Web aplikacije (WA)

Nositelj: doc. dr. sc. Nikola Tanković
Asistent: Luka Blašković, mag. inf.

Ustanova: Sveučilište Jurja Dobrile u Puli, Fakultet informatike u Puli

(3) Komunikacija s klijentskom stranom

Do sad smo vidjeli kako izraditi Express poslužitelj i osnovne rute koje odgovaraju na HTTP zahtjeve. Dosad smo HTTP zahtjeve slali pomoću web preglednika i HTTP klijenata. Naučili smo dijelove HTTP zahtjeva i odgovora, kako slati i očekivati JSON podatke kroz odgovor, parametre zahtjeva i statusne kodove. Vidjeli smo i kako strukturirati aplikaciju kroz Express Router objekt. U ovoj skripti napokon ćemo spojiti naš Express poslužitelj s klijentskom stranom. Preciznije, realizirat ćemo komunikaciju s Vue razvojnim poslužiteljem kroz Axios HTTP klijent.

 **Posljednje ažurirano: 11.11.2024.**

Sadržaj

1. Postavljanje `Express` poslužitelja

Krenimo s definiranjem osnovnog Express poslužitelja. Izradit ćemo jednostavni web shop jedne **male trgovine koja prodaje odjeću i obuću**.

Do sad smo već naučili kako izraditi node projekt i definirati Express poslužitelj uz relevantne rute na primjeru aplikacije za naručivanje pizze. U ovom primjeru ćemo izraditi potpuno novi poslužitelj. Kako ćemo imati i klijentsku stranu, definirat ćemo zasebne direktorije za klijentsku (`vue`) i serversku stranu (`server`). Sve možemo definirati unutar direktorija `webshop`.

```
webshop
├── server
└── vue
```

Unutar `server` direktorija ćemo izraditi novi node projekt, instalirati Express i definirati osnovni poslužitelj.

```
import express from 'express';

const app = express();

app.use(express.json());

const PORT = 3000;

app.get('/', (req, res) => {
  res.send('Webshop API');
});

app.listen(PORT, error => {
  if (error) {
    console.error(`Greška prilikom pokretanja poslužitelja: ${error.message}`);
  } else {
    console.log(`Server dela na http://localhost:${PORT}`);
  }
});
```

1.1 Definiranje osnovnih ruta

Želimo da korisnik može pregledati sve proizvode u trgovini, pogledati detalje o pojedinom proizvodu te napraviti narudžbu.

Prema tome, prvo ćemo osmisliti backend dizajn naše aplikacije:

- GET `/proizvodi` - dohvaća sve proizvode
- GET `/proizvodi/:id` - dohvaća proizvod s određenim ID-om
- POST `/narudzba` - stvara novu narudžbu

So far, so good. Sada ćemo definirati boilerplate za rute za svaku od ovih akcija:

```
app.get('/proizvodi', (req, res) => {
  res.send('Dohvati sve proizvode');
});

app.get('/proizvodi/:id', (req, res) => {
  const id_req = req.params.id;
  res.send(`Dohvati proizvod s ID: ${id_req}`);
});

app.post('/narudzba', (req, res) => {
  res.send('Napravi novu narudžbu');
});
```

Proizvode ćemo spremati *in-memory*, tj. u polju objekata. Možemo definirati konstruktor `Proizvod` koji će nam pomoći pri stvaranju novih proizvoda.

Definirat ćemo i polje `proizvodi` te u njega pohraniti nekoliko instanciranih `Proizvod` objekata.

```
function Proizvod(id, naziv, cijena, velicine) {
  this.id = id;
  this.naziv = naziv;
  this.cijena = cijena;
  this.velicine = velicine;
}

const proizvodi = [
  new Proizvod(1, 'Obična crna majica', 100, ['XS', 'S', 'M', 'L']),
  new Proizvod(2, "Levi's 501 traperice", 110, ['S', 'M', 'L']),
  new Proizvod(3, 'Zimska kapa', 40, 'onesize'),
  new Proizvod(4, 'Čarape Adidas', 20, ['34-36', '37-39', '40-42']),
  new Proizvod(5, 'Tenisice Nike', 200, ['38', '39', '40', '41', '42', '43', '44', '45'])
];
```

Sada možemo definirati logiku za dohvaćanje **svih proizvoda** i **proizvoda s određenim ID-om**:

```
app.get('/proizvodi', (req, res) => {
  res.status(200).json(proizvodi);
});

app.get('/proizvodi/:id', (req, res) => {
  const id_req = req.params.id;

  if (isNaN(id_req)) {
    res.status(400).send('ID mora biti broj');
    return;
  }

  const proizvod = proizvodi.find(proizvod => proizvod.id == id_req);

  if (proizvod) {
```

```
    res.status(200).json(proizvod);
  } else {
    res.status(404).send('Proizvod nije pronađen');
  }
});
```

Testirajte poslužitelj koristeći HTTP klijent ili web preglednik.

1.2 Router za proizvode

Nakon što ste definirali i testirali obje rute, definirat ćemo **Express Router** u datoteci `proizvodi.js`. Struktura projekta će izgledati ovako:

```
webshop
├── server
│   ├── index.js
│   ├── node_modules
│   ├── package-lock.json
│   ├── package.json
│   └── routes
│       └── proizvodi.js
└── vue
```

Definirajmo Express Router u datoteci `proizvodi.js` i prebacimo logiku iz `index.js` datoteke kako smo naučili na prethodnim vježbama.

```
// routes/proizvodi.js

import express from 'express';

const router = express.Router();

function Proizvod(id, naziv, cijena, velicine) {
  this.id = id;
  this.naziv = naziv;
  this.cijena = cijena;
  this.velicine = velicine;
}

const proizvodi = [
  new Proizvod(1, 'Obična crna majica', 100, ['XS', 'S', 'M', 'L']),
  new Proizvod(2, "Levi's 501 traperice", 110, ['S', 'M', 'L']),
  new Proizvod(3, 'Zimska kapa', 40, 'onesize'),
  new Proizvod(4, 'Čarape Adidas', 20, ['34-36', '37-39', '40-42']),
  new Proizvod(5, 'Tenisice Nike', 200, ['38', '39', '40', '41', '42', '43', '44', '45'])
];

router.get('/', (req, res) => {
  res.status(200).json(proizvodi);
});

router.get('/:id', (req, res) => {
  const id_req = req.params.id;

  if (isNaN(id_req)) {
    res.status(400).send('ID mora biti broj');
    return;
  }
```

```
  const proizvod = proizvodi.find(proizvod => proizvod.id == id_req);
  if (proizvod) {
    res.status(200).json(proizvod);
  } else {
    res.status(404).send('Proizvod nije pronađen');
  }
});

export default router;
```

Kako smo uklonili prefikse `/proizvodi` i `/proizvodi/:id`, moramo prefiks definirati prilikom uključivanja router objekta u `index.js` datoteku.

```
// index.js

import proizvodiRouter from './routes/proizvodi.js';

app.use('/proizvodi', proizvodiRouter);
```

Testirajte poslužitelj koristeći HTTP klijent ili web preglednik.

1.3 Router za narudžbe

Definirat ćemo još jedan router za narudžbe, za sada samo s jednom rutom koja stvara novu narudžbu.

```
// routes/narudzbe.js

import express from 'express';

const router = express.Router();

router.post('/', (req, res) => {
  res.send('Napravi novu narudžbu');
});

export default router;
```

Na jednak način kao i za proizvode, uključite i router za narudžbe u `index.js` datoteku.

Možemo definirati konstruktor `Narudzba` koji će nam pomoći pri stvaranju novih narudžbi.

```
// routes/narudzbe.js

function Narudzba(id, naruceni_proizvodi) {
  this.id = id;
  this.naruceni_proizvodi = naruceni_proizvodi;
}
```

Atribut `naruceni_proizvodi` će biti polje objekata koji sadrže: **ID proizvoda**, **veličinu** i **naručenu količinu**.

Primjer:

```
// routes/narudzbe.js

const narudzba = new Narudzba(1, [
  { id: 1, velicina: 'M', narucena_kolicina: 2 },
  { id: 3, velicina: 'onesize', narucena_kolicina: 1 }
]);
```

Možemo još dodati u konstruktor metodu `ukupnaCijena` koja će izračunati ukupnu cijenu narudžbe.

> Kako smo narudžbu definirali kao polje objekata gdje svaki objekt sadrži ID proizvoda, veličinu i naručenu količinu, moramo za svaki naručeni proizvod pronaći odgovarajući proizvod iz polja `proizvodi`, pomnožiti cijenu proizvoda s naručenom količinom i zbrojiti sve proizvode. Iskoristit ćemo metodu `Array.reduce` za zbrajanje za bržu implementaciju.

```
// routes/narudzbe.js

function Narudzba(id, naruceni_proizvodi) {
  this.id = id;
```

```
  this.naruceni_proizvodi = naruceni_proizvodi;
  this.ukupnaCijena = function () {
    let ukupno = this.naruceni_proizvodi.reduce((suma, trenutni_proizvod) => {
      let proizvod_obj = proizvodi.find(proizvod => proizvod.id == trenutni_proizvod.id);
      return suma + proizvod_obj.cijena * trenutni_proizvod.narucena_kolicina;
    }, 0); // 0 je početna vrijednost sume
    return ukupno;
  };
}

const narudzba = new Narudzba(1, [
  { id: 1, velicina: 'M', narucena_kolicina: 2 },
  { id: 3, velicina: 'onesize', narucena_kolicina: 1 }
]);

// Testiramo metodu ukupnaCijena za narudžbu koja sadrži 2 majice i 1 kapu
console.log(narudzba.ukupnaCijena()); // 240
```

Obzirom da sad koristimo `ES6` sintaksu (definirali smo u prošloj skripti kroz `package.json`), nije loše ostati dosljedan pa možemo funkcijske konstruktore pretvoriti u klase. Koristimo ključnu riječ `class` i metodu `constructor` za inicijalizaciju objekta, dok metodu `ukupnaCijena` možemo definirati kao `get` metodu klase jer nema potrebe da se poziva kao zasebna funkcija ili da se nalazi unutar konstruktora.

```
// routes/narudzbe.js
class Proizvod {
  constructor(id, naziv, cijena, velicine) {
    this.id = id;
    this.naziv = naziv;
    this.cijena = cijena;
    this.velicine = velicine;
  }
}

const proizvodi = [
  new Proizvod(1, 'Obična crna majica', 100, ['XS', 'S', 'M', 'L']),
  new Proizvod(2, "Levi's 501 traperice", 110, ['S', 'M', 'L']),
  new Proizvod(3, 'Zimska kapa', 40, 'onesize'),
  new Proizvod(4, 'Čarape Adidas', 20, ['34-36', '37-39', '40-42']),
  new Proizvod(5, 'Tenisice Nike', 200, ['38', '39', '40', '41', '42', '43', '44', '45'])
];

class Narudzba {
  constructor(id, naruceni_proizvodi) {
    this.id = id;
    this.naruceni_proizvodi = naruceni_proizvodi;
  }

  get ukupnaCijena() {
    let ukupno = this.naruceni_proizvodi.reduce((suma, currProizvod) => {
      let pronadeni_proizvod = proizvodi.find(p => p.id == currProizvod.id);
      console.log(pronadeni_proizvod);
```

```
      return suma + pronadeni_proizvod.cijena * currProizvod.narucena_kolicina;
    }, 0);
    return ukupno;
  }
}

// dummy narudžba
const narudzba = new Narudzba(1, [
  { id: 1, velicina: 'M', narucena_kolicina: 2 },
  { id: 3, velicina: 'onesize', narucena_kolicina: 1 }
]);

console.log(narudzba.ukupnaCijena()); // 240
```

No, sada se javlja problem ponavljanja koda u router objektima. Kako bismo to riješili, možemo izdvojiti konstruktor i polje za spremanje proizvoda i narudžbi u zasebnu datoteku `data.js`.

```
// server/data.js

class Proizvod {
  constructor(id, naziv, cijena, velicine) {
    this.id = id;
    this.naziv = naziv;
    this.cijena = cijena;
    this.velicine = velicine;
  }
}

const proizvodi = [
  new Proizvod(1, 'Obična crna majica', 100, ['XS', 'S', 'M', 'L']),
  new Proizvod(2, "Levi's 501 traperice", 110, ['S', 'M', 'L']),
  new Proizvod(3, 'Zimska kapa', 40, 'onesize'),
  new Proizvod(4, 'Čarape Adidas', 20, ['34-36', '37-39', '40-42']),
  new Proizvod(5, 'Tenisice Nike', 200, ['38', '39', '40', '41', '42', '43', '44', '45'])
];
```

Sada moramo još samo izvesti ove podatke iz `data.js` datoteke.

```
// server/data.js

export { Proizvod, proizvodi };
```

Za kraj ćemo još samo dodati POST rutu za izradu i pohranu nove narudžbe u polje narudžbi `narudzbe`.

```
// narudzbe.js

let narudzbe = [];

router.post('/', (req, res) => {
  let podaci = req.body;
```

```
  let naruceni_proizvodi = podaci.naruceni_proizvodi;

  if (!Array.isArray(naruceni_proizvodi) || naruceni_proizvodi.length == 0) {
    return res.status(400).json({ message: 'Nema podataka' });
  }

  let latest_id = narudzbe.length ? narudzbe.at(-1).id + 1 : 1; // one-liner za
generiranje ID-a

  let narudzba_obj = new Narudzba(latest_id, naruceni_proizvodi);

  narudzbe.push(narudzba_obj);

  return res.status(201).json(podaci); // vraćamo poslane podatke o narudžbi i statusni
kod 201 (Created) budući da smo stvorili novi resurs
});
```

To je to za sada! Prebacujemo se na izradu klijentske strane (aka. frontend).

2. Izrada jednostavne `Vue3` aplikacije s `Tailwind CSS`

2.1 Konfiguracija projekta

Definirali smo osnovni Express poslužitelj s rutama za proizvode i narudžbe. Sada ćemo izraditi jednostavnu **Vue.js** aplikaciju koja će komunicirati s poslužiteljem.

Tailwind CSS popularni je CSS razvojni okvir koji omogućuje brzo i jednostavno stiliziranje web stranica upotrebom predefiniranih CSS klasa. Njegova glavna prednost je što ne koristi gotove komponente već se stilovi definiraju direktno u HTML-u, a samim time omogućuje **veću fleksibilnost** i **prilagodbu**.

Koristit ćemo Tailwind 3 za potrebe naše aplikacije. **Napomena**, Tailwind CSS zahtjeva Node.js verziju 12.13.0 ili noviju.

Vue aplikaciju možemo instalirati i zasebno pa dodati Tailwind CSS, ali brže i jednostavnije je kroz `Vite` i postojeći `Vue Tailwind` template.

> Upute za instalaciju nalaze se na sljedećoj poveznici: https://tailwindcss.com/docs/guides/vite#vue

- **1. korak**: otvorite poveznicu iznad i provjerite da ste odabrali `Using Vue`. Zatim kopirajte naredbu za inicijalizaciju projekta s predloškom:

```
npm create vite@latest my-project -- --template vue
```

- **2. korak**: unutar direktorija projekta, pokrenite `npm install` kako biste instalirali sve potrebne pakete:

```
cd my-project

npm install
```

Nakon toga možete (ali i ne morate) prebaciti sadržaj gdje se nalazi vue projekt u `webshop/vue` direktorij, obzirom da smo sada dobili novi direktorij `my-project`.

- **3. korak**: instalirajte `Tailwind CSS`, `PostCSS` i `Autoprefixer` pakete koji su potrebni za ispravan rad Tailwinda:

```
npm install -D tailwindcss postcss autoprefixer

npx tailwindcss init -p
```

Uočite da smo dobili dvije nove datoteke u korijenskom direktoriju projekta: `tailwind.config.js` i `postcss.config.js`.

- **4. korak**: konfigurirajte `tailwind.config.js` datoteku prema uputama na poveznici:

Zamjenjujemo sadržaj datoteke s:

```
// tailwind.config.js

/** @type {import('tailwindcss').Config} */
export default {
  content: ['./index.html', './src/**/*.{vue,js,ts,jsx,tsx}'],
  theme: {
    extend: {}
  },
  plugins: []
};
```

- **5. korak**: unutar `./src/style.css` postavljamo sljedeće 3 direktive (brišemo sve ostalo):

```
@tailwind base;
@tailwind components;
@tailwind utilities;
```

Uspješno te instalirali Vue aplikaciju s Tailwind CSS-om. Sada ćemo malo srediti strukturu projekta, a zatim pokrenuti projekt i testirati funkcionira li sve.

2.2 Struktura projekta i testiranje

Nakon što ste sve instalirali, trebali biste imati sljedeću strukturu `vue` projekta:

```
vue
  ├── node_modules
  ├── public
  ├── src
  │   ├── assets
  │   ├── components
  │   ├── App.vue
  │   ├── main.js
  │   └── style.css
  ├── index.html
  ├── package-lock.json
  ├── package.json
  ├── postcss.config.js
  ├── README.md
  ├── tailwind.config.js
  └── vite.config.js
```

Kako biste se uvjerili da je Tailwind instaliran, otvorite `./src/App.vue` datoteku i dodajte sljedeći `template` kod:

```
<template>
  <h1 class="text-3xl font-bold underline">Hello world!</h1>
</template>
```

Pokrenite aplikaciju koristeći naredbu:

```
npm run dev
```

Trebali biste vidjeti poruku `Hello world!` s velikim podebljanim fontom i podcrtanim tekstom budući da je tako definirano Tailwind CSS klasama `text-3xl`, `font-bold` i `underline`.

Ako vidite, znači da je sve uspješno instalirano i konfigurirano. Možete još za sigurnost pokušati izmijeniti neku klasu, npr. `text-3xl` u `text-5xl` i vidjeti hoće li se promjena odraziti na stranici.

Općenito, klase za Tailwind CSS ne želite mijenjati direktno u CSS kodu, već koristiti i **kombinirati predefinirane klase** koje dolaze s Tailwindom. Nema ih puno smisla niti učiti napamet iako su u pravilu intuitivno imenovane. **Preporuka je naučiti služiti se** TailwindCSS dokumentacijom.

2.3 Komponenta za prikaz proizvoda

Izradit ćemo komponentu `Proizvod.vue` koja će prikazivati **određeni proizvod s njegovim detaljima**. U `./src/components` direktoriju izradite novu datoteku `Proizvod.vue`.

Kako frontend dizajn korisničkog sučelja nije predmet ovog predmeta, upotrijebit ćemo gotovi template stiliziran pomoću Tailwinda te raditi na funkcionalnostima komponente. Ako hoćete, možete uređivati stilove prema vlastitim željama i/ili izraditi vlastiti dizajn.

Template možete kopirati iz `/snippets/ProductTemplate.html` koji se nalazi u ovom repozitoriju.

Nakon što ste kopirali template i izradili komponentu, morate omotati html kod u `template` tag.

```
<!-- Proizvod.vue -->
<template>
  <!-- HTML kod ovdje -->
</template>
```

Zatim ćemo u `App.vue` uključiti ovu komponentu i iscrtati ju na stranici.

```
<template>
  <div>
    <!-- Uključujemo komponentu -->
    <ProductView />
  </div>
</template>
<script setup>
import ProductView from './components/ProductView.vue';
</script>
```

Sada možete pokrenuti aplikaciju i vidjeti kako izgleda komponenta `Proizvod.vue`.

Naslov proizvoda

Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur. Excepteur sint occaecat cupidatat non proident, sunt in culpa qui officia deserunt mollit anim id est laborum.

Karakteristike

- Sed ut perspiciatis unde omnis iste natus error sit voluptatem accusantium
- doloremque laudantium, totam rem aperiam, eaque ipsa quae ab illo inventore veritatis et quasi architecto
- beatae vitae dicta sunt explicabo
- Nemo enim ipsam voluptatem quia voluptas sit aspernatur aut odit aut fugit

Detalji

Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore

100€

Boje

Veličina

	–	–	–
–	–	–	–

Dodaj u košaricu

3. `Axios` HTTP klijent

Za komunikaciju s Express poslužiteljem imamo na raspolaganju više opcija. Moguće je koristiti i `fetch` API koji smo upoznali na Skriptnim jezicima i Programskom inženjerstvu, međutim kroz neke vanjske biblioteke možemo definirati konciznu i čitljivu sintaksu za slanje HTTP zahtjeva te rukovanje odgovorima.

Jedna od takvih biblioteka je i `Axios` koji ćemo koristiti na ovom kolegiju. **Axios** je HTTP klijent za Node i web preglednika koji se bazira na sintaksi `Promise` objekata.

Axios pojednostavljuje slanje HTTP zahtjeva na vanjske API servise, omogućava detaljnu konfiguraciju zahtjeva, upravljanje odgovorima i greškama itd.

Instalirajte Axios paket koristeći npm:

```
npm install axios
```

Jednom kad ste ga instalirali, možete ga koristiti u svim `vue` i `js` datotekama.

3.1 Slanje GET zahtjeva

Krenimo sa slanjem jednostavnog GET zahtjeva za dohvaćanje svih proizvoda iz naše trgovine. Možete otvoriti `ProductView.vue` datoteku i dodati sljedeći kod unutar `script` taga:

```
// ProductView.vue

<script setup>
import axios from 'axios';

axios
  .get('http://localhost:3000/proizvodi')
  .then(response => {
    console.log(response.data);
  })
  .catch(error => {
    console.error('Greška u dohvatu podataka:', error);
  });
</script>
```

Vidimo da Axios koristi `Promise` objekte za rukovanje odgovorima i greškama. Prisjetimo se ukratko kako `Promise` objekti funkcioniraju. `Promise` predstavlja eventualni rezultat asinkronog procesa i može biti u jednom od tri stanja: `pending`, `fulfilled` ili `rejected`.

- `pending`: inicijalno stanje, očekuje se rezultat
- `fulfilled`: proces je završen uspješno
- `rejected`: proces je završen s greškom

Kada se proces završi, `Promise` objekt prelazi u jedno od dva završna stanja: `fulfilled` ili `rejected`. Ukoliko je proces završen uspješno, `then` metoda se poziva s rezultatom, dok se u slučaju greške poziva `catch` metoda.

Sintaksa:

```
const myPromise = new Promise((resolve, reject) => {
  // asinkroni proces
  if (uspjeh) {
    resolve('Uspješno');
  } else {
    reject('Greška');
  }
});
```

Kada pozovemo `Promise`, rukujemo njime pomoću `then` i `catch` metoda:

```
myPromise
  .then(result => {
    // ako je rezultat uspješan (resolve)
    console.log(result); // ispisuje se "Uspješno"
  })
  .catch(error => {
    // ako je došlo do greške (reject)
    console.error(error); // ispisuje se "Greška"
  });
```

Rekli smo da možemo koristiti i `async/await` sintaksu za rukovanje `Promise` objektima. **Asinkrone funkcije** su funkcije koje vraćaju `Promise` objekt. Ukoliko se u funkciji koristi `await` ključna riječ, **funkcija se zaustavlja** dok se ne razriješi `Promise` objekt.

```
let result = await myPromise;

console.log(result); // ispisuje se "Uspješno"
```

Međutim sad ako želimo rukovati greškama, moramo koristiti `try/catch` blok.

```
try {
  let result = await myPromise;
  console.log(result); // ispisuje se "Uspješno"
} catch (error) {
  console.error(error); // ispisuje se "Greška"
}
```

Vratimo se na Axios. U gornjem primjeru, Axios šalje GET zahtjev na adresu `http://localhost:3000/proizvodi` i **očekuje odgovor**.

- Ukoliko je odgovor uspješan, ispisuje se odgovor u konzoli.
- Ukoliko dođe do greške, ispisuje se poruka o grešci.

> Pokrenite Express aplikaciju i pokušajte poslati ovaj GET zahtjev na način da ćete samo osvježiti stranicu. Nakon toga otvorite web konzolu i provjerite ispis podataka.

3.1.1 CORS policy

Na našu žalost, ispisa nema već se u konzoli pojavljuje greška.

Prvo što možemo uočiti jest da se zahtjev definitivno poslao, budući da vidimo nekakav ispis u konzoli. Također možemo vidjeti URI na kojem je poslan zahtjev: `http://localhost:3000/proizvodi`.

Greška kaže: `Access to XMLHttpRequest at 'http://localhost:3000/proizvodi' from origin 'http://localhost:5173' has been blocked by CORS policy: No 'Access-Control-Allow-Origin' header is present on the requested resource.`

U prijevodu, slanje zahtjeva i pristup resursu su blokirani jer poslužitelj ne dopušta pristup resursu iz vanjskog izvora (naša vue aplikacija). Ovo je sigurnosna mjera koja se zove **CORS policy** (Cross-Origin Resource Sharing).

Statusni kod `200` znači da je zahtjev uspješno poslan, ali vidimo da je odgovor blokiran i grešku pod statusnim kodom: `ERR_NETWORK`.

Ono što moramo napraviti je omogućiti pristup resursima iz vanjskog izvora na Express poslužitelju.

U Express poslužitelju, instalirajte sljedeću biblioteku:

```
npm install cors
```

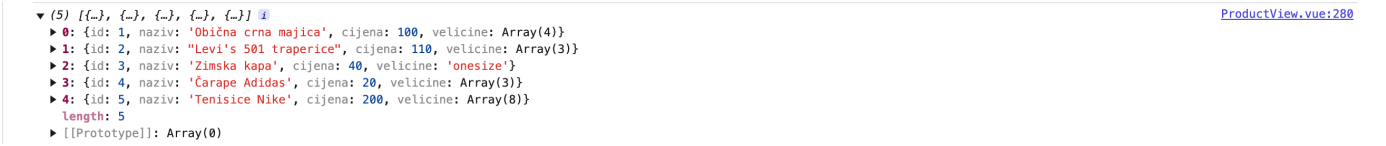
Zatim uključite `cors` u `index.js` datoteci:

```
// index.js

import express from 'express';
import cors from 'cors';

const app = express();
app.use(cors());
```

Linija `app.use(cors())` omogućuje pristup svim resursima iz vanjskih izvora (nikako pustiti ovo kad se radi o produkcijskom okruženju). Sada bi trebali moći poslati zahtjev i dobiti odgovor.

> **Napomena**: CORS policy je sigurnosna mjera koja se koristi kako bi se spriječilo izvršavanje zlonamjernog koda na strani klijenta. U produkcijskom okruženju, uvijek je preporučljivo postaviti odgovarajuće postavke za CORS policy.

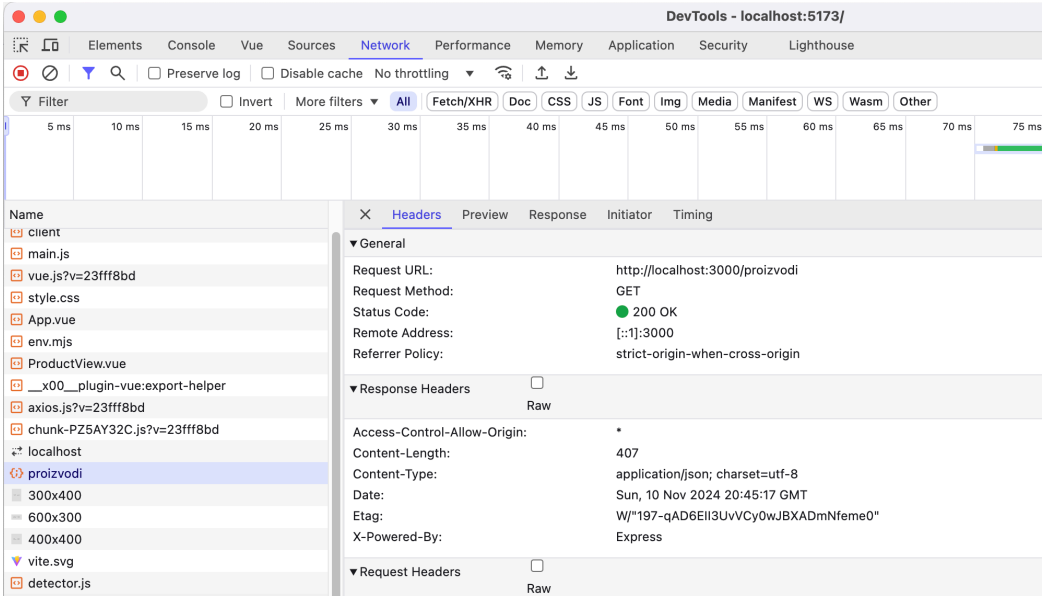
Sada možemo vidjeti da je odgovor uspješno primljen i da se podaci ispisuju u konzoli.



Možete otvoriti i `Network` tab u Developer alatima vašeg web preglednika i vidjeti detalje svih zahtjeva i odgovora. Ovdje ćete pronaći puno korisnih informacija o svakom zahtjevu, uključujući i zaglavlja, tijelo zahtjeva i odgovora, statusni kod itd.

Međutim vidjet ćete i puno zahtjeva koji se pošalju/primaju prilikom učitavanja stranice. Ovo je zato što se u pozadini izvršava razvojni Vue.js poslužitelj koji nam servira mnoge datoteke i resurse kako bi se aplikacija ispravno prikazala. Koga više zanima ova tema, možete pročitati kako funkcioniraju SSA (Single-page Application) aplikacije, poput Vue.js-a.

Osim toga, možete pronaći i zahtjev koji smo poslali na naš Express poslužitelj i vidjeti neke detalje o njemu:



Primjerice, uočite generalne informacije o zahtjevu:

- **Request URL**: `http://localhost:3000/proizvodi`
- **Request Method**: `GET`
- **Status Code**: `200 OK`

Ako otvorite `Preview` ili `Response` možete vidjeti tijelo odgovora koje je poslano s poslužitelja, odnosno podatke o proizvodima.

> Testirajte slanje GET zahtjeva za dohvaćanje pojedinog proizvoda.

3.1.2 Prikazivanje proizvoda na frontendu

Ideja je da upotrijebimo podatke koje smo dobili iz Express poslužitelja i prikažemo ih korisniku. Prvo moramo naravno podatke spremiti u neku varijablu, a zatim ih prikazati na stranici.

U `ProductView.vue` datoteci, definirajmo varijablu `proizvodi` koja će sadržavati podatke o proizvodima.

```
// ProductView.vue

<script setup>
import axios from 'axios';

let proizvodi = [];

axios
  .get('http://localhost:3000/proizvodi/1')
  .then(response => {
    proizvodi = response.data;
  })
  .catch(error => {
    console.error('Greška u dohvatu podataka:', error);
  });
</script>
```

Međutim ovaj pristup nije dobar budući da Vue komponente nemaju ugrađenu reaktivnost za promjene varijabli poput `proizvodi`. Potrebno je definirati reaktivnu varijablu koristeći `ref` ili `reactive` funkcije.

Reactivity API u Vue 3 omogućuje nam da pratimo promjene varijabli na razne načine. Do sada je vjerojatno većina vas učila Vue2 koji koristi `data` objekt za definiranje reaktivnih varijabli. U Vue 3, `data` objekt je zamijenjen s `setup` funkcijom koja vraća objekt s reaktivnim varijablama. Više informacija o tome na sljedećoj poveznici.

`ref` funkciju koristimo za definiranje **reaktivne reference** na primitivne tipove podataka** (string, number, boolean) ili **objekte**.

- vrijednost vraćamo pomoću `value` svojstva ako ga čitamo unutar skriptnog dijela komponente, u template dijelu možemo izostaviti `value` svojstvo.
- ako se promijeni vrijednost reaktivne reference, Vue će automatski osvježiti komponentu.

`reactive` funkciju koristimo za definiranje **reaktivnih objekata** koji sadrže više svojstava.

- iako možemo koristiti i `ref` za objekte, `reactive` je bolji izbor jer omogućuje reaktivnost za dublja svojstva objekta (npr. kada definiramo config aplikacije, korisničke postavke itd.)
- promjene unutar objekta će se automatski pratiti i osvježiti komponentu, a za pristupanje vrijednostima ne koristimo `value` svojstvo.

Kako ne bi previše zakomplicirali stvari, možemo koristiti `ref` funkciju za definiranje reaktivne varijable `proizvod`. Oprez, ako koristite `reactive` funkciju morate paziti da ne pregazite cijeli objekt dohvaćenim podacima, već samo pojedinačne atribute (npr. greška bi bila: `proizvod = response.data`).

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

let proizvod = ref(null);

axios
  .get('http://localhost:3000/proizvodi/1')
  .then(response => {
    console.log(response.data);
    proizvod.value = response.data;
  })
  .catch(error => {
    console.error('Greška u dohvatu podataka:', error);
  });

console.log('proizvod', proizvod); // prazan Reference objekt budući da se asinkroni
zahtjev još nije izvršio
</script>
```

Problem je što se `console.log` izvršava prije nego što se asinkroni zahtjev izvrši. Kako bismo riješili ovaj problem, možemo koristiti `await` ključnu riječ unutar `setup` funkcije, međutim kako želimo da se asinkroni zahtjev izvrši samo jednom, kada korisnik učita stranicu, koristit ćemo `onMounted` hook.

`onMounted` je Lifecycle Hook u Vue.js kojim se može definirati **callback funkcija koja će se izvršiti kada se komponenta učita** (montira).

Kako se radi o asinkronoj callback funkciji, moramo koristiti `async` ključnu riječ ispred definicije funkcije.

```
<script setup>
import { ref, onMounted } from 'vue';
import axios from 'axios';

let proizvod = ref(null);

// asinkroni callback (hook)
onMounted(async () => {
  try {
    const response = await axios.get('http://localhost:3000/proizvodi/1');
    proizvod.value = response.data; // postavljanje podataka u reaktivnu varijablu
  } catch (error) {
    console.error('Greška u dohvatu podataka: ', error);
  }
});
</script>
```

Jednom kad imamo podatke o proizvodu, možemo ih prikazati na stranici. U `template` tagu možemo prikazati podatke pomoću `{{ }}` interpolacije odnosno `v-model` direktive ako se radi o atributima HTML elemenata.

```
<template>
  ...
  <a href="#" aria-current="page" class="font-medium text-gray-500 hover:text-gray-600">{{
proizvod.naziv }}</a>
  ...
  <h1 class="text-2xl font-bold tracking-tight text-gray-900 sm:text-3xl">{{
proizvod.naziv }}</h1>
</template>
```

Vidimo da stvari rade, ali dobivamo grešku u konzoli `Cannot read properties of null`. Ovo je zato što se komponenta renderira prije nego što se podaci o proizvodu dohvate s poslužitelja, a početna vrijednost reaktivne varijable `proizvod` je `null`.

Kako bismo to riješili, možemo koristiti `v-if` direktivu koja će prikazati element samo ako je on istinit. U ovom slučaju, prikazat ćemo elemente samo ako postoje dohvaćeni podaci o proizvodu.

```
<template>
  <div v-if="proizvod">
    <a href="#" aria-current="page" class="font-medium text-gray-500 hover:text-gray-600">
{{ proizvod.naziv }}</a>
    <h1 class="text-2xl font-bold tracking-tight text-gray-900 sm:text-3xl">{{
proizvod.naziv }}</h1>
  </div>
</template>
```

- ili možemo definirati početne vrijednosti za reaktivnu varijablu `proizvod` koristeći `ref` funkciju.

```
<script setup>
let proizvod = ref({
  id: 0,
  naziv: '',
  cijena: 0,
  velicine: []
});
});
</script>
```

Nadopunit ćemo i preostale podatke o proizvodu: `cijena` i `velicine`.

```
<template>
  ...
  <div class="mt-4 lg:row-span-3 lg:mt-0">
    <h2 class="sr-only">Product information</h2>
    <p class="text-3xl tracking-tight text-gray-900">{{ proizvod.cijena }}€</p>
  ...
</template>
```

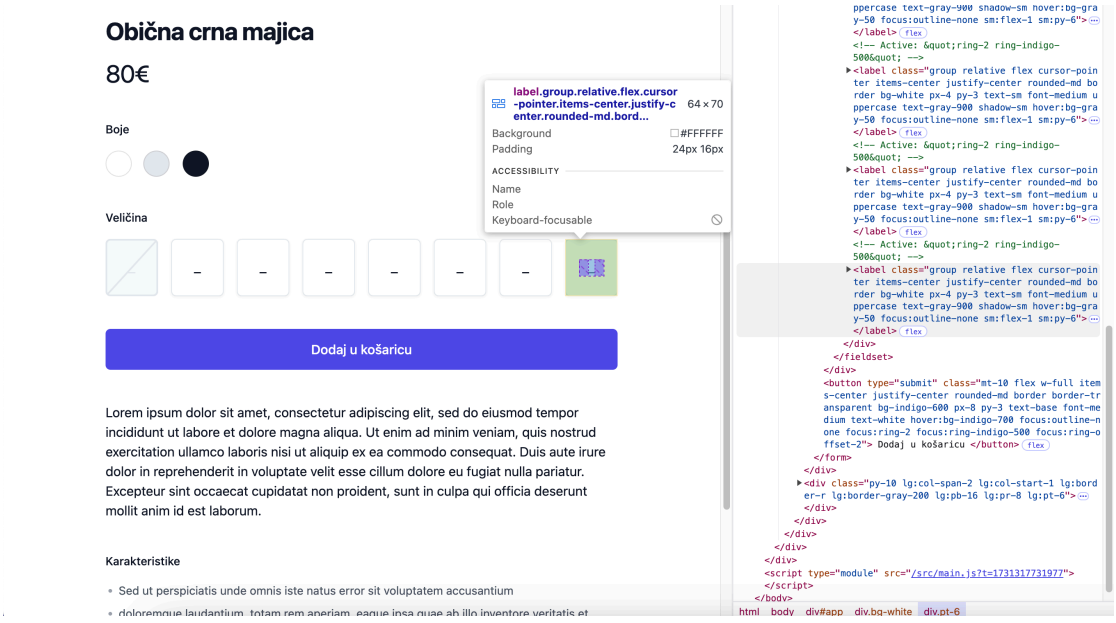
Za veličine nemamo jednostavne inline tekst elemente, već ćemo koristiti `v-for` direktivu za ponavljanje HTML elemenata koji prikazuju pravokutnik s dostupnim veličinama.

Koristeći Find (`CTRL/CMD + f`) pronađite HTML sekciju s veličinama:

```
<div class="mt-10">
  <div class="flex items-center justify-between">
    <h3 class="text-sm font-medium ■text-gray-900">Veličina</h3>
  </div>

  <fieldset aria-label="Choose a size" class="mt-4">
    <div class="grid grid-cols-4 gap-4 sm:grid-cols-8 lg:grid-cols-4">
      <!-- Active: "ring-2 ring-indigo-500" -->
      <label
        class="group relative flex cursor-not-allowed items-center justify-center rounded-md border □bg-gray-50 px-4 py-3 t
      >
        <input type="radio" name="size-choice" value="_" disabled class="sr-only" />
        <span>_</span>
        <span aria-hidden="true" class="pointer-events-none absolute -inset-px rounded-md border-2 □border-gray-200">
          <svg
            class="absolute inset-0 h-full w-full stroke-2 □text-gray-200"
            viewBox="0 0 100 100"
            preserveAspectRatio="none"
            stroke="currentColor"
          >
            <line x1="0" y1="100" x2="100" y2="0" vector-effect="non-scaling-stroke" />
          </svg>
        </span>
      </label>
      <!-- Active: "ring-2 ring-indigo-500" -->
      <label
        class="group relative flex cursor-pointer items-center justify-center rounded-md border □bg-white px-4 py-3 text-sm
      >
        <input type="radio" name="size-choice" value="_" class="sr-only" />
        <span>_</span>
        <!--
        Active: "border", Not Active: "border-2"
        Checked: "border-indigo-500", Not Checked: "border-transparent"
        -->
        <span class="pointer-events-none absolute -inset-px rounded-md" aria-hidden="true"></span>
      </label>
      <!-- Active: "ring-2 ring-indigo-500" -->
      <label
        class="group relative flex cursor-pointer items-center justify-center rounded-md border □bg-white px-4 py-3 text-sm
      >
        <input type="radio" name="size-choice" value="S" class="sr-only" />
        <span>_</span>
        <!--
        Active: "border", Not Active: "border-2"
        Checked: "border-indigo-500", Not Checked: "border-transparent"
        -->
        <span class="pointer-events-none absolute -inset-px rounded-md" aria-hidden="true"></span>
      </label>
      <!-- Active: "ring-2 ring-indigo-500" -->
```

Ako analizirate HTML strukturu, uočit ćete da je svaka veličina definirana unutar `label` oznake, a sama oznaka je unutar `span` elementa. Trenutno su za sve prikazane `_` oznake, a za prvu imamo i dodatni `svg` element koji prikazuje prekriženu veličinu (nije dostupna).

Ako vam nije potpuno jasno, vrlo je korisno kroz Developer alate preglednika analizirati HTML strukturu i pronaći odgovarajuće elemente.



Na ovaj način možete jednostavno uočiti strukturu koja definira neki podatak.

Budući da se radi o dinamičkim podacima, odnosno veličine se mogu mijenjati ovisno o proizvodu, a imamo i ponavljajući HTML kod koji definira jednu veličinu, moramo koristiti `v-for direktivu` za iscrtavanje ponavljajućih HTML elemenata.

Sintaksa `v-for`:

```
<div v-for="element in list" :key="element" >{{element}}</div
```

- ili ako koristimo element unutar HTML oznake:

```
<div v-for="element in list" :key="element" :value = "element"></div
```

> OPREZ: razlikuje se od sintakse JavaScript `for` petlje koja koristi `of` ključnu riječ. Ovo više nalikuje na `for` petlju u Pythonu.

U našem slučaju, izbrisat ćemo sve osim jedne ponavljajuće veličine (preskačemo i prvu jer je ta prekrižena) i pišemo `v-for` za iscrtavanje za svaku veličinu u objektu koji definira proizvod. Dodatno, obzirom da `label` sadrži dosta CSS klasa, možda je bolje da sve omotamo jednostavnim `div` elementom i tu definiramo `v-for` direktivu.

```
<div v-for="velicina in proizvod.velicine" :key="velicina">
  <!-- v-for direktiva za iscrtavanje veličina -->
  <label
    class="group relative flex cursor-pointer items-center justify-center rounded-md
border bg-white px-4 py-3 text-sm font-medium uppercase text-gray-900 shadow-sm hover:bg-
gray-50 focus:outline-none sm:flex-1 sm:py-6"
  >
    <input type="radio" name="size-choice" value="_" class="sr-only" />
    <span>{{ velicina }}</span>
    <!--
    Active: "border", Not Active: "border-2"
    Checked: "border-indigo-500", Not Checked: "border-transparent"
    -->
    <span class="pointer-events-none absolute -inset-px rounded-md" aria-hidden="true">
</span>
  </label>
  <!-- Active: "ring-2 ring-indigo-500" -->
</div>
```

3.1.3 Dodavanje podataka na backendu

Sada kada smo prikazali sve podatke, dodat ćemo još nekoliko podataka za naš proizvod na Express poslužitelju kako bi upotpunili prikaz u našoj aplikaciji.

Dodat ćemo `slike` i `opis` proizvoda.

Proširit ćemo definiciju konstruktora koji definira proizvod u `data.js` datoteci.

```
// server/data.js

class Proizvod {
  constructor(id, naziv, cijena, velicine) {
    this.id = id;
    this.naziv = naziv;
    this.cijena = cijena;
    this.velicine = velicine;
  }
}
```

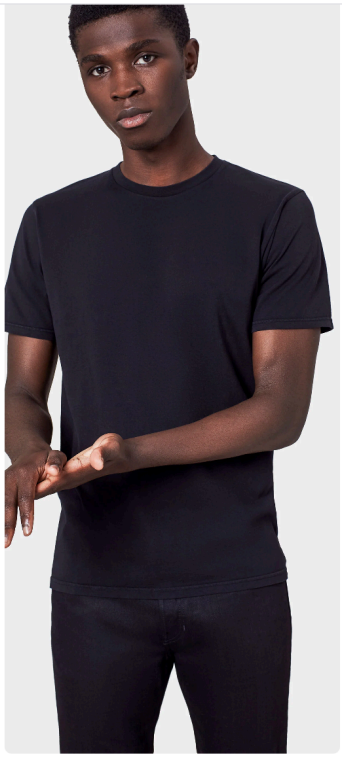
Za slike ćemo pronaći nekoliko javnih URL-ova slika proizvoda na internetu, a opis možemo izmisliti.

```
// server/data.js

class Proizvod {
  constructor(id, naziv, cijena, velicine, opis, slike) {
    this.id = id;
    this.naziv = naziv;
    this.cijena = cijena;
    this.velicine = velicine;
    this.opis = opis;
    this.slike = slike;
  }
}

const proizvodi = [
  new Proizvod(1, 'Obična crna majica', 80, ['XS', 'S', 'M', 'L']), // dodajte opis i
polje poveznica na slike
  new Proizvod(2, "Levi's 501 traperice", 110, ['S', 'M', 'L']),
  new Proizvod(3, 'Zimska kapa', 40, 'onesize'),
  new Proizvod(4, 'Čarape Adidas', 20, ['34-36', '37-39', '40-42']),
  new Proizvod(5, 'Tenisice Nike', 200, ['38', '39', '40', '41', '42', '43', '44', '45'])
];

export { proizvodi, Proizvod };
```

U `vue` aplikaciji, odnosno komponenti `ProductView.vue` možemo dodati prikaz slika i opisa proizvoda. Pronađite odgovarajuće HTML elemente i dodajte ih u template na isti način. Dodat ćemo slike ručno, bez korištenja `v-for` direktive budući da su različitih dimenzija.

```
<template>
  ...
  <!-- Image gallery -->
  <div class="mx-auto mt-6 max-w-2xl sm:px-6 lg:grid lg:max-w-7xl lg:grid-cols-3 lg:gap-x-8 lg:px-8">
    <div class="aspect-h-4 aspect-w-3 hidden overflow-hidden rounded-lg lg:block">
      <img :src="proizvod.slike[0]" alt="Two each of gray, white, and black shirts laying flat." class="h-full w-full object-cover object-center" />
    </div>
    <div class="hidden lg:grid lg:grid-cols-1 lg:gap-y-8">
      <div class="aspect-h-2 aspect-w-3 overflow-hidden rounded-lg">
        <img
          :src="proizvod.slike[3]"
          alt="illo inventore veritatis et quasi architecto beatae vitae"
          class="h-full w-full object-cover object-center"
        />
      </div>
      <div class="aspect-h-2 aspect-w-3 overflow-hidden rounded-lg">
        <img
          :src="proizvod.slike[2]"
          alt="accusantium doloremque laudantium, totam rem aperiam, eaque ipsa quae ab"
          class="h-full w-full object-cover object-center"
        />
      </div>
    </div>
    <div class="aspect-h-2 aspect-w-3 overflow-hidden rounded-lg">
      <img
        :src="proizvod.slike[1]"
        alt="Sed ut perspiciatis unde omnis iste natus error sit voluptatem"
        class="h-full w-full object-cover object-center"
      />
    </div>
  </div>
  ...
  <div class="space-y-6">
    <p class="text-base text-gray-900">{{ proizvod.opis }}</p>
  </div>
  ...
</template>
```

Uspješno smo pročitali sve podatke o proizvodu i prikazali ih na klijentskoj strani! Čestitke!



Obična crna majica

Otkrijte ultimativni spoj udobnosti i stila s našom premium crnom majicom kratkih rukava. Ova majica nije samo odjevni predmet; ona je vaš tihi saveznik u svakodnevnim avanturama, idealna za sve prilike i sve stilove. Izrađena je od visokokvalitetnog, prozračnog pamuka koji miluje vašu kožu poput nježnog povjetarca. Njena izrada pažljivo prati svaki šav, jamčeći trajnost i oblik kroz bezbrojna pranja, dok njezin besprijekorni kroj ističe vašu figuru, bez obzira na tjelesnu građu.

Karakteristike

- Sed ut perspiciatis unde omnis iste natus error sit voluptatem accusantium
- doloremque laudantium, totam rem aperiam, eaque ipsa quae ab illo inventore veritatis et quasi architecto
- beatae vitae dicta sunt explicabo
- Nemo enim ipsam voluptatem quia voluptas sit aspernatur aut odit aut fugit

80€

Boje

Veličina

XXS XS S M L

3.2 Slanje POST zahtjeva

Za kraj ćemo još samo pokazati kako slati POST zahtjeve na Express poslužitelju. U ovom slučaju, slat ćemo zahtjev za dodavanje novog proizvoda u našu trgovinu.

Jedina razlika je što POST zahtjev ima dodatno tijelo koje sadrži podatke koje želimo poslati na poslužitelj. U Axiosu, tijelo se definira kao drugi argument funkcije `axios.post`.

Sintaksa je sljedeća:

```
axios.post(url, data, config);
```

gdje su:

- `url`: URL adresa na koju šaljemo zahtjev
- `data`: podaci (tijelo zahtjeva) koje šaljemo na poslužitelj
- `config`: dodatne postavke zahtjeva (npr. zaglavlja, autentifikacija itd.)

Narudžbu možemo poslati na sljedeći način:

```
<script setup>
axios.post('http://localhost:3000/narudzbe', podaci);
</script>
```

- gdje su `podaci` objekt koji sadrži podatke o narudžbi

Na poslužitelju smo definirali da se naša narudžba sastoji od `id` i `naruceni_proizvodi` gdje su `naruceni_proizvodi` polje objekata koji sadrže `id` proizvoda i `narucena_kolicina` proizvoda. Međutim, logično je da se `id` narudžbe generira automatski na poslužitelju, što smo i implementirali ranije.

Dakle, naše tijelo zahtjeva može izgledati ovako:

```
let podaci = ref({
  naruceni_proizvodi: [
    { id: 1, narucena_kolicina: 2 },
    { id: 3, narucena_kolicina: 1 }
  ]
});
```

Sada ćemo poslati POST zahtjev s hardkodiranim podacima o narudžbi, čisto da vidite kako se to radi. Naravno, podaci se inače šalju nakon što korisnik doda sve proizvode u košaricu, uz ostale podatke o narudžbi i odradi plaćanje.

U Vue3 koristimo **Composition API** pa definiramo podatke i metode drugačije unutar skriptnog dijela. Već smo vidjeli kako definiramo podatke, a metode možemo definirate kao varijable koje pohranjuju složene funkcijske izraze.

Dummy podatke smo definirali iznad, a metodu za slanje narudžbe možemo definirati na sljedeći način:

```
// ProductView.vue
<script setup>
const posaljiNarudzbu = async () => {
  try {
    let response = await axios.post('http://localhost:3000/narudzbe', podaci.value); //
axios.post(url, data)
    console.log(response);
  } catch (error) {
    console.error('Greška u dohvatu podataka: ', error);
  }
};
</script>
```

Metodu možemo pozvati na kraju `onMounted` hooka ili pritiskom na gumb "Dodaj u košaricu" koristeći direktivu `@click`.

```
<button
  type="submit"
  @click="posaljiNarudzbu"
  class="mt-10 flex w-full items-center justify-center rounded-md border border-
transparent bg-indigo-600 px-8 py-3 text-base font-medium text-white hover:bg-indigo-700
focus:outline-none focus:ring-2 focus:ring-indigo-500 focus:ring-offset-2"
>
  Dodaj u košaricu
</button>
```

To je to! Ako ste dodali ispis na poslužitelju, možete vidjeti da se narudžba uspješno poslala, proizvodi su pronađeni te je izračunata ukupna cijena narudžbe.

Ako pogledamo karticu `Network` u Developer alatima, možemo pronaći zahtjev na `/narudzbe` i vidjeti da smo dobili statusni kod `201 Created` i nazad dobili podatke o narudžbi, upravo kako smo i definirali na poslužitelju.

Samostalni zadatak za Vježbu 3

Vaš zadatak je nadograditi aplikaciju `webshop` dodavanjem novih funkcionalnosti.

Kako već imate iskustva u razvoju klijentske strane aplikacije, morate implementirati sljedeće funkcionalnosti:

1. Instalirajte i podesite `vue-router` za navigaciju između stranica. Aplikacija mora imati dvije stranice: početnu stranicu (`/proizvodi`) koja će prikazivati sve proizvode kao kartice s osnovnim detaljima (vi ih odaberite i dizajnirajte kartice) i stranicu za prikaz pojedinog proizvoda koju imate već definiranu (`/proizvodi/:id`).
2. Implementirajte funkcionalnost gdje korisnik može stisnuti na određenu karticu proizvoda i biti preusmjeren na stranicu s detaljima proizvoda.
3. Jednom kad korisnik pristupi stranici s detaljima (`/proizvodi/:id`), mora se poslati GET zahtjev na Express poslužitelj za dohvat podataka o jednom proizvodu s odgovarajućim `id` parametrom. Naravno, na početnoj stranici `/proizvodi` morate u `onMounted` hook u kojem ćete poslati GET zahtjev za dohvat svih proizvoda.

Nadogradite Express poslužitelj na način da ćete za svaki proizvod dodati još nekoliko atributa:

- `dostupne_boje` (npr. crna, bijela, plava, crvena)
- `karakteristike` (npr. materijal, težina, informacije o održavanju)

Definirajte nekoliko proizvoda i nadopunite ih s novim atributima u `data.js` datoteci.

Nadogradite klijentsku stranu na način da ćete prikazati sve nove podatke o proizvodu na stranici s detaljima.

Kada korisnik odabere proizvod i pritisne gumb "Dodaj u košaricu", nemojte poslati narudžbu kao što je to sad slučaj, već spremite podatke u vanjsku `js` datoteku, npr. `narudzbe.js` ili `kosarica.js` ili u `localStorage`/`sessionStorage`. Na ovaj način pohranjujete podatke o narudžbi lokalno na strani klijenta (naučit ćemo kako to raditi bolje).

Nakon što korisnik doda proizvod, preusmjerite ga na početnu stranicu. Na početnoj stranici prikažite broj proizvoda u košarici, a ispod broja proizvoda dodajte još jedan gumb "Naruči proizvode" koji će poslati POST zahtjev `/narudzbe` na Express poslužitelj s podacima o narudžbi u tijelu zahtjeva.

> Rješenje učitajte na svoj GitHub repozitorij, a poveznicu na Merlin.