SVEUČILIŠTE U ZAGREBU
FAKULTET ORGANIZACIJE I INFORMATIKE
V A R A Ž D I N

Grgo Penava

Razvoj aplikacija temeljen na Vue.js razvojnom okviru

DIPLOMSKI RAD

Varaždin, 2025.

SVEUČILIŠTE U ZAGREBU

FAKULTET ORGANIZACIJE I INFORMATIKE

V A R A Ž D I N

Grgo Penava

Matični broj: 0016142422

Studij: Informacijsko i programsko inženjerstvo

Razvoj aplikacija temeljen na Vue.js razvojnom okviru

DIPLOMSKI RAD

Mentor:

Prof. dr. sc. Neven Vrček

Varaždin, srpanj 2025.

Grgo Penava

Izjava o izvornosti

Izjavljujem da je moj završni/diplomski rad izvorni rezultat mojeg rada te da se u izradi istoga nisam koristio drugim izvorima osim onima koji su u njemu navedeni. Za izradu rada su korištene etički prikladne i prihvatljive metode i tehnike rada.

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

Sažetak

Moderne web tehnologije i razvojni okviri postaju ključni elementi u izradi suvremenih aplikacija koje zahtijevaju responzivnost, skalabilnost i jednostavno održavanje. Jedan od takvih okvira je Vue.js, progresivni JavaScript okvir koji omogućuje izradu interaktivnih korisničkih sučelja na učinkovit i modularan način. Vue.js se temelji na komponentnom pristupu razvoju aplikacija, što omogućuje bolju organizaciju koda, ponovnu iskoristivost i lakšu suradnju među programerima. Posebno je pogodan za izradu jednostraničnih aplikacija (engl. Single Page Application, skraćeno SPA), gdje se korisničko sučelje dinamički ažurira bez ponovnog učitavanja stranice. Razvojem web aplikacija temeljenih na Vue.js okviru dolazi do izražaja i potreba za upravljanjem složenijim stanjima aplikacije, što se rješava korištenjem dodatnih alata poput Pinia – službene biblioteke za upravljanje globalnim stanjem. Uz to, alati poput Vite omogućuju brzu izgradnju i optimizaciju Vue aplikacija, čime se značajno smanjuje vrijeme razvoja i poboljšava korisničko iskustvo. Razvojem tehnologije povećava se i dostupnost serverskih resursa, pa je moguće cjelokupni sustav postaviti na vlastiti VPS poslužitelj i učiniti ga javno dostupnim putem vlastite domene. Svrha ovog rada je prikazati konkretan proces razvoja frontend web aplikacije koristeći Vue.js 3, s posebnim naglaskom na organizaciju koda, upravljanje stanjem i povezivanje s vanjskim servisima.

Ključne riječi: vue.js; TypeScript; aplikacija; razvoj; servisi; komponenta; stanje

Sadržaj

1. Uvod

Razvoj web aplikacija u suvremenom IT okruženju zauzima ključno mjesto u gotovo svim granama industrije. Digitalna transformacija, sve veća potreba za dostupnošću informacija i usluga putem interneta dovela je do porasta interesa za razvoj rješenja koja su brza, prilagodljiva i jednostavna za korištenje. U tom kontekstu, frontend razvoj – koji se odnosi na izradu korisničkog sučelja i interakcije korisnika s aplikacijom – igra izuzetno važnu ulogu.

Vue.js je jedan od modernih JavaScript okvira koji omogućava izgradnju responzivnih i komponentno organiziranih web aplikacija. Za razliku od nekih kompleksnijih okvira, Vue se ističe jednostavnošću učenja, čistom sintaksom i lakoćom integracije s drugim tehnologijama. Upravo zbog toga sve češće se koristi u izradi malih i srednje velikih aplikacija, ali i u projektima većeg opsega.

Tema ovog diplomskog rada usmjerena je na analizu i primjenu Vue.js razvojne tehnologije kroz praktičnu izradu aplikacije za upravljanje zadacima. Takve aplikacije koriste se u timskom radu za praćenje tijeka zadataka, evidentiranje statusa, delegiranje i organizaciju poslovnih procesa. Kroz razvoj ove aplikacije demonstrira se kako se Vue.js može koristiti za implementaciju dinamičnih korisničkih sučelja, dok se za pohranu podataka koristi poseban backend servis, a za autentifikaciju korisnika koristi se Supabase.

Cilj rada je prikazati prednosti Vue.js okvira u razvoju aplikacija te istaknuti konkretne funkcionalnosti poput upravljanja stanjima, povezivanja s vanjskim servisima te integracije s backend servisom. Praktični dio rada daje jasan uvid u tijek razvoja jedne aplikacije od početne ideje do gotovog rješenja, s naglaskom na strukturu koda, korištenje alata i dobre prakse razvoja.

2. Metode i tehnike rada

Razvoj moderne web aplikacije zahtijeva pažljiv odabir alata i tehnologija koje omogućuju učinkovitu implementaciju funkcionalnosti, dobru organizaciju koda te jednostavno održavanje i proširivost sustava. U sklopu izrade ovog rada korišten je skup tehnologija koje su se pokazale kao pouzdane i praktične u kontekstu izgradnje jednostavne, ali funkcionalne aplikacije za upravljanje zadacima.

Frontend dio aplikacije razvijen je korištenjem Vue.js 3 – progresivnog JavaScript okvira koji omogućuje brzu izradu dinamičkih korisničkih sučelja uz jasan fokus na modularnost i ponovnu iskoristivost komponenata. Uz Vue.js korišten je i TypeScript, jezik koji proširuje JavaScript statičkom tipizacijom, čime se smanjuje broj grešaka u kodu i povećava čitljivost većih projekata.

Kao glavno razvojno okruženje korišten je Visual Studio Code, zbog svoje jednostavnosti, bogatog ekosustava proširenja te odlične podrške za rad s Vue.js projektima i TypeScriptom. Kombinacija ovih tehnologija omogućila je razvoj stabilne i responzivne aplikacije, uz relativno brz tijek izrade i testiranja.

U nastavku su detaljno opisane ključne tehnologije koje su korištene u ovom radu, kao i njihova uloga u procesu razvoja.

2.1. Vue.js 3

Vue.js je progresivni JavaScript okvir koji se koristi za razvoj korisničkih sučelja i jednostraničnih aplikacija (SPA). Njegova osnovna funkcionalnost usredotočena je na View layer, ali se zahvaljujući svojoj fleksibilnosti vrlo jednostavno može integrirati s drugim bibliotekama i alatima. Vue.js koristi Model-View-ViewModel (skraćeno MVVM) arhitekturu, što dodatno pojednostavljuje povezivanje podataka i strukturu aplikacije [1].

Vue.js 3, kao najnovija verzija ovog okvira, donosi niz poboljšanja u odnosu na prethodne verzije, uključujući veću brzinu izvođenja, manju veličinu paketa i poboljšanu podršku za TypeScript. Sam okvir je u potpunosti prepisan u TypeScriptu, ali njegovo korištenje ostaje neobavezno, čime se programerima ostavlja fleksibilnost u razvoju [1].

Jedna od najvažnijih novosti u Vue.js 3 je Composition API, novi način strukturiranja logike komponenata koji omogućuje bolju organizaciju i ponovnu iskoristivost koda. Za razliku od Options API pristupa koji je korišten u ranijim verzijama, Composition API omogućuje

funkcionalni pristup razvoju komponenata, što je posebno korisno kod kompleksnijih aplikacija [1].

Vue.js Composition API je kao set dodatnih, funkcijski temeljenih API-ja koji omogućuju fleksibilno komponiranje logike komponenata [2].

Osim toga, Vue.js 3 uvodi i značajku *teleport* (poznatu i kao *portal*), koja omogućuje renderiranje dijelova komponente izvan njenog matičnog DOM stabla. Ova funkcionalnost je posebno korisna za *modal* prikaz i *pop-up* prozore te je sada sastavni dio Vue.js jezgre, dok je ranije bila dostupna samo kroz vanjske biblioteke [1].

Također, u novoj verziji više nije obavezno imati jedan korijenski element unutar <template> tag-a. Ova funkcionalnost poznata je kao Multi-root components [1].

Nadalje, performanse Vue.js 3 su znatno poboljšane – nova verzija koristi optimizirani *virtual* DOM i algoritam za *diffing*, što rezultira bržim renderiranjem i manjim opterećenjem preglednika. Osnovni *runtime* smanjen je na otprilike 12 kB, što dodatno doprinosi efikasnosti aplikacija [1].

Vue.js 3 dodatno dolazi s podrškom za nove web standarde i bolje alate za razvoj, uključujući poboljšani sustav za reakciju na promjene (engl. *Reactivity system*), bolje rukovanje s props i događajima, te bolju integraciju sa alatima za razvoj [2].

Zahvaljujući svim navedenim značajkama, Vue.js 3 se pokazao kao snažan i moderan okvir pogodan za razvoj kako manjih tako i složenijih web aplikacija, pružajući visoku razinu kontrole i efikasnosti prilikom implementacije korisničkog sučelja [2] [3].

Slika 1. Vue.js; prema [4]

2.2. Typescript

TypeScript je snažan *superset* JavaScripta koji uvodi statičku tipizaciju i napredne značajke s ciljem povećanja sigurnosti koda, poboljšanja alata za razvoj te lakšeg skaliranja aplikacija svih veličina. U usporedbi s JavaScriptom, TypeScript omogućuje robusniji i održiviji razvoj, posebno kod velikih projekata gdje je važno rano otkrivanje grešaka i dosljednost u strukturi koda [5].

TypeScript je razvio Microsoft, a prvi puta je predstavljen 2012. godine. U njegovom dizajnu ključnu je ulogu imao Anders Hejlsberg, poznat i kao tvorac C# jezika. Njegova svrha bila je poboljšati JavaScript u kontekstu većih i kompleksnijih aplikacija, pružajući jaču osnovu za razvoj na razini poduzeća [6].

Kao nadskup JavaScripta, TypeScript omogućuje korištenje postojećeg JavaScript koda, ali nadopunjuje ga dodatnim značajkama poput statičkog tipiziranja, sučelja (engl. *Interfaces*), generičkih tipova (engl. *Generics*) i poboljšanih razvojnih alata. Tipizacija u TypeScriptu je opcionalna, što omogućuje fleksibilnost u primjeni – od potpunog tipiziranog sustava do labavijeg pristupa korištenjem tipa any [5].

Budući da web preglednici ne mogu direktno interpretirati TypeScript, on se mora prethodno pretvoriti u čisti JavaScript koristeći TypeScript prevoditelj (skraćeno tsc). Na taj način, TypeScript ostaje kompatibilan sa svim modernim preglednicima, serverima i operativnim sustavima [6].

Velike tehnološke kompanije, poput Googlea (za Angular), Microsofta (VS Code, Teams), kao i Airbnb, Uber, Slack i mnoge druge, koriste TypeScript u svakodnevnom razvoju, što potvrđuje njegovu pouzdanost i skalabilnost [6].

TypeScript je potpuno otvorenog koda, a njegov izvorni kod je dostupan na GitHubu. Projekt aktivno održava Microsoft uz veliku podršku globalne zajednice programera. Do lipnja 2025., TypeScript repozitorij broji više od 105.000 zvjezdica i 12.000 „forkova“, što potvrđuje njegovu široku primjenu i popularnost u *open source* zajednici [7].

Slika 2. Typescript; prema [8]

2.3. Visual Studio Code

Visual Studio Code, poznatiji kao VS Code, besplatan je i lagan uređivač izvornog koda koji je u kratkom vremenu postao jedan od najkorištenijih alata među programerima svih razina znanja i iskustva, neovisno o programskom jeziku koji koriste [9].

Njegova jednostavnost, fleksibilnost i napredne funkcionalnosti osigurale su mu vodeće mjesto u brojnim anketama među programerima te široku primjenu u različitim industrijama. Zahvaljujući bogatom sustavu proširenja, intuitivnom korisničkom sučelju i snažnim ugrađenim mogućnostima, VS Code olakšava razvojni proces i povećava produktivnost [9].

Visual Studio Code je uređivač izvornog koda koji je razvio Microsoft i dostupan je na Windows, Linux, macOS i u web-preglednicima. Iako nije klasično integrirano razvojno okruženje (IDE), pruža mnoge napredne značajke poput „debuggiranja“, automatskog dovršavanja koda, isticanja sintakse, refaktoriranja koda i ugrađene kontrole verzija putem „Git-a“. VS Code je u vrlo kratkom roku stekao iznimnu popularnost, a prema anketi „Stack Overflow“ programera iz 2024. godine koristi ga više od 73% ispitanika, što ga čini najpopularnijim alatom te vrste [10].

Podržava velik broj jezika kao što su JavaScript, Python, C++, Java i mnogi drugi, a uz pomoć proširenja moguće je dodati podršku i za dodatne jezike i alate. Uređivač uključuje alate poput *IntelliSense* za pametno dovršavanje koda, pregled strukture koda i automatsko zatvaranje zagrada. Također omogućuje korisnicima da prilagode temu, prečace na tipkovnici i druge postavke sučelja prema vlastitim preferencijama [10].

VS Code ima ugrađeni terminal koji podržava različite tipove terminala (Bash, PowerShell, Zsh i sl.) i omogućuje izvršavanje naredbi bez napuštanja uređivača. Paleta naredbi (engl. *Command Palette*) omogućuje pristup gotovo svim funkcionalnostima editora putem

tipkovnice, čime se značajno ubrzava radni tijek i čini alat posebno pristupačnim korisnicima koji preferiraju rad bez miša [10].

Također, omogućava i funkciju *Live Share* koja omogućuje programerima zajednički rad na istom projektnom kodu u stvarnom vremenu, neovisno o njihovoj fizičkoj lokaciji. Kroz ovu funkcionalnost, sudionici mogu pregledavati i uređivati iste datoteke, dijeliti terminale te zajedno koristiti alate za ispravljanje pogrešaka. Ova opcija posebno je korisna za udaljene timove, online edukaciju i mentoriranje jer omogućuje jednostavnu i učinkovitu suradnju bez potrebe za dodatnim alatima ili infrastrukturom [9].

„Lagan“ je, ali moćan editor koda koji se brzo pokreće i radi na svim glavnim operacijskim sustavima, što ga čini pogodnim za razne razvojne potrebe. Zahvaljujući velikom broju dostupnih ekstenzija, lako se prilagođava različitim projektima i tehnologijama, a podrška zajednice i česta ažuriranja dodatno doprinose njegovoj popularnosti. Ipak, kod vrlo velikih projekata može doći do problema s performansama, a preveliko oslanjanje na dodatke može zakomplicirati održavanje radnog okruženja [9].

Slika 3. Visual Studio Code; prema [10]

3. Povijest razvojnog okvira Vue.js

Vue.js je razvojni okvir kojeg je 2014. godine osmislio Evan You, bivši zaposlenik Googlea. Tijekom rada na AngularJS-u, You je uočio potrebu za jednostavnijim i lakšim rješenjem koje bi bilo pristupačnije širem krugu developera. Kao odgovor na tu potrebu, razvio je Vue – lagan i fleksibilan *framework* namijenjen izradi korisničkih sučelja i jednostranih (*single-page*) aplikacija. Prva verzija Vue.js-a objavljena je u veljači 2014. godine, a brzo je stekla popularnost zahvaljujući jednostavnoj sintaksi, jasnoj dokumentaciji i lakoći korištenja [3].

Tijekom godina, Vue.js je značajno porastao u popularnosti i danas se ubraja među najkorištenije *frontend* okvire, rame uz rame s Reactom i Angularom. Prošao je kroz nekoliko velikih verzija, od kojih je svaka donijela nove značajke i poboljšanja u performansama i strukturi koda. Najnovija stabilna verzija Vue.js-a trenutno Vue.js 3, a zahvaljujući snažnoj zajednici i stalnom razvoju, očekuje se da će okvir i dalje imati svijetlu budućnost [3].

Vue.js je tijekom godina izrastao u jedan od najvažnijih *frontend* okvira, kombinirajući jednostavnost s moćnim značajkama i snažnom podrškom zajednice. Od svog početka, pa do danas, prošao je kroz nekoliko ključnih faza razvoja koje su ga oblikovale u robusnu platformu za izradu web aplikacija [11].

3.1. Vue 1.0 (Evangelion)

Prva stabilna verzija Vue.js-a razvijena je u listopadu 2015. godine. Donijela je funkcionalnosti poput dvosmjernog povezivanja podataka (engl. *Two-way data binding*), komponentnog pristupa, prilagođenih direktiva i podrške za prijelaze i animacije. Brzo je privukla veliki broj korisnika, osobito u Kini, gdje je postala vodeći *frontend* alat [11].

3.2. Vue 2.0 (Ghost in the Shell)

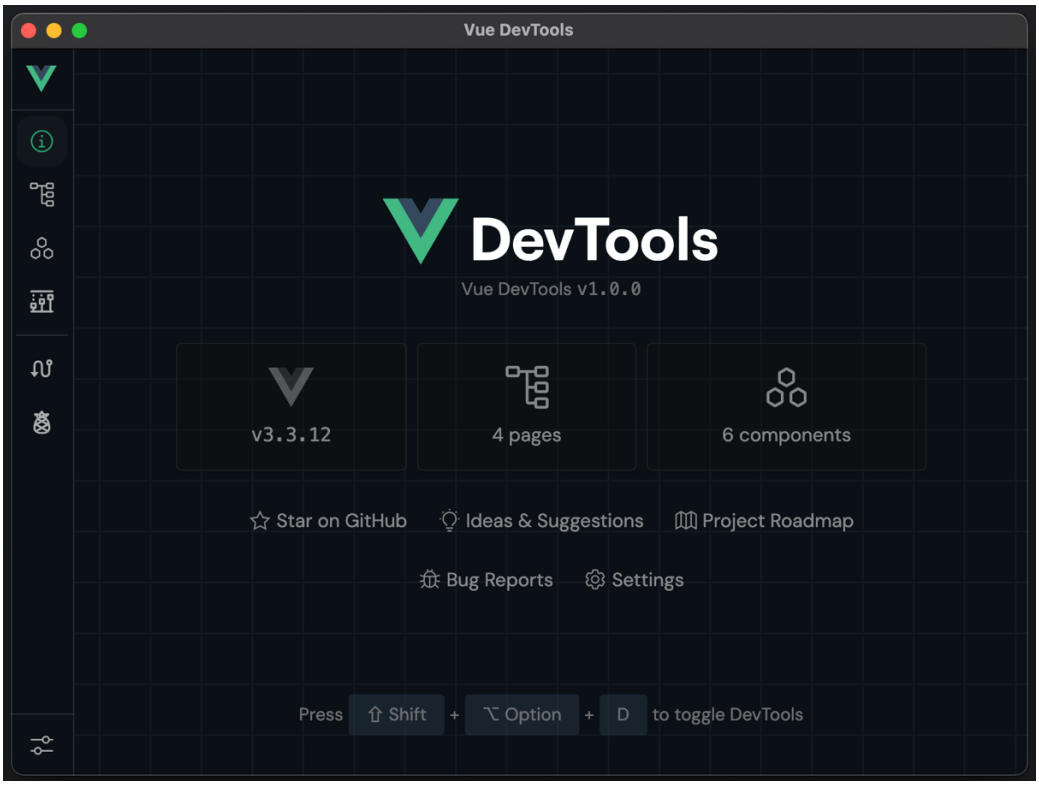
Druga verzija označila je velik iskorak – uveden je virtualni DOM za brže renderiranje, unaprijeđen je sustav komponenti te je dodana podrška za renderiranje na strani poslužitelja (SSR). Uz to, razvijeni su i ključni alati poput Vue Routera, Vuex-a i Vue CLI-ja. Ova verzija učvrstila je Vue.js kao ravnopravnog konkurenta Angularu i Reactu. Ova je verzija objavljena u rujnu 2016. godine [11].

3.3. Vue 3.0 (One Piece)

Treća verzija objavljena je u rujnu 2020. godine, a donijela je potpuni redizajn okvira uz implementaciju u TypeScriptu. Uveden je Composition API, koji omogućuje bolju organizaciju i ponovnu uporabu logike, dok je nova reaktivnost temeljena na Proxy objektima značajno ubrzala rad aplikacija. Paket Vue.js 3 postao je manji i efikasniji, a poboljšana je i podrška za TypeScript [11].

3.4. Vue danas

Danas Vue predstavlja više od samog okvira – on je temelj cijelog ekosustava modernih alata za razvoj web aplikacija. Uključuje Nuxt.js za SSR (*Server Side Rendering*) i statičke stranice, Vite kao ultra brzi alat za izgradnju aplikacija (također djelo Evana You-a), Pinia kao modernu zamjenu za Vuex te Vue Devtools za napredno otklanjanje grešaka. Vue.js se razvijao iz lagane biblioteke u cjelovitu platformu sposobnu za izgradnju malih, ali i vrlo složenih aplikacija [11].



Slika 4. Vue DevTools; prema [12]

4. TypeScript

TypeScript je programski jezik otvorenog koda koji je razvio Microsoft, a predstavlja nadskup (superset) JavaScripta. To znači da TypeScript proširuje JavaScript dodavanjem sintakse za statičku tipizaciju, zadržavajući pritom istu osnovnu sintaksu i sve funkcionalnosti JavaScripta. Osnovna ideja iza TypeScripta je omogućiti razvojnim inženjerima da dodaju tipove podataka varijablama, parametrima funkcija i povratnim vrijednostima, što nije standardna karakteristika JavaScripta koji je dinamički tipiziran jezik [13].

Glavna prednost TypeScripta leži u njegovoj sposobnosti da detektira pogreške u kodu tijekom procesa razvoja (engl. *compile-time*), prije nego što se kod izvrši u pregledniku ili na poslužitelju. Time se smanjuje vjerojatnost pogrešaka i olakšava održavanje kompleksnih aplikacija. Budući da web preglednici ne mogu izravno interpretirati TypeScript, kod napisan u TypeScriptu se pomoću TypeScript kompajlera (skraćeno tsc) prevodi u standardni JavaScript kod [14].

Zatim, JavaScript kod može se izvršavati u bilo kojem JavaScript okruženju. TypeScript također podržava datoteke s definicijama tipova (.d.ts) koje omogućuju korištenje postojećih JavaScript biblioteka kao da su napisane s TypeScript tipovima [15].

4.1. TypeScript osnove

U usporedbi s JavaScriptom, TypeScript omogućuje robusniji i održiviji razvoj, posebno kod velikih projekata gdje je važno rano otkrivanje grešaka i dosljednost u strukturi koda. Kao nadskup JavaScripta, TypeScript omogućuje korištenje postojećeg JavaScript koda, ali nadopunjuje ga dodatnim značajkama poput statičkog tipiziranja, sučelja (engl. *Interfaces*), generičkih tipova (engl. *Generics*) i poboljšanih razvojnih alata [14].

4.1.1. Tipovi podataka

TypeScript uvodi statičku tipizaciju, što znači da se tipovi podataka varijabli moraju eksplicitno definirati. Ovo omogućuje kompajleru da provjeri tipove tijekom razvoja i spriječi pogreške koje bi se inače dogodile tek tijekom izvođenja [16].
TypeScript podržava sljedeće primitivne tipove podataka [16]:

- string: Tekstualni podaci
- number: Brojčani podaci (cijeli brojevi i decimalni brojevi)

- boolean: Logički podaci (true ili false)
- bigInt: Brojčani podaci koji predstavljaju vrlo velike cijele brojeve
- symbol: Jedinstveni identifikatori
- null i undefined: Predstavljaju odsutnost vrijednosti
- array: Nizovi podataka istog ili različitog tipa
- any: Tip koji može predstavljati bilo koju vrijednost (poželjno je izbjegavati i koristiti samo u posebnim slučajevima)

Isječak koda 4.1: Različiti tipovi podataka

```
// string: Tekstualni podaci
let imePrezime: string = "Marko Marković";

// number: Brojčani podaci (cijeli brojevi i decimalni brojevi)
let godine: number = 30;
let cijena: number = 19.99;

// boolean: Logički podaci (true ili false)
let jePunoljetan: boolean = true;

// bigInt: Brojčani podaci koji predstavljaju vrlo velike cijele brojeve
let drugiVelikiBroj: bigint = BigInt("12345678901234567890");

// symbol: Jedinstveni identifikatori
let jedinstveniId: symbol = Symbol("id");

// null: Predstavlja namjernu odsutnost vrijednosti objekta
let nista: null = null;

// undefined: Predstavlja varijablu koja je deklarirana, ali joj nije
dodijeljena vrijednost
let nedefinirano: undefined;

// array: Nizovi podataka istog ili različitog tipa
let brojeviNiz: number[] = [1, 2, 3, 4, 5];

// any: Tip koji može predstavljati bilo koju vrijednost (poželjno je
izbjegavati)
let biloSto: any = "Ovo može biti string";
```

Isječak koda 4.2: Niz (engl. Array) s različitim tipovima podataka

```
let mjesovitiNiz: (string | number | boolean)[] = ["Jabuka", 15, true,
"Kruška", 23];
let drugiMjesovitiNiz: any[] = ["Tekst", 100, false, {naziv: "Objekt"}];

console.log(mjesovitiNiz[0]); // Ispisuje: Jabuka
console.log(drugiMjesovitiNiz[3].naziv); // Ispisuje: Objekt
```

4.1.2. Definiranje varijabli i funkcija

Varijable u TypeScriptu definiraju se pomoću ključnih riječi „let“ ili „const“, uz navođenje tipa podataka. Ključna riječ „let“ koristi se za deklariranje varijabli čija se vrijednost može kasnije promijeniti. S druge strane, „const“ se koristi za deklariranje konstanti, odnosno varijabli čija se vrijednost ne može mijenjati nakon inicijalizacije. Korištenje „const“ preporučuje se za vrijednosti koje trebaju ostati nepromijenjene tijekom izvršavanja programa, čime se povećava sigurnost i čitljivost koda [17].

Funkcije se također mogu tipizirati, specificiranjem tipova parametara i povratne vrijednosti [17].

Isječak koda 4.3: Definiranje varijable

```
let ime: string = "Ivan"; // Vrijednost varijable 'ime' se može kasnije
promijeniti
const brojGodina: number = 25; // Vrijednost konstante 'brojGodina' se ne
može mijenjati

ime = "Marko"; // Ovo je dozvoljeno
brojGodina = 30; // Ovo bi uzrokovalo grešku jer je 'brojGodina' konstanta
```

Isječak koda 4.4: Definiranje i poziv funkcije

```
function pozdravi(ime: string, godine: number): string {
  return "Pozdrav, " + ime + "! Ti imaš " + godine + " godina.";
}

let poruka: string = pozdravi("Ana", 30);
console.log(poruka); // Ispisuje: Pozdrav, Ana! Ti imaš 30 godina.
```

4.1.3. *Type*, *Class* i *Interface*

TypeScript pruža tri ključna mehanizma za definiranje oblika podataka i strukture koda: „type“, „class“ i „interface“. Svaki od njih ima svoju svrhu i karakteristike, a razumijevanje razlika i sličnosti pomaže u pisanju čitljivijeg, održivijeg i skalabilnijeg koda [16].

Class se koristi za definiranje predložaka za stvaranje objekata. Klase sadrže svojstva (podatke) i metode (funkcije) koje definiraju ponašanje objekta, a TypeScript klase podržavaju koncepte objektno orijentiranog programiranja kao što su enkapsulacija, nasljeđivanje i polimorfizam [16] [15].

Isječak koda 4.5: Primjer korištenja klase

```
class Osoba {
    private ime: string;
    private dob: number;

    constructor(ime: string, dob: number) {
        this.ime = ime;
        this.dob = dob;
    }
```

```
    public pozdravi(): void {
        console.log(
            `Pozdrav, moje ime je ${this.ime} i imam ${this.dob} godina.`
        );
    }
}

const osoba = new Osoba('Ivan', 25);
osoba.pozdravi(); // Ispisuje: Pozdrav, moje ime je Ivan i imam 25 godina.
```

U ovom primjeru, klasa „Osoba“ ima privatna svojstva „ime“ i „dob“, konstruktor za inicijalizaciju tih svojstava te javnu metodu „pozdravi“ koja ispisuje poruku. Ključna riječ „private“ ograničava pristup svojstvima samo unutar klase.

Konstruktori su posebne metode unutar klase koje se koriste za inicijalizaciju svojstava objekta prilikom stvaranja instance klase. Konstruktor se definira pomoću ključne riječi „constructor“. U TypeScriptu je moguće definirati više konstruktora (engl. *Overload*), ali je dozvoljena samo jedna implementacija konstruktora koja mora biti kompatibilna sa svim preopterećenjima. To se može postići korištenjem opcionalnog parametra [13] [16].

Isječak koda 4.6: Primjer preopterećenja konstruktora

```
type Spol = 'm' | 'ž';

class Osoba {
    ime: string;
    dob: number;
    spol: Spol;

    constructor(ime: string, dob: number, spol?: Spol);
    constructor(ime: string, dob: number, spol: Spol) {
        this.ime = ime;
        this.dob = dob;
        this.spol = spol ?? 'm';
    }
}

const osoba1 = new Osoba('Ivan', 25);
const osoba2 = new Osoba('Ana', 22, 'ž');
```

U ovom primjeru, konstruktor je preopterećen kako bi se omogućilo stvaranje instance klase s ili bez specificiranja spola. Ukoliko spol nije specificiran, postavlja se na zadanu vrijednost „m“.

Pristupni modifikatori „private“, „protected“ i „public“ koriste se za kontrolu vidljivosti i pristupa članovima klase (svojstvima i metodama) [16]:

- private: Članovi klase označeni kao „private“ dostupni su samo unutar same klase.
- protected: Članovi klase označeni kao „protected“ dostupni su unutar klase i njenih izvedenih klasa.

- public: Članovi klase označeni kao „public“ dostupni su svugdje (unutar klase, izvan klase, u izvedenim klasama).

„Interface“ definira strukturu objekata, specificirajući nazive i tipove svojstava ili metoda koje objekt mora imati. „Type“ definira oblik podataka i koristi se za provjeru tipova [15].

Isječak koda 4.7: Primjer kreiranja Interface

```
interface Osoba {
    ime: string;
    dob: number;
    pozdravi(): void;
}
```

Isječak koda 4.8: Primjer kreiranja Type

```
type OsobaType = {
    ime: string;
    dob: number;
    pozdravi(): void;
};
```

4.1.4. Unije i presjeci tipova

U TypeScriptu, unije (engl. *Union Types*) i presjeci (engl. *Intersection Types*) tipova su moćni alati koji omogućuju kombiniranje postojećih tipova na fleksibilne načine kako bi se stvorili novi, složeniji tipovi. Ovi mehanizmi omogućuju preciznije modeliranje podataka u aplikacijama [18] [19].

Unija tipova dopušta da varijabla, parametar ili povratna vrijednost funkcije može biti jednog od nekoliko definiranih tipova. Koristi se simbol vertikalne crte („|“) za označavanje unije. Vrijednost koja odgovara uniji tipova mora biti barem jednog od tipova navedenih u uniji. Ovo je korisno kada očekujemo vrijednost koja može imati različite oblike [18] [19].

Isječak koda 4.9: Primjer korištenja unije tipova

```
// Unija tipova: varijabla 'identifikator' može biti ili string ili broj
type Identifikator = string | number;

let korisnickiID: Identifikator;

korisnickiID = "user123"; // Valjano
console.log(korisnickiID); // Ispisuje: user123

korisnickiID = 404; // Valjano
console.log(korisnickiID); // Ispisuje: 404

// korisnickiID = true; // Greška: Tip 'boolean' nije dodjeljiv tipu
'string | number'.
```

```
// Primjer funkcije koja prihvaća uniju tipova
function ispisiID(id: Identifikator): void {
  if (typeof id === "string") {
    console.log(`ID korisnika (string): ${id.toUpperCase()}`);
  } else {
    console.log(`ID korisnika (broj): ${id}`);
  }
}

ispisiID("abcDef"); // Ispisuje: ID korisnika (string): ABCDEF
ispisiID(789);    // Ispisuje: ID korisnika (broj): 789
```

Presjek tipova kombinira više tipova u jedan. Varijabla koja odgovara presjeku tipova mora zadovoljavati sva svojstva i metode svih tipova uključenih u presjek. Koristi se simbol ampersanda (“&”) za označavanje presjeka. Presjeci su korisni kada želimo stvoriti novi tip koji ima sve značajke nekoliko postojećih tipova [19] [18].

Isječak koda 4.10: Primjer korištenja presjeka tipova

```
interface Osoba {
  ime: string;
  godine: number;
}

interface Zaposlenik {
  idZaposlenika: string;
  odjel: string;
}

// Presjek tipova: ZaposlenaOsoba mora imati sva svojstva iz Osoba I
Zaposlenik
type ZaposlenaOsoba = Osoba & Zaposlenik;

let djelatnik: ZaposlenaOsoba = {
  ime: "Ana Anić",
  godine: 30,
  idZaposlenika: "ZAP001",
  odjel: "Marketing"
};

console.log(djelatnik.ime);        // Ispisuje: Ana Anić
console.log(djelatnik.odjel);      // Ispisuje: Marketing
```

Unije i presjeci tipova omogućuju razvojnim inženjerima da kreiraju vrlo specifične i prilagođene tipove koji točno opisuju podatke s kojima rade, čime se povećava sigurnost tipova podataka i čitljivost koda [19] [20].

4.1.5. Operatori

Operator je specijalni simbol koji se koristi za izvršavanje operacija nad operandima (vrijednostima i varijablama). Budući da je TypeScript snažno tipizirani nadskup JavaScripta,

on podržava sve standardne operatore koje podržava i JavaScript. Operatori se, prema broju operanada koje primaju, mogu podijeliti u tri vrste [21]:

- Unarni - jedan operand
- Binarni - dva operanda, a većina operanada dostupnih u TypeScriptu su binarni operandi
- Ternarni - tri operanda. Prvi je operand uvjet, a TypeScript podržava ternarni operator „?“

Isječak koda 4.11: Primjer korištenja unarnog operatora

```
// Unarni operator (npr. negacija)
let broj: number = 5;
let negiraniBroj = -broj; // Unarni minus
console.log("Unarni operator (negacija):", negiraniBroj); // Ispis: -5
```

Isječak koda 4.12: Primjer korištenja binarnog operatora

```
// Binarni operator (npr. zbrajanje)
let prviBroj: number = 10;
let drugiBroj: number = 3;
let zbroj = prviBroj + drugiBroj; // Binarni operator zbrajanja
console.log("Binarni operator (zbrajanje):", zbroj); // Ispis: 13
```

Isječak koda 4.13: Primjer korištenja ternarnog operatora

```
// Ternarni operator (uvjetni izraz)
let prijavljen: boolean = true;
let poruka = prijavljen ? "Dobrodošli natrag!" : "Molimo prijavite se.";
console.log("Ternarni operator:", poruka); // Ispis: Dobrodošli natrag!
```

U nastavku su detaljno opisane glavne kategorije operatora koje se koriste u TypeScriptu.

4.1.5.1. Aritmetički operatori (engl. *Arithmetic operators*)

Aritmetički operatori uzimaju numeričke vrijednosti kao operande i kao rezultat vraćaju jednu numeričku vrijednost. U TypeScriptu razlikujemo 7 aritmetičkih operatora [21]:

Tablica 1: Aritmetički operatori

Operator	Naziv	Opis	Primjer

+	Zbrajanje (engl. *Addition*)	Vraća zbroj operanada	a + b
-	Oduzimanje (engl. *Subtraction*)	Vraća razliku operanada	a - b
*****	Množenje (engl. *Multiplication*)	Vraća umnožak operanada	a * b
/	Dijeljenje (engl. *Division*)	Vraća kvocijent operanada	a / b
%	Modul (engl. *Modulus*)	Vraća ostatak pri dijeljenju	a % b
++	Inkrement (engl. *Increment*)	Povećava vrijednost operanda za jedan	a++
–	Dekrement (engl. *Decrement*)	Smanjuje vrijednost operanda za jedan	a--

(Izvor: Tutorialspoint, bez dat.)

4.1.5.2. Logički operatori (engl. *Logical operators*)

Operatori koji se koriste za povezivanje dvaju ili više uvjeta u jedan izraz koji vraća logičku vrijednost (true ili false) obično se svrstavaju u kategoriju logičkih operatora. U TypeScriptu razlikujemo 3 logička operatora: [21].

Tablica 2: Logički operatori

Operator	Naziv	Primjer	Opis
&&	Logičko I (AND)	a&&b	Vraća true ako su 'a' i 'b' true
||	Logičko ILI (OR)	a||b	Vraća true ako je 'a' ili 'b' true
!	Logičko NE (NOT)	!a	Vraća true ako je 'a' false

(Izvor: Nitin Bhardwaj, 2020)

4.1.5.3. Operatori usporedbe (engl. *Comparision operators*)

Operatori usporedbe koriste se za usporedbu dvaju operanada, a kao rezultat uvijek vraćaju Boolean vrijednost (true ili false) [21].

Tablica 3: Operatori usporedbe

Operator	Naziv	Primjer (za x = 5)	Rezultat
==	Jednakost (engl. *equal to*)	x == 8	False
		x == "5"	True
===	Striktna jednakost (engl. *equal value and equal type*)	x === 5	True
		x === "5"	False
!=	Nejednakost (engl. *not equal*)	x != 8	True
!==	Striktna nejednakost (engl. *not equal value or not equal type*)	x !== 5	False
		x !== "5"	True
>	Veće od (engl. *greater than*)	x > 8	False
<	Manje od (engl. *less than*)	x < 8	True
>=	Veće ili jednako (engl. *greater than or equal to*)	x >= 8	False
<=	Manje ili jednako (engl. *less than or equal to*)	x <= 8	True

(Izvor: W3schools, bez dat.)

4.1.5.4. Operatori pridruživanja (engl. *Assignment operators*)

Operatori koji se koriste za dodjelu vrijednosti operandima ili varijablama obično se svrstavaju u kategoriju operatora pridruživanja, a u TypeScriptu razlikujemo 6 različitih operatora pridruživanja [21]:

Tablica 4: Operatori pridruživanja

Operator	Naziv	Primjer
=	Operator dodjele (engl. *Assign operator*)	a: number = 5
+=	Operator množenja i dodjele (engl. *Add and assign operator*)	b += a (isto kao b = b + a)
-=	Operator oduzimanja i dodjele (engl. *Subtract and assign operator*)	b -= a (isto kao b = b – a)
***=**	Operator množenja i dodjele (engl. *Multiply and assign operator*)	b *= a (isto kao b = b * a)
/=	Operator dijeljenja i dodjele (engl. *Divide and assign operator*)	b /= a (isto kao b = b / a)
%=	Operator ostatka i dodjele (engl. *Modulus and assign operator*)	b %= a (isto kao b = b % a)

(Izvor: Shammi Anand, 2024)

4.1.5.5. Bitovni operatori (engl. *Bitwise operators*)

Bitovni operatori koriste se za manipulaciju bitovima operanada, odnosno za izvođenje niskorazinskih operacija nad pojedinačnim bitovima. U TypeScriptu postoji ukupno sedam bitovnih operatora, a svaki od njih ima specifičnu svrhu i ponašanje [21]:

Tablica 5: Bitovni operatori

Operator	Naziv	Primjer
&	Bitovni AND operator (engl. *Bitwise AND operator*)	2 & 3 // rezultat = 2
|	Bitovni OR operator (engl. *Bitwise OR operator*)	2 | 3 // rezultat = 3
^	Bitovni XOR operator (engl. *Bitwise XOR operator*)	2 ^ 3 // rezultat = 1
~	Bitovni NOT operator (engl. *Bitwise NOT operator*)	~2 // rezultat = -3
>>	Bitovni pomak udesno (engl. *Bitwise Right Shift operator*)	2 >> 3 // rezultat = 0
<<	Bitovni pomak ulijevo (engl. *Bitwise Left Shift operator*)	2 << 3 // rezultat = 16
>>>	Bitovni pomak udesno bez predznaka (engl. Bitwise *Right Shift With Zero operator*)	3 >>> 4 // rezultat = 0

(Izvor: Shammi Anand, 2024)

4.1.5.6. Ternarni operatori (engl. *Ternary operators*)

Ternarni operator u TypeScriptu koristi se kao skraćeni zapis za if...else izraze, kada je potrebno izvršiti jednostavnu odluku i dodijeliti vrijednost na temelju uvjeta. Njegova sintaksa je prikazana u sljedećem isječku koda [22]:

Isječak koda 4.14: Sintaksa ternarnog operatora

```
uvjet ? izraz_ako_je_true : izraz_ako_je_false;
```

Ternarni operator je idealan za kraće uvjetne izraze i dodjelu vrijednosti unutar jedne linije koda, čime se povećava preglednost i smanjuje broj redova. Također je moguće ugnijezditi (engl. *nest*) više ternarnih operatora unutar jednog izraza, ali se takav pristup preporučuje koristiti umjereno zbog smanjene čitljivosti programskog koda [22].

Isječak koda 4.15: Primjer ternarnog operatora

```
let dob: number = 20;
let rezultat: string = (dob >= 18) ? "Punoljetan" : "Maloljetan";
console.log(rezultat); // Punoljetan
```

Isječak koda 4.16: Primjer ugniježđenog ternarnog operatora

```
let broj: number = 0;
let rezultat: string = (broj > 0) ? "pozitivan" : (broj == 0 ? "nula" :
"negativan");
console.log(rezultat); // nula
```

4.1.5.7. Operatori spajanja (engl. *Concatenation operators*)

Operator za spajanje (konkatenaciju) koristi se za spajanje dva ili više “stringova” u jedan jedinstveni “string” [21].

Isječak koda 4.17: Primjer operatora spajanja

```
let ime: string = "Ivan";
let prezime: string = "Ivic";
// Spajanje stringova pomoću operatora +
let punoIme: string = ime + " " + prezime;
console.log(punoIme); // Ispis: Ivan Ivic
```

4.1.5.8. Operatori tipa (engl. *Type operators*)

U TypeScriptu, operator typeof omogućuje dohvaćanje tipa neke varijable na razini statičke provjere (u vrijeme prevođenja). Za razliku od JavaScript verzije koja vraća tipove u obliku stringa, TypeScript verzija typeof koristi se za deklaraciju ili dohvaćanje stvarnih tipova podataka u kodu [23].

4.1.6. Petlje i uvjetno grananje

Kao i većina programskih jezika, TypeScript omogućuje upravljanje tokom izvršavanja programa na temelju postavljenih uvjeta. To znači da se određeni blokovi koda izvršavaju samo ako su zadovoljeni definirani logički izrazi. TypeScript, kao nadskup JavaScripta, podržava sve konstrukte za donošenje odluka poznate iz ES6 standarda, uključujući if, if...else, else if i switch [24].

Takve strukture omogućuju programeru da definira jedan ili više uvjeta koji se provjeravaju pri izvršavanju programa, čime se određuje koji će dio koda biti izvršen. Ova fleksibilnost ključna je za izgradnju dinamičkih i interaktivnih aplikacija koje reagiraju na ulazne podatke i različite scenarije u stvarnom vremenu [24].

4.1.6.1. If i if-else naredba

U TypeScriptu, naredba if koristi se za provjeru istinitosti određenog logičkog izraza (uvjeta). Ako uvjet evaluira kao true, tada se izvršava pripadajući blok koda unutar vitičastih zagrada {}. Ukoliko uvjet nije ispunjen, kod unutar if bloka se preskače [24].

Za slučajeve kada je potrebno izvršiti alternativni blok koda ako uvjet nije ispunjen, koristi se naredba if...else. U tom slučaju, ako je uvjet false, izvršava se else blok. TypeScript također podržava proširenu formu else if, koja omogućuje višestruke grananja u odlučivanju toka izvršavanja programa [24].

Isječak koda 4.18: Primjer if else naredbe

```
let ocjena: number = 86;
if (ocjena >= 90) {
    console.log("Izvrstan");
} else if (ocjena >= 80) {
    console.log("Vrlo dobar");
} else {
    console.log("Potrebno je više truda");
}
```

4.1.6.2. Else-if naredba

Else if naredba koristi se kada je potrebno provjeriti više uzastopnih uvjeta koji su povezani s početnim if izrazom. Ova naredba, poznata je i kao „else if ljestve“ (engl. *else if ladder*), a omogućuje postepenu evaluaciju uvjeta — jedan po jedan — sve dok se ne pronađe onaj koji je istinit [25].

Isječak koda 4.19: Sintaksa else-if naredbe

```
if (uvjet1) {
// izvršava se ako je uvjet1 točan
}
else if (uvjet2) {
// izvršava se ako je uvjet1 netočan,
```

```
// a uvjet2 točan
}
```

Isječak koda 4.19: Primjer else-if naredbe

```
let broj1: number = 50;
let broj2: number = 15;
if (broj1 < broj2) {
console.log("if uvjet je istinit");
} else if (broj1 > broj2) {
console.log("else if uvjet je istinit");
}
```

4.1.6.3. Switch naredba

Switch naredba koristi se kada želimo usporediti jednu vrijednost s više mogućih slučajeva i izvršiti odgovarajući blok koda za onu vrijednost koja se poklapa. Omogućuje pregledniju zamjenu za višestruke if...else if uvjete [26].

Ključne značajke switch naredbe su [26]:

- switch prima izraze bilo kojeg podatkovnog tipa (broj, string itd.).
- Svaki case predstavlja moguću vrijednost koju izraz može imati.
- Blok koda unutar case se izvršava kada se nađe podudaranje.
- break se koristi na kraju svakog case kako bi se spriječilo „propadanje“ u sljedeći case.
- default blok je opcionalan i izvršava se ako nijedan case nije zadovoljen.
- Vrijednost izraza u switch i case moraju biti kompatibilnog tipa.

Isječak koda 4.20: Primjer switch naredbe

```
let izborPica: string = "sok";

switch (izborPica) {
  case "voda":
    console.log("Odabrali ste vodu.");
    break;
  case "kava":
    console.log("Odabrali ste kavu.");
    break;
  case "sok":
    console.log("Odabrali ste sok.");
    break;
  case "čaj":
    console.log("Odabrali ste čaj.");
    break;
  default:
    console.log("Nepoznat odabir pića.");
    break;
}

// Rezultat: Odabrali ste sok.
```

4.1.6.4. For petlja

For petlja u TypeScriptu je temeljna naredba za kontrolu toka koja omogućuje višekratno izvršavanje bloka koda [27]. Koristi se kada je unaprijed poznat broj ponavljanja ili za iteraciju kroz elemente kolekcija poput nizova [28]. Budući da je TypeScript nadskup JavaScripta, podržava nekoliko varijanti *for* petlje, od kojih svaka ima svoju specifičnu namjenu [28].

Sintaksa *for* petlje prikazana je u sljedećem isječku koda:

Isječak koda 4.21: Sintaksa *for* petlje

```
for(inicijalizacija; uvjet; izraz) {
 // statement
}
```

For petlja, sastoji se od sljedećih dijelova [29]:

- inicijalizacija – izraz koji se izvršava jednom, prije početka petlje. Tipično se koristi za inicijalizaciju brojača petlje.
- uvjet – izraz koji se evaluira na kraju svake iteracije petlje. Ako je uvjet ispunjen (*true*), izvršava se tijelo petlje.
- izraz – izraz koji se evaluira prije evaluacije uvjeta na kraju svake iteracije. Najčešće se koristi za ažuriranje brojača petlje.

Isječak koda 4.22: Primjer jednostavne for petlje

```
for (let i = 1; i <= 5; i++) {
  console.log(`Broj: ${i}`);
}

//Broj: 1
//Broj: 2
//Broj: 3
//Broj: 4
//Broj: 5
```

4.1.6.5. While petlja

U TypeScriptu, while petlja omogućuje ponavljanje izvršavanja bloka koda sve dok je određeni uvjet istinit. Ova vrsta petlje posebno je korisna kada unaprijed nije poznat broj ponavljanja, već se iteracija temelji na logičkom uvjetu koji se provjerava prije svake iteracije [30].

Petlja while najprije evaluira zadani uvjet. Ako je uvjet istinit (true), izvršava se kod unutar vitičastih zagrada. Kada uvjet postane neistinit (false), izvršavanje se prekida i program nastavlja s narednim dijelom koda [30].

Zbog toga što se uvjet provjerava prije ulaska u petlju, while petlja se naziva petlja s prethodnim testiranjem (engl. *Pretest loop*) [30].

Ukoliko je potrebno prekinuti petlju prije nego što uvjet postane neistinit, može se koristiti if uvjet unutar petlje zajedno s naredbom break [30].

Isječak koda 4.23: Primjer jednostavne while petlje

```
let broj: number = 1;
let brojac: number = 0;

while (broj <= 1000) {
if (broj % 5 === 0) {
console.log(broj); // Broj djeljiv sa 5
brojac++;
}
broj++;
}

console.log(Ukupno brojeva između 1 i 1000 djeljivih sa 5: ${brojac});
```

4.1.6.6. Do…while petlja

U programskom jeziku TypeScript, do…while petlja koristi se kada želimo osigurati da se blok naredbi izvrši barem jednom, bez obzira na to je li uvjet ispunjen na samom početku. To ju razlikuje od obične while petlje, koja najprije provjerava uvjet pa tek onda (ako je uvjet true) ulazi u petlju [31].

Isječak koda 4.24: Primjer jednostavne *do…while* petlje

```
let broj: number = 1;
do {
console.log("Broj je: " + broj);
broj++;
} while (broj <= 5);
// Broj je: 1
// Broj je: 2
// Broj je: 3
// Broj je: 4
// Broj je: 5
```

4.1.6.7. For…of petlja

For...of petlja koristi se za iteraciju nad iterabilnim objektima poput nizova (engl. *Array*), stringova i mapa (engl. *Map*). Ovo je napredna i “tipno” sigurna petlja koja omogućuje elegantno pristupanje svakom elementu kolekcije bez potrebe za pristupom putem indeksa [32].

Sintaksa for..of petlje prikazana je na sljedećem isječku koda [32]:

Isječak koda 4.25: Sintaksa for…of petlje

```
for (const element of iterable) {
  // izvrši nešto s elementom
}
```

U prethodno prikazanoj sintaksi, Element predstavlja trenutni element iz kolekcije (npr. niz ili string), a *iterable* je object koji se može iterirati.

4.1.6.8. For..in petlja

For...in petlja koristi se za iteraciju kroz svojstva objekta. Umjesto da vraća vrijednosti kao for...of, for...in vraća ključeve (nazive svojstava) objekta. Također se može koristiti za iteraciju po indeksima niza, iako to nije preporučeno u TypeScriptu zbog tipne nesigurnosti [32].

Sintaksa for..in petlje prikazana je na sljedećem isječku koda [32]:

Isječak koda 4.26: Sintaksa for…in petlje

```
for (const key in object) {
  // izvrši nešto s key ili object[key]
}
```

Isječak koda 4.27: Primjer korištenja for…in petlje

```
const osoba = {
ime: "Ivan",
godine: 20,
zanimanje: "Kuhar"
};

for (const svojstvo in osoba) {
console.log(${svojstvo}: ${osoba[svojstvo]});
}
```

Prethodni primjer ispisat će sljedeće vrijednosti kao rezultat for…in petlje:

```
ime: Ivan
godine: 20
zanimanje: Kuhar
```

4.2. Naprednije značajke

Osim osnovnih koncepata i tipova, TypeScript nudi i niz naprednijih značajki koje programerima omogućuju pisanje lakše održivog koda. Ove značajke proširuju mogućnosti

jezika i pomažu u rješavanju složenijih programskih izazova, te unaprjeđuju ukupnu kvalitetu i robusnost aplikacija. Postoje razni napredni koncepti programskog jezika TypeScript, uključujući pomoćne tipove (engl. *Utility Types*), generičke tipove, dekoratore, enumeratore, alate za statičku analizu koda te striktni način rada [33] [34].

4.2.1. Utility Types

TypeScript pruža značajku poznatu kao pomoćni tipovi (engl. *Utility Types*) koji pomažu u uobičajenim transformacijama tipova. Oni pomažu pri jednostavnom manipulacijom tipovima, čineći aplikacije robusnijima i lakšima za održavanje [35].

Pomoćni tipovi su skup generičkih tipova koje pruža TypeScript, a koji izvršavaju određene operacije na tipovima. Oni su dio TypeScript sustava tipova koji pomaže u konstruiranju novih tipova na temelju postojećih, često radi modificiranja svojstava ili izdvajanja informacija iz njih [35].

Ovo su neki od najčešće korištenih pomoćnih tipova [35]:

- Partial - Pretvara sva svojstva tipa u opcionalna. Ovo je korisno kada treba ažurirati samo nekoliko svojstava objekta
- Required – Čini sva svojstva tipa obaveznima, suprotno od Partial
- Readonly – Čini sva svojstva tipa samo za čitanje, što znači da ih nije moguće ponovo dodijeliti nakon njihove prve deklaracije
- Record – Kreira tip sa skupom svojstava zadanog tipa, a korisno je za mapiranje svojstava na isti tip
- Pick – Kreira tip odabirom skupa svojstava
- Omit – Stvara tip izostavljanjem skupa svojstava
- Exclude – Stvara tip isključivanjem članova unije
- Extract – Kreira tip izdvajanjem svih članova unije
- NonNullable – Kreira tip isključivanjem „null“ i „undefined“

Isječak koda 4.28: Primjer korištenja Partial

```
interface Korisnik {
  id: number;
  ime: string;
  godine: number;
}

type DjelomicniKorisnik = Partial<Korisnik>;

// Sva svojstva u DjelomicniKorisnik su opcionalna
const azurirajKorisnika: DjelomicniKorisnik = { ime: 'Ivan' };
```

Isječak koda 4.29: Primjer korištenja Required

```
interface Postavke {
  ime?: string;
  godine?: number;
}

type ObaveznePostavke = Required<Postavke>;

// Sva svojstva u ObaveznePostavke su obavezna
const postavke: ObaveznePostavke = { ime: 'Ivan', godine: 30 };
```

Isječak koda 4.30: Primjer korištenja Pick

```
interface Korisnik {
  id: number;
  ime: string;
  godine: number;
  email: string;
}

type PregledKorisnika = Pick<Korisnik, 'ime' | 'email'>;

// PregledKorisnika ima samo svojstva ime i email
const korisnik: PregledKorisnika = { ime: 'Ivan', email: 'ivan@example.com'
};
```

Isječak koda 4.31: Primjer korištenja Omit

```
// Koristeći prethodno definiran interface Korisnik
type KorisnikOsjetljivo = Omit<Korisnik, 'godine'>;

// KorisnikOsjetljivo ima id, ime, i email, ali ne i godine
const korisnikOsjetljivo: KorisnikOsjetljivo = {
  id: 1,
  ime: 'Ivan',
  email: 'ivan@example.com'
};
```

4.2.2. Generički tipovi

Generički tipovi (engl. *Generics*) u TypeScriptu predstavljaju moćan alat za kreiranje višekratno upotrebljivih komponenti, kao što su klase, funkcije i „aliasi“ tipova, bez potrebe za eksplicitnim definiranjem tipova koje koriste . Oni omogućuju pisanje koda koji može raditi s različitim tipovima podataka, zadržavajući pritom sigurnost tipova i izbjegavajući potrebu za korištenjem tipa „any“. Generički tipovi se u TypeScript kodu pojavljuju unutar uglatih zagrada, u formatu „<T>“, gdje „T“ predstavlja oznaku mjesta (engl. *Placeholder*) za tip koji će biti proslijeđen prilikom korištenja komponente. Oznaka „T“ djeluje slično kao parametri u funkcijama, ali umjesto vrijednosti, predstavlja tip [36] [37].

Generičke funkcije omogućuju stvaranje općenitijih metoda koje preciznije predstavljaju tipove koji se koriste i vraćaju . Umjesto da se koristi any tip, koji bi prihvatio bilo koji tip, ali ne bi pružio informaciju o vraćenom tipu, generički tipovi omogućuju specificiranje točnog tipa koji je prihvaćen i vraćen [38].

Isječak koda 4.32: Primjer korištenja generičkih tipova

```
// Definicija generičke funkcije 'kreirajPar'
// Ova funkcija prima dva argumenta, varijabla1 tipa S i varijabla2 tipa T,
// i vraća uređeni par tipa [S, T].
function kreirajPar<S, T>(varijabla1: S, varijabla2: T): [S, T] {
  return [varijabla1, varijabla2];
}

// Poziv funkcije s eksplicitno navedenim tipovima
let par1 = kreirajPar<number, string>(10, "primjer");
console.log(par1); // Ispisuje: [10, "primjer"]

// TypeScript također može zaključiti tipove generičkih parametara
// iz argumenata proslijeđenih funkciji.
let par2 = kreirajPar(false, "foi"); // S postaje boolean, T postaje string
console.log(par2); // Ispisuje: [false, "foi"]
```

U gornjem primjeru, funkcija “kreirajPar” koristi dva generička tipa, “S” i “T”. Kada se funkcija pozove, TypeScript može ili eksplicitno prihvatiti tipove navedene u uglatim zagradama (“<number, string>”) ili ih automatski zaključiti na temelju vrijednosti argumenata (“false” i ”foi”) .

Generički se tipovi također mogu koristiti za stvaranje generaliziranih klasa. Nakon definiranja imena klase, specificira se generički tipski parametar u uglatim zagradama (“<>”) [37] [38].

4.2.3. Dekoratori

Dekoratori u TypeScriptu predstavljaju posebnu vrstu deklaracija koje se mogu pridružiti klasama, njihovim metodama, pristupnicima (tzv. „getterima/setterima“), svojstvima ili parametrima. Omogućuju metaprogramiranje, odnosno promjenu ili proširenje ponašanja tih elemenata bez izmjene njihove izvorne implementacije. Dekoratori se označavaju simbolom „@“ ispred naziva funkcije dekoratora i primjenjuju se direktno na element koji se dekorira.
Za korištenje dekoratora u TypeScriptu potrebno je u konfiguracijskoj datoteci „tsconfig.json“ omogućiti eksperimentalnu podršku [39] [40]:

Isječak koda 4.33: Modifikacija datoteke „tsconfig.json“

```
{
  "compilerOptions": {
    "experimentalDecorators": true,
    "emitDecoratorMetadata": true
  }
}
```

Razlikujemo nekoliko tipova dekoratora [39]:

- Class Decorators – primjenjuje se na deklaraciju klase i može nadzirati ili mijenjati konstruktor klase

- Method Decorators – primjenjuje se na metodu klase i može mijenjati njeno ponašanje
- Accessor Decorators – deklarira se neposredno prije deklaracije pristupnika (gettera ili settera) i primjenjuje se na deskriptor svojstva pristupnika, omogućujući promatranje, izmjenu ili zamjenu definicija pristupnika
- Property Decorators – deklarira se neposredno prije deklaracije svojstva i koristi se prvenstveno za promatranje da je svojstvo određenog imena deklarirano za klasu, često za bilježenje metapodataka o tom svojstvu
- Parameter Decorators – deklarira se neposredno prije deklaracije parametra i primjenjuje se na funkciju za konstruktor klase ili deklaraciju metode, prvenstveno kako bi se promatralo da je parametar deklariran na metodi

Isječak koda 4.34: Primjer implementacije dekoratora metode

```
// Definicija dekoratora metode @logirajPozivMetode
function logirajPozivMetode(target: any, propertyKey: string, descriptor:
PropertyDescriptor) {
  // Spremi originalnu metodu
  const originalnaMetoda = descriptor.value;

  // Izmijeni deskriptor svojstva (PropertyDescriptor) tako da 'value' sada
pokazuje na novu funkciju
  descriptor.value = function (...args: any[]) {
    // Logika koja se izvršava prije poziva originalne metode
    console.log(`[DEKORATOR] Priprema za poziv metode: ${propertyKey}`);
    console.log(`[DEKORATOR] Argumenti proslijeđeni metodi:
${JSON.stringify(args)}`);

    // Pozovi originalnu metodu s originalnim kontekstom (this) i
argumentima
    const rezultat = originalnaMetoda.apply(this, args);

    // Logika koja se izvršava nakon poziva originalne metode
    console.log(`[DEKORATOR] Metoda ${propertyKey} je završila s
izvršavanjem.`);
    console.log(`[DEKORATOR] Vraćeni rezultat:
${JSON.stringify(rezultat)}`);

    // Vrati rezultat originalne metode
    return rezultat;
  };

  return descriptor;
}

// Primjena dekoratora na metodu klase
class UslugaPozdrava {
  prefiksPoruke: string;

  constructor(prefiks: string) {
    this.prefiksPoruke = prefiks;
  }
```

```
  @logirajPozivMetode // Primjena dekoratora na metodu 'kreirajPozdrav'
  kreirajPozdrav(imeKorisnika: string): string {
    console.log("   (Unutar originalne metode 'kreirajPozdrav')");
    return `${this.prefiksPoruke}, ${imeKorisnika}! Dobrodošao/la.`;
  }

  @logirajPozivMetode
  odjaviKorisnika(imeKorisnika: string): string {
    console.log("   (Unutar originalne metode 'odjaviKorisnika')");
    return `Doviđenja, ${imeKorisnika}! Hvala na posjeti.`;
  }
}

// Testiranje
const pozdravnaUsluga = new UslugaPozdrava("Pozdravi");

const porukaDobrodoslice = pozdravnaUsluga.kreirajPozdrav("Ana");
console.log("-----------------------------------------------------");
console.log(`Dobivena poruka dobrodošlice: ${porukaDobrodoslice}`);
console.log("=====================================================");

const porukaOdlaska = pozdravnaUsluga.odjaviKorisnika("Marko");
console.log("-----------------------------------------------------");
console.log(`Dobivena poruka odlaska: ${porukaOdlaska}`);
```

U primjeru iznad, dekorator „@logirajPozivMetode“ “omata” originalnu metodu na koju je primijenjen, u ovom slučaju „kreirajPozdrav“ i „odjaviKorisnika“. Kada se pozove dekorirana metoda, prvo se izvršava kod unutar dekoratora koji ispisuje informacije o pripremi poziva i argumentima. Nakon toga, dekorator poziva originalnu metodu, dopuštajući joj da obavi svoj stvarni posao. Na kraju, nakon što originalna metoda završi i vrati rezultat, dekorator ponovno preuzima kontrolu kako bi ispisao poruku o završetku i vraćenom rezultatu, prije nego što taj isti rezultat proslijedi dalje pozivatelju.

4.2.4. Enumeratori

Enumeratori (skraćeno „enumi“) u TypeScriptu omogućuju definiranje skupa imenovanih konstanti, čime se grupi srodnih vrijednosti daje smisleniji i čitljiviji oblik. Iako JavaScript izvorno ne podržava enume, TypeScript uvodi ovu značajku koja poboljšava čitljivost koda zamjenom “čarobnih” brojeva ili stringova jasnim nazivima, centralizira definiciju povezanih konstanti olakšavajući održavanje te pruža podršku za automatsko dovršavanje koda u razvojnim okruženjima [41].

Definiraju se ključnom riječi „enum“, a članovi unutar enuma prema zadanim postavkama dobivaju numeričke vrijednosti počevši od 0, no moguće je eksplicitno dodijeliti numeričke ili string vrijednosti. Korištenje enuma smanjuje se vjerojatnost grešaka i općenito doprinosi pisanju robusnijeg i lakše održivog koda [41].

Isječak koda 4.35: Primjer korištenja enumeratora

```
// Definiranje enumeratora
```

```
enum Day {
  Monday,
  Tuesday,
  Wednesday,
  Thursday,
  Friday,
  Saturday,
  Sunday
}

function isWeekend(day: Day): boolean {
  return day === Day.Saturday || day === Day.Sunday;
}

// Primjena:
const today = Day.Friday;

console.log("Jel vikend?", isWeekend(today)); // Jel vikend? False
```

4.2.5. Alati za statičku analizu koda

Statička analiza koda predstavlja proces automatskog ispitivanja izvornog koda bez njegovog izvršavanja, s ciljem otkrivanja potencijalnih grešaka, sigurnosnih ranjivosti, neusklađenosti sa standardima i drugih problema u ranoj fazi razvoja [42]. U TypeScript projektima, alati za statičku analizu značajno doprinose povećanju kvalitete, održivosti i sigurnosti koda, kao i smanjenju troškova otklanjanja grešaka [43].

Osnovni alat za statičku analizu u TypeScriptu je TypeScript kompajler (tsc). On provodi statičku provjeru tipova, identificira neslaganja tipova, varijable koje nisu inicijalizirane i druge tipične greške prije samog pokretanja aplikacije. Konfiguriranjem strožih pravila, moguće je dodatno povećati sigurnost tipova i smanjiti rizik od pogreške [44].

Linting je posebna vrsta statičke analize koja se fokusira na sintaksu, stil pisanja i pridržavanje dogovorenih standarda [42]. Najpopularniji alat za TypeScript danas je ESLint, koji omogućuje definiranje i prilagodbu pravila, integraciju s raznim editorima (npr. Visual Studio Code) i podršku za automatsko ispravljanje manjih problema. Iako je TSLint bio specifičan za TypeScript, danas se preporučuje prelazak na ESLint zbog šire podrške i aktivnog razvoja [43].

SonarQube je alat za dinamičku analizu koda koji podržava TypeScript i mnoge druge programske jezike. Omogućuje otkrivanje čestih grešaka u kodiranju te mjerenje kvalitete koda. Kroz bogat skup dodataka i preglednu nadzornu ploču, programerima olakšava prepoznavanje i brzo otklanjanje problema u kodu. Jedna od ključnih funkcionalnosti SonarQube-a je analiza pokrivenosti koda testovima (engl. *Code coverage*). Na taj način timovi mogu uočiti dijelove aplikacije koji nisu obuhvaćeni testiranjem i time bolje planirati daljnji razvoj i testiranje softvera [43].

4.2.6. Striktni način rada (engl. *Strict mode*)

Striktni način rada (engl. *Strict mode*) u TypeScriptu je opcija unutar „tsconfig.json“ datoteke koja uključuje širok spektar provjera tipova s ciljem pružanja snažnijih garancija ispravnosti programa [45].

Kada je ova opcija omogućena, programski kod podliježe strožim pravilima statičke provjere tipova, što u konačnici znači sigurniji kod [46]. Preporučuje se uključivanje striktnog načina rada za sve nove projekte kako bi se od samog početka osigurala najviša razina sigurnosti tipova. Međutim, implementacija može biti zahtjevnija prilikom migracije postojećih JavaScript projekata zbog potencijalno velikog broja prijavljenih grešaka tipova [47].

Striktni način rada aktivira se postavljanjem opcije „strict“ na vrijednost „true“ unutar „compilerOptions“ objekta (u „tsconfig.json“ datoteci), kao što je prikazano u sljedećem isječku koda [46]:

Isječak koda 4.36: Uključivanje strict mode u tsconfig.json datoteci

```
{
  "compilerOptions": {
    "strict": true,
    // ... ostale opcije
  }
}
```

Opcija strict zapravo je prečac koji automatski omogućuje skup pojedinačnih, strožih pravila provjere. Moguće je imati uključenu opciju strict, ali po potrebi isključiti određena pojedinačna pravila. Glavne opcije koje strict način rada obuhvaća su [47]:

- noImplicitAny: Prijavljuje grešku na izrazima i deklaracijama koje imaju implicitno definiran any tip
- noImplicitThis: Prijavljuje grešku kada „this“ izraz ima implicitno definiran any tip
- alwaysStrict: Analizira kod u JavaScript striktnom načinu i emitira "use strict"; na vrhu svake izlazne datoteke [46]
- strictBindCallApply: Omogućuje strožu provjeru bind, call i apply metoda na funkcijama
- strictNullChecks: Osigurava da vrijednosti „null“ i „undefined“ nisu prihvatljive vrijednosti za tipove koji ih eksplicitno ne uključuju
- strictFunctionTypes: Onemogućuje „dvojnu“ provjeru parametara za tipove funkcija
- strictPropertyInitialization: Osigurava da su svojstva klase inicijalizirana u konstruktoru ili da imaju zadanu vrijednost. Kako bi ova opcija radila, potrebno je omogućiti strictNullChecks opciju

Isječak koda 4.37: Opcija noImplicitAny

```
// Greška uz noImplicitAny: parametri x i y imaju implicitan tip "any"
function zbrojNeIspravno(x, y) {
return x + y;
}
function zbrojIspravno(x: number, y: number): number {
return x + y;
}
console.log(zbrojNeIspravno(2,3));
console.log(zbrojIspravno(2,3));
```

U prethodnom primjeru, ako je opcija noImplicitAny isključena, obje funkcije radit će bez greške. Ukoliko uključimo provjeru za noImplicitAny, tada se će funkcija „zbrojNeIspravno“ javiti pogreške:

- Parameter 'x' implicitly has an 'any' type.
- Parameter 'y' implicitly has an 'any' type.

Isječak koda 4.38: Primjer ispravnog i neispravnog programskog koda s obzirom na uključenu opciju „strictPropertyInitialization“

```
// Primjer neispravne klase
class Osoba {
ime: string;
// Javlja se pogreška: Property 'ime' has no initializer and is not
definitely assigned in the constructor.

constructor() {
// ime vrijednost nije postavljena
}
}

// Primjer ispravne klase
class Osoba2 {
ime: string;

constructor() {
this.ime = "Nepoznato";
}
}
```

U prethodnom isječku koda, ukoliko je opcija „strictPropertyInitialization“ postavljena na vrijednost „true“, tada se javlja pogreška da nije postavljena vrijednost za „ime“ u konstruktoru klase. Ukoliko opciju isključimo, pogreška se neće javiti za istu tu klasu.

5. Vue.js

Vue.js progresivni je JavaScript okvir namijenjen izradi korisničkih sučelja. Temelji se na standardnim web tehnologijama: HTML, CSS i JavaScript. Pruža deklarativan i komponentno orijentiran model programiranja koji omogućuje efikasnu izradu jednostavnih i složenih web aplikacija [48].

Vue.js je osmislio Evan You tijekom rada u Googleu na AngularJS aplikacijama. Razvijen je kao rješenje za postizanje bolje performansi i fleksibilnosti u usporedbi s tadašnjim postojećim rješenjima. Vue.js koristi neke ideje iz Angulara (poput predložaka), ali odbacuje kompleksnost i strogu strukturu te nudi jednostavniji i lakši pristup razvoju [48].

Za razliku od mnogih monolitnih okvira, Vue.js temelji se na postupnom modelu izgradnje. Vue.js se ističe jednostavnošću, početna aplikacija može se napisati u svega nekoliko redaka koda. Automatizirani mehanizmi poput povezivanja podataka s DOM-om, organizacije konfiguracija i automatskog kreiranja promatrača (engl. *Watcher*) za komponente značajno olakšavaju razvoj [48].

5.1. Prednosti korištenja Vue.js

Jedna od najvećih prednosti Vue.js okvira jest njegova jednostavnost. Razvoj Vue.js aplikacije može započeti s vrlo malo koda, dok sam okvir u pozadini automatski izvršava niz zadataka koji bi inače zahtijevali dodatni napor programera. Na primjer, Vue.js automatski povezuje podatke s prikazom (DOM-om), organizira konfiguracije te za svaku komponentu stvara watcher koji reagira na promjene stanja. Upravo ta jednostavnost čini Vue.js posebno pogodnim za brzi razvoj modernih web aplikacija [48].

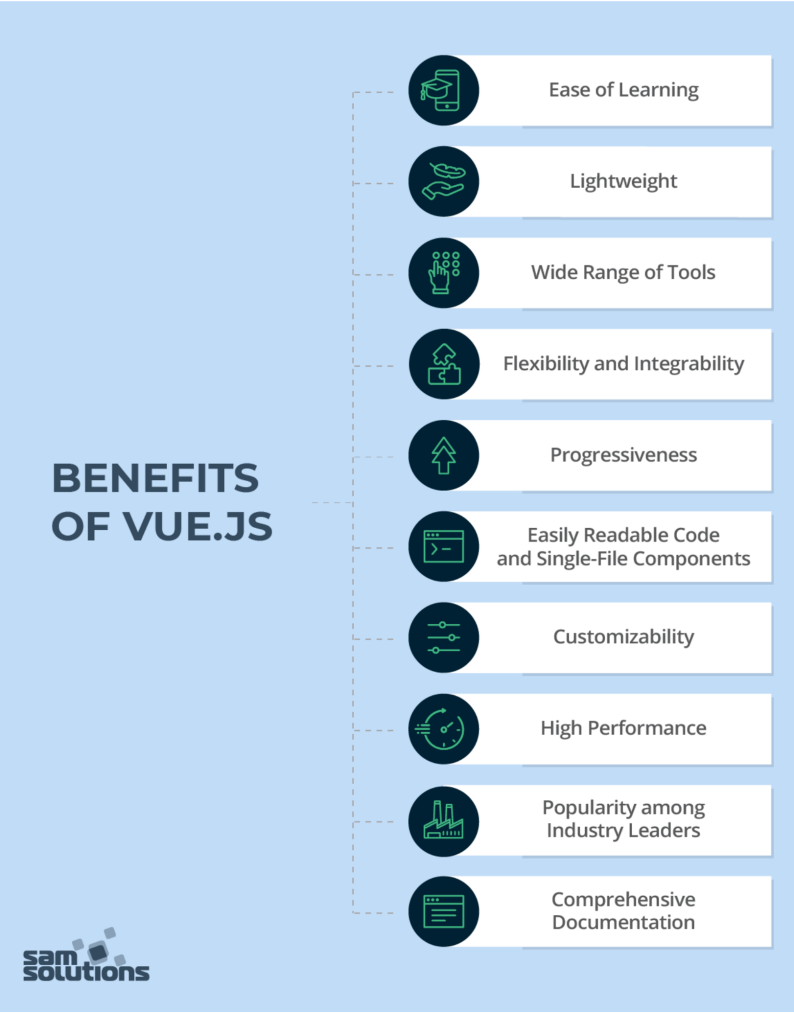
Svaki dio aplikacije razdvaja se u samostalne i neovisne komponente koje obuhvaćaju logiku, strukturu i stil. Time se postiže modularnost i jasnoća u kodu, ali i omogućava njegovo jednostavno testiranje, ponovno korištenje i održavanje. Komponente su lagane, pregledne i mogu se višekratno koristiti kroz različite dijelove aplikacije, što doprinosi bržem razvoju i smanjenju dupliciranja koda [48].

Vue.js je izuzetno lagan okvir, s veličinom od svega oko 20 KB (18 KB nakon gzip kompresije). Ova mala veličina osigurava brzo preuzimanje i instalaciju, što izravno doprinosi bržem učitavanju stranica i boljem korisničkom iskustvu te pozitivno utječe na SEO rangiranje [49] [50].

Dostupni su brojni alati i okviri koji dodatno olakšavaju razvoj višeplatformskih aplikacija pomoću Vue.js okvira. „Quasar Framework**“**, izgrađen na Vue.js-u, omogućuje kreiranje responzivnih web aplikacija, mobilnih aplikacija putem „Cordove“ ili „Capacitor“ alata, te

desktop aplikacija korištenjem „Electron“ okvira. NativeScript-Vue integrira Vue.js s NativeScriptom i omogućuje izradu izvornih mobilnih aplikacija za Android i iOS, pri čemu se zadržava pristup izvornim API-jima i komponentama, što osigurava izvrsne performanse i prirodan osjećaj korištenja. Za izradu responzivnih web aplikacija posebno se ističe Vuetify, biblioteka temeljena na principima Material Designa, koja pruža bogat skup unaprijed definiranih komponenti pogodnih i za desktop i za mobilno okruženje. Također, Electron-Vue omogućava izradu desktop aplikacija kombiniranjem Vue.js-a s Electronom, nudeći unaprijed konfigurirano razvojno okruženje za brzu izradu i distribuciju aplikacija [49].

Još jedna važna značajka Vue.js-a jest korištenje virtualnog DOM-a, što omogućuje učinkovitije ažuriranje korisničkog sučelja jer se promjene prvo vrše na virtualnoj razini, a tek potom reflektiraju u stvarnom prikazu. Također, Vue.js implementira reaktivno dvosmjerno vezivanje podataka, što znači da su svi dijelovi aplikacije uvijek međusobno sinkronizirani. Bilo kakva promjena modela automatski se odražava u prikazu i obrnuto, čime se smanjuje potreba za dodatnim kodom i olakšava održavanje [51].



Slika 5. Prednosti Vue.js; prema [50]

5.1.1. Vue.js 3

S izlaskom verzije Vue.js 3, ovaj popularni *JavaScript* okvir napravio je značajan iskorak naprijed, donoseći niz poboljšanja koja dodatno potvrđuju njegovu poziciju među vodećim alatima za razvoj korisničkih sučelja. Iako zadržava jednostavnost i fleksibilnost po kojoj je poznat, Vue.js 3 uvodi nove funkcionalnosti koje ga čine još moćnijim i učinkovitijim [52].

Jedna od najvažnijih novosti u Vue.js 3 jest optimizacija performansi. Zahvaljujući potpunom prepisivanju u TypeScript i uvođenju reaktivnog sustava temeljenog na *Proxy* objektima, Vue.js 3 postiže brže izvođenje i manju veličinu paketa. Time se značajno skraćuje vrijeme učitavanja aplikacija, što je posebno korisno kod velikih i složenih projekata. Druga ključna novost je Composition API, koji uz postojeći Options API nudi alternativni način strukturiranja komponenata [52].

Vue.js 3 također omogućuje korištenje više v-model veza unutar jedne komponente, čime se poboljšava podrška za dvosmjerno povezivanje podataka. Poboljšan je i reaktivni sustav, omogućujući izravne manipulacije podacima bez potrebe za pomoćnim funkcijama [53].

Nove značajke, poput portala i fragmenta dodatno proširuju fleksibilnost Vue.js aplikacija. Portali omogućuju prikaz sadržaja izvan roditeljskog DOM stabla, dok Fragmenti omogućuju više korijenskih čvorova bez dodatnih omotača. Također, uvedene su promjene u globalnom montiranju i konfiguraciji, pojednostavljujući način inicijalizacije aplikacije i rada s globalnim opcijama [53].

U konačnici, Vue.js 3 donosi značajna poboljšanja u pogledu performansi, fleksibilnosti i razvojne ergonomije, čineći ga izuzetno atraktivnim rješenjem za moderne web aplikacije [52].

5.1.2. Composition API

Composition API predstavlja skup funkcionalnosti koje olakšavaju razumijevanje i organizaciju poslovne logike unutar aplikacije. Za razliku od Options API, koji strukturira kod prema tehničkim segmentima, Composition API omogućuje da se kod organizira prema funkcionalnostima koje se razvijaju. Takav pristup omogućuje jasnije odvajanje logičkih cjelina i preglednije strukturiranje aplikacijskog koda [53].

Korištenjem Composition API programeri dobivaju veću fleksibilnost u pisanju i ponovnoj upotrebi logike, uz niz dodatnih prednosti. Uvođenjem setup metode, kod postaje pregledniji i lakši za održavanje, što je vidljivo i u jednostavnom primjeru gdje se vrijednost varijable udvostručuje [53].

Composition API koristan je iz sljedećih razloga [52]:

- Omogućuje grupiranje povezane logike, što olakšava upravljanje složenim komponentama
- Pojednostavljuje ponovnu upotrebu koda, posebno kada je riječ o dijeljenju funkcionalnosti između komponenti
- Prirodno se nadopunjuje s TypeScriptom, što olakšava pisanje koda sa snažnim tipovima

Isječak koda 5.1: Primjer korištenja Options API

```
<template>
  <div>
    <p>Broj: {{ count }}</p>
    <button @click="increment">Povećaj</button>
  </div>
</template>

<script>
export default {
  data() {
    return {
      count: 0
    }
  },
  methods: {
    increment() {
      this.count++;
    }
  }
}
</script>
```

Isječak koda 5.2: Primjer korištenja Composition API

```
<template>
  <div>
    <p>Broj: {{ count }}</p>
    <button @click="increment">Povećaj</button>
  </div>
</template>

<script setup>
import { ref } from 'vue'

const count = ref(0)

function increment() {
  count.value++
}
</script>
```

U prethodim primjerima, oba programska koda imaju jednaku funkcionalnost. Razlika je u tome kako se definiraju varijable i programska logika:

- Options API koristi data() i metode (engl. *methods*)

- Composition API koristi ref() i script za bolju organizaciju i ponovnu iskoristivost programske logike

5.2. Temeljni koncepti

U nastavku rada, bit će opisani ključni koncepti potrebni za izradu web aplikacije koristeći Vue.js. Virtual DOM i životni ciklus komponente samo su neki od temeljnih koncepata, a oni čine Vue.js izrazito pogodnim alatom za razvoj modernih web aplikacija koje su lako održive i proširive [52] [53].

5.2.1. Sustav reaktivnosti

Sustav reaktivnosti je ključna značajka Vue.js-a koja omogućuje automatsko ažuriranje komponenti kada se njihovi podaci promijene [54]. Kada se vrijednost koju komponenta koristi promijeni, Vue.js automatski detektira promjenu i ažurira *DOM* u skladu s njom [55].

Vue.js 3 donosi unaprijeđeni sustav reaktivnosti koji omogućuje jednostavno praćenje i ažuriranje stanja aplikacije. Ključni alati u ovom sustavu su funkcije ref() i reactive(), koje omogućuju definiranje reaktivnog stanja. Korištenjem ref() moguće je stvoriti reaktivnu vrijednost koja se može pratiti i mijenjati. Kada se ref koristi unutar Vue.js komponente, promjene njegove vrijednosti automatski ažuriraju prikaz u korisničkom sučelju [56].

Također, Vue.js nudi i reactive() funkciju koja omogućuje da cijeli objekt postane reaktivan. Sve promjene svojstava takvih objekata, uključujući i one duboko ugniježdene, automatski se detektiraju zahvaljujući Vue.js mehanizmu temeljenom na JavaScript Proxy objektima. Međutim, važno je napomenuti da reactive() radi isključivo s objektima i nije pogodan za primitivne tipove podataka, zbog čega se za njih preporučuje korištenje ref() [56].

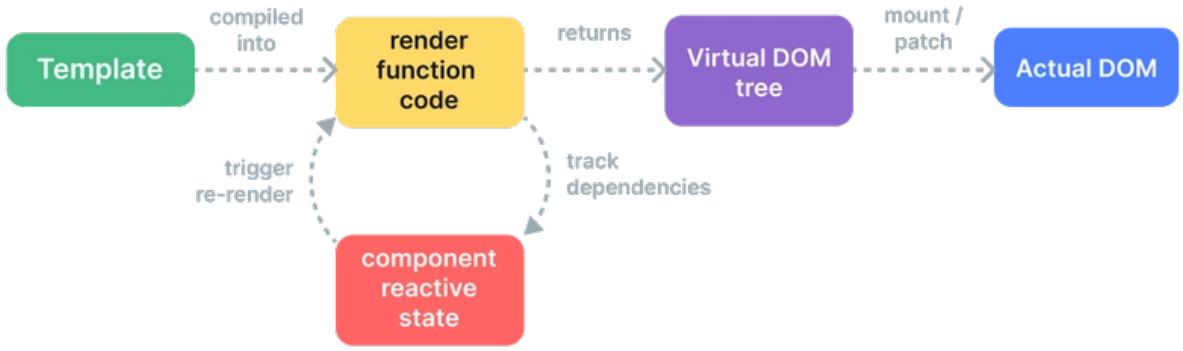
Vue.js također optimizira performanse tako što odgađa ažuriranje DOM-a dok se ne završi trenutni ciklus ažuriranja. Za čekanje da se ažuriranje DOM-a završi nakon promjene stanja, moguće je koristiti globalni API nextTick() [56].

Isječak koda 5.3: Primjer korištenja nextTick() metode [56]

```
import { nextTick } from 'vue'

async function increment() {
  count.value++
  await nextTick()
  // Now the DOM is updated
}
```

5.2.2. Virtualni DOM (engl. *Virtual* DOM)

Vue.js koristi virtualni DOM (skraćeno V-DOM) kako bi optimizirao performanse manipulacije DOM-om. Virtualni DOM je koncept gdje se idealna, odnosno “virtualna”, reprezentacija korisničkog sučelja čuva u memoriji i sinkronizira sa stvarnim DOM-om. Kada se komponenta renderira, Vue.js kreira JavaScript stablo objekata koje predstavlja strukturu stvarnog DOM-a. Prilikom promjene stanja komponente, stvara se novo stablo Virtualnog DOM-a, a Vue.js zatim uspoređuje novo i staro virtualno stablo kako bi identificirao razlike (proces poznat kao *diffing*). Nakon toga, Vue.js primjenjuje minimalan skup promjena na stvarni DOM (proces zvan *patching*), što značajno smanjuje broj skupih DOM operacija i dovodi do znatnih poboljšanja performansi [57].



Slika 6. Vue.js Virtual DOM; prema [58]

5.2.3. Životni ciklus komponente (engl. *Lifecycle Hooks*)

Komponente u Vue.js-u prolaze kroz različite faze tijekom svog „života“: od stvaranja, preko ažuriranja, do uklanjanja iz DOM-a. U svakoj od tih faza moguće je izvršiti vlastiti kod pomoću posebnih funkcija koje se nazivaju metode životnog ciklusa (engl. *lifecycle hooks*). One omogućuju programeru da precizno upravlja ponašanjem komponente u svakom trenutku njenog postojanja [59].

Na primjer, beforeCreate metoda se poziva prije nego što je komponenta u potpunosti inicijalizirana, što znači da podatkovna svojstva još nisu dostupna. S druge strane, created se izvršava nakon inicijalizacije, kada su već dostupni podaci, metode i reaktivnost. U fazi mounted, komponenta je već dodana u DOM, što omogućuje manipulaciju elementima stranice putem referenci. Metode beforeUpdate i updated služe za reagiranje na promjene podataka i ponovna kreiranje komponente, dok beforeUnmount i unmounted omogućuju čišćenje resursa prije i nakon uklanjanja komponente iz DOM-a [59].

Dodatne metode poput activated i deactivated koriste se kod keširanih komponenti unutar KeepAlive, dok errorCaptured omogućuje hvatanje grešaka u komponentama “djeci” (engl. *Child component*). Posebne metode poput renderTracked i renderTriggered pomažu u dijagnostici i praćenju reaktivnih promjena tijekom razvoja. Tu je i serverPrefetch metoda, koja je specifična za aplikacije koje koriste SSR (engl. *Server-Side Rendering) [59]*.

Korištenjem ovih metoda, programer dobiva snažnu kontrolu nad ponašanjem komponenti, olakšavajući organizaciju koda, upravljanje memorijom i otklanjanje grešaka [59].

Isječak koda 5.4: Primjer korištenja unmounted metode

```
<template>
  <h2>Primjer koristenja unmounted</h2>
</template>

<script>
export default {
  unmounted() {
    alert("Komponenta se maknula (unmounted)!");
  }
}
</script>
```

5.3. Struktura Vue.js komponente

U Vue.js, komponente su osnovni dijelovi aplikacije. Svaka komponenta sadrži vlastiti izgled, logiku i stilove, što olakšava organizaciju i ponovno korištenje koda. Tako se aplikacije grade kao skup manjih, povezivih dijelova [60] [61].

5.3.1. Komponenta kao osnova

Komponente su temeljni gradivni blokovi Vue.js aplikacija koje omogućuju podjelu korisničkog sučelja na neovisne i višekratno upotrebljive dijelove. Ovakav pristup dopušta da se o svakom dijelu sučelja razmišlja u izolaciji, što uvelike olakšava razvoj i održavanje složenih aplikacija. Cjelokupna aplikacija se obično organizira kao stablo ugniježđenih komponenti, slično kao što se gnijezde „nativni“ (engl. *Native*) HTML elementi [60].

U središtu razvoja aplikacija u Vue.js nalazi se koncept komponente. Svaka komponenta predstavlja samostalnu cjelinu koja može sadržavati vlastiti predložak (engl. *Template*), podatke, metode i stilove. Najčešći način definiranja komponente u modernom Vue.js okruženju jest korištenje tzv. *Single File Components* (skraćeno SFC), odnosno “.vue”

datoteka. Unutar jedne takve datoteke istovremeno se definiraju struktura, logika i izgled komponente [60].

5.3.2. Predlošci (engl. *Template*) i direktive (engl. *Directives*)

U Vue.js-u, predložak (engl. *Template*) predstavlja HTML dio aplikacije i temelj je za izgradnju korisničkog sučelja. Predlošci omogućuju deklarativno definiranje strukture aplikacije, povezivanje s podacima i interakciju s komponentama na jasan i intuitivan način [62].

Vue.js koristi HTML temeljenu sintaksu predložaka koja omogućuje deklarativno povezivanje prikazanog DOM-a s podacima komponente. Svi Vue.js predlošci su sintaktički ispravan HTML, što znači da ih mogu analizirati standardni preglednici i HTML parseri [63].

Najosnovniji oblik povezivanja podataka je interpolacija teksta pomoću dvostrukih vitičastih zagrada (tzv. “mustache” sintaksa), npr. „{{ message }}“, gdje se vrijednost automatski ažurira kad se podaci promijene [63].

Predlošci u Vue.js-u proširuju mogućnosti standardnog HTML-a putem direktiva – posebnih atributa s prefiksom „v-„ (npr. „v-if“, „v-for“, „v-bind“ i „v-on“). Direktive omogućuju uvjetno renderiranje, petlje, obradu događaja i druge napredne funkcionalnosti izravno u HTML-u predloška.

Popis Vue.js direktiva prikazan je u tablici:

Tablica 6: Vue.js direktive

Direktiva	Opis
v-bind	Povezuje atribut HTML elementa s varijablom unutar Vue instance
v-if	Kreira HTML elemente ovisno o uvjetu. Direktive v-else-if i v-else koriste se zajedno s v-if
v-show	Određuje hoće li HTML element biti vidljiv ili ne, ovisno o uvjetu
v-for	Generira listu HTML elemenata na temelju niza iz Vue instance koristeći “for” petlju
v-on	Povezuje događaj na HTML elementu s JavaScript izrazom ili metodom Vue instance. Također omogućuje specifičnije reagiranje na događaje pomoću modifikatora događaja

v-model	Koristi se u HTML formama s elementima poput <form>, <input> i <button>. Omogućuje dvosmjerno povezivanje između elementa za unos i podatka u Vue instanci

(Izvor: W3schools, bez dat.)

5.3.3. Komunikacija među komponentama

Props (engl. *Properties*) omogućuju jednostavan način prijenosa podataka iz roditeljske komponente u njezinu djecu. Roditeljska komponenta definira podatke koje želi proslijediti, a djetetu ih dostavlja preko atributa koji su unaprijed deklarirani u samoj komponenti. Prednost ovakvog pristupa jest reaktivnost - svaki put kada se vrijednost props-a u roditelju promijeni, promjena se automatski odražava i u djetetu. Ovo čini props idealnim za jednostavan prijenos podataka s roditelja na dijete [64].

Suprotan smjer komunikacije ostvaruje se emitiranjem događaja (od djeteta prema roditelju). Umjesto izravnog prijenosa podataka, dijete šalje “signal” roditelju da se nešto dogodilo. Ovo se postiže korištenjem metode “$emit”, kojom dijete pokreće vlastiti događaj i može uz njega poslati i podatke. Roditelj tada može presresti taj događaj pomoću v-on direktive ili skraćenog oblika “@”, čime definira što treba napraviti kada se događaj dogodi. Važno je da ime događaja u roditelju odgovara onome što je dijete emitiralo. Ova metoda je posebno korisna kada komponenta treba obavijestiti roditelja o nekoj akciji [64].

Komponente se mogu višestruko koristiti u aplikaciji, pri čemu svaka instanca održava vlastito stanje. Na taj način postiže se modularnost i ponovna iskoristivost koda. Da bi se komponenta koristila unutar druge, potrebno ju je najprije importirati, a zatim uključiti unutar njezinog predloška. Vue.js omogućuje i globalnu registraciju komponenata, što olakšava njihovo korištenje u više dijelova aplikacije bez potrebe za ponovnim uvozom [60].

Isječak koda 5.5: Primjer korištenja $emit i props (roditeljska komponenta)

```
<template>
  <div>
    <h2>Roditelj komponenta</h2>
    <ChildComponent
      message="Zdravo dijete!"
      @reply="handleReply"
    />
    <p>Poruka od djeteta: {{ childResponse }}</p>
  </div>
</template>

<script setup>
import { ref } from 'vue'
```

```
import ChildComponent from './ChildComponent.vue'

const childResponse = ref('')

function handleReply(msg) {
  childResponse.value = msg
}
</script>
```

Isječak koda 5.6: Primjer korištenja $emit i props (dijete komponenta)

```
<template>
  <div>
    <p>Poruka od roditelja: {{ message }}</p>
    <button @click="notifyParent">Pošalji odgovor roditelju</button>
  </div>
</template>

<script setup>
import { defineProps, defineEmits } from 'vue'

const props = defineProps(['message'])
const emit = defineEmits(['reply'])

function notifyParent() {
  emit('reply', 'Pozdrav od djeteta!')
}
</script>
```

U prethodnom primjeru programskog koda, Props omogućuje slanje podataka iz "roditeljske" u komponentu "dijete" tako da se atributi definiraju i prenose kao ulazne vrijednosti. S druge strane, emit omogućuje komponenti "dijete" da pošalje događaj natrag "roditeljskoj" komponenti.

5.4. Upravljanje stanjem i navigacijom

U modernim Vue.js aplikacijama, upravljanje stanjem (engl. *State management*) i usmjeravanje (engl. *Routing*) predstavljaju dva ključna aspekta razvoja složenih korisničkih sučelja. Upravljanje stanjem omogućuje sinkronizaciju i dijeljenje podataka između različitih komponenti, dok usmjeravanje omogućuje korisnicima jednostavno kretanje kroz različite "stranice" ili dijelove aplikacije bez ponovnog učitavanja cijele stranice [65] [66].

5.4.1. Upravljanje stanjem (engl. *State*)

U Vue.js-u, svaka komponenta već ima vlastito reaktivno stanje, ali kako aplikacija raste, često je potrebno dijeliti podatke između više komponenti. Osnovni način dijeljenja

stanja je “podizanje” (engl. *lifting*) stanja na zajedničkog roditelja i prosljeđivanje putem props-a, no za veće aplikacije preporučuje se korištenje centraliziranog spremišta (engl. Store) [66].

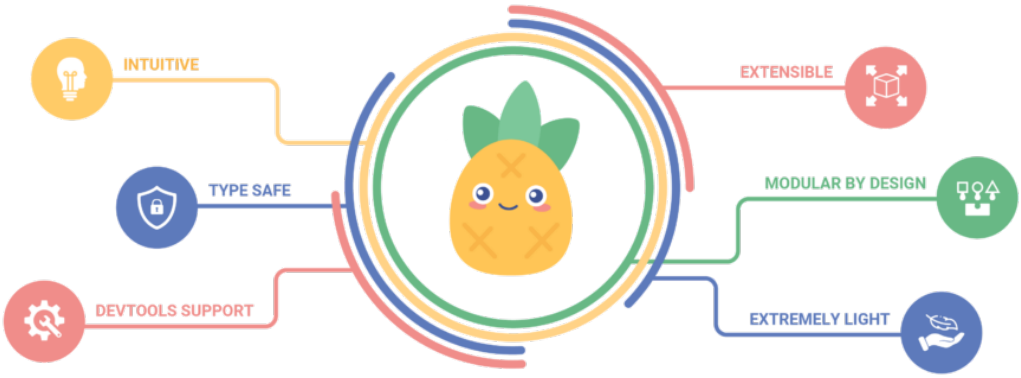
Uvođenjem Composition API-ja u Vue.js 3, „Pinia“ se istaknula kao jednostavnije, lakše i modernije rješenje za upravljanje globalnim stanjem u odnosu na prethodni „Vuex“. Pinia se prirodno integrira s Composition API-jem, omogućujući intuitivno i modularno upravljanje podacima kroz cijelu aplikaciju [67].

Pinia je službena biblioteka za upravljanje stanjem u Vue.js aplikacijama, razvijena od strane „Vue core“ tima kao nasljednik Vuex-a. Za razliku od Vuex-a, koji je sada u fazi održavanja, Pinia nudi jednostavniji i moderniji *API*, bolju podršku za Composition API i napredno tipiziranje u TypeScriptu [66].

Da bi Pinia radila u aplikaciji, potrebno ju je registrirati u glavnoj „.js“ (ili „.ts“ ako se koristi TypeScript) datoteci pomoću „app.use(pinia)“, čime postaje dostupna svim komponentama [67].

Isječak koda 5.7: Primjer korištenja Pinia biblioteke

```
import { defineStore } from 'pinia'

export const koristiBrojacStore = defineStore('brojac', {
  state: () => ({
    vrijednost: 0
  }),
  actions: {
    povecaj() {
      this.vrijednost++
    },
    resetiraj() {
      this.vrijednost = 0
    }
  }
})
```



Slika 7. Pinia; prema [68]

5.4.2. Usmjeravanje (engl. *Routing*)

U suvremenim web aplikacijama temeljenim na Vue.js-u, usmjeravanje (engl. *Routing*) omogućuje korisnicima navigaciju između različitih prikaza ili "stranica" bez ponovnog učitavanja cijele aplikacije, što je temelj za izradu Single-Page Applications (skraćeno SPA) [69].

Za razliku od tradicionalnih web aplikacija gdje svaki klik na poveznicu uzrokuje novo učitavanje stranice sa servera, SPA aplikacije koriste klijentsko usmjeravanje: JavaScript aplikacija presreće navigaciju, dinamički dohvaća podatke i ažurira prikaz bez da se mora u potpunosti izvršiti osvježavanje stranice, čime se postiže brže i ugodnije korisničko iskustvo [69].

Vue Router službena je i najčešće korištena biblioteka za upravljanje klijentskim usmjeravanjem u Vue.js aplikacijama. Ona omogućuje mapiranje URL-ova na određene komponente, što znači da svaka ruta može prikazati određenu Vue.js komponentu. Vue Router koristi deklarativnu konfiguraciju ruta, što olakšava razumijevanje i održavanje navigacijske logike aplikacije [70].

Ključne značajke Vue Router-a su [70]:

- Declarative Routing – Vue Router koristi deklarativnu konfiguraciju za definiranje ruta aplikacije, što omogućuje jednostavno razumijevanje i održavanje.
- Nested Routes – Omogućeno je kreiranje ugniježđenih ruta, što omogućuje izgradnju složenijih struktura aplikacije gdje su komponente ugniježđene jedna u drugu
- Route Parameters – Vue Router podržava dinamičke parametre ruta, čime se omogućuje prijenos podataka putem „URL-a"
- Navigation Guards – Implementacija zaštita (engl. *Navigation guards*) omogućuje kontrolu pristupa određenim rutama i osigurava provjeru da korisnici imaju potrebne dozvole za pristup

Isječak koda 5.8: Primjer definiranja ruta za Vue Router

```
const routes: Array<RouteRecordRaw> = [
  {
    path: "/login",
    name: "Login",
    component: Login,
    meta: {
      requiresAuth: false,
      showInSidebar: false,
    },
  },
  {
    path: "/register",
    name: "Register",
```

```
    component: Register,
    meta: {
      requiresAuth: false,
      showInSidebar: false,
    },
  }
// Ostatak ruta…
];
```

Isječak koda 5.9: Prebacivanja korisnika na stranicu za prijavu ako nije prijavljen

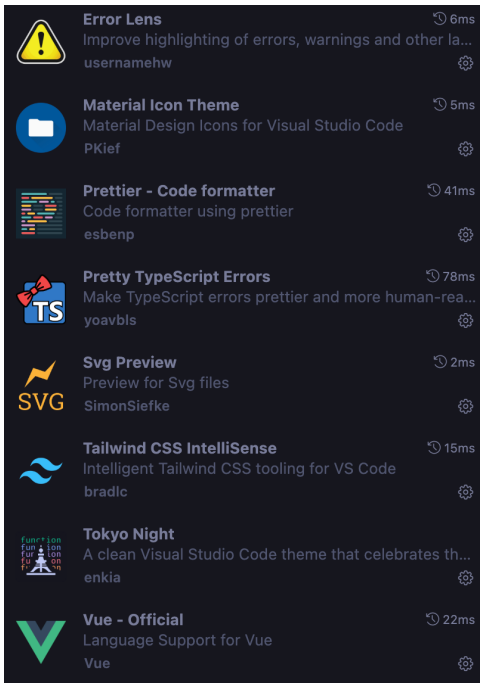
```
router.beforeEach(async (to) => {
  if (to.meta.requiresAuth) {
    const user = await getCurrentUser();

    if (!user) {
      return { name: "Login" };
    }
  }

  return true;
});
```

6. Izrada praktičnog dijela

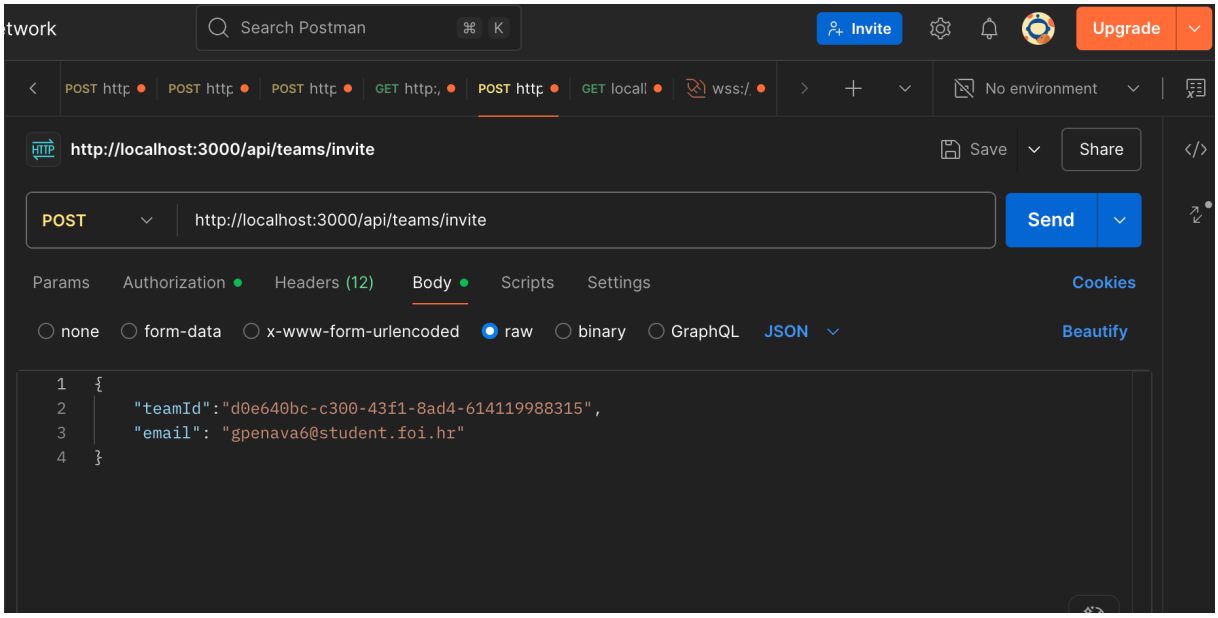
U sklopu praktičnog dijela ovog rada izrađena je moderna web aplikacija čiji je cilj omogućiti korisnicima upravljanje timovima i raspodjelu zadataka među članovima tima. Aplikacija funkcionalno podsjeća na alat kao što je “Jira”, s naglaskom na jednostavnost korištenja i brzu komunikaciju između korisnika.

Za samu izradu aplikacije, korišten je Visual Studio Code editor s instaliranim nadogradnjama koje su značajno olakšale razvojni proces. Ekstenzije koje sam koristio uključuju Vue - Official za podršku Vue komponentama i sintaksi, Error Lens za brzo vizualno uočavanje grešaka unutar koda, Prettier za automatsko formatiranje koda prema definiranim pravilima, Pretty TypeScript Errors za pregledniji prikaz TypeScript pogrešaka, te Tailwind CSS IntelliSense za automatsko dovršavanje Tailwind klasa i prikaz mogućih stilova unutar editora. Korištenje ovih alata doprinijelo je većoj produktivnosti, dosljednosti u pisanju koda i boljoj organizaciji cijelog projekta.

Osim Visual Studio Code editora, u razvoju aplikacije korišteni su i dodatni alati koji su bili ključni za testiranje i upravljanje sustavom. Postman je korišten za testiranje REST API endpointa backend sustava, što je omogućilo jednostavno slanje zahtjeva i provjeru odgovora API-ja. Za upravljanje PostgreSQL bazom podataka korišten je DataGrip, napredni alat za rad s bazama koji je omogućio pregled, uređivanje i izvršavanje SQL upita na jednostavan i pregledan način.



Slika 8. Visual Studio Code nadogradnje [autorski rad]



Slika 9. Testiranje REST endpoint-a pomoću alata Postman [autorski rad]

6.1. Komponente sustava

Frontend aplikacije razvijen je korištenjem Vue.js 3 uz Composition API i TypeScript, a za brzu i efikasnu izradu korisničkog sučelja korišteni su Vite kao alat za razvoj, TailwindCSS za

stilizaciju, Vue Router za upravljanje rutama te Supabase za autentifikaciju korisnika (prijava i registracija). Za komunikaciju u stvarnom vremenu među članovima tima implementiran je klijentski dio WebSocket protokola korištenjem biblioteke socket.io-client.

Backend dio sustava razvijen je u Express.js okviru, gdje se koristi Prisma ORM za rad s PostgreSQL bazom podataka. Također je implementiran WebSocket server koji omogućuje dvosmjernu komunikaciju između korisnika u stvarnom vremenu.

6.2. Inicijalizacija web aplikacije

U početnoj fazi razvoja aplikacije, web projekt je inicijaliziran korištenjem alata Vite, koji omogućuje brzo i moderno razvojno okruženje usklađeno s najnovijim web standardima. Vite pruža izuzetno jednostavan početak i brze nadogradnje tijekom razvoja, što značajno poboljšava iskustvo rada na frontend aplikacijama temeljenima na Vue.js-u.

Za stilizaciju korisničkog sučelja korišten je TailwindCSS, *utility-first framework* koji omogućuje brzu i fleksibilnu izradu dizajna izravno unutar HTML strukture. Konfiguracija Tailwinda postavljena je u style.css datoteci, gdje su definirane osnovne boje, varijable i podrška za tamni način rada putem klasa.

Glavna ulazna točka aplikacije definirana je u main.ts datoteci, gdje se pokreće Vue.js aplikacija, učitavaju svi potrebni moduli (router, Pinia za upravljanje stanjem, PrimeVue komponentna biblioteka i tema), te se aplikacija montira na HTML element s ID “#app”. Ovaj korak postavlja osnovu za rad cjelokupne aplikacije.

Isječak koda 6.1: Implementacija main.ts datoteke

```
import { createApp } from "vue";
import "./style.css";
import App from "./App.vue";
import PrimeVue from "primevue/config";
import Aura from "@primeuix/themes/aura";
import router from "./router.ts";
import { createPinia } from "pinia";

const app = createApp(App);
const pinia = createPinia();

app
.use(PrimeVue, {
theme: {
preset: Aura,
```

```
},
})
.use(router)
.use(pinia);
app.mount("#app");
```

6.3. Komunikacija aplikacije s backend servisom

Za komunikaciju između Vue.js aplikacije i backend servisa korištene su dvije glavne tehnologije: REST API putem axios biblioteke te web socket putem socket.io-client biblioteke. Ova kombinacija omogućuje klasičnu asinkronu komunikaciju s poslužiteljem za dohvat i ažuriranje podataka, a i dvosmjernu komunikaciju u stvarnom vremenu za funkcionalnosti poput razgovora između članova tima.

6.3.1. REST komunikacija putem ApiService klase

Za dohvat, slanje, ažuriranje i brisanje podataka putem HTTP zahtjeva implementirana je posebna klasa ApiService.ts, koja koristi axios instancu s unaprijed definiranim zaglavljima i autentifikacijskim tokenom iz Supabase sesije. Token se automatski dodaje svakom zahtjevu korištenjem axios interceptora:

Isječak koda 6.2: Automatsko dodavanje Supabase tokena

```
this.api.interceptors.request.use(
async (config) => {
try {
const token = await this.getAuthToken();
if (token) {
config.headers.Authorization = Bearer ${token};
}
} catch (error) {
console.error("Error adding auth token:", error);
}
return config;
},
(error) => Promise.reject(error)
);
```

Osim toga, klasa ApiService pruža metode poput get, post, put, patch i delete koje se koriste u cijeloj aplikaciji za komunikaciju s backendom. Ova apstrakcija omogućuje centralizirano upravljanje komunikacijom s API-jem i obradu pogrešaka:

Isječak koda 6.3: Implementacija GET metode

```
public async get<T = any>(url: string, config?: AxiosRequestConfig):
Promise<T> {
try {
const response: AxiosResponse<T> = await this.api.get(url, config);
return response.data;
} catch (error) {
this.handleApiError(error);
throw error;
}
}
```

6.3.2. Komunikacija u stvarnom vremenu putem Web Socketa

Za potrebe razmjene poruka i komentara između članova tima u stvarnom vremenu, implementirana je klasa SocketService.ts. Prilikom inicijalizacije socket veze dohvaća se Supabase token za autentifikaciju te se uspostavlja veza s backend socket-om:

Isječak koda 6.4: Korištenje Supabase tokena u Web socket zahtjevima

```
this.socket = io(this.socketUrl, {
auth: { token },
transports: ["websocket", "polling"],
});
```

Klasa SocketService omogućuje funkcionalnosti poput:

- Pridruživanja komentarima za određeni zadatak
- Emitiranja novog komentara
- Praćenja događaja poput dolaska i odlaska korisnika, te dolaznih poruka
- Odspajanja i čišćenja veza

Isječak koda 6.5: Implementacija čitanja novog komentara

```
this.socket?.on("new-comment", (comment: Comment) => {
  console.log("New comment received:", comment);
  onNewComment(comment);
});

this.socket?.emit("join-task-comments", taskId);
```

Na taj način korisnicima aplikacije omogućena je sinkronizirana komunikacija, gdje se poruke prikazuju u stvarnom vremenu svim sudionicima istog zadatka bez potrebe za ručnim osvježavanjem stranice.

Ovakav pristup kombinira pouzdanost REST API komunikacije za standardne operacije i brzinu socket komunikacije za *real-time* interakciju, čime se korisnicima pruža moderno korisničko iskustvo.

6.4. Implementacija Vue router-a

Za upravljanje navigacijom unutar aplikacije korišten je Vue Router, službena biblioteka za promjenu ruta u Vue.js aplikacijama. Omogućuje povezivanje komponenti s URL putanjama, čime se upravlja prikazom sadržaja ovisno o trenutnoj ruti u web pregledniku. U ovoj aplikaciji implementiran je pomoću createRouter i createWebHistory, čime se omogućava tzv. “clean URL” pristup bez “#” znaka u putanjama.

Rute su organizirane tako da se javno dostupne stranice poput prijave i registracije definiraju izvan glavnog layouta, dok su sve ostale zaštićene i grupirane unutar komponente MainLayout.vue. Unutar nje se nalaze “djeca” rute kao što su početna stranica, upravljanje timom, ploča sa zadacima (engl. *Board*) i postavke. Svaka ruta dodatno sadrži metapodatke (meta) koji definiraju pravila pristupa, ikonice, naslove i prikaz u bočnoj navigaciji.

Isječak koda 6.6: Definiranje rute za početnu stranicu

```
{
  path: "/",
  name: "Home",
  component: Home,
  meta: {
    requiresAuth: true,
    icon: HomeIcon,
    title: "Home",
    showInSidebar: true,
    sidebarGroup: "Platform",
    order: 1,
  },
}
```

Kako bi se omogućila zaštita ruta koje zahtijevaju autentifikaciju, koristi se beforeEach navigacijski *guard* koji provjerava je li korisnik prijavljen.

Isječak koda 6.7: Implementacija zaštite ruta

```
router.beforeEach(async (to) => {
  if (to.meta.requiresAuth) {
    const user = await getCurrentUser();
    if (!user) {
```

```
      return { name: "Login" };
    }
  }
  return true;
});
```

Također, definirana je ruta s koja će se prikazati ako korisnik pokuša pristupiti stranici (ruti) koja ne postoji.

Isječak koda 6.8: Implementacija NotFound rute

```
{
path: "/:catchAll(.*)",
name: "NotFound",
component: NotFoundPage,
meta: {
requiresAuth: false,
showInSidebar: false,
},
}
```

Prikaz odgovarajuće komponente prema trenutnoj ruti upravlja se putem “<router-view>” direktive u App.vue datoteci, čime se omogućuje dinamički prikaz sadržaja.

Isječak koda 6.9: Dodavanje Vue-router-a u App.vue

```
<template>
<div id="app">
<router-view></router-view>
</div>
</template>
```

6.5. Implementacija autentifikacije korisnika

Za autentifikaciju korisnika u aplikaciji korišten je Supabase Auth, koji omogućava jednostavno upravljanje prijavom, registracijom, odjavom i provjerom trenutnog korisnika. Autentifikacija je implementirana kroz posebnu servisnu datoteku auth.ts, gdje se nalaze funkcije za osnovne operacije. Za prikaz korisničkog sučelja kreirane su Vue.js komponente Register.vue i Login.vue.

Komponente Login.vue i Register.vue koriste vee-validate i yup za validaciju obrazaca. Nakon uspješne prijave ili registracije korisnik se automatski preusmjerava na odgovarajuću stranicu.

Korisnik prilikom registracije unosi svoj e-mail, lozinku i ime (ime nije obavezan podatak). Podaci se šalju funkciji registerUser() koja koristi Supabase signUp metodu.

Isječak koda 6.10: Registracija korisnika pomoću Supabase

```
const { data, error } = await supabase.auth.signUp({
  email,
  password,
  options: {
    data: {
      name,
    },
  },
});
```

Ako je registracija uspješna, korisnik dobiva poruku s uputama za potvrdu e-maila. Ako dođe do pogreške, vraća se odgovarajuća poruka korisniku.

Upravljanje korisničkom sesijom može biti izazovno, ali Supabase nudi metodu za dohvat trenutnog korisnika, pa je za Vue-router zaštitu korištena Supabase metoda za dohvat trenutnog korisnika.

Isječak koda 6.11: Dohvat trenutno prijavljenog korisnika

```
export const getCurrentUser = async () => {
  const { data } = await supabase.auth.getUser();
  return data.user;
};
```

U nastavku će biti prikazana Vue.js komponenta Login.vue, kao i logika koja se koristi unutar komponente.

Isječak koda 6.12: Implementacija programske logike Vue.js komponente

```
<script setup lang="ts">
import { ref } from "vue";
import { useRouter } from "vue-router";
import { loginUser, loginWithGithub } from "../services/auth";
import { Button } from "@/components/ui/button";
import { Input } from "@/components/ui/input";
import { Form, Field } from "vee-validate";
import * as yup from "yup";

const router = useRouter();
const isLoading = ref(false);
const message = ref("");
const isSuccess = ref(false);

const schema = yup.object({
```

```
  email: yup
    .string()
    .required("Email is required")
    .email("Please enter a valid email address"),
  password: yup
    .string()
    .required("Password is required")
    .min(6, "Password must be at least 6 characters long"),
});

const handleLogin = async (values: any) => {
  message.value = "";
  isSuccess.value = false;

  try {
    isLoading.value = true;

    const response = await loginUser({
      email: values.email,
      password: values.password,
    });

    isSuccess.value = response.success;
    message.value = response.message;

    if (response.success) {
      router.push("/");
    }
  } catch (error: any) {
    isSuccess.value = false;
    message.value = error.message || "Login failed. Please try again.";
  } finally {
    isLoading.value = false;
  }
};

const handleGithubLogin = async () => {
  try {
    isLoading.value = true;
    await loginWithGithub();
  } catch (error: any) {
    message.value = "GitHub login failed. Please try again.";
    isSuccess.value = false;
  } finally {
    isLoading.value = false;
  }
};
</script>
```

U gornjem isječku koda iz Login.vue komponente prikazana je logika za obradu korisničke prijave pomoću e-maila i lozinke, kao i putem GitHub autentifikacije. Programski kod napisan je korištenjem Composition API-ja i TypeScripta.

Na početku se uvoze potrebni moduli: ref za reaktivne varijable iz Vue.js, useRouter za preusmjeravanje korisnika nakon prijave, te loginUser i loginWithGithub metode iz servisnog sloja za autentifikaciju. Također se koristi vee-validate biblioteka za validaciju obrazaca i yup za definiranje pravila validacije.

U kodu su definirane tri reaktivne varijable:

- isLoading – prikazuje stanje učitavanja prilikom autentifikacije,
- message – prikazuje korisničke poruke (uspjeh ili greška),
- isSuccess – označava je li autentifikacija bila uspješna.

Funkcija handleLogin poziva se prilikom slanja obrasca za prijavu. Ona ima više funkcionalnosti:

1. Resetira poruke i postavlja stanje na početno
2. Poziva *loginUser* funkciju s unesenim e-mailom i lozinkom
3. Ako je prijava uspješna (*response.success*), korisnik se preusmjerava na početnu stranicu (*router.push*("/"))
4. U slučaju pogreške, prikazuje se odgovarajuća poruka

Funkcija handleGithubLogin omogućuje prijavu putem GitHub računa. Poziva loginWithGithub metodu, a u slučaju pogreške korisniku se prikazuje poruka.

U nastavku teksta slijedi prikaz važnijih “template” dijelova koda iz Login.vue komponente.

Isječak koda 6.13: Prikaz poruke uspjeha ili pogreške

```
<div
  v-if="message"
  class="rounded-md p-4"
  :class="{
    'bg-red-50 text-red-700 border border-red-200': !isSuccess,
    'bg-green-50 text-green-700 border border-green-200': isSuccess,
  }"
>
  {{ message }}
</div>
```

Prethodni element prikazuje poruku korisniku nakon pokušaja prijave. Poruka se prikazuje samo ako varijabla message sadrži tekst (što znači da se dogodio neki ishod prijave). Vizualna boja poruke ovisi o vrijednosti isSuccess – zelena za uspjeh, a crvena za grešku.

Isječak koda 6.14: Implementacija obrasca za prijavu

```
<Form
@submit="handleLogin"
```

```
class="mt-8 space-y-6"
:validation-schema="schema"
>
<div class="space-y-4">
<div>
<label for="email" class="block text-sm font-medium text-gray-700"
>Email</label
>
<Field
name="email"
type="email"
placeholder="Enter your email"
v-slot="{ field, errorMessage, meta }"
>
<Input
v-bind="field"
id="email"
class="mt-1 w-full"
:class="{ 'border-red-500': meta.touched && errorMessage }"
/>
<p
v-if="meta.touched && errorMessage"
class="mt-1 text-sm text-red-600"
>
{{ errorMessage }}
</p>
</Field>
</div>
…
```

Prethodni dio koda prikazuje implementaciju obrasca za unos email adrese unutar Vue komponente pomoću biblioteke vee-validate, koja omogućuje jednostavno upravljanje validacijom korisničkih unosa. Komponenta <Form> služi kao kontejner za cijelu formu i koristi @submit="handleLogin" kako bi se prilikom slanja pokrenula funkcija handleLogin definirana u script dijelu. Također, putem :validation-schema="schema" veže se validacijska shema izrađena pomoću yup biblioteke, koja definira pravila (npr. da email mora biti u ispravnom formatu). Unutar forme koristi se komponenta <Field> koja je povezana s nazivom "email" i koristi v-slot za pristup ključnim informacijama o unosu – konkretno field, errorMessage i meta. Slot field se direktno veže na <Input> komponentu, čime se osigurava sinkronizacija unosa i validacije. Ako korisnik dodirne polje i pogrešno unese email (npr. ostavi ga praznim ili upiše nevažeći format), prikazuje se poruka greške unutar <p> elementa, a obrub inputa postaje crven (border-red-500), što se postiže dinamičkom klasom vezanom na meta.touched && errorMessage. Također, obrazac sadrži i polje za unos lozinke, ali zbog kompleksnosti, prikazano je samo polje za unos email-a. Ovaj dio programskog koda omogućuje elegantno, vizualno jasno i automatski validirano korisničko iskustvo prilikom unosa email adrese u login formu.

6.6. Definiranje Pinia *store*

Kako bi se Pinia store mogao koristiti, potrebno ga je registrirati u main.ts datoteci.

Isječak koda 6.15: Registracija Pinia store u main.ts

```
const pinia = createPinia();
app.use(pinia);
```

Pinia omogućava centralizirano pohranjivanje i pristup podacima koje koriste različite komponente. Store se definira pomoću funkcije defineStore, pri čemu se dodjeljuje jedinstveno ime i definiraju varijable stanja, funkcije za dohvat podataka i metode za ažuriranje stanja.

U nastavku teksta, prikazana je implementacija za store “team” koji upravlja timovima korisnika. Store koristi ref za reaktivno pohranjivanje aktivnog tima, svih timova, indikator učitavanja i moguće pogreške. Pomoću metode fetchTeams() dohvaćaju se timovi s backend API-ja, dok setActiveTeam() omogućuje postavljanje trenutno aktivnog tima i njegovo spremanje u localStorage kako bi se zadržao između osvježavanja stranice. Samim time, aktivni tim je globalno dostupan u aplikaciji te se na svim komponentama aplikacije može dohvatiti trenutno aktivni tim za korisnika (npr. promjenom aktivnog tima, ažurira se “Board” komponenta s pripadajućim zadacima).

Isječak koda 6.16: Definiranje “team” *store*

```
import { defineStore } from "pinia";
import { ref } from "vue";
import type { Team } from "@/models/Team";
import TeamService from "@/services/TeamService";

export const useTeamStore = defineStore("team", () => {
const activeTeam = ref<Team | null>(null);
const teams = ref<Team[]>([]);
const isLoading = ref(false);
const error = ref<string | null>(null);

async function fetchTeams() {
isLoading.value = true;
error.value = null;

try {
const teamsData = await TeamService.getAllTeams();
teams.value = teamsData || [];
if (teamsData && teamsData.length > 0) {
const storedTeamId = localStorage.getItem("activeTeamId");
if (storedTeamId) {
const storedTeam = teamsData.find((team) => team.id === storedTeamId);
if (storedTeam) {
setActiveTeam(storedTeam);
return;
}
}
const personalTeam = teamsData.find((team) => team.isPersonal);
```

```
setActiveTeam(personalTeam || teamsData[0]);
} else {
activeTeam.value = null;
}
} catch (err: any) {
console.error("Error fetching teams:", err);
error.value = err.message;
} finally {
isLoading.value = false;
}
}
function setActiveTeam(team: Team) {
activeTeam.value = team;
localStorage.setItem("activeTeamId", team.id);
}
return {
activeTeam,
teams,
isLoading,
error,
fetchTeams,
setActiveTeam,
};
});
```

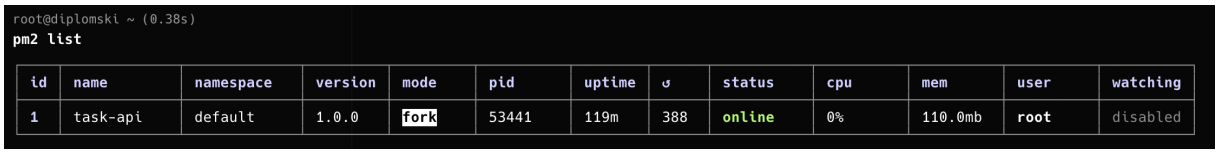
6.7. *Deploy* svih komponenti sustava

Po završetku razvoja, sustav je u potpunosti “deployan” na vlastiti VPS server, a pristup se vrši putem domene “tasky.live”. Sustav je podijeljen na frontend i backend, a za obje komponente je konfiguriran Nginx i omogućena HTTPS komunikacija putem SSL certifikata.

Backend servis razvijen u Express.js, koristi Prisma ORM i izložen je putem REST API-ja. Deploy je izveden tako da je aplikacija smještena na server, a za njeno pokretanje koristi se PM2 — procesni menadžer koji omogućava automatski restart, monitoring i upravljanje Node.js servisima. Nakon svakog ažuriranja koda, pokreće se pm2 restart, čime se backend servis automatski osvježava. Backend servis izložen je preko Nginx *reverse proxy* na adresi https://api.tasky.live, uz podršku za HTTPS preko besplatnog SSL certifikata dobivenog putem “Let’s Encrypt”.

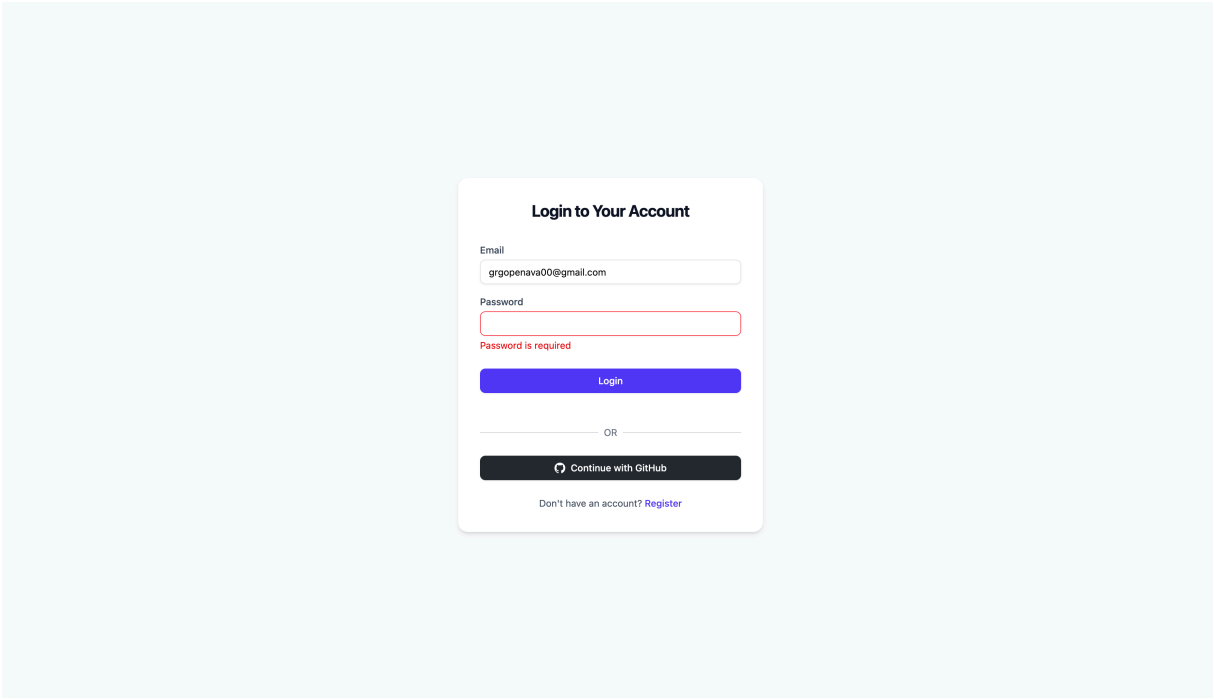
Web aplikacija “builda” se pomoću ugrađenog Vite build, odnosno pokretanjem naredbe “npm run build”. Ukoliko aplikacije nema niti jednu pogrešku, kreira se “dist” direktorij u kojem se nalazi sav sadržaj aplikacije koji je potreban da bi server mogao pokrenuti istu.

Dobiveni sadržaj iz “dist” direktorija se zatim postavlja na VPS i “servira” preko Nginxa kao statička stranica. Nginx konfiguracija preusmjerava sve zahtjeve prema statičkom sadržaju te omogućava HTTPS pristup.

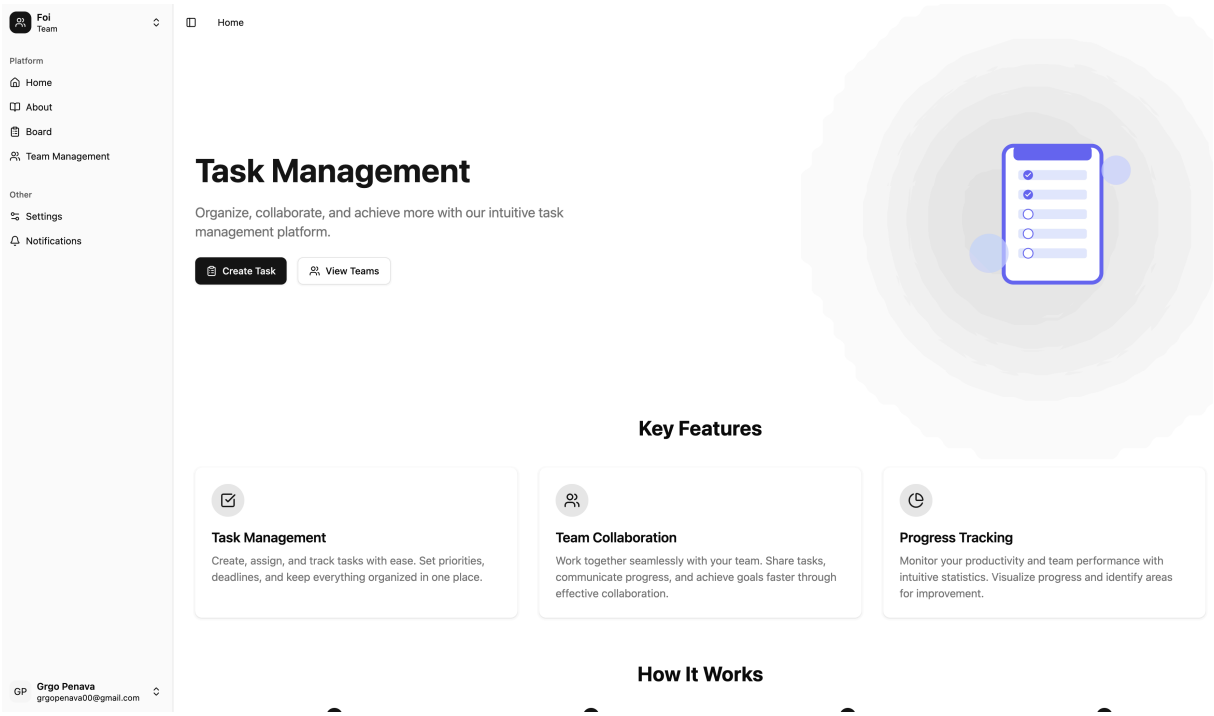


Slika 10. Prikaz aktivnog backend servisa pomoću PM2 [autorski rad]

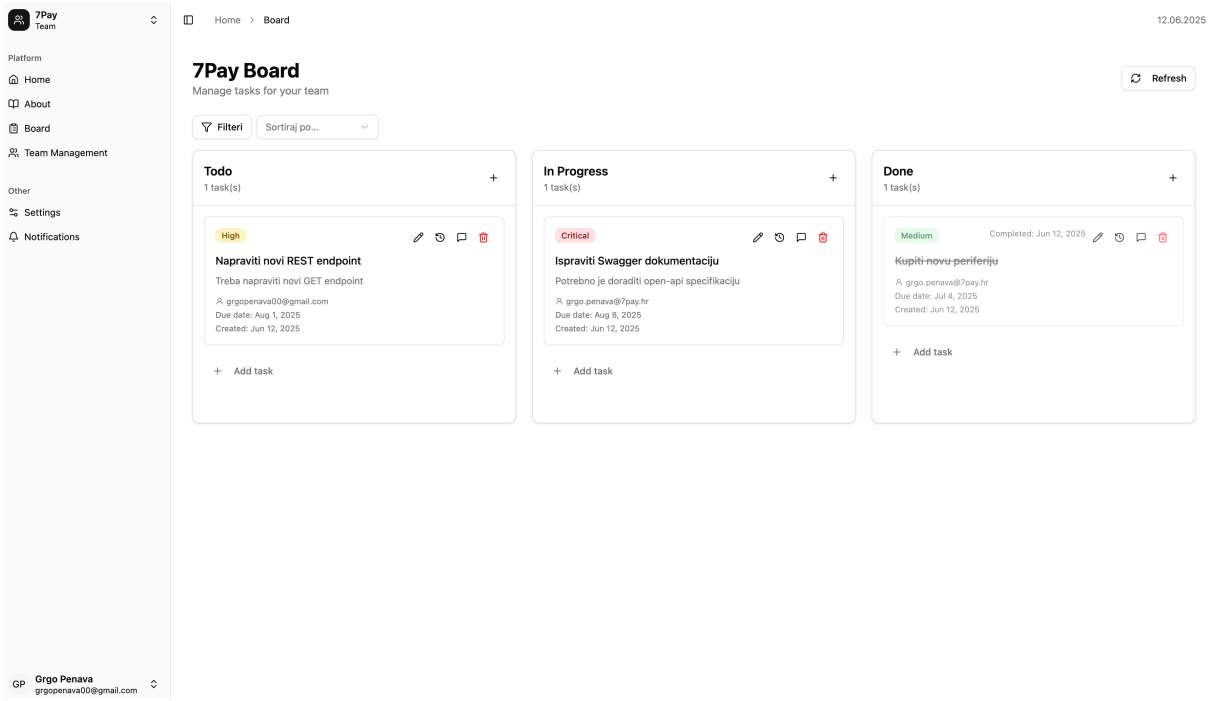
7. Prikaz kreirane Web aplikacije



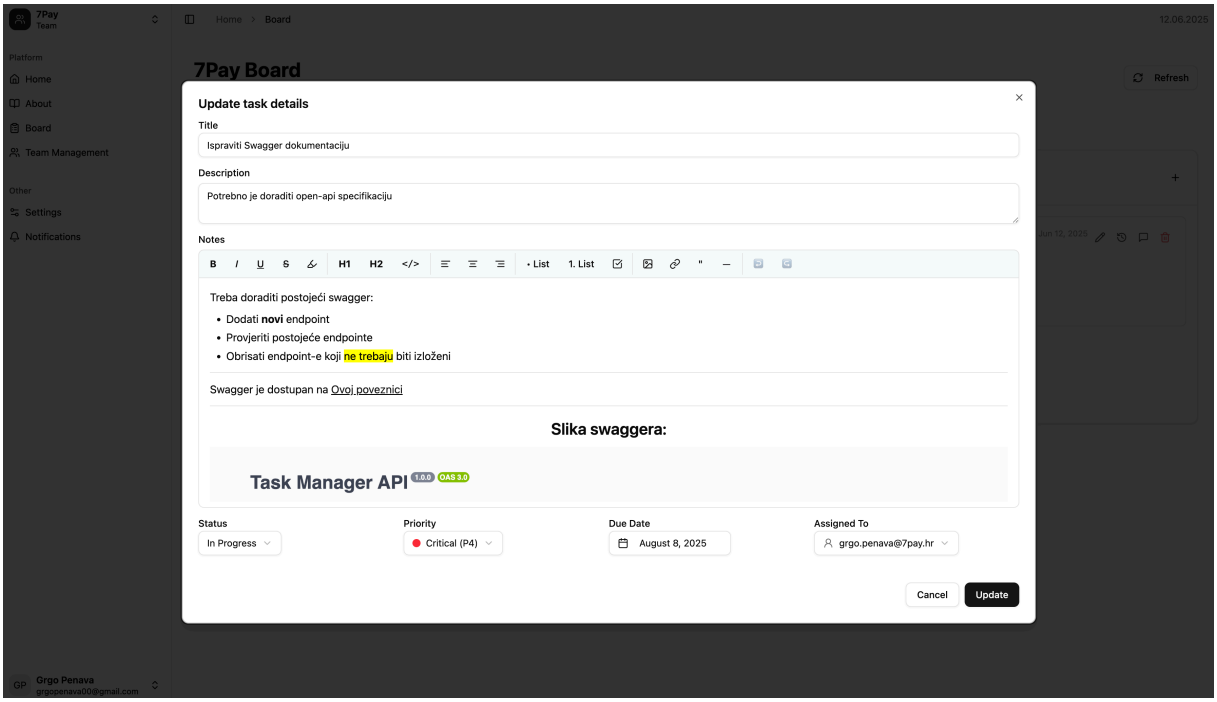
Slika 11. Prikaz ekrana za prijavu [autorski rad]



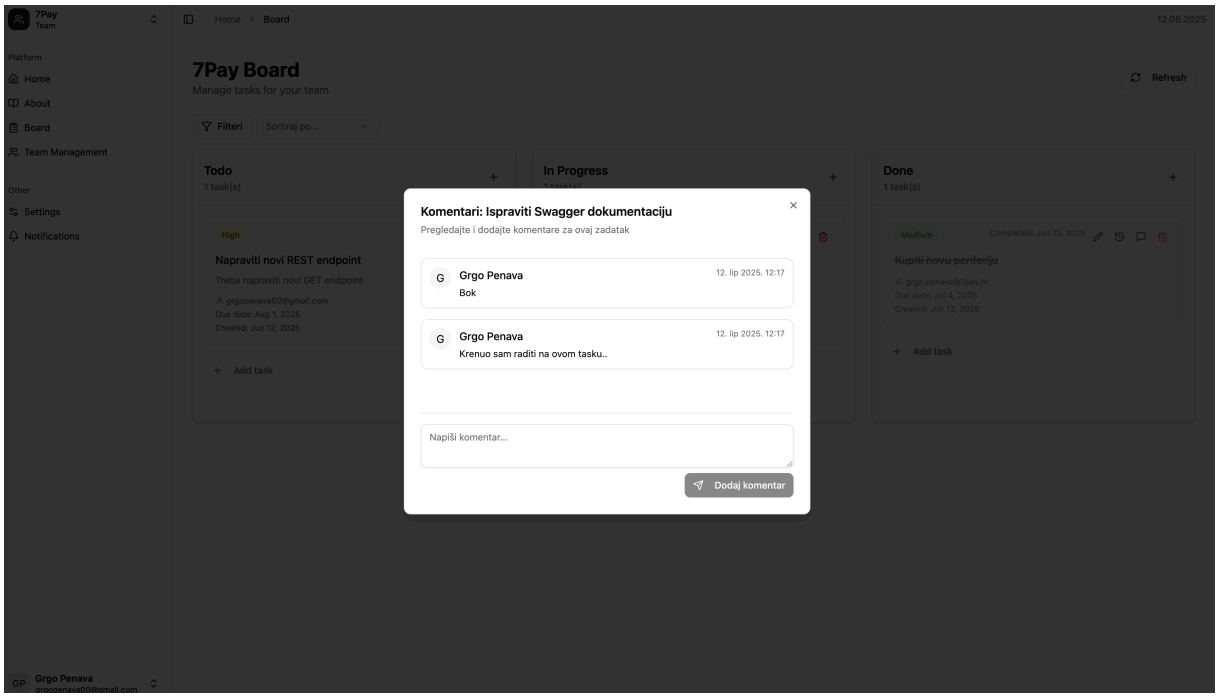
Slika 12. Prikaz početnog zaslona [autorski rad]



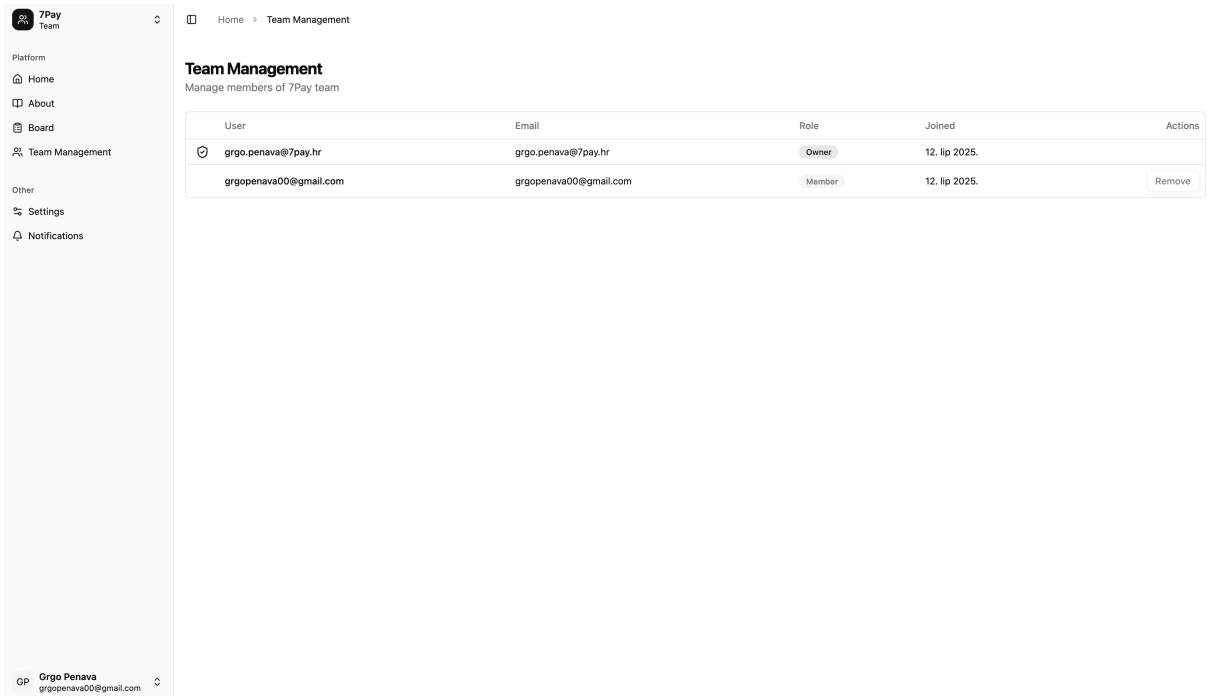
Slika 13. Prikaz ekrana Board [autorski rad]



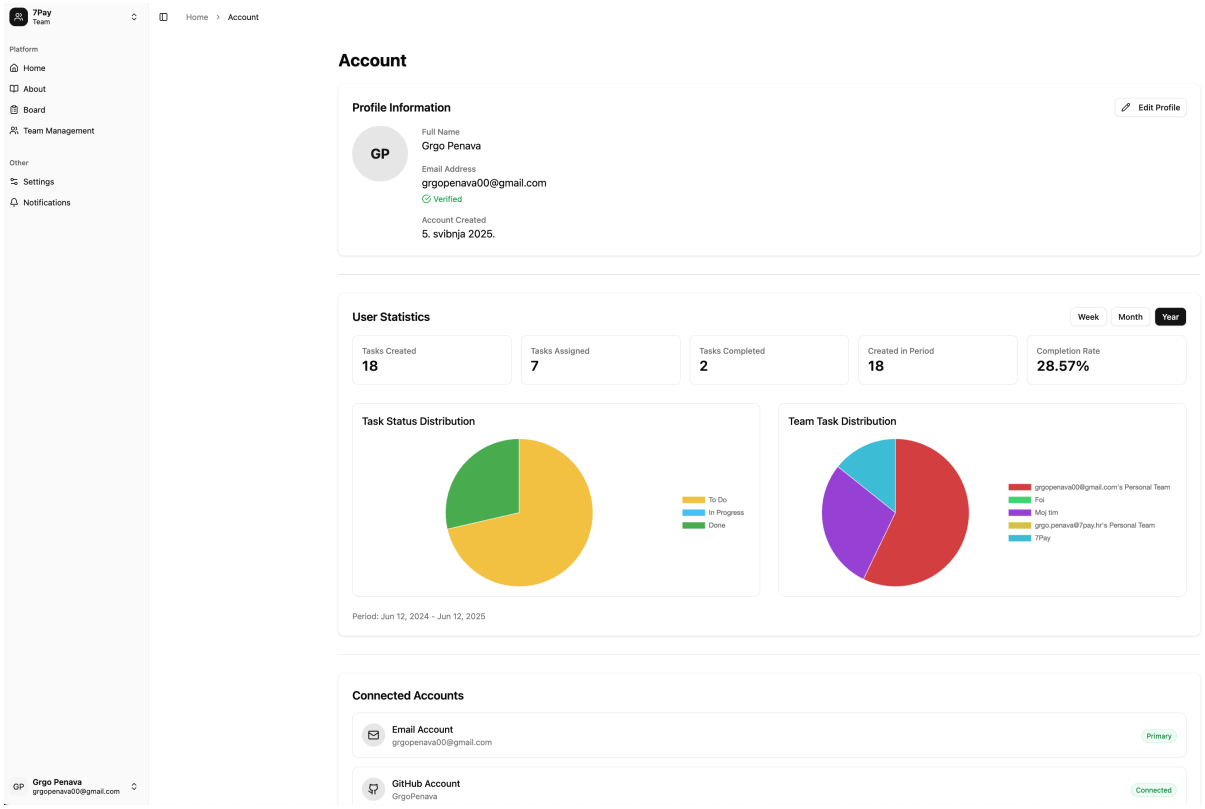
Slika 14. Prikaz forme za kreiranje/ažuriranje zadatka [autorski rad]



Slika 15. Prikaz ekrana za komunikaciju putem Web Socketa [autorski rad]



Slika 16. Prikaz ekrana za upravljanje članovima tima [autorski rad]



Slika 17. Prikaz ekrana s informacijama o prijavljenom korisniku [autorski rad]

8. Zaključak

U okviru ovog diplomskog rada izrađena je web aplikacija za upravljanje zadacima, s posebnim naglaskom na razvoj korisničkog sučelja korištenjem tehnologije Vue.js 3. Cilj rada bio je razvoj funkcionalne i responzivne web aplikacije koja omogućuje jednostavno i učinkovito upravljanje zadacima unutar timova, uz primjenu suvremenih tehnologija i principa razvoja web-aplikacija.

Aplikacija je implementirana kao jednostranična aplikacija, pri čemu su korišteni alati i biblioteke kao što su Pinia za upravljanje stanjem, te Vite za brzu izgradnju i optimizaciju projekta. Posebna pažnja posvećena je organizaciji koda, reaktivnosti korisničkog sučelja te povezivanju s REST API servisom.

Frontend aplikacija je, nakon procesa buildanja, postavljena kao statički sadržaj na vlastiti VPS server, uz osiguranu HTTPS komunikaciju putem Nginx poslužitelja i Let’s Encrypt SSL certifikata. Sustav je dostupan putem vlastite domene “tasky.live”, dok je backend servis izložen na poddomeni “api.tasky.live”.

Rezultat ovog rada je funkcionalna i skalabilna aplikacija spremna za produkcijsku upotrebu, koja omogućuje daljnje nadogradnje i proširenja. Kroz provedbu projekta demonstrirana je primjena suvremenih frontend tehnologija, njihova integracija s backend servisom te implementacija kompletne aplikacije u stvarnom okruženju.

Popis literature

[1] „What's new in Vue 3?“, What's new in Vue 3? Pristupljeno: 02. lipanj 2025. [Na internetu]. Dostupno na: https://www.geeksforgeeks.org/whats-new-in-vue-3/

[2] „Vue.js“, Vue. Pristupljeno: 01. lipanj 2025. [Na internetu]. Dostupno na: https://vuejs.org/guide/introduction.html

[3] „All You Need to Know About VueJS“. flexiple. Pristupljeno: 06. veljača 2025. [Na internetu]. Dostupno na: https://flexiple.com/vue/deep-dive

[4] „Vue.JS online coding platform“. RunCode. Pristupljeno: 01. lipanj 2025. [Na internetu]. Dostupno na: https://runcode.io/vue-online-coding-platform

[5] TNS Staff, „Introduction to Typescript“. TheNewStack, 26. srpanj 2022. Pristupljeno: 02. lipanj 2025. [Na internetu]. Dostupno na: https://thenewstack.io/what-is-typescript/

[6] „Interesting Facts About Typescript“. Geeksforgeeks, 01. ožujak 2025. Pristupljeno: 01. lipanj 2025. [Na internetu]. Dostupno na: https://www.geeksforgeeks.org/interesting-facts-about-typescript/?utm_source=chatgpt.com

[7] „Typescript“. Microsoft. Pristupljeno: 01. lipanj 2025. [Na internetu]. Dostupno na: https://github.com/microsoft/TypeScript

[8] „Typescript“. Wikipedia. Pristupljeno: 01. lipanj 2025. [Na internetu]. Dostupno na: https://en.wikipedia.org/wiki/TypeScript

[9] Dominykas J., „What is VS Code: an overview of the popular code editor and its features“. Hostinger, 23. travanj 2025. Pristupljeno: 02. lipanj 2025. [Na internetu]. Dostupno na: https://www.hostinger.com/tutorials/what-is-vs-code

[10] „Visual Studio Code“. Wikipedia. Pristupljeno: 02. lipanj 2025. [Na internetu]. Dostupno na: https://en.wikipedia.org/wiki/Visual_Studio_Code

[11] Jimmy Wong, „A Brief History of Vue.js: From a Side Project to a Global Framework“. Jimmy Wong. Pristupljeno: 02. lipanj 2025. [Na internetu]. Dostupno na: https://jimmywong.co.uk/2025/03/brief-history-of-vuejs.html

[12] „Standalone App“. Devtools.vuejs. Pristupljeno: 02. lipanj 2025. [Na internetu]. Dostupno na: https://devtools.vuejs.org/guide/standalone

[13] „TypeScript Introduction“. W3schools. Pristupljeno: 06. veljača 2025. [Na internetu]. Dostupno na: https://www.w3schools.com/typescript/typescript_intro.php

[14] ASHUTOSH PAWAR, „TypeScript for Beginners: A Gentle Introduction“. dev.to, 12. studeni 2023. Pristupljeno: 06. ožujak 2025. [Na internetu]. Dostupno na: https://dev.to/gurrudev/typescript-for-beginners-a-gentle-introduction-33ca

[15] „Introduction to TypeScript“. nodejs. Pristupljeno: 06. ožujak 2025. [Na internetu]. Dostupno na: https://nodejs.org/en/learn/typescript/introduction

[16] „TypeScript Book“. gibbok. Pristupljeno: 06. ožujak 2025. [Na internetu]. Dostupno na: https://gibbok.github.io/typescript-book/book/primitive-types/

[17] „Variable Declaration“. TypeScript. Pristupljeno: 06. ožujak 2025. [Na internetu]. Dostupno na: https://www.typescriptlang.org/docs/handbook/variable-declarations.html

[18] Fabien Schlegel, „Handling complex types: Union, Intersection and Typeguards in TypeScript“. Devoreur2code, 29. studeni 2025. Pristupljeno: 06. ožujak 2025. [Na internetu]. Dostupno na: https://www.devoreur2code.com/blog/typescript-union-intersection-typeguards

[19] Ikeh Akinyemi, „Understanding discriminated union and intersection types in TypeScript“. LogRocket, 11. travanj 2025. Pristupljeno: 06. ožujak 2025. [Na internetu]. Dostupno na: https://blog.logrocket.com/understanding-discriminated-union-intersection-types-typescript/

[20] „Everyday Types“. TypeScript. Pristupljeno: 06. ožujak 2025. [Na internetu]. Dostupno na: https://www.typescriptlang.org/docs/handbook/2/everyday-types.html#union-types
[21] Shammi Anand, „TypeScript Operators“. scaler, 23. travanj 2024. Pristupljeno: 04. lipanj 2025. [Na internetu]. Dostupno na: https://www.scaler.com/topics/typescript/typescript-operator/
[22] „TypeScript Ternary Operator“. Programiz.
[23] Kenny DuMez, „TypeScript typeof operator“. Graphite. Pristupljeno: 06. lipanj 2025. [Na internetu]. Dostupno na: https://graphite.dev/guides/typescript-typeof-operator
[24] „Typescript - If Statement“. tutorialspoint. Pristupljeno: 06. lipanj 2025. [Na internetu]. Dostupno na: https://www.tutorialspoint.com/typescript/typescript_if_statement.htm
[25] „TypeScript if, else & switch Conditional Control Tutorial“. Koderhq. Pristupljeno: 06. lipanj 2025. [Na internetu]. Dostupno na: https://www.koderhq.com/tutorial/typescript/conditional-control/
[26] „TypeScript - switch“. tutorialsteacher. Pristupljeno: 06. lipanj 2025. [Na internetu]. Dostupno na: https://www.tutorialsteacher.com/typescript/typescript-switch
[27] By Sahil, „For Loop in TypeScript“. scaler, 04. svibanj 2023. Pristupljeno: 06. lipanj 2025. [Na internetu]. Dostupno na: https://www.scaler.com/topics/typescript/for-loop-in-typescript/
[28] „TypeScript - For Loop“. Pristupljeno: 06. lipanj 2025. [Na internetu]. Dostupno na: https://www.tutorialspoint.com/typescript/typescript_for_loop.htm
[29] „TypeScript for“. typescripttutorial. Pristupljeno: 06. lipanj 2025. [Na internetu]. Dostupno na: https://www.typescripttutorial.net/typescript-tutorial/typescript-for/
[30] „TypeScript while“. typescripttutorial. Pristupljeno: 06. lipanj 2025. [Na internetu]. Dostupno na: https://www.typescripttutorial.net/typescript-tutorial/typescript-while/
[31] „TypeScript - dowhile loop“. tutorialspoint. Pristupljeno: 06. lipanj 2025. [Na internetu]. Dostupno na: https://www.tutorialspoint.com/typescript/typescript_do_while_loop.htm
[32] Bob Junior, „Typescript: For..in vs For..of“. 04. travanj 2023. Pristupljeno: 06. lipanj 2025. [Na internetu]. Dostupno na: https://medium.com/@bobjunior542/typescript-for-in-vs-for-of-44800351510e
[33] Sumita Kevat, „8 Advanced features of Typescript: A deeper dive into the language“. Canopas, 23. siječanj 2023. Pristupljeno: 06. travanj 2025. [Na internetu]. Dostupno na: https://canopas.com/8-advanced-features-of-typescript-a-deeper-dive-into-the-language-cb630b1079e2
[34] Kevin Kreuzer, „Advanced TypeScript“. AngularExperts, 10. studeni 2022. Pristupljeno: 06. travanj 2025. [Na internetu]. Dostupno na: https://angularexperts.io/blog/advanced-typescript
[35] Kenny DuMez, „TypeScript utility types“. Graphite. Pristupljeno: 06. ožujak 2025. [Na internetu]. Dostupno na: https://graphite.dev/guides/typescript-utility-types
[36] Jonathan Cardoso, „How To Use Generics in TypeScript“. DigitalOcean, 16. studeni 2021. Pristupljeno: 06. ožujak 2025. [Na internetu]. Dostupno na: https://www.digitalocean.com/community/tutorials/how-to-use-generics-in-typescript
[37] „TypeScript Basic Generics“. W3schools. Pristupljeno: 06. ožujak 2025. [Na internetu]. Dostupno na: https://www.w3schools.com/typescript/typescript_basic_generics.php
[38] „TypeScript Generics“. 11. rujan 2023. Pristupljeno: 03. lipanj 2025. [Na internetu]. Dostupno na: https://pieces.app/blog/understanding-typescript-generics
[39] „Decorators“. TypeScript. Pristupljeno: 03. lipanj 2025. [Na internetu]. Dostupno na: https://www.typescriptlang.org/docs/handbook/decorators.html
[40] „TypeScript Decorators in Brief“. refine, 09. siječanj 2025. Pristupljeno: 03. lipanj 2025. [Na internetu]. Dostupno na: https://refine.dev/blog/typescript-decorators/#introduction

[41] „How to Use Enums in TypeScript for Cleaner Code“. geshan, 02. siječanj 2024. Pristupljeno: 06. travanj 2025. [Na internetu]. Dostupno na: https://geshan.com.np/blog/2024/01/typescript-enum/
[42] Chris Janes, „The differences between static analysis and linting“. imperfectdev, 03. listopad 2023. Pristupljeno: 06. travanj 2025. [Na internetu]. Dostupno na: https://www.imperfectdev.com/static-analysis-vs-linting/
[43] Matías Hernández Arellano, „How to Use Static Code Analysis Tools to Improve Your TypeScript Codebase“. dev.to, 12. lipanj 2023. Pristupljeno: 06. travanj 2025. [Na internetu]. Dostupno na: https://dev.to/documatic/how-to-use-static-code-analysis-tools-to-improve-your-typescript-codebase-b6g
[44] Sara Verdi, „Existing code review tools for TypeScript“. graphite.dev. Pristupljeno: 04. lipanj 2025. [Na internetu]. Dostupno na: https://graphite.dev/guides/existing-code-review-tools-for-typescript
[45] „Typescript Strict“. Typescriptlang. Pristupljeno: 04. lipanj 2025. [Na internetu]. Dostupno na: https://www.typescriptlang.org/tsconfig/#strict
[46] Andy Li, „‚Strict Mode‘ TypeScript config options“. dev.to, 25. siječanj 2021. Pristupljeno: 04. lipanj 2025. [Na internetu]. Dostupno na: https://dev.to/jsdev/strict-mode-typescript-j8p
[47] „Controlling type checking strictness“. learntypescript. Pristupljeno: 04. lipanj 2025. [Na internetu]. Dostupno na: https://learntypescript.dev/11/l6-strictness
[48] „Everything you want to know about Vue.js“. tildeloop. Pristupljeno: 06. lipanj 2025. [Na internetu]. Dostupno na: https://tildeloop.com/blog/everything-you-want-to-know-about-vue-js/
[49] Aly Ninh, „Exploring Vue.js for Cross-Platform Development“. dev.to, 04. srpanj 2024. Pristupljeno: 06. lipanj 2025. [Na internetu]. Dostupno na: https://dev.to/ngocninh123/exploring-vuejs-for-cross-platform-development-2070
[50] Maryia Shapel, „10 Reasons Why Vue.js Is Best for App Development [+ Benefits]“. sam-solutions. Pristupljeno: 06. lipanj 2025. [Na internetu]. Dostupno na: https://sam-solutions.com/blog/why-vue-js/
[51] „React vs Angular vs Vue“. belitsoft, 09. lipanj 2023. Pristupljeno: 06. lipanj 2025. [Na internetu]. Dostupno na: https://belitsoft.com/front-end-development-services/react-vs-angular
[52] Emre Deniz, „Vue 3: What’s New“. medium, 25. studeni 2024. Pristupljeno: 06. lipanj 2025. [Na internetu]. Dostupno na: https://medium.com/@emre.deniz/vue-3-whats-new-0d2a97e14125
[53] Paridhi Wadhwani i Nikita Sakhuja, „What’s New in Vue 3? – A Comprehensive Overview on Exciting New Features in Vue 3“. bacancytechnology, 03. lipanj 2024. Pristupljeno: 06. lipanj 2025. [Na internetu]. Dostupno na: https://www.bacancytechnology.com/blog/whats-new-in-vue-3
[54] Doshi, „Advanced Concepts: Deep Dive in Reactivity“. medium, 27. ožujak 2023. Pristupljeno: 06. lipanj 2025. [Na internetu]. Dostupno na: https://blog.stackademic.com/advanced-concepts-deep-dive-in-reactivity-7ad22192e58
[55] Domenico Tenace, „Reactivity: ref() vs reactive()“. medium, 29. kolovoz 2023. Pristupljeno: 06. lipanj 2025. [Na internetu]. Dostupno na: https://javascript.plainenglish.io/basic-concepts-of-vue-js-ref-vs-reactive-7f20b5b08d32
[56] „Reactivity Fundamentals“. Vuejs. Pristupljeno: 06. lipanj 2025. [Na internetu]. Dostupno na: https://vuejs.org/guide/essentials/reactivity-fundamentals.html
[57] Mehul Mohan, „Efficient DOM Manipulation with Vue.js Virtual DOM“. codedamn, 25. ožujak 2023. Pristupljeno: 07. lipanj 2025. [Na internetu]. Dostupno na: https://codedamn.com/news/vuejs/efficient-dom-manipulation-vuejs-virtual-dom

[58] Jakub Andrzejewski, „Understanding Vue's Virtual DOM". dev.to, 24. veljača 2025. Pristupljeno: 07. lipanj 2025. [Na internetu]. Dostupno na: https://dev.to/jacobandrewsky/understanding-vues-virtual-dom-108p
[59] „Vue Lifecycle Hooks". W3schools. Pristupljeno: 07. lipanj 2025. [Na internetu]. Dostupno na: https://www.w3schools.com/vue/vue_lifecycle-hooks.php
[60] „Components Basics". vuejs. Pristupljeno: 07. lipanj 2025. [Na internetu]. Dostupno na: https://vuejs.org/guide/essentials/component-basics
[61] „VueJS Component". Geeksforgeeks. Pristupljeno: 07. lipanj 2025. [Na internetu]. Dostupno na: https://www.geeksforgeeks.org/javascript/vuejs-component/
[62] „Vue Templates". W3schools. Pristupljeno: 07. lipanj 2025. [Na internetu]. Dostupno na: https://www.w3schools.com/vue/vue_templates.php
[63] „Template Syntax". vuejs. Pristupljeno: 07. lipanj 2025. [Na internetu]. Dostupno na: https://vuejs.org/guide/essentials/template-syntax
[64] Sanchithasr, „How to Communicate between Components in Vue.js". dev.to, 24. travanj 2021. Pristupljeno: 06. srpanj 2025. [Na internetu]. Dostupno na: https://dev.to/sanchithasr/how-to-communicate-between-components-in-vue-js-kjc
[65] „Vue Router". router.vuejs. Pristupljeno: 08. lipanj 2025. [Na internetu]. Dostupno na: https://router.vuejs.org/
[66] „State Management". vuejs. Pristupljeno: 08. lipanj 2025. [Na internetu]. Dostupno na: https://vuejs.org/guide/scaling-up/state-management
[67] Dharmendra Kumar, „State Management with Pinia and Vue.js (Composition API) Lifecycle Hooks". dev.to, 12. rujan 2024. Pristupljeno: 08. lipanj 2025. [Na internetu]. Dostupno na: https://dev.to/dharamgfx/state-management-with-pinia-and-vuejs-composition-api-lifecycle-hooks-50bh
[68] Mohammad Rashedul Hasan, „Vuex vs Pinia: A Comparison of Vue State Management Libraries". linkedin, 16. rujan 2023. Pristupljeno: 08. lipanj 2025. [Na internetu]. Dostupno na: https://www.linkedin.com/pulse/vuex-vs-pinia-comparison-vue-state-management-libraries-hasan/
[69] „Routing". vuejs. Pristupljeno: 08. lipanj 2025. [Na internetu]. Dostupno na: https://vuejs.org/guide/scaling-up/routing
[70] chintanonweb, „Vue.js and Vue Router Unveiled: Mastering Seamless Routing for Vue Apps". dev.to, 19. rujan 2023. Pristupljeno: 08. lipanj 2025. [Na internetu]. Dostupno na: https://dev.to/chintanonweb/vuejs-and-vue-router-unveiled-mastering-seamless-routing-for-vue-apps-2bf8

Popis slika

Popis tablica