: ..., use: "...", exclude: ... }, },
      {  test: ..., use: [..., "..."] },
    ],
},
```

3.3.3.4. Dodaci

Dok se učitavači koriste za transformaciju određenih vrsta modula, dodaci se mogu koristiti za izvođenje šireg raspona zadataka poput: optimizacije node paketa (eng. *node modules*), upravljanja svojstvima (eng. *property*) i ubacivanja varijabli okruženja (.env datoteke) [17].

Da bi se dodatak mogao koristiti, potrebno ga je priključiti korištenjem ključne riječi `require` i dodati u niz dodataka. Većina dodataka može se prilagoditi putem opcija. Budući da se dodatak može koristiti više puta u konfiguraciji za različite svrhe, potrebno je stvoriti instancu pozivanjem operatora `new` [17].

Webpack pruža širok spektar dodataka, a neki od najčešće korištenih su sljedeći. HtmlWebpackPlugin dodatak automatski generira HTML datoteke s već dodanom `<script>` oznakom koja sadrži putanju do optimizirane JavaScript datoteke. MiniCssExtractPlugin služi za optimizaciju CSS datoteka, čime se poboljšava učitavanje stranica i brzina rada aplikacije. TerserPlugin se koristi za minifikaciju JavaScript datoteka, što smanjuje njihovu veličinu i poboljšava performanse aplikacije. CleanWebpackPlugin, prije svake generacije datoteka naredbom `build`, čisti izlazni direktorij kako bi se svaki put svježe dodale najnovije generirane datoteke.

```
plugins: [
    new CleanWebpackPlugin(),
    new HtmlWebpackPlugin({ template: "./src/index.html" }),
    new MiniCssExtractPlugin({ filename: "style.css" }),
  ],
  optimization: {
    minimize: true,
    minimizer: [
      new TerserPlugin({
        minify: true,
        parallel: true,
      }),
```

```
    ],
  },
```

3.3.3.5. Način rada

Postavljanjem parametra načina rada na razvoj (eng. *development*), proizvodnju ili ništa, omogućuje ugrađene optimizacije webpacka koje odgovaraju svakoj vrsti okruženju. Inicijalna zadana vrijednost nakon instalacije je proizvodnja (eng. *production*) [17].

Način rada definira se svojstvom `mode`. Svojstvu se definira vrijednost "*production*" za proizvodni način rada ili "*development*" za razvojni način rada. Moguće je kreirati dvije različite webpack konfiguracije stvaranjem dodatne `webpack.config.js` datoteke, koristeći jednu za razvoj, a drugu za produkciju. Alternativno, moguće je definirati različite skripte u datoteci package.json koje mijenjaju način rada unutar svojstva `mode` u webpack konfiguracijskoj datoteci.

```
mode: '<način_rada>'
```

3.3.4. Primjer konfigurirane webpack.config.js datoteke

U nastavku je prikazan primjer konfigurirane `webpack.config.js` datoteke.

```
const path = require("path");
const HtmlWebpackPlugin = require("html-webpack-plugin");
const { CleanWebpackPlugin } = require("clean-webpack-plugin");
const MiniCssExtractPlugin = require("mini-css-extract-plugin");
const TerserPlugin = require("terser-webpack-plugin");
module.exports = {
  entry: "./src/index.js",
  output: {
    path: path.resolve(__dirname, "dist"),
    filename: "optimizirana_datoteka.js",
  },
  mode: "development",
  module: {
    rules: [
      { test: /\.tsx?$/, use: "ts-loader", exclude: /node_modules/
},
      { test: /\.jsx?$/, use: "babel-loader", exclude:
/node_modules/ },
      { test: /\.css$/, use: [MiniCssExtractPlugin.loader, "css-
loader"] },
```

```
      {
        test: /\.scss$/,
        use: [MiniCssExtractPlugin.loader, "css-loader", "sass-
loader"],
      },
      { test: /\.txt$/, use: "raw-loader" },
      {
        test: /\.(png|jpg|gif|svg)$/,
        use: [
          {
            loader: "file-loader",
            options: { name: "[path][name].[ext]" },
          },
        ],
      },
    ],
  },
  plugins: [
    new CleanWebpackPlugin(),
    new HtmlWebpackPlugin({ template: "./src/index.html" }),
    new MiniCssExtractPlugin({ filename: "style.css" }),
  ],
  optimization: {
    minimize: true,
    minimizer: [
      new TerserPlugin({
        minify: true,
        parallel: true,
      }),
      new OptimizeCSSAssetsPlugin({}),
    ],
  },};
```

4. Usporedba alata

U ranim fazama razvoja web aplikacija, kada su projekti bili manji i manje složeni, programeri nisu imali potrebu za korištenjem alata poput Grunta, Gulpa ili Webpacka. U to vrijeme, preglednici su mogli izravno izvršavati kod bez potrebe za dodatnom obradom. Međutim, kako su projekti postajali sve veći i kompleksniji, pojavili su se novi izazovi koji su zahtijevali povećanje produktivnosti, poboljšanje kvalitete aplikacija te jačanje sigurnosti. Kako bi se nosili s tim izazovima, developeri su razvili alate za automatizaciju, kao što su Webpack, Grunt.js i Gulp.js, koji su osmišljeni kako bi rasteretili programere od raznih rutinskih i vremenski intenzivnih zadataka [10], [11], [12]. Grunt.js i Gulp.js alati su pokretači zadataka (eng. *task runners*) za razliku od webpacka koji je sakupljač statičkih modula (eng. *module bundler*). Pokretači zadataka automatski izvršavaju procese iza štedeći posao i vrijeme, dok sakupljači modula obavljaju zadatke pokretača zadataka i poduzimaju nekoliko koraka dalje. Grunt.js i Gulp.js, iako rade ili mogu raditi isti posao rade ga na različite načine.

Kako bi se dobio bolji uvid u razlike među alatima Webpack, Grunt.js i Gulp.js, provedeno je istraživanja koje će usporediti kriterije kao što su: jednostavnost korištenja, brzina izvršavanja, popularnost te konačna veličina obrađenih datoteka. Cilj analize je pružiti detaljan pregled prednosti i nedostataka svakog alata te pomoći u odabiru najprikladnijeg alata za specifične potrebe.

4.1. Jednostavnost korištenja

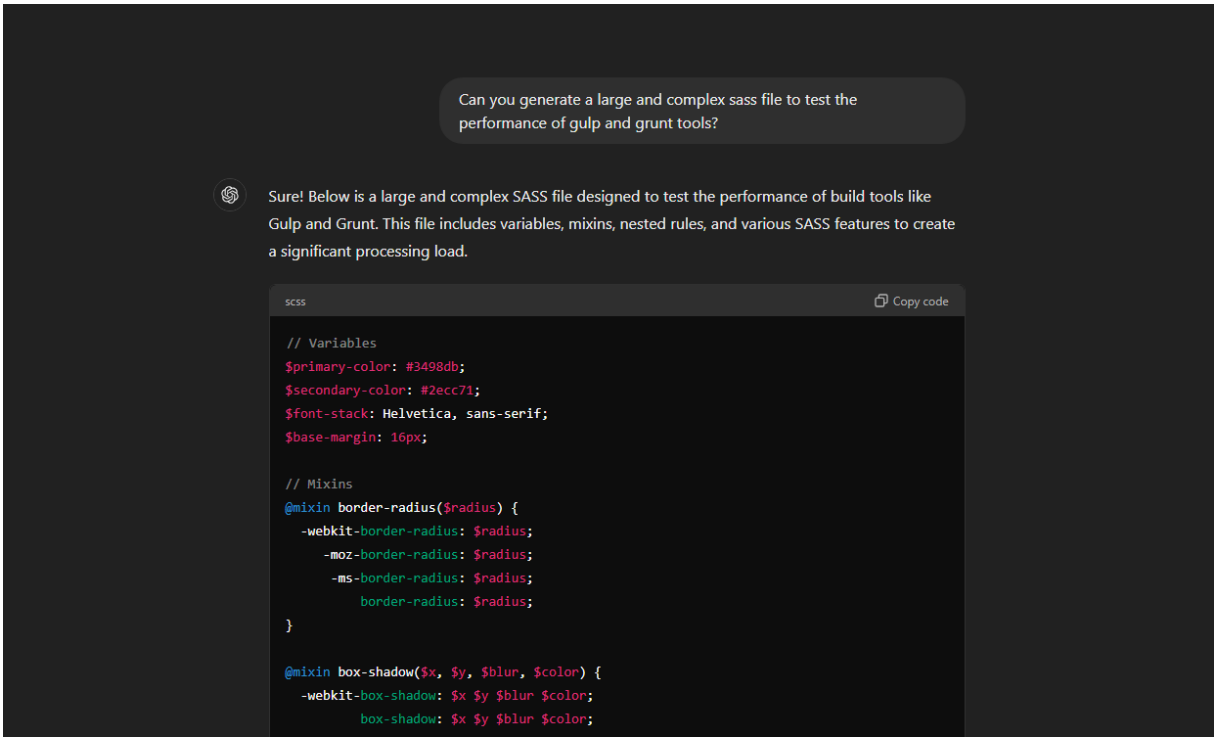
Gulp.js koristi Node tokove koji omogućuju kreiranje cjevovoda za spajanje i paralelno izvršavanje više zadataka, dok Grunt.js koristi datotečni sustav za sinkrono izvršavanje zadataka. Grunt.js prilikom pokretanja kreira privremene datoteke, dok Gulp.js upravlja zadacima u memoriji i rezultate sprema direktno u konačne datoteke. Memorijske operacije su značajno brže od operacija nad datotečnim sustavom, što sugerira da je Gulp.js učinkovitiji i brži alat od Grunt.js-a [10], [11], [12].

Konfiguracija webpacka, iako može biti složena, pruža izuzetno visoku razinu fleksibilnosti u odnosu na Grunt ili Gulp. Konfiguracijska datoteka `webpack.config.js` omogućava potpunu kontrolu nad procesom izgradnje, što olakšava razvoj i održavanje kompleksnih projekata [10], [11], [12]. Webpack može obavljati razne zadatke bez potrebe za aplikacijama treće strane, čineći ga potpunim rješenjem za mnoge aspekte razvoja web aplikacija. Jedna od većih prednosti Webpack alata naspram alata poput Grunt ili Gulp je `webpack-dev-server`. Prema [15], Webpack razvojni server omogućuje pokretanje

lokalnog servera s automatskim spremanjem, povezivanjem ovisnosti i osvježavanjem stranica prilikom promjena u kodu, što značajno ubrzava i pojednostavljuje razvojni proces. Prilikom odabira alata za jednostavnost korištenja preporučuje se Gulp.js.

4.2. Brzina izvršavanja

Prema testiranju koje je proveo TMWtech [11], u kojem se uspoređivala kompilacija Sass datoteka koristeći alate Gulp.js i Grunt.js, rezultati su pokazali da je Gulp.js bio gotovo dvostruko brži od Grunt.js-a, s vremenom od 1,3 sekunde naspram 2,3 sekunde potrebne za Grunt.js. Kako bi se potvrdili rezultati testiranja provedenih od strane TMWtech, provedeno je dodatno testiranje kompilacije Sass datoteke. U svrhu testiranja korišten je generativni AI alat ChatGPT za generaciju kompleksne Sass, odnosno Scss datoteke koja će se obraditi pojedinim alatom.



Slika 8. Upit generativnom AI alatu ChatGPT za generaciju Sass datoteke

Rezultati provedenog testiranja su dokazali da je Gulp.js nakon pet provedenih kompilacija bio dvostruko brži u odnosu na Grunt.js.

Grunt.js je dulje prisutan na tržištu i ima širi spektar dodataka, ali kada je brzina ključni kriterij, preporučuje se korištenje Gulpa. Prilikom odabira alata, važno je uzeti u obzir kompleksnost projekta i potrebu za kompleksnim alatom kao što je Webpack. Webpack je posebno pogodan za složene projekte s velikim brojem ovisnosti. S druge strane, za

jednostavne i manje projekte s manjim brojem ovisnosti, preporučuje se korištenje Gulpa, koji je jednostavniji za upotrebu i lakši za konfiguraciju.

Tablica 1. Rezultati testiranja Grunt.js i Gulp.js alata

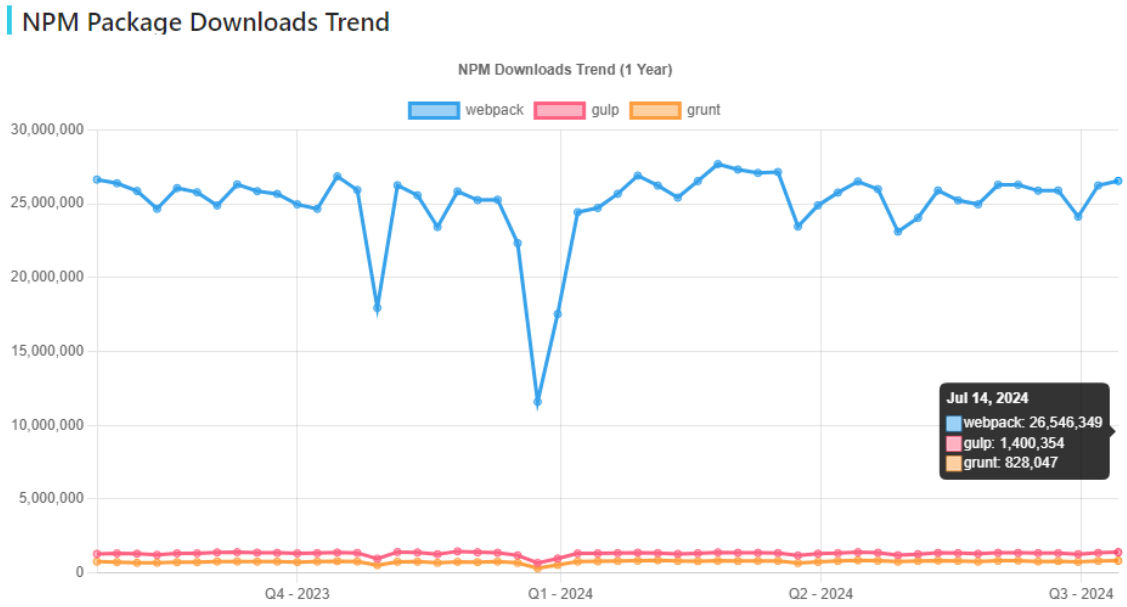
Broj mjerenja	Grunt	Gulp
1.	2.3 sek	821 ms
2.	2.2 sek	976 ms
3.	2.1 sek	878 ms
4.	2.2 sek	845 ms
5.	2.2 sek	830 ms
Prosjek	**2.2 sek**	**870 ms**

4.3. Popularnost

Grunt.js bio je jedan od prvih alata za automatizaciju zadataka u Node.js okruženju. S druge strane, Gulp.js, razvijen je od strane Blainea Bublitza i Erica Schoffstalla, te predstavljen 2013. godine. Unatoč kasnijem dolasku uspio je nadmašiti Grunt.js u pogledu popularnosti. Prema podacima dostupnim na npm-compare stranici, na dan 14. lipnja 2024. godine, Gulp.js bilježi više od milijun i četiristo tisuća preuzimanja, dok Grunt.js ima nešto više od 828 tisuća preuzimanja. Što se tiče učestalosti ažuriranja, Grunt.js je posljednji put ažuriran sredinom 2023. godine, dok je Gulp.js doživio posljednje ažuriranje početkom 2024. godine [11], [18], [19]. Uzimajući u obzir visoku popularnost i aktivnu podršku zajednice, preporuka za izbor alata za automatizaciju zadataka je Gulp.js.

Nadalje, Webpack je jedan od najpopularnijih alata za razvoj JavaScript projekata i sadrži značajnu i aktivnu podršku zajednice koja kontinuirano doprinosi njegovom unapređenju i ispravljanju grešaka. S obzirom na njegovu popularnost i širok spektar mogućnosti koje nudi, interes za učenje i korištenje webpacka i dalje raste. Webpack ne samo da pruži učinkovito upravljanje modulima i njihovim ovisnostima, već pruži i visoku razinu prilagodbe konfiguracije. Također, webpack je integriran u Angular okvir, gdje se automatski instalira prilikom preuzimanja Angulara putem npm alata [11], [12], [18]. Osim toga, prema podacima s npm-compare stranice, Webpack ima znatno veći broj preuzimanja u usporedbi s Gruntom i

Gulpom, što dodatno potvrđuje njegovu popularnost u odnosu na druge alate za automatizaciju procesa prilikom razvoja web aplikacija.



Slika 9. Popularnost preuzimanja prema npm-compare.com

5. Opis razvijene web aplikacije

Aplikacija razvijena za praktični dio rada je web aplikacija za upravljanje svakodnevnim poslovima u restoranu. Aplikacija se sastoji od klijentske i poslužiteljske strane, klijentska strana je izrađena koristeći Typescript, React.js i Webpack alat za automatizaciju, dok je poslužiteljska strana razvijena koristeći Node.js i Express okvir.

Web aplikacija sadrži niz korisnih funkcionalnosti, uključujući, registraciju i prijavu korisnika, pregled jelovnika, izvršavanje i pregled rezervacija za određeni vremenski okvir, ali i za cijeli dan u svrhu posebnih prilika. Nadalje, aplikacija omogućuje funkcionalnost naručivanja hrane, pregled aktivnih i prošlih narudžbi, pregled statistika te upravljanje profilima korisnika. Aplikacija podržava različite vrste korisnika, a svaka vrsta korisnika ima pristup specifičnim funkcionalnostima u skladu sa svojom ulogom. Definirane uloge uključuju gost korisnika, registriranog korisnika, konobara, kuhara i administratora. Uloga gost sadrži funkcionalnosti: pregled početne stranice i jelovnika, standardnu rezervaciju, rezervaciju za posebne prigode, registraciju i prijavu u sustav. Uloga korisnik ima pristup svim funkcionalnosti uloge gost, te funkcionalnosti profila, pregled prošlih rezervacija i automatsko popunjavanje podataka prilikom rezervacije. Uloga konobar sadrži funkcionalnosti: pregled aktivnih rezervacija, pregled jelovnika, dodavanje novih jela i pića, pregled aktivnih narudžba, te mijenjanje stanja aktivnih narudžba. Uloga kuhar sadrži funkcionalnosti: pregled jelovnika, dodavanje novih jela i pića, te pregled i mijenjanje stanja aktivnih narudžba. Uloga administrator ima pristup svim funkcionalnostima dostupnim drugim ulogama, što omogućuje upravljanje i nadzor nad poslovanjem restorana. Kompletan kod implementirate aplikacije dostupan je na sustavu FOI radovi u zip arhivi

5.1. Tehnologije

Poslužiteljska strana aplikacije razvijena je koristeći Node.js, Express, te PostgreSQL za bazu podataka, dok je za klijentsku stranu korišten Typescript, s React.js-om i Webpack za automatizaciju. Za razvoj klijentske strane osim Reacta i Webpacka korištene su i druge biblioteke poput Redux Toolkita za upravljanje stanjem (eng. *state management*), `react-toastify` za prikaz tost notifikacija o uspješnim, odnosno neuspješnim zahtjevima i `react-datepicker` za prikaz kalendara. Na poslužiteljskoj strani, uz Node.js i Express, korištene su i biblioteke kao što su `jsonwebtoken` za autentifikaciju korisnika, `bcrypt` za raspršivanje (eng. *hashing*) lozinki prilikom registracije i provjere lozinke prilikom prijave, te `nodemailer` za slanje elektroničke pošte prilikom registracije i rezervacije. Razvojna okolina korištena za projekt je bila Visual Studio Code. Za testiranje aplikacije korišteni su Mozilla Firefox preglednik

s dodatkom React Developer Tools i Google Chrome preglednik za provjeru performansi aplikacije koristeći Lighthouse alat.

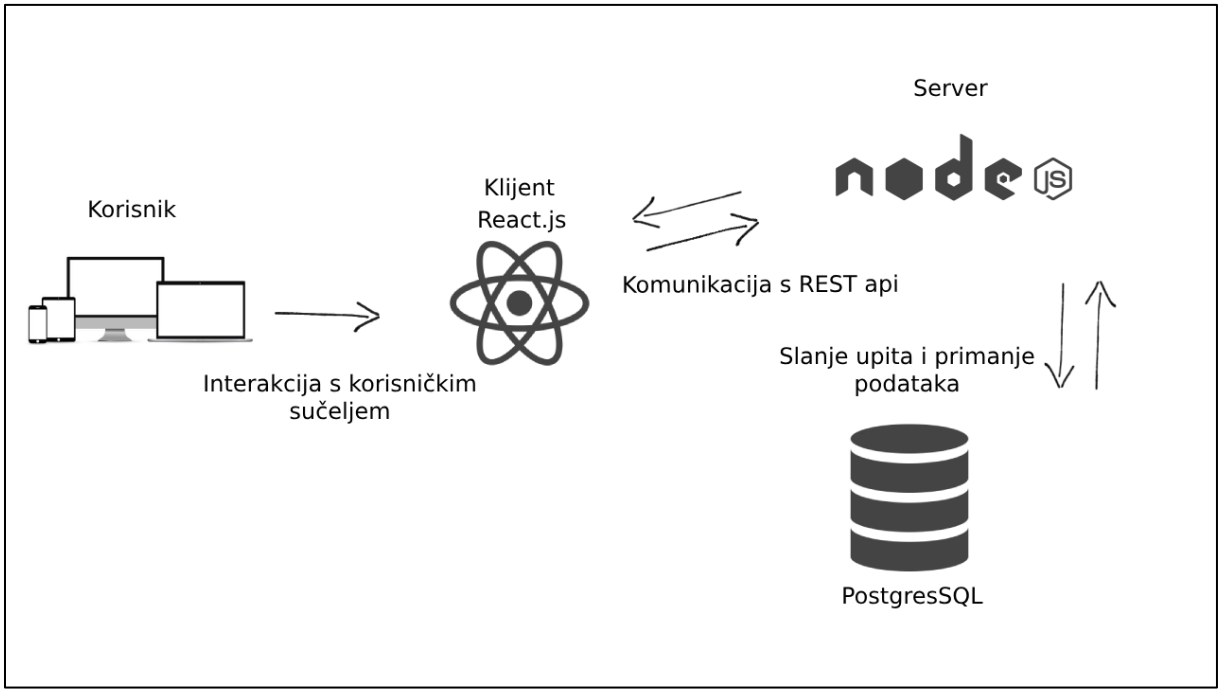
5.2. Arhitektura

Arhitektura aplikacije je postavljena tako da klijentska i poslužiteljska strana komuniciraju putem HTTP zahtjeva koristeći REST api. Poslužiteljska strana zatim komunicira s PostgreSQL bazom podataka. React komponente predstavljaju korisničko sučelje za interakciju korisnika s aplikacijom. Kada korisnik poduzme neku akciju, šalje se odgovarajući HTTP zahtjev poslužiteljskoj strani. Poslužiteljska strana koristi Node.js i Express za definiranje API putanja i upravljanje HTTP zahtjevima. Svaka putanja pripada specifičnom zahtjevu aplikacije, npr `/api/orders` je putanja za narudžbe. Za autentifikaciju korisnika koristi se jwt token i bcrypt za raspršivanje (eng. *hashing*) lozinki. Prilikom prijave server generira jwt token i vraća ga klijentu. Klijent zatim koristi taj token za autentifikaciju prilikom slanja daljnjih zahtjeva. Za bazu podataka koristi se PostgreSQL. Komunikacija s bazom ostvaruje se putem odgovarajućih servisa za upravljanje tablicama u bazi podataka.

Struktura projekta se sastoji od klijentske i poslužiteljske strane, koje su organizirane u odgovarajuće direktorije. Poslužiteljska strana smještena je u direktoriju `server` i sastoji se od nekoliko dijelova. Datoteka `server.js` sadrži poslužiteljsku logiku za kreiranje servera, definiranje API putanja i posluživanje klijentskog dijela. Uz `server.js` datoteku, poslužiteljska strana uključuje direktorij `node_modules` koji sadrži sve potrebne Node.js module, direktorij `helpers` s pomoćnim datotekama za upravljanje bibliotekama poput `nodemailer` i `jsonwebtoken`, te direktorij `servis` koji sadrži datoteku `database.js` s klasom za komunikaciju s bazom podataka i zasebne direktorije s datotekama za upravljanje svakom od tablica u bazi podataka.

Klijentska strana projekta smještena je u direktoriju `app` i sastoji se od nekoliko datoteka i direktorija. Glavne datoteke i direktoriji u app direktoriju su `webpack.config.js`, `tsconfig.json` i `package.json` datoteke i `public` i `src` direktoriji. Datoteke `webpack.config.js` i `tsconfig.json` definiraju konfiguracije za Webpack i TypeScript. Direktorij `public` sadrži rezultate pokretanja Webpacka i ikonu aplikacije, dok direktorij src sadrži sav kod za klijentski dio aplikacije. U direktoriju `src` nalaze se poddirektoriji: `assets` koji sadrži slike i ikone korištene u aplikaciji, `authorization` koji sadrži TypeScript datoteku s putanjama i dozvolama aplikacije, `components` koji sadrži sve korištene komponente, `helpers` s pomoćnim datotekama, `hooks` koji sadrži logiku za slanje HTTP zahtjeva poslužiteljskoj strani, `layout` s komponentom za glavni raspored aplikacije, `pages` koji sadrži

glavne stranice aplikacije, `state` koji sadrži logiku za Redux Toolkit i `validation` koji sadrži datoteke za validaciju različitih vrsta unosa. Uz navedene direktorije, direktorij app sadrži i glavne početne datoteke klijentskog dijela kao što su `main.tsx` kao glavna datoteka aplikacije, `App.tsx` kao glavna komponenta aplikacije i `index.html` kao glavna HTML datoteka aplikacije.



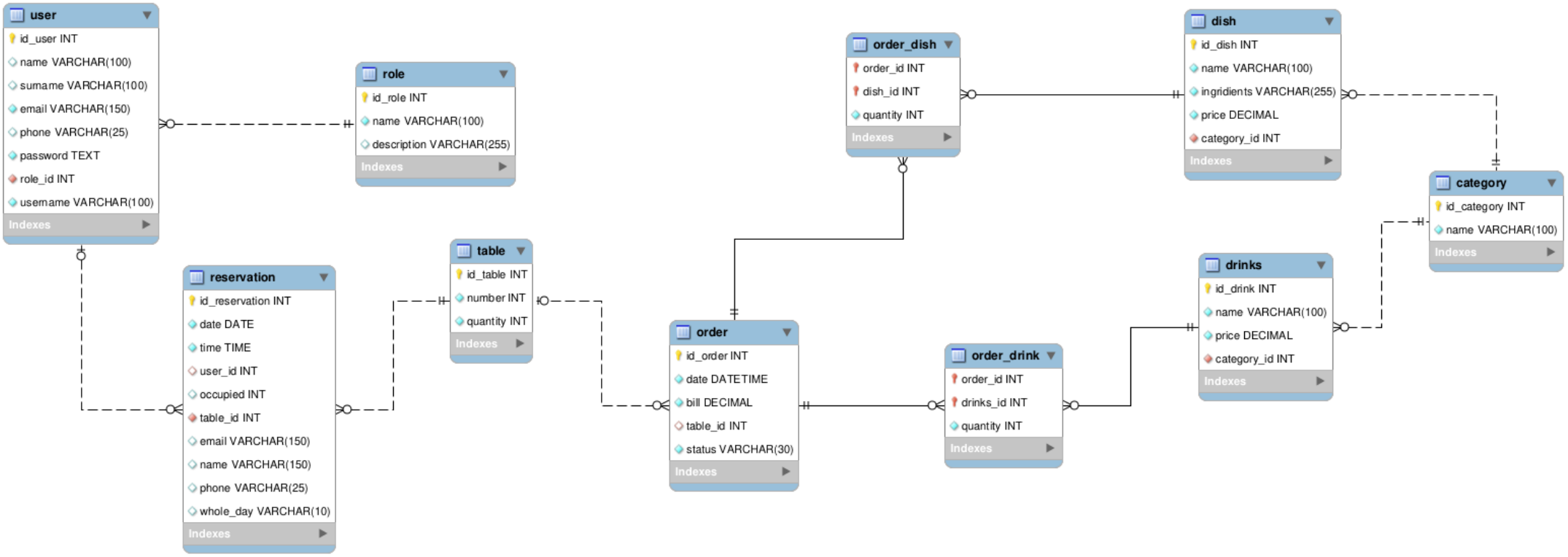
Slika 10. Arhitektura web aplikacije

5.3. Baza podataka

Za bazu podataka odabrana je PostgreSQL baza podataka s pgAdmin4 admin sučeljem za rukovanje. Baza podataka sadrži jedanaest tablica koje omogućuju upravljanje različitim vrstama korisnika poput: admin, gost, kuhar, konobar i registrirani korisnik. Nadalje tablice omogućuju upravljanje rezervacijama, narudžbama, jelima i pićima, te kategorizaciju jela i pića. Tablica `user` sadrži informacije o korisnicima, uključujući ime, prezime, email, telefon, lozinku, korisničko ime i ulogu. Ova tablica je povezana s tablicom `role` putem vanjskog ključa `role_id`, što omogućava dodjeljivanje različitih uloga korisnicima, kao što su navedeni administrator, konobar, kuhar, gost i registrirani korisnik. Tablica `reservation` sadrži podatke o rezervacijama stolova. Informacije koje su obuhvaćene su datum i vrijeme rezervacije, identifikator korisnika koji je napravio rezervaciju, identifikator stola, email adresa korisnika i stupac `whole_day` za indikaciju rezervacije restorana na cijeli dan. Ova tablica sadrži veze s tablicama `user` i `table`. Povezana je s tablicom user putem vanjskog ključa `user_id` i s tablicom `table` putem vanjskog ključa `table_id`. Tablica `table` sadrži

informacije o stolovima u restoranu, uključujući broj stola i kapacitet odnosno broj mjesta. Ova tablica omogućuje praćenje dostupnosti stolova. Tablica `order` sadrži podatke o narudžbama, uključujući datum i vrijeme narudžbe, ukupni iznos računa, te identifikator stola za koji je narudžba napravljena. Povezana je s tablicom `table` putem vanjskog ključa `table_id`. Za praćenje pojedinačnih jela u narudžbama koristi se tablica `order_dish`. Ova tablica povezuje narudžbe s jelima putem vanjskih ključeva `order_id_order` i `dish_id_dish`, te sadrži količinu za svako naručeno jelo. Tablica `dish` sadrži informacije o jelima koja su dostupna u restoranu, uključujući ime jela, sastojke, cijenu i kategoriju. Povezana je s tablicom `category` putem vanjskog ključa `category_id`, što omogućava svrstavanje jela u različite kategorije. Tablica `order_drink` koristi se za praćenje pića u narudžbama. Povezuje narudžbe s pićima putem vanjskih ključeva `order_id_order` i `drinks_id_drink`, te sadrži količinu za svako naručeno piće. Tablica `drinks` pohranjuje informacije o pićima koja su dostupna u restoranu, uključujući ime pića, cijenu i kategoriju. Povezana je s tablicom `category` putem stupca `category_id`. Tablica `category` definira različite kategorije za jela i pića. Ovakva struktura baze podataka je omogućila upravljanje najvažnijim funkcijama restorana, uključujući, rezervacije, narudžbe, jelovnik, jela i pića. Za uspostavu veze s bazom podataka, poslužiteljska strana koristi klasu DB datoteke `database.js`. Prvo se učitaju potrebni moduli `pg` za kreiranje instance pg baze podataka i `dotenv` kako bi se mogle koristiti varijable okruženja. Nadalje, unutar konstruktora, potrebno je učitati konfiguraciju varijabla okruženja, te u metodi `Pool` definirati parametre za spajanje na bazu. Nakon toga, klasa sadrži metodu za slanje upita. Unutar metode nalazi se logika za slanje upita, stvaranje i uništavanje veza na bazu. Klasa sadrži dodatnu metodu za ručno uništavanje veze na bazu.

```
class DB {
  constructor() {
    env.config();
    this.pool = new pg.Pool({
      user: process.env.DB_USER,
      ...
    });
  }
  async query(sql, params) {
    const client = await this.pool.connect();
    try { ... } catch { ... }
    finally { client.release(); }
  }
  async endPool() { ... }
} export default DB;
```



Slika 11. Era model baza podataka

5.4. Poslužiteljska strana

Tehnologije za razvoj poslužiteljske strane odabrani su Node.js i Express. Poslužiteljska strana omogućuje postavljanje Express servera, komuniciranje s bazom podataka i definiranje ruta za posluživanje klijentskog dijela aplikacije i API zahtjeva. Sastoji se od niza datoteka koje omogućuju dohvaćanje, pisanje, spremanje i brisanje podataka iz i u bazu podataka.

Svaka tablica u bazi podataka sadrži odgovarajući servisni direktorij na poslužiteljskoj strani. Servisni direktoriji sadrže dvije datoteke od kojih je prva datoteka s nastavkom DAO (Data Access Object). DAO datoteka sadrži metode za interakciju s bazom podataka, kao što su čitanje, pisanje, ažuriranje i brisanje. Cilj DAO datoteke je razdvajanje logike i stvaranje apstraktnog sloja za rad s bazom podataka. Primjer takve datoteke je `userDAO.js` datoteka, u ovoj datoteci se nalazi sva logika za interakciju s tablicom `user`. Datoteka sadrži metode poput: `getAll()`, `getOne(id_user)`, `insert(user)`. Primjer `getAll()` metode izgleda ovako:

```
async getAll() {
      try {
            let sql = `SELECT * FROM "user";`;
            const data = await this.db.query(sql, []);
            const rows = data.rows;
            return rows;
       } catch (error) { ... }
}
```

Sljedeća datoteka u servisnom direktoriju je datoteka s ključnom riječi `rest`. Ova datoteka sadrži logiku za kreiranje API poziva za komunikaciju s klijentskom stranom aplikacije. Koristi metode definirane u DAO datoteci za izvršavanje operacija nad bazom podataka. Glavna zadaća ove datoteke je rukovanje zahtjevima prema bazi podataka te vraćanje odgovarajućih odgovora, ovisno o uspjehu ili neuspjehu operacija. Primjer metode iz ove datoteke bi izgledao ovako:

```
postUsers: async function (req, res) {
  res.type("application/json");
  try {
    let data = req.body;
    let udao = new UserDAO();
    const users = await udao.insert(data);
```

```
    res.send(JSON.stringify(users));
  } catch (error) {
    console.error(error);
    res.status(500).send(JSON.stringify({ error: "Server Error" }));
  }
},
```

Osim servisnih datoteka za tablicu user, poslužiteljska strana sadrži direktorije, rest i DAO datoteke vezane uz tablice: `drink`, `dish`, `order`, `order_dish`, `order_drink`, `reservation`, `statistics`, i `table`. Tablice `drink`, `dish`, `table`, `order_drink`, `order_dish` i `statistics` sadrže jednostavnu logiku za dohvaćanje, ažuriranje i pisanje, dok datoteke tablica `reservation` i `order` sadrže malo složeniju logiku. Primjer složenije metode vezane uz rezervaciju je metoda datoteke `reservationDAO.js` `getBookedDates(`. Metoda dohvaća sve datume s popunjenim rezervacijama. Prvo, unutar SQL upita stvara se privremena tablica `time_slots` koja sadrži vremenske termine u kojim je moguće obaviti rezervaciju, termini su 12:00:00, 13:30:00, pa sve do 21:00:00. Nadalje, podaci o datumu se dohvaćaju iz tablice `reservation` kojoj je pridijeljena oznaka `r1`. Tablici `r1` lijevim pridruživanjem (eng. *left join*) pridružuje se podupit s privremenom tablicom `r2` koja grupira rezervacije prema datumu, vremenskom terminu, te obavlja izračun koliko je stolova rezervirano u svakom od vremenskih termina. Nadalje, tablici se lijevim pridruživanjem pridružuje privremena tablica `t` koja sadrži ukupan broj stolova u restoranu. Završni dio upita grupira rezultate prema datumu i parametru `whole_day`. Parametar `whole_day` služi kao indikator da je restoran taj dan rezerviran za posebne prilike. Za kraj uvjet `HAVING` filtrira datume u kojim su svi vremenski okviri popunjeni ili je parametar `whole_day` postavljen na `yes`.

```
async getBookedDates() {
    try {
      let sql = `
      WITH time_slots AS (
        SELECT '12:00:00'::time AS time_slot UNION
        ...
        SELECT '21:00:00'::time AS time_slot
        )
        SELECT r1.date
        FROM reservation r1
        LEFT JOIN (
```

```
            SELECT date, time, COUNT(DISTINCT table_id) AS
total_tables
            FROM reservation
            GROUP BY date, time
        ) r2 ON r1.date = r2.date AND r2.time IN (
            SELECT time_slot FROM time_slots
        )
        LEFT JOIN (
            SELECT COUNT(DISTINCT id_table) AS total_tables
            FROM "table"
        ) t ON r2.total_tables = t.total_tables
        GROUP BY r1.date, r1.whole_day
        HAVING COUNT(DISTINCT r2.time) = (SELECT COUNT(*) FROM
time_slots)
            OR r1.whole_day = 'yes';
      `;
      const data = await this.db.query(sql, []);
      const rows = data.rows;
      return rows;
    } catch (error) { ... }
  }
```

Primjer složenije metode vezane za tablicu order je metoda `getAll()`. Metoda dohvaća sve narudžbe tako da spaja podatke iz pet tablica. Tablice koje se spajaju su sljedeće: `order`, `order_dish`, `dish`, `order_drink` i `drink`. Upit se sastoji od dva `SELECT` upita povezana operatorom `UNION`. Operator `UNION` omogućava spajanje rezultata dvaju upita u jedan skup podataka koji se zatim može dohvatiti. Prvi `SELECT` upit prikuplja podatke o narudžbama koje sadrže jela iz tablice `order` označena skraćenicom „`o`“. Nadalje, tablica `order` se povezuje lijevim spajanjem (eng. *left join*) na tablice `order_dish` i `dish` pomoću vanjskog ključa `order_id`, odnosno `dish_id`. Prvi dio upita završava klauzulom `WHERE` koja definira dohvaćanje samo onih podataka koji imaju parametar `status` kao `pending`, `preparing` ili `done`. Provjera statusa je važna radi funkcionalnosti aplikacije koja prikazuje trenutni status narudžbe. Naime, postoji još jedan status naziva `finished` koji označava da je narudžba završena, no takve narudžbe nije potrebno prikazivati na stranici s aktivnim narudžbama pa ih je potrebno izbaciti. Drugi `SELECT` upit prati jednaki postupak kao i prvi upit, samo vezan uz tablice `order_drink` i `drink`. Na kraju, svi dohvaćeni podaci se svrstavaju klauzulom `ORDER BY` prema identifikatoru narudžbe.

```
async getAll() {
    try {
      let sql = `SELECT o.id_order, o.date, ...
      FROM "order" o
      LEFT JOIN order_dish od ON o.id_order = od.order_id
      LEFT JOIN dish d ON od.dish_id = d.id_dish
      WHERE o.status IN ('pending', 'preparing', 'done')
      UNION
      SELECT o.id_order, o.date, ...
      FROM "order" o
      LEFT JOIN order_drink odr ON o.id_order = odr.order_id
      LEFT JOIN drink dr ON odr.drink_id = dr.id_drink
      WHERE o.status IN ('pending', 'preparing', 'done')
      ORDER BY id_order
      `;
      const data = await this.db.query(sql, []);
      const rows = data.rows;
      return rows;
    } catch (error) { ... }
  }
```

Osim servisnih datoteka poslužiteljska strana sadrži i pomoćne datoteke za upravljanje autentifikacijom korisnika i slanje elektroničke pošte. Ove datoteke su `jwt.js` i `mailer.js`. Datoteka `jwt.js` sadrži metode za kreiranje i provjeru tokena, te metodu za provjeru uloge korisnika. Metod za provjeru tokena i uloge korisnika se kasnije koriste kao middleware, kako bi se korisnicima s nedovoljnim pravima zabranio pristup određenim putanjama.

```
const verifyRole = function (allowedRoles) {
  return (req, res, next) => {
    if (!req.user || !allowedRoles.includes(req.user.role)) {
      return res.status(403).send("Forbidden");
    }
    next();
  };
};
```

Datoteka `mailer.js` sadrži logiku za slanje elektroničke pošte. Za implementaciju slanja pošte korišten je modul `nodemailer`. Za početak je potrebno uključiti `nodemailer` i `dotenv` ključnom riječi `import`. Zatim je potrebno kreirati transporter korištenjem metode

`createTransport`. Metodi je potrebno proslijediti potrebne postavke, uključujući parametre za host, port, te korisničko ime i lozinku. Nadalje, slanje elektroničke pošte obavlja metoda `sendMail`. Ova metoda prima četiri parametra: from, to, subject i message. Zatim, email se šalje koristeći `sendMail` metodu, te se na kraju vraća odgovor s informacijama o poslanom emailu.

```
import nodemailer from "nodemailer";
import env from "dotenv";
env.config();

let mailer = nodemailer.createTransport({
  host: "smtp.gmail.com",
  port: ...,
  auth: { user: ..., pass: ..., },
});

const sendMail = async function (from, to, subject, message) {
  message = { from: from, to: to, subject: subject, text: message,
};

  let response = await mailer.sendMail(message);
  return response;
};
export default sendMail;
```

Glavna datoteka poslužiteljske strane je datoteka `server.js`. Ova datoteka sadrži logiku za pokretanje servera, definiranje ruta, korištenje međusoftvera (eng. *middleware*) i rukovanje svim potrebnih servisnim funkcijama. Prvo se učitavaju potrebni moduli, uključujući `express` za kreiranje servera, `dotenv` za upravljanje varijablama okruženja, `express-session` za rad sa sesijama, te svi potrebni servisni moduli za upravljanje korisnicima, rezervacijama, jelima, pićima, narudžbama, stolovima i statistikama, te moduli za `cors` i `jwt`.

```
import express from "express";
import env from "dotenv";
import session from "express-session";
import restUser from "./servis/user/restUser.js";
...
import cors from "cors";
import jwt from "./modules/jwt.js";
```

Nadalje, nakon učitavanja svi potrebnih modula kreira se instanca za korištenje varijabla okruženja `env.config()`, instanca Express servera i postavljaju se osnovni parametri za server kao što su port i tajni ključ sesije.

```
env.config();
const server = express();
const port = process.env.SERVER_PORT || 3000;
const SESSION_SECRET = process.env.SESSION_SECRET;
```

Glavna metoda `server.js` datoteke je metoda `startServer()`. Ova metoda koristi se za pokretanje servera, konfiguraciju različitih middleware funkcija, uključujući `urlencoded` za omogućavanje obrade složenijih podataka poput nizova i objekata, funkcije za parsiranje JSON podataka i postavljanje zaglavlja za cors.

```
server.use(express.urlencoded({ extended: true }));
server.use(express.json());
server.use(cors());
server.use((req, res, next) => {
  res.setHeader("Access-Control-Allow-Origin",
  "http://localhost:8080");
  ...
  next();
});
```

Nadalje, metoda sadrži rukovatelja za putanje koje nisu pronađene, te razne funkcije za pripremu API putanja. Funkcije za pripremu putanja definiraju specifične rute za dohvaćanje, pisanje, ažuriranje i brisanje podataka vezanih uz korisnike, rezervacije, jela, pića, narudžbe, stolove i statistiku.

```
server.use(
    session({
      secret: SESSION_SECRET,
      cookie: { maxAge: 1000 * 60 * 60 * 3 }
      ...
    })
  );
prepareUserPaths();
prepareReservationPaths();
...
prepareStatisticsPaths();
```

Primjer takve funkcije je `prepareDishPaths()`. Funkcija `prepareDishPaths()` definira putanje `/api/dishes i /api/dishes/:id` za slanje GET, POST, PUT i DELETE HTTP zahtjeva. Prije obrade zahtjeva, u svrhu autentifikacije i autorizacije korisnika, koriste se middleware funkcije `verifyToken` i `verifyRole`. Funkcija `verifyToken` provjerava valjanost jwt tokena, dok `verifyRole` kao parametre prima identifikatore uloga korisnika i provjerava korisnikova prava pristupa. Nakon uspješne provjere, zahtjev se obrađuje s pomoću odgovarajuće metode iz modula `restDish.js`, koja sadrži logiku za obradu zahtjeva.

```
function prepareDishPaths() {
  server.get("/api/dishes", restDish.getDishes);
  server.post("/api/dishes", jwt.verifyToken, jwt.verifyRole([1, 2,
3]), restDish.postDishes);
  ...
  server.get("/api/dishes/:id", restDish.getDish);
  server.post("/api/dishes/:id", jwt.verifyToken, jwt.verifyRole([1,
2, 3]), restDish.postDish);
  server.put("/api/dishes/:id", jwt.verifyToken, jwt.verifyRole([1,
2, 3]), restDish.putDish);
  server.delete("/api/dishes/:id", jwt.verifyToken,
jwt.verifyRole([1, 2]), restDish.deleteDish);
}
```

Express server je moguće pokrenuti pozicioniranjem u direktorij `server` i izvršavanjem naredbe `node server.js`.

5.5. Klijentska strana

Za razvoj klijentske strane odabrani su TypeScript, React i Webpack kao alat za automatizaciju procesa. Klijentska strana sadrži korisničko sučelje koje izravno komunicira s poslužiteljskom stranom. Osim Webpacka i Reacta, korišten je i Redux Toolkit, koji olakšava upravljanje stanjem u aplikaciji. Datoteke u Reactu koriste ekstenziju `.tsx`, koja omogućuje korištenje TypeScripta s JSX-om za kreiranje komponenti.

Konfiguracija Webpacka obavlja se u korijenu projekta u `webpack.config.js` datoteci. Konfiguracijska datoteka sadrži konfiguraciju za razvojni i produkcijski način rada. Konfiguracija omogućuje generiranje HTML datoteka i umetanje JavaScript `script` oznake u HTML datoteku koristeći `HtmlWebpackPlugin` dodatak. Nadalje, omogućuje minimiziranje i razdvajanje CSS-a u više datoteka pomoću `MiniCssExtractPlugin` dodatka, kompilaciju TypeScript datoteka u izvorne JavaScript datoteke, te minimiziranje i praćenje ovisnosti u

JavaScript datotekama pomoću `TerserWebpackPlugin` dodatka. Na kraju, datoteka omogućuje automatizirano čišćenje izlaznog direktorija koristeći `CleanWebpackPlugin` dodatak.

Prvo se pomoću ključne riječi `require` učitavaju navedeni dodaci za rad sa zadacima.

```
const HtmlWebpackPlugin = require("html-webpack-plugin");
const MiniCssExtractPlugin = require("mini-css-extract-plugin");
...
const path = require("path");
```

Definira se varijabla `dev` koja definira način rada, ovisno o vrsti okruženja u kojem se aplikacija pokreće. Varijabla koristi ternarni operator za provjeru varijable `NODE_ENV`. Ako je `NODE_ENV` različit od `production`, varijabla `dev` se postavlja na `true`, inače na `false`.

```
const dev = process.env.NODE_ENV !== "production";
```

Nadalje, u glavnom dijelu konfiguracije postavljaju se ključni parametri kao što su način rada, ulazna i izlazna točka. Način rada se postavlja na temelju vrijednosti varijable `dev`, ako je varijabla postavljena na `true`, koristi se razvojni način, u suprotnom, koristi se produkcijski način rada. Ulazna točka definirana je svojstvom `entry` na datoteku `main.tsx` koja se nalazi u direktoriju `src`. Izlazna točka postavljena je tako da se sve generirane datoteke pohranjuju u direktorij `dist`, uz javnu putanju odnosno `publicPath` definiranu kao „/“. Naziv generirane JavaScript datoteke je postavljen na `bundle.js`, a za optimizirane slike definirana je putanja unutar direktorija `images`. Unutar direktorija `images` svaka slika se generira s jedinstvenim hashom kako bi se izbjegli duplikati.

```
module.exports = {
  mode: dev ? "development" : "production",
  entry: "./src/main.tsx",
  output: {
    path: path.resolve(__dirname, "dist"),
    publicPath: "/",
    filename: "bundle.js",
    assetModuleFilename: "images/[hash][ext][query]",
  },
```

Optimizacije kao što su minifikacija JavaScript datoteka, obavljaju se pomoću `TerserWebpackPlugin` dodatka, koji je definiran unutar svojstva `optimization`.

```
optimization: {
  minimize: true,
  minimizer: [new TerserWebpackPlugin()],
},
```

U svojstvu `plugins` definiran je niz svih potrebnih dodataka koji se koriste u konfiguraciji. Svaki dodatak se uključuje kreiranjem nove instance dodatka ključnom riječi `new`. Dodatku `HtmlWebpackPlugin` proslijeđen je dodatni parametar `template`, unutar kojeg je definirana putanja do glavne HTML datoteke `index.html`.

```
plugins: [
  new CleanWebpackPlugin(),
  new HtmlWebpackPlugin({
    template: "./src/index.html",
  }),
  new MiniCssExtractPlugin(),
],
```

Svojstvo `module` sadrži niz pravila za obradu različitih vrsta datoteka. Pravila se definiraju kreiranjem novog objekta za pravilo, te unutar objekta definiranjem potrebnih svojstva. Svojstvo `test` definira koju vrstu datoteke je potrebno obraditi, svojstvo `use` definira koji učitavač će se koristiti za obradu datoteke. Pravilo za slike sadrži još jedno svojstvo, `type` koje definira vrstu resursa koja će se obraditi, ona mogu biti slike, fontovi, ikone i slično. Prvo pravilo odnosi se na datoteke s nastavkom `.tsx`. U svojstvu `use` definirano je da se `.tsx` datoteke obrađuju pomoću `ts-loader` učitavača, dok je u svojstvu `exclude` definirano da se direktorij `node_modules` zanemari prilikom obrade.

```
module: {
  rules: [
    {
      test: /\.tsx?$/, use: "ts-loader", exclude: /node_modules/,
    },
    { ... },
    {
      test: /\.(png|svg|jpg|gif|webp)$/, type: "asset",
    },
  ],
},
```

Svojstvo `resolve`, u nizu `extensions` sadrži listu ekstenzija koje omogućuju Webpacku da prepozna različite vrste datoteka kao što su: `.tsx`, `.ts`, `.js`, `.jsx`. Svojstvo `devtool` služi za konfiguraciju source mapa koje se koriste u projektu. Ponovno se koristi varijabla `dev` koja postavljena na `true`, svojstvo `devtool` postavlja na `inline-source-map`, što je prikladno za razvojni način rada, dok suprotnom slučaju postavlja svojstvo na vrijednost `source-map`, što je prikladno za produkcijski način rada.

```
resolve: {
  extensions: [".tsx", ".ts", ".js", ".jsx"],
},
devtool: dev ? "inline-source-map" : "source-map",
```

Posljednje svojstvo u konfiguraciji je `devServer`. Ovo svojstvo omogućuje konfiguriranje razvojnog servera. Unutar ovog svojstva definirana su sljedeća svojstva: `historyApiFallback` postavljeno na `true`, što je potrebno za ispravan rad s ruterom unutar aplikacije, svojstvo `hot`, koje omogućuje automatsko učitavanje stranice prilikom primjene promjena u kodu i svojstvo `open`, koje automatski otvara aplikaciju u pregledniku nakon pokretanja razvojnog servera.

```
devServer: {
  historyApiFallback: true,
  hot: true,
  open: true,
 },
```

Klijentska strana organizirana je u niz modula koji zajedno čine jednostraničnu web aplikaciju. Glavni moduli klijentske strane organizirani su u direktoriju `src` i podijeljeni na direktorije: `pages`, `layout`, `hooks`, `state` i `components`.

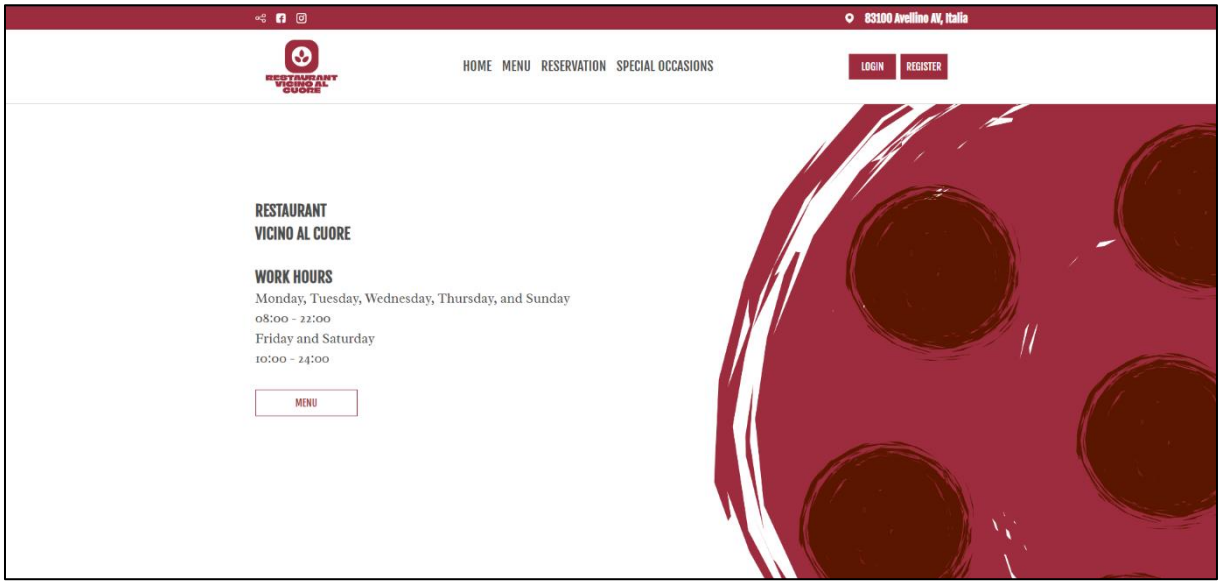
Direktorij `pages` sadrži sve stranice koje koriste komponente za izgradnju korisničkog sučelja. Stranice ne sadrže logiku dohvaćanja podataka, manipulacije ili slične funkcionalnosti ta logika se odvija unutar komponenti. Stranice u ovom direktoriju uključuju: `LoginPage.tsx`, `RegistrationPage.tsx`, `HomePage.tsx`, `MenuPage.tsx`, `ReservationPage.tsx`, `ReservationsPage.tsx`, `ReservationsPageUser.tsx`, `SpecialOcassionsPage.tsx`, `ProfilePage.tsx`, `AddDishPage.tsx`, `AddDrinkPage.tsx`, `OrderPage.tsx`, `OrderList.tsx` i `StatisticsPage.tsx`.

Na početku svake stranice uključuju se sve potrebne komponente koje se koriste za prikaz korisničkog sučelja. Sve stranice i komponente definirane su kao React funkcionalne

komponente. Unutar `HomePage` funkcije vraća se fragment `<> </>` koji sadrži sve potrebne komponente. Primjer takve stranice je komponenta `HomePage.tsx`, koja izgleda ovako:

```
import React from "react";
import Hero from "../components/Hero/Hero";
import Footer from "../components/Footer/Footer";
...
const HomePage = () => {
  return (
    <>
      <Hero />
      <About />
      <Info />
      <Menu />
      <Footer />
    </>
  );
};

export default HomePage;
```



Slika 12. Prikaz početne stranice

Direktorij `layout` sadrži datoteku `MainLayout.tsx`. Ova datoteka služi kao predložak za sve ostale stranice. Datoteka sadrži dvije komponente koje se ponavljaju kroz sve stranice u aplikaciji. Navedene komponente su `Header` i `Outlet`. Komponenta `Header` služi za prikaz zaglavlja s navigacijom. Komponenta `Outlet` je dio biblioteke `react-`

router-dom te služi kao placeholder za prikaz trenutno prikazane stranice. Na početku je potrebno uključiti navedene komponente ključnom riječi import. Nadalje, potrebno je u funkciji MainLayout vratiti navedene komponente.

```
import React from "react";
import { Outlet } from "react-router-dom";
import Header from "../components/Header/Header";

const MainLayout = () => {
  return (
    <>
      <Header />
      <Outlet />
    </>
  );
};
export default MainLayout;
```

Direktorij hooks sadrži dva poddirektorija s pomoćnim datotekama koje sadrže funkcije za izvršavanje HTTP zahtjeva prema poslužiteljskoj strani. Navedeni poddirektoriji su user i order. Unutar direktorija order nalazi se datoteka orderUtils.ts. Ova datoteka sadrži funkcije za unos podataka vezanih uz narudžbe, jela i pića. Na početku su definirana sučelja za podatke vezane u narudžbu OrderData, jelo DishData i piće DrinkData.

```
interface OrderData {
  date: string;
  bill: number;
  table_id: number;
}
interface DishData { order_id: number; ... }
interface DrinkData { order_id: number; ... }
```

Nakon toga, definirane su funkcije za unos navedenih podataka. Jedna od tih funkcija je insertOrder. Funkcija insertOrder je asinkrona funkcija definirana ključnom riječi async. Prije naziva funkcije definirana je ključna riječ export koja omogućuje pozivanje funkcije u drugim datotekama. Nadalje, funkcija prima argument orderData tipa OrderData. Unutar funkcije ,u try-catch bloku, iz spremišta sesije, dohvaća se jwt token. Nakon dohvaćanja tokena provjerava se njegovo postojanje. Ako token ne postoji, funkcija prekida izvršavanje i vraća poruka u konzoli. Ako token postoji funkcija koristi fetch metodu za slanje

POST zahtjeva na poslužitelj. Rezultati zahtjeva spremaju se u varijablu Zahtjev sadrži url putanju do API-a, vrstu metode postavljenu na `POST`, postavlja zaglavlje Content-type na `„Content-type“: „application/json“` i u zaglavlju `Authorization` šalje token na autentifikaciju. Podaci za unos šalju se u tijelu zahtjeva u JSON formatu pomoću metode `JSON.stringify()`.

```
const res = await fetch("http://localhost:12413/api/orders", {
      method: "POST",
      headers: {
        "Content-type": "application/json",
        Authorization: "Bearer " + token,
      },
      body: JSON.stringify(orderData),
    });
```

Na kraju, ako je odgovor poslužitelja pozitivan, funkcija vraća dobivene podatke, u suprotnom funkcija se prekida i vraća se greška.

```
if (res.ok) {
      const data = await res.json();
      return data;
    } else { ... }
  } catch (error) { ... }
};
```

Direktorij `user` sadrži pomoćnu datoteku `userUtils.ts`. Ova datoteka sadrži funkciju za ažuriranje korisničkog profila.

Direktorij `state` sadrži poddirektorije i datoteke povezane uz Redux Toolkit. Redux Toolkit je popularno rješenje za upravljanje stanjem (eng. *state management*) unutar React web aplikacija. Glavni `state` direktorij sadrži poddirektorije `store` i `slices`. Direktorij `store` sadrži datoteku `store.ts`. Ova datoteka sadrži glavnu konfiguraciju za Redux store. Redux store predstavlja kompletnu konfiguraciju stabla stanja (eng. *state tree*) i omogućuje upravljanje stanjem koristeći dispatch akcije.

Na početku, potrebno je uključiti metodu `configureStore` biblioteke `reduxjs/toolkit` koja omogućuje konfiguraciju i registriranje slice reducer-a koji omogućuju upravljanje stanjem u specifičnom dijelu aplikacije. Nakon toga potrebno je uključiti sve potrebne slice datoteke koje će se koristiti za upravljanje stanjem.

```
import { configureStore } from "@reduxjs/toolkit";
```

```
import userReducer from "../slices/user/userSlice";
...
import orderReducer from "../slices/order/orderSlice";
import orderListReducer from "../slices/order/orderListSlice";
```

Redux store ili spremište konfigurira se kreiranjem funkcije koja koristi `configureStore` metodu. Funkciji je zatim potrebno omogućiti pozivanje u drugim datotekama korištenjem ključne riječi `export`. Nadalje, u metodi `configureStore` definira se `reducer` objekt koji mapira slice reducer-e odgovarajućim imenima. Redux radi tako da prati pojavu akcije, te na njenu pojavu, gleda koji slice treba biti iskorišten za tu akciju, te zatim prosljeđuje akciju odgovarajućem reducer-u

```
export const store = configureStore({
  reducer: {
    reservations: reservationsReducer,
    category: categoryReducer,
    user: userReducer,
    ...
    statistics: statisticsReducer,
  },
});
```

Poddirektorij `slices` sadrži skup poddirektorija i datoteka unutar kojih su definirani slice reducer-i za upravljanje akcijama nad promjenom stanja u aplikaciji. Neki od poddirektorija su: `category`, `dish`, `drink`, `order`, `reservations` i `user`. Direktorij `order` sadrži datoteke `orderSlice.ts` i `orderListSlice.ts`. Datoteka `orderSlice.ts` sadrži logiku za kreiranje slice-a koji upravlja stanjem dodavanja jela i pića u narudžbu. Na početku, potrebno je uključiti sve potrebne metode biblioteke `reduxjs/toolkit`, `createSlice` i `PayloadAction`.

```
import { createSlice, PayloadAction } from "@reduxjs/toolkit";
```

Nadalje, potrebno je definirati sučelja za tipove jela `Dish` i pića `Drink`, te novi tip `OrderItem` koji sadrži kombinaciju sučelja `Dish` i `Drink`. Također, potrebno je definirati inicijalno stanje narudžbe tako da se nizu `initialState` dodijeli tip podatka `OrderState` koji je jednak nizu tipa `OrderItem`.

```
export interface Dish { id_dish: number; ... }
export interface Drink { id_drink: number; ... }
export type OrderItem = Dish | Drink;
type OrderState = OrderItem[];
```

```
const initialState: OrderState = [];
```

Nakon toga, potrebno je kreirati slice korištenjem metode `createSlice`. Kreirani slice sadrži ime `order`, te mu se prosljeđuje inicijalno stanje `initialState`.

```
const orderSlice = createSlice({
  name: "order",
  initialState,
```

Također, slice sadrži objekt `reducer` sa skupom funkcija koje upravljaju stanjem narudžbe. Prva funkcija je funkcija `addItem` koja ima funkciju dodavanja jela, odnosno pića u narudžbu. Funkcija kao argumente prima trenutno stanje `state OrderItem` objekta, te akciju `action`.

```
addItem(state, action: PayloadAction<OrderItem>) {
```

Nadalje, u varijablu `existingIndex` sprema se rezultat metode `findIndex` unutar koje se metodama `isDish` i `isDrink` provjerava već postojanje jela i pića u narudžbi. Ova provjera je bitna kako bi se kasnije u funkciji takvim jelima i pićima mogla samo nadodati količina i povećati ukupna cijena.

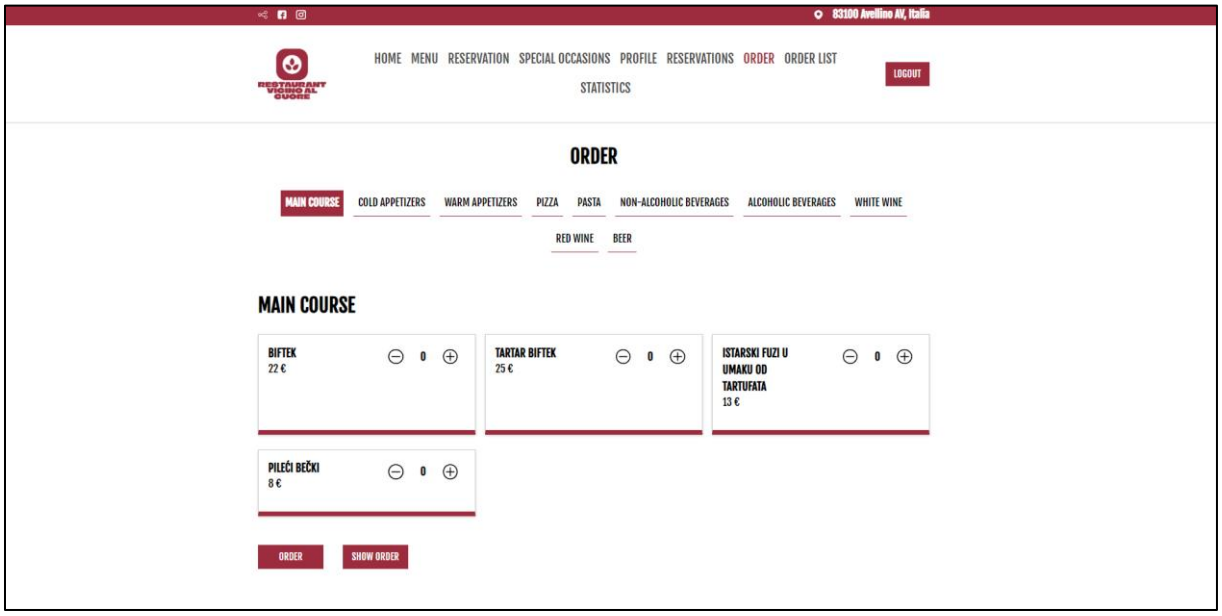
```
function isDish(item: OrderItem): item is Dish {
  return (item as Dish).id_dish !== undefined;
}
function isDrink(item: OrderItem): item is Drink { ... }

const existingIndex = state.findIndex((item) =>
     isDish(newItem) && isDish(item)
       ? item.id_dish === newItem.id_dish
       : isDrink(newItem) && isDrink(item)
       ? item.id_drink === newItem.id_drink
       : false
);
```

Nadalje, izvršava se provjerava ako je jelo ili piće pronađeno, ako je, pomoću metode `map` kreira se kopija trenutnog niza s ažuriranom cijenom i količinom za to jelo, u suprotnom dodaje se novo jelo ili piće u narudžbu.

```
if (existingIndex !== -1) {
   return state.map((item, index) =>
     index === existingIndex
       ? {
```

```
            ...item,
            quantity: item.quantity + 1,
            totalPrice: price * (item.quantity + 1),
          }
        : item
    );
} else {
    return [...state, { ...newItem, quantity: 1, totalPrice: price }];
```



Slika 13. Prikaz stranice narudžba

Unutar direktorija `user` nalazi se datoteka `userSlice.ts`. Ova datoteka sadrži funkciju `fetchUserData` za dohvaćanje korisnika, te spremanje rezultata u spremište. Na početku, potrebno je uključiti dodatne metode `createAsyncThunk` i `jwtDecode` biblioteka `reduxjs/toolkit` i `jwt-decode`.

```
import { createAsyncThunk, createSlice } from "@reduxjs/toolkit";
import { jwtDecode } from "jwt-decode";
```

Metoda `createAsyncThunk` omogućuje kreiranje asinkronih funkcija koje vraćaju Promise. Metoda prilikom izvršavanja generira tri tipa akcija: `pending`, `fulfilled` i `rejected`. Ove akcije omogućuju upravljanje asinkronim operacijama u Reduxu. Kao argumente, metoda prima naziv akcije tipa string i callback funkciju koja izvršava operaciju i vraća Promise.

```
export const fetchUserData = createAsyncThunk<User>(
  "user/fetchUserData",
```

```
  async () => {
```

Unutar callback funkcije, u `try-catch` bloku, dohvaća se token iz spremišta sesije i provjerava se njegova prisutnost. Ako token ne postoji, funkcija se prekida. Ako token postoji, koristi se metoda `jwtDecode` za dekodiranje tokena i vađenje id-a korisnika iz tijela tokena. Kasnije, id korisnika se koristi u fetch metodi za dohvaćanje podataka o korisniku. Ako je zahtjev uspješan, metoda vraća podatke o korisniku u JSON obliku, u suprotnom, baca grešku.

```
try {
   const token = sessionStorage.getItem("token");
   if (!token) {
     return [];
   }
   const decodedToken = jwtDecode(token) as DecodedToken;
   const id_user = decodedToken.user.id_user;
   const res =
     await fetch(`http://localhost:12413/api/users/${id_user}`,
     {...}, });

   if (!res.ok) { ... }
     const data = await res.json();
     return data;
} catch (error) { ... }
```

Na kraju, unutar metode `createSlice`, potrebno je definirati svojstvo `extraReducers`. Ovo svojstvo omogućuje definiranje kako se stanje ažurira ovisno o različitim stanjima asinkrone funkcije. Za konfiguraciju ponašanja funkcije ovisno o stanju koristi se argument `builder`. Svako stanje definira se unutar metode `addCase`, koja prihvaća tri moguća stanja: `pending`, `fulfilled` ili `rejected`, te callback funkciju. Callback funkcija prima dva argumenta: `state` i `action`, te omogućuje ažuriranje globalnog stanja na temelju trenutnog stanja asinkrone funkcije.

```
const userSlice = createSlice({
  ...
  extraReducers: (builder) => {
    builder
      .addCase(fetchUserData.pending, (state) => {
        state.status = "loading";
      })
      .addCase(fetchUserData.fulfilled, (state, action) => {
```

```
        state.status = "succeded";
        state.user = [action.payload];
      })
      .addCase(fetchUserData.rejected, (state, action) => {
        state.status = "failed";
        state.error = action.error.message;
      });
  },
});
```

Direktorij `components` organiziran je u niz poddirektorija i datoteka. Svaka komponenta sadrži poddirektorij koji sadrži jednu `.tsx` i jednu `.css` datoteku. Datoteke s nastavkom `.tsx` predstavljaju glavne datoteke komponente, te je unutar njih implementirana sva logika specifična za tu komponentu. Komponente su implementirane da prate uzorak dizajna da svaka komponenta obavlja jednu zadaću. Na kraju skup svih komponenti stvara izgled korisničkog sučelja aplikacije.

Stranica `reservation` sastoji se od niza različitih komponenti koje omogućuju korisnicima rezervaciju kroz tri koraka. Komponente uključuju `ReservationDatePicker`, `ReservationTimeSelect`, `ReservationTableSelect`, `ReservationUserInput` i `ReservationSubmitButton`. Stranica je dizajnirana da prikazuje različite komponente rezervacije ovisno o koraku na kojem se korisnik nalazi. Prikaz sljedećih koraka ovisi o tome je li korisnik odabrao datum ili kliknuo na gumb za prijelaz na sljedeći korak.

Komponenta `ReservationDatePicker` služi za odabir datuma rezervacije. Za prikaz kalendara koristi se biblioteka `react-datepicker`. Na početku, potrebno je uključiti sve potrebne module i funkcije, kao što su `useState`, `useEffect`, `useDispatch`, `useSelector`, `reservationSlice`, `AppDispatch`, `RootState`, `DatePicker`, `maxDate`, `today`, te odgovarajuće CSS datoteke.

```
import React, { useState, useEffect } from "react";
import { useDispatch, useSelector } from "react-redux";
...
import DatePicker from "react-datepicker";
import "react-datepicker/dist/react-datepicker.css";
import "../ReservationForm/ReservationForm.css";
```

Nakon toga, kreira se sučelje `ReservationDatePickerProps` za definiranje prop-ova koje komponenta prima. U Reactu, prop-ovi predstavljaju argumente koji se mogu proslijediti komponenti. U glavnoj funkciji komponente, prosljeđuju se prop-ovi `setStep`,

`reservationData` i `setReservationData`. Ovi argumenti služe za ažuriranje stanja koraka i podataka o rezervaciji.

```
interface ReservationDatePickerProps { ... }
const ReservationDatePicker = ({
  setStep,
  reservationData,
  setReservationData,
}: ReservationDatePickerProps) => {
```

Dalje, definiraju se varijable: `dispatch`, koja predstavlja metodu Redux Toolkita za pokretanje funkcija definiranih u slice-u, `bookDatesList`, koja sadrži listu rezerviranih datuma radi blokiranja istih prilikom pokretanja komponente, `bookedDatesStatus`, koja prati status asinkrone funkcije dohvaćanja rezerviranih datuma, i `bookedDatesError`, koja sadrži greške u slučaju da asinkrona funkcija u slice-u baci grešku. Na kraju, definira se `useState` hook za spremanje i postavljanje blokiranih datuma.

```
const dispatch = useDispatch<AppDispatch>();
const bookedDatesList = useSelector(
  (state: RootState) => state.reservations.bookedDates
);
const bookedDatesStatus = useSelector(
  (state: RootState) => state.reservations.status
);
const bookedDatesError = useSelector(
  (state: RootState) => state.reservations.error
);
const [blockedDatesState, setBlockedDatesState] = useState([]);
```

Komponenta koristi dva `useEffect` hook-a. Prvi `useEffect` dohvaća blokirane datume putem `dispatch` metode i sprema ih u `bookedDatesList`. Drugi `useEffect` koristi `setBlockedDatesState` za postavljanje dohvaćenih blokiranih datuma u stanje.

```
useEffect(() => {
  dispatch(fetchBookedDates());
}, [dispatch]);

if (bookedDatesStatus === "failed") {
  return <div>Error: {bookedDatesError}</div>;
}
```

```
useEffect(() => {
  if (bookedDatesList.length > 0) {
    const blockedDatesArray = bookedDatesList.map(
      (reservation) => new Date(reservation.date)
    );
    setBlockedDatesState(blockedDatesArray);
  }
}, [bookedDatesList]);
```

U funkciji `handleDateChange`, ažuriraju se podaci o rezervaciji pomoću `setReservationData` prop-a na odabrani datum. Formatira se datum, dohvaćaju se blokirana vremena za taj datum, te se postavlja `setStep` prop na 2 kako bi se učitala sljedeća komponenta.

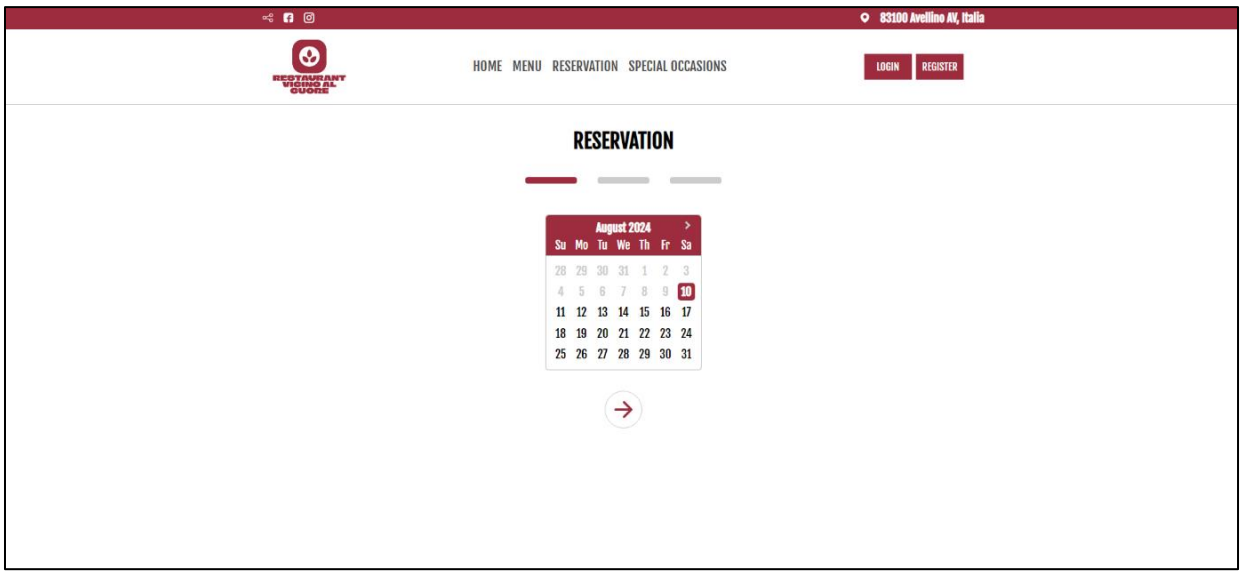
```
const handleDateChange = (date: Date) => {
  setReservationData((prevState: any) => ({
    ...prevState,
    date: date,
  }));

  const formattedDate = date.toISOString().split("T")[0];
  dispatch(fetchBookedTime(formattedDate));
  setStep(2);
};
```

Na kraju, komponenta vraća kalendar s parametrima: `selected`, koji sadrži odabrani datum, `onChange`, koji povezuje navedenu funkciju `handleDateChange`, `minDate`, koji postavlja minimalni datum na današnji dan, `maxDate`, koji definira maksimalni datum za rezervaciju, `excludedDates`, koji sadrži listu blokiranih datuma kako bi se spriječio odabir rezerviranih datuma, i `inline`, koji određuje izgled kalendara.

```
return (
    <DatePicker
      selected={new Date(reservationData.date)}
      onChange={handleDateChange}
      minDate={today}
      maxDate={maxDate}
      excludeDates={blockedDatesState}
      inline
    />
```

```
    );
};
```



Slika 14. Prikaz stranice rezervacija

6. Zaključak

Analiza alata Grunt, Gulp i Webpack, te usporedba njihovih dobrih i loših strana, pokazala je da svaki od njih ima svoje prednosti i nedostatke. Grunt, s jednostavnim načinom postavljanja uz pomoć JSON-a, daje lakšu opciju na početku, ali može stvarati ograničenje u većim i složenijim projektima. Gulp, koristeći node-stream metodu omogućuje brže obavljanje zadataka i bolju prilagodljivost, dok Webpack kao najkompleksniji alat nudi široke mogućnosti za prilagodbu i optimizaciju, no zahtjeva dobro razumijevanje osnovnih koncepata za ispravnu upotrebu. Usporedba ovih alata pokazuje da izbor pravog alata zavisi o specifičnim potrebama projekta, složenosti aplikacije, te željama timova koji rade na razvoju aplikacije. Dok Grunt i Gulp pružaju odlične opcije za manje projekte ili ranije faze razvoja, Webpack se pokazao kao najbolji alat za složene projekte koji traže visoku razinu prilagodbe i konfiguracije.

Na kraju, implementacija vlastite web aplikacije potvrdila je značaj pravilne konfiguracije i odabira pravih alata za automatizaciju kako bi se osiguralo da razvojni proces bude što brži i učinkovitiji. Automatizacija procesa se pokazala ne samo kao sredstvo za ubrzanje razvoja, već i ključan resurs za održavanje kvalitete, ovisnosti i skalabilnosti modernih web aplikacija.

Popis literature

[1] H. Piirainen, *Optimizing web development workflow*. Metropolia Ammattikorkeakoulu, 2016. Pristupljeno: 24. kolovoz 2024. [Na internetu]. Dostupno na: http://www.theseus.fi/handle/10024/118611

[2] R. Sharma, „Why do we need a module bundler?“, Medium. Pristupljeno: 27. srpanj 2024. [Na internetu]. Dostupno na: https://medium.com/@rajatgms/why-do-we-need-a-module-bundler-c5ff221523f5

[3] D. Flanagan, *JavaScript: The Definitive Guide: Activate Your Web Pages*. O'Reilly Media, Inc., 2011.

[4] E. A. Meyer, *CSS: The Definitive Guide: The Definitive Guide*. O'Reilly Media, Inc., 2006.

[5] G. Svaiko, „Font Psychology: Here's Everything You Need to Know About Fonts“, Designmodo. Pristupljeno: 27. srpanj 2024. [Na internetu]. Dostupno na: https://designmodo.com/font-psychology/

[6] „Image file type and format guide“. Pristupljeno: 27. srpanj 2024. [Na internetu]. Dostupno na: https://developer.mozilla.org/en-US/docs/Web/Media/Formats/Image_types

[7] C. A. Devarapalli, „Employing Build Tools: Optimizing Frontend Development with Webpack, Gulp, or Grunt“, *J. Technol. Innov.*, sv. 1, izd. 1, Art. izd. 1, velj. 2020, doi: 10.93153/x4pxh005.

[8] „Grunt: The JavaScript Task Runner“. Pristupljeno: 14. srpanj 2024. [Na internetu]. Dostupno na: https://gruntjs.com/

[9] J. Cryer, *Pro Grunt.js*. Apress, 2015.

[10] „Grunt vs gulp vs Webpack | What are the differences?“, StackShare. Pristupljeno: 14. srpanj 2024. [Na internetu]. Dostupno na: https://stackshare.io/stackups/grunt-vs-gulp-vs-webpack

[11] „Webpack vs Gulp vs Grunt vs Browserify Comparison Guide“, Cleveroad Inc. - Web and App development company. Pristupljeno: 22. srpanj 2024. [Na internetu]. Dostupno na: https://www.cleveroad.com/blog/gulp-browserify-webpack-grunt/

[12] „Gulp vs Grunt vs Webpack: Which Technology is Better? | Artjoker“. Pristupljeno: 22. srpanj 2024. [Na internetu]. Dostupno na: https://artjoker.net/blog/gulp-vs-grunt-vs-webpack-tools-and-task-runners-which-technology-is-better/

[13] T. Maynard, *Getting Started with Gulp – Second Edition*. Packt Publishing Ltd, 2017.

[14] „GitHub - gulpjs/gulp“. Pristupljeno: 14. srpanj 2024. [Na internetu]. Dostupno na: https://github.com/gulpjs/gulp

[15] IndiDev, „Webpack Demystified: A Beginner's Guide and Pros & Cons“, Medium. Pristupljeno: 27. srpanj 2024. [Na internetu]. Dostupno na: https://medium.com/@IndiDev/webpack-demystified-a-beginners-guide-and-pros-cons-2c10dc71e85

[16] M. Bouzid, *Webpack for Beginners: Your Step-by-Step Guide to Learning Webpack 4*. Apress, 2020.

[17] „Concepts“, webpack. Pristupljeno: 14. srpanj 2024. [Na internetu]. Dostupno na: https://webpack.js.org/concepts/

[18] „webpack vs gulp vs grunt | Compare Similar NPM Packages“. Pristupljeno: 11. kolovoz 2024. [Na internetu]. Dostupno na: https://npm-compare.com/grunt,gulp,webpack

[19] „webpack vs gulp vs grunt: Which is Better JavaScript Bundlers and Task Runners?“ Pristupljeno: 22. srpanj 2024. [Na internetu]. Dostupno na: https://npm-compare.com/grunt,gulp,webpack

Popis slika

Popis tablica

Prilozi

Prilog 1: ZIP datoteka s programskim kodom