Web aplikacije (WA)

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

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

(6) Middleware funkcije

Middleware funkcije predstavljaju funkcijske komponente koje djeluju kao posrednici između dolaznog HTTP zahtjeva i odgovora poslužitelja. Njihova je uloga obrada zahtjeva prije nego što on dosegne krajnju logiku aplikacije. Među najčešćim primjenama *middleware* funkcija ističu se validacija podataka dolaznih zahtjeva te autorizacija pristupa resursima.

Validacijom podataka osigurava se da informacije koje korisnik šalje ispunjavaju unaprijed definirane kriterije, kao što su ispravnost sadržaja, struktura, duljina i tip podataka. U sklopu ove skripte obradit će se validacija na razini pojedine rute i na aplikacijskoj razini, uz poseban naglasak na biblioteku `express-validator`, koja pojednostavljuje proces validacije dolaznih zahtjeva korištenjem unaprijed pripremljenih *middleware* funkcija. Osim za validaciju, *middleware* ćemo također iskoristiti za strukturiranje koda u više datoteka, čime se postiže bolja organizacija i čitljivost aplikacije te za autentifikaciju i autorizaciju HTTP zahtjeva koju ćemo obraditi u sljedećoj skripti.

🆙 **Posljednje ažurirano: 13.1.2026.**

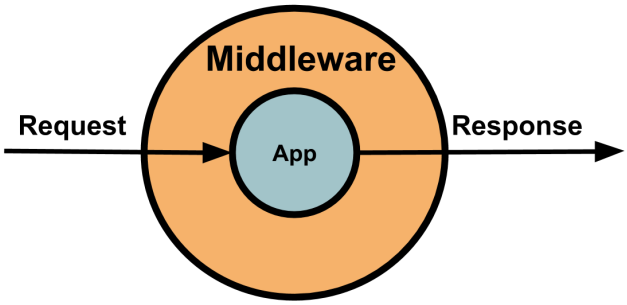
Sadržaj

1. Što su *middleware* funkcije?

Middleware funkcije (*eng. Middleware functions*) su funkcije koje se izvršavaju u različitim fazama obrade HTTP zahtjeva, tj. *request-response* ciklusa. U Express.js razvojnom okruženju, u pravilu se koriste u trenutku kad HTTP zahtjev stigne na poslužitelj, a prije konkretne obrade zahtjeva (*eng. route handler*) definirane u **implementaciji rute** odnosno endpointa. Međutim, mogu se koristiti i na **aplikacijskoj razini** (*eng. Application level middleware*) ili na **razini rutera** (*eng. Router level middleware*).

Middleware funkcije se koriste prvenstveno za:

- izvođenje koda koji se ponavlja u više različitih ruta ponovnom upotrebom `req` i `res` objekata
- izvođenje koda prije ili nakon obrade zahtjeva, npr. u svrhu validacije podataka, autentifikacije, autorizacije, logiranja itd.



> Slika 1. Ilustracija middleware funkcija u *request-response* ciklusu

1.1 *Middleware* na razini definicije rute

Najčešći oblik korištenja middleware funkcija je na razini definicije rute. U tom slučaju, middleware funkcija se definira kao argument metode `app.METHOD()`:

Sintaksa:

```
app.METHOD(path, [middleware], callback);
```

odnosno:

```
app.METHOD(path, [middleware_1, middleware_2, ..., middleware_n], callback);
```

gdje su:

- `app` - instanca Express aplikacije
- `METHOD` - HTTP metoda
- `path` - putanja na koju se odnosi ruta
- `middleware` - **middleware funkcija** ili **niz od** `N` **middleware funkcija - nema ograničenja**
- `callback` - funkcija koja se izvršava kad se HTTP zahtjev "poklopi" s definiranom rutom/endpointom

Middleware funkcije navodimo u **uglatim zagradama nakon putanje.**

Ako koristimo više *middleware* funkcija, **svaka od njih se izvršava redom**, a navodimo ih kao niz elemenata, identično kao elemente u polju.

Primjer definicije rute s middleware funkcijom:

```
app.get('/korisnici', middleware_fn, (req, res) => {
    // Obrada zahtjeva
});
```

- `middleware_fn` - middleware funkcija koja se izvršava prije obrade zahtjeva

Middleware funkcije imaju minimalno 3 parametra, i to:

- `req` - **objekt** dolaznog HTTP zahtjeva
- `res` - **objekt** HTTP odgovora koji se šalje korisniku
- `next` - **funkcija** koja se poziva kako bi se prešlo na sljedeću *middleware* funkciju ili na obradu zahtjeva tj. *route handler*

Dakle, *middleware* funkcije **imaju pristup *request* (`req`) i *response* (`res`) objektima**, jednako kao i *route handler* funkcija tj. *callback* funkcija rute.

Osnovna sintaksa middleware arrow funkcije s 3 parametra:

```
const middleware_fn = (req, res, next) => {
    // Izvođenje koda
    next(); // pozivanjem funkcije next() prelazimo na sljedeću middleware funkciju ili na
obradu zahtjeva
};
```

ili koristeći klasičnu funkcijsku deklaraciju:

```
function middleware_fn(req, res, next) {
    // Izvođenje koda
    next(); // pozivanjem funkcije next() prelazimo na sljedeću middleware funkciju ili na
obradu zahtjeva
}
```

Primjer: Definirat ćemo jednostavni Express poslužitelj koji obrađuje zahtjeve na putanji `GET /korisnici`. Korisnike ćemo definirati kao niz *in-memory* objekata s ključevima `id`, `ime` i `prezime`.

```
import express from 'express';

const app = express();
app.use(express.json());

let PORT = 3000;

app.listen(PORT, error => {
```

```
    if (error) {
        console.error(`Greška prilikom pokretanja poslužitelja: ${error.message}`);
    } else {
        console.log(`Poslužitelj dela na http://localhost:${PORT}`);
    }
});
```

Dodajemo rute za dohvat svih korisnika (`GET /korisnici`) i pojedinog korisnika (`GET /korisnici/:id`):

```
let korisnici = [
    { id: 983498354, ime: 'Ana', prezime: 'Anić', email: 'aanic@gmail.com' },
    { id: 983498355, ime: 'Ivan', prezime: 'Ivić', email: 'iivic@gmail.com' },
    { id: 983498356, ime: 'Sanja', prezime: 'Sanjić', email: 'ssanjic123@gmail.com' }
];

// dohvat svih korisnika
app.get('/korisnici', async (req, res) => {
    if (korisnici) {
        return res.status(200).json(korisnici);
    }
    return res.status(404).json({ message: 'Nema korisnika' });
});

// dohvat pojedinog korisnika
app.get('/korisnici/:id', async (req, res) => {
    const id_route_param = parseInt(req.params.id);
    const korisnik = korisnici.find(korisnik => korisnik.id === id_route_param);
    if (korisnik) {
        return res.status(200).json(korisnik);
    }
    return res.status(404).json({ message: 'Korisnik nije pronađen' });
});
```

U redu, do sad nismo koristili *middleware* funkcije niti imamo potrebu za njima u kodu iznad.

Međutim, što ako **želimo dodati još jednu rutu koja će ažurirati email adresu** pojedinog korisnika, primjerice `PATCH /korisnici/:id`?

```
// ažuriranje email adrese pojedinog korisnika
app.patch('/korisnici/:id', async (req, res) => {
    const id_route_param = parseInt(req.params.id);
    const korisnik = korisnici.find(korisnik => korisnik.id === id_route_param);
    if (korisnik) {
        korisnik.email = req.body.email;
        console.log(korisnici);
        return res.status(200).json(korisnik);
    }
    return res.status(404).json({ message: 'Korisnik nije pronađen' });
});
```

Primjerice: želimo ažurirati email `Sanje Sanjić` na `"saaaanja123@gmail.com"`. Kako bismo to učinili, koristimo HTTP `PATCH` metodu i šaljemo sljedeći HTTP zahtjev:

```
PATCH http://localhost:3000/korisnici/983498356
Content-Type: application/json

{
  "email": "saaaanja123@gmail.com"
}
```

Polako možemo uočavati potrebu za korištenjem *middleware* funkcija **na razini definicije rute**. Potreba se javlja **prilikom validacije tijela dolaznog HTTP zahtjeva**, odnosno želimo provjeriti je li korisnik poslao ispravnu JSON strukturu (objekt) s ključem `email` te je li vrijednost ključa `email` tipa string, a naposljetku i je li email adresa ispravna.

Do sad smo isto provjeravali u samoj *callback* funkciji rute, recimo na sljedeći način:

```
app.patch('/korisnici/:id', async (req, res) => {
    const id_route_param = parseInt(req.params.id);
    const korisnik = korisnici.find(korisnik => korisnik.id === id_route_param);
    if (korisnik) {
        // postoji li ključ email i je li tipa string
        if (req.body.email && typeof req.body.email === 'string') {
            // trebali bi dodati još provjera za ispravnost strukture email adrese
            korisnik.email = req.body.email;
            console.log(korisnici);
            return res.status(200).json(korisnik);
        }
        return res.status(400).json({ message: 'Neispravna struktura tijela zahtjeva' });
    }
    return res.status(404).json({ message: 'Korisnik nije pronađen' });
});
```

Što ako još želimo provjeriti ispravnost email adrese?

- Praktično bi bilo to implementirati u vanjskoj funkciji, koristiti neku biblioteku ili regularni izraz.
- U svakom slučaju, to je posao koji može obaviti *middleware* funkcija budući da **kod postaje sve složeniji** s **previše** `if` **grananja** te ga je **potrebno ponavljati u više ruta**.

Idemo vidjeti kako bismo ove provjere implementirali u *middleware* funkciji. Znamo da one imaju pristup *request* (`req`) i *response* (`res`) objektima.

Middleware funkciju možemo nazvati `validacijaEmaila`:

```
// middleware funkcija
const validacijaEmaila = (req, res, next) => {
    //implementacija
    next();
};
```

Jednostavno preslikamo istu provjeru od ranije:

```
// middleware funkcija
const validacijaEmaila = (req, res, next) => {
    if (req.body.email && typeof req.body.email === 'string') {
        // ako postoji ključ email i tipa je string
        next(); // prelazimo na sljedeću middleware funkciju odnosno na obradu zahtjeva
(route handler/callback)
    }
    // u suprotnom?
};
```

- U suprotnom, tj. ako uvjet nije zadovoljen, želimo poslati korisniku odgovor s statusom `400` i porukom `"Neispravna struktura tijela zahtjeva"`

```
// middleware funkcija
const validacijaEmaila = (req, res, next) => {
    if (req.body.email && typeof req.body.email === 'string') {
        // ako postoji ključ email i tipa je string
        next(); // prelazimo na sljedeću middleware funkciju odnosno na obradu zahtjeva
    }
    // u suprotnom
    return res.status(400).json({ message: 'Neispravna struktura tijela zahtjeva' });
};
```

Jednom kad smo definirali *middleware* funkciju, **dodajemo ju kao drugi argument** *endpoint definition* metode `app.patch()`, a prethodnu provjeru uklanjamo iz *callback* funkcije rute:

- ako ruta ima samo jedan *middleware*, možemo i **izostaviti uglate zagrade** `[...]`

```
// dodajemo validacijaEmaila kao drugi argument
app.patch('/korisnici/:id', [validacijaEmaila], async (req, res) => {
    const id_route_param = parseInt(req.params.id);
    const korisnik = korisnici.find(korisnik => korisnik.id === id_route_param);
    if (korisnik) {
        korisnik.email = req.body.email;
        console.log(korisnici);
        return res.status(200).json(korisnik);
    }
    return res.status(404).json({ message: 'Korisnik nije pronađen' });
});
```

> **Važno!** *Middleware* `validacijaEmaila` će se izvršiti prije obrade zahtjeva u *callback* funkciji rute. Ako uvjeti nisu zadovoljeni, *middleware* će poslati odgovor korisniku sa statusom `400` i porukom `"Neispravna struktura tijela zahtjeva"`, dok se *callback* funkcija nikada neće izvršiti

Druga velika prednost korištenja *middleware* funkcija je **ponovna upotrebljivost koda** (eng. *reusability*).

- Naime, često je slučaj da više ruta zahtjeva iste provjere, i to istim redoslijedom.

- U tom slučaju, umjesto da kopiramo isti kod u svaku rutu, možemo ga jednostavno izdvojiti u zasebnu *middleware* funkciju i koristiti ju u svakoj ruti koja zahtjeva tu provjeru.

Sada imamo sljedeće rute:

- `GET /korisnici` - dohvat svih korisnika
- `GET /korisnici/:id` - dohvat pojedinog korisnika
- `PATCH /korisnici/:id` - ažuriranje email adrese pojedinog korisnika

Ako pogledamo implementacije, vidimo da u svakoj ruti koristimo `parseInt(req.params.id)` kako bismo dobili brojčanu vrijednost `id` parametra rute te zatim pretražujemo korisnika po tom `id`-u.

Ovo je odličan primjer gdje možemo koristiti *middleware* funkciju!

Nazvat ćemo ju `pretragaKorisnika`

Prve dvije linije *middlewarea* `pretragaKorisnika` su identične kao i u metodama `GET /korisnici/:id` i `PATCH /korisnici/:id`:

```
const pretragaKorisnika = (req, res, next) => {
    const id_route_param = parseInt(req.params.id);
    const korisnik = korisnici.find(korisnik => korisnik.id === id_route_param);
};
```

Ako korisnik postoji, želimo nastaviti s izvođenjem sljedeće *middleware* funkcije ili s obradom zahtjeva u *callbacku*, u suprotnom želimo poslati korisniku odgovor s statusom `404` i porukom `"Korisnik nije pronađen"`.

```
const pretragaKorisnika = (req, res, next) => {
    const id_route_param = parseInt(req.params.id);
    const korisnik = korisnici.find(korisnik => korisnik.id === id_route_param);
    if (korisnik) {
        next(); // prelazimo na sljedeću middleware funkciju odnosno na obradu zahtjeva
    }
    return res.status(404).json({ message: 'Korisnik nije pronađen' });
};
```

Dodatno, kako su `req` i `res` objekti **globalni na razini/opsegu definicije rute**, možemo jednostavno dodati svojstvo `korisnik` u `req` objekt kako bismo ga mogli koristiti u svim drugim *middlewareima* ili u *callback* funkciji rute 🚀

```
const pretragaKorisnika = (req, res, next) => {
    const id_route_param = parseInt(req.params.id);
    const korisnik = korisnici.find(korisnik => korisnik.id === id_route_param);
    if (korisnik) {
        req.korisnik = korisnik; // Ključna linija: dodajemo svojstvo korisnik na req
objekt
        next(); // prelazimo na sljedeću middleware funkciju odnosno na obradu zahtjeva
    }
    return res.status(404).json({ message: 'Korisnik nije pronađen' });
};
```

> **Dodatno pojašnjenje**: Zašto je ovo moguće? Objekt `req` je isti objekt koji se prosljeđuje kroz sve *middleware* funkcije i *callback* funkciju rute, a budući da se radi o standardnom JavaScript objektu, znamo da možemo dinamički dodavati (ili brisati) svojstva na objektu. Stoga, dodavanjem svojstva `korisnik` na `req` objekt unutar *middleware* funkcije `pretragaKorisnika`, to svojstvo postaje dostupno u svim sljedećim *middleware* funkcijama i u *callback* funkciji rute.

Sada možemo refaktorirati rute `GET /korisnici/:id` i `PATCH /korisnici/:id`. Prvo ćemo rutu `GET /korisnici/:id`

Pogledajmo trenutačnu implementaciju:

```
app.get('/korisnici/:id', async (req, res) => {
    const id_route_param = parseInt(req.params.id);
    const korisnik = korisnici.find(korisnik => korisnik.id === id_route_param);
    if (korisnik) {
        return res.status(200).json(korisnik);
    }
    return res.status(404).json({ message: 'Korisnik nije pronađen' });
});
```

Vidimo da možemo izbaciti gotovo sve! Ostaje nam samo slanje `korisnik` objekta sa statusom `200`.

Dodajemo *middleware* `pretragaKorisnika`:

```
app.get('/korisnici/:id', [pretragaKorisnika], async (req, res) => {
    // implementacija
});
```

> Čitaj:
>
> - Prije obrade zahtjeva, izvrši *middleware* `pretragaKorisnika`.
> - Ako *middleware* prođe (tj. vrati `next()`), nastavi s obradom zahtjeva odnosno izvrši *callback* funkciju rute.

Dakle, samo vraćamo korisnika koji se sad nalazi u `req.korisnik`:

```
app.get('/korisnici/:id', [pretragaKorisnika], async (req, res) => {
    return res.status(200).json(req.korisnik);
});
```

To je to! Idemo refaktorirati i rutu `PATCH /korisnici/:id`.

Pogledajmo trenutačnu implementaciju:

```
app.patch('/korisnici/:id', [validacijaEmaila], async (req, res) => {
    const id_route_param = parseInt(req.params.id);
    const korisnik = korisnici.find(korisnik => korisnik.id === id_route_param);
    if (korisnik) {
        korisnik.email = req.body.email;
        console.log(korisnici);
        return res.status(200).json(korisnik);
    }
    return res.status(404).json({ message: 'Korisnik nije pronađen' });
});
```

Ruta već sadrži *middleware* `validacijaEmaila`. Međutim, mi moramo **prvo provjeriti ispravnost** `id`**-a, odnosno provjeriti postojanje korisnika**.

- To ćemo jednostavno učiniti dodavanjem *middlewarea* `pretragaKorisnika` **prije** `validacijaEmaila`:

```
// dodajemo middleware pretragaKorisnika prije validacijaEmaila
app.patch('/korisnici/:id', [pretragaKorisnika, validacijaEmaila], async (req, res) => {
    // implementacija
});
```

Sada možemo izbaciti sve provjere iz *callback* funkcije rute:

```
app.patch('/korisnici/:id', [pretragaKorisnika, validacijaEmaila], async (req, res) => {
    req.korisnik.email = req.body.email; // ostavljamo samo promjenu emaila
    console.log(korisnici); // možemo pustiti i ispis strukture
    return res.status(200).json(req.korisnik); // vraćamo korisnika
});
```

To je to! 😎 Uočite koliko *middleware* funkcije čine kod čitljivijim!

Međutim, prije nego nastavimo, uočite sljedeće:

- slanjem zahtjeva na `GET /korisnici/:id` dobivamo korisnika s određenim `id`-em, što je OK ali dobivamo i sljedeću grešku u konzoli:

```
Error [ERR_HTTP_HEADERS_SENT]: Cannot set headers after they are sent to the client
    at ServerResponse.setHeader (node:_http_outgoing:699:11)
    ...
```

- slanjem zahtjeva na `PATCH /korisnici/:id` odradit ćemo izmjenu email adrese, ali dobivamo dvaput istu grešku u konzoli:

```
Error [ERR_HTTP_HEADERS_SENT]: Cannot set headers after they are sent to the client
    at ServerResponse.setHeader (node:_http_outgoing:699:11)
    ...
```

Zašto dobivamo ove greške? 🤔

▸ Spoiler alert! Odgovor na pitanje

Dodat ćemo ispise na početku svake *middleware* funkcije kako bismo pratili redoslijed njihova izvršavanja:

```
const validacijaEmaila = (req, res, next) => {
    console.log('Middleware: validacijaEmaila');
    if (req.body.email && typeof req.body.email === 'string') {
        next(); // prelazimo na sljedeću middleware funkciju odnosno na obradu zahtjeva
    }
    return res.status(400).json({ message: 'Neispravna struktura tijela zahtjeva' });
};

const pretragaKorisnika = (req, res, next) => {
    console.log('Middleware: pretragaKorisnika');
    const id_route_param = parseInt(req.params.id);
    const korisnik = korisnici.find(korisnik => korisnik.id === id_route_param);
    if (korisnik) {
        req.korisnik = korisnik;
        next(); // prelazimo na sljedeću middleware funkciju odnosno na obradu zahtjeva
    }
    return res.status(404).json({ message: 'Korisnik nije pronađen' });
};
```

Kako bismo sigurno prekinuli izvršavanje trenutne *middleware* funkcije, dodajemo `return` ispred `next()`:

```
const validacijaEmaila = (req, res, next) => {
    console.log('Middleware: validacijaEmaila');
    if (req.body.email && typeof req.body.email === 'string') {
        return next(); // prelazimo na sljedeću middleware funkciju odnosno na obradu
zahtjeva
    }
    return res.status(400).json({ message: 'Neispravna struktura tijela zahtjeva' });
};

const pretragaKorisnika = (req, res, next) => {
    console.log('Middleware: pretragaKorisnika');
    const id_route_param = parseInt(req.params.id);
    const korisnik = korisnici.find(korisnik => korisnik.id === id_route_param);
    if (korisnik) {
        req.korisnik = korisnik;
```

```
        return next(); // prelazimo na sljedeću middleware funkciju odnosno na obradu
zahtjeva
    }
    return res.status(404).json({ message: 'Korisnik nije pronađen' });
};
```

Ili, koristimo `else` uvjetni izraz kada šaljemo statusni kod `4xx`:

```
const validacijaEmaila = (req, res, next) => {
    console.log('Middleware: validacijaEmaila');
    if (req.body.email && typeof req.body.email === 'string') {
        return next(); // prelazimo na sljedeću middleware funkciju odnosno na obradu
zahtjeva
    } else {
        return res.status(400).json({ message: 'Neispravna struktura tijela zahtjeva' });
    }
};

const pretragaKorisnika = (req, res, next) => {
    console.log('Middleware: pretragaKorisnika');
    const id_route_param = parseInt(req.params.id);
    const korisnik = korisnici.find(korisnik => korisnik.id === id_route_param);
    if (korisnik) {
        req.korisnik = korisnik;
        return next(); // prelazimo na sljedeću middleware funkciju odnosno na obradu
zahtjeva
    } else {
        return res.status(404).json({ message: 'Korisnik nije pronađen' });
    }
};
```

1.2 Strukturiranje programa u više datoteka

Rekli smo da je jedna od glavnih prednosti korištenja *middleware* funkcija **ponovna upotrebljivost koda** (*eng. Code reusability*). Međutim, vidite da već sad `index.js` datoteka postaje nečitljiva zbog miješanja definicija ruta i *middleware* funkcija. Uobičajena praksa je odvojiti *middleware* funkcije u zasebne datoteke, jednako kao što smo radili i za definicije rute koristeći `express.Router()`.

Napravit ćemo dva nova direktorija, jedan za rute i jedan za *middleware* funkcije:

```
→ mkdir routes
→ mkdir middleware
```

Obzirom da u pravilu želimo koristiti istu skupinu ruta s istim *middleware* funkcijama, možemo jednako nazvati datoteke u direktoriju `routes` i `middleware`: nazvat ćemo ih `korisnici.js`.

Naša struktura poslužitelja sada izgleda ovako:

```
.
├── middleware
│   └── korisnici.js
├── routes
│   └── korisnici.js
├── index.js
├── node_modules
├── package.json
└── package-lock.json
```

Prvo ćemo jednostavno prebaciti definicije middleware funkcija iz `index.js` u `middleware/korisnici.js`:

```
// middleware/korisnici.js

const validacijaEmaila = (req, res, next) => {
    console.log('Middleware: validacijaEmaila');
    if (req.body.email && typeof req.body.email === 'string') {
        return next(); // prelazimo na sljedeću middleware funkciju odnosno na obradu
zahtjeva
    } else {
        return res.status(400).json({ message: 'Neispravna struktura tijela zahtjeva' });
    }
};

const pretragaKorisnika = (req, res, next) => {
    console.log('Middleware: pretragaKorisnika');
    const id_route_param = parseInt(req.params.id);
    const korisnik = korisnici.find(korisnik => korisnik.id === id_route_param);
    if (korisnik) {
        req.korisnik = korisnik;
```

```
        return next(); // prelazimo na sljedeću middleware funkciju odnosno na obradu
zahtjeva
    } else {
        return res.status(404).json({ message: 'Korisnik nije pronađen' });
    }
};
// izvoz middleware funkcija
export { validacijaEmaila, pretragaKorisnika };
```

Moramo još prebaciti podatke:

```
// middleware/korisnici.js

let korisnici = [
    { id: 983498354, ime: 'Ana', prezime: 'Anić', email: 'aanic@gmail.com' },
    { id: 983498355, ime: 'Ivan', prezime: 'Ivić', email: 'iivic@gmail.com' },
    { id: 983498356, ime: 'Sanja', prezime: 'Sanjić', email: 'ssanjic123@gmail.com' }
];
```

Prebacujemo definicije ruta iz `index.js` u `routes/korisnici.js`:

```
// routes/korisnici.js

import express from 'express';
// uključujemo middleware funkcije iz middleware/korisnici.js
import { validacijaEmaila, pretragaKorisnika } from '../middleware/korisnici.js';

const router = express.Router();

let korisnici = [
    { id: 983498354, ime: 'Ana', prezime: 'Anić', email: 'aanic@gmail.com' },
    { id: 983498355, ime: 'Ivan', prezime: 'Ivić', email: 'iivic@gmail.com' },
    { id: 983498356, ime: 'Sanja', prezime: 'Sanjić', email: 'ssanjic123@gmail.com' }
];

// endpoint ne koristi middleware
router.get('/', async (req, res) => {
    if (korisnici) {
        return res.status(200).json(korisnici);
    }
    return res.status(404).json({ message: 'Nema korisnika' });
});

// endpoint koristi middleware pretragaKorisnika
router.get('/:id', [pretragaKorisnika], async (req, res) => {
    return res.status(200).json(req.korisnik);
});

// endpoint koristi middleware pretragaKorisnika → validacijaEmaila, tim redom
router.patch('/:id', [pretragaKorisnika, validacijaEmaila], async (req, res) => {
    req.korisnik.email = req.body.email;
```

```
    console.log(korisnici);
    return res.status(200).json(req.korisnik);
});

export default router;
```

U `index.js` datoteci uključujemo router i dodajemo odgovarajući prefiks:

```
// index.js

import express from 'express';
import korisniciRouter from './routes/korisnici.js';

const app = express();
app.use(express.json()); // ovaj aplikacijski middleware uvijek dodajemo prije importanja routera jer ga želimo primijeniti na sve rute
app.use('/korisnici', korisniciRouter);

let PORT = 3000;

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

1.3 Middleware na aplikacijskoj razini

Pokazali smo kako definirati *middleware* funkcije na razini definicije rute, unutar metoda `app.METHOD(URL, [middleware_1, middleware_2, ... middleware_N], callback)`.

Međutim, ***middleware* funkcije možemo definirati i na razini aplikacije**, tj. na razini objekta `app`.

app objekt konvencionalno se koristi za konfiguraciju ukupne Express.js aplikacije, a instancira se pozivanjem funkcije `express()` → `const app = express();`.

Primjerice, ako želimo da se neka *middleware* funkcija izvrši prije svake rute, neovisno je li to `GET /korisnici`, `GET /korisnici/:id` ili `PATCH /korisnici/:id` ili pak `GET /pizze` itd, možemo ju na razini aplikacijskog objekta (*eng. Application-level middleware*).

Primjer: Možemo definirati *middleware* `timestamp` koja će ispisati u konzolu **trenutni datum i vrijeme** svaki put kad se zaprimi zahtjev na poslužitelju:

```
// index.js

const timer = (req, res, next) => {
    console.log(`Trenutno vrijeme: ${new Date().toLocaleString()}`);
    next();
};

// koristimo timer middleware na aplikacijskoj razini
app.use(timer);
```

Međutim, uključivanje ovog *middlewarea* moramo definirati **prije** uključivanja routera, **inače će se odnositi samo na rute koje slijede nakon njega**:

```
// index.js

import express from 'express';
import korisniciRouter from './routes/korisnici.js';

const app = express();
app.use(express.json());

const timer = (req, res, next) => {
    console.log(`Trenutno vrijeme: ${new Date().toLocaleString()}`);
    next();
};

// koristimo timer middleware na aplikacijskoj razini
app.use(timer);

app.use('/korisnici', korisniciRouter);

let PORT = 3000;

app.listen(PORT, error => {
```

```
    if (error) {
        console.error(`Greška prilikom pokretanja poslužitelja: ${error.message}`);
    } else {
        console.log(`Poslužitelj dela na http://localhost:${PORT}`);
    }
});
```

Pogledajte malo bolje kod. Uočavate li još negdje *middleware* koji smo do sad uvijek koristili? 🤔

▸ Spoiler alert! Odgovor na pitanje

> Pokušajte poslati zahtjev na bilo koju rutu i uočite ispis trenutnog vremena u konzoli.

Dobra praksa, pogotovo u produkcijskom okruženju, jest definirati *middleware* na razini aplikacije koji ispisuje *logove* o svakom zahtjevu koji stigne na poslužitelj. Ovo je korisno za praćenje i analizu ponašanja poslužitelja, kao i za *debugging*.

Primjerice, želimo ispisati trenutni datum, vrijeme, metodu HTTP zahtjeva i URL zahtjeva:

```
[1/6/2025, 12:30:40 PM] : GET /korisnici
```

Vrijeme znamo izračunati, HTTP metoda se nalazi u `req.method`, a URL zahtjeva u `req.originalUrl`.

Rješenje:

```
// index.js

const requestLogger = (req, res, next) => {
    const date = new Date().toLocaleString();
    const method = req.method; // HTTP metoda
    const url = req.originalUrl; // URL zahtjeva
    console.log(`[${date}] : ${method} ${url}`);
    next();
};

app.use(requestLogger);
```

Testirajmo slanjem zahtjeva na `GET http://localhost:3000/korisnici/983498356`

Ispis u konzoli:

```
Poslužitelj dela na http://localhost:3000
[1/6/2025, 12:33:31 PM] : GET /korisnici/983498356
Middleware: pretragaKorisnika
```

---

ili slanjem zahtjeva na: `PATCH http://localhost:3000/korisnici/983498356` s tijelom zahtjeva:

```
{
    "email": "sanja.sanjić@gmail.com"
}
```

Ispis u konzoli:

```
[1/6/2025, 12:34:49 PM] : PATCH /korisnici/983498356
Middleware: pretragaKorisnika
Middleware: validacijaEmaila
```

Osim pozivanja *middlewarea* na **aplikacijskog razini na svim rutama**, možemo ga pozvati i na definiranoj ruti za sve HTTP metode.

- *Primjerice*: ako imamo skupinu ruta URL-a `/admin`. Želimo u konzoli *logirati* da je pristigao zahtjev na bilo koju `/admin` rutu, **neovisno o metodi HTTP zahtjeva**.

Rješenje: Koristimo funkciju `app.all()` odnosno `router.all()`:

```
// index.js

const adminLogger = (req, res, next) => {
    console.log('Oprez! Pristigao zahtjev na /admin rutu');
    // u pravilu ovdje moramo provjeriti autorizacijski token - ovo ćemo raditi na WA7
    next();
};

app.all('/admin', adminLogger); // na svim /admin rutama pozovi adminLogger middleware funkciju
// odnosno
router.all('/admin', adminLogger);
```

Primjerice: Ako pošaljemo zahtjev na `GET http://localhost:3000/admin`, u konzoli ćemo dobiti ispis:

```
Poslužitelj dela na http://localhost:3000
Oprez! Pristigao zahtjev na /admin rutu
[1/6/2025, 12:50:04 PM] : PATCH /admin
```

Kada definiramo middleware na razini aplikacije, ponekad želimo uključiti i 4. neobavezni parametar (`err`) kako bismo mogli uhvatiti greške koje se dogode u *middleware* funkciji. Ovaj parametar se koristi za hvatanje grešaka koje se dogode u *middleware* funkciji.

Primjer:

```
// index.js

const errorHandler = (err, req, res, next) => {
    console.log(err);
    res.status(500).json({ message: 'Greška na poslužitelju' });
};

app.use(errorHandler);
```

Kada će se izvršiti ovaj *middleware*? 🤔

▸ Spoiler alert! Odgovor na pitanje

Možemo uvijek provjeriti simulacijom greške u nekoj ruti:

```
// index.js

app.get('/error', (req, res) => {
    throw new Error('Simulirana greška na poslužitelju');
});
```

> Zapamti: *Middleware* funkcije na razini rutera (*eng. Router level middleware*) definiramo na **identičan način kao i na razini aplikacije/rute**, samo što ih dodajemo kao drugi argument metode `router.METHOD()`, gdje je `router` instanca `express.Router()` konstruktora.

2. `express-validator` biblioteka

express-validator biblioteka nudi **skup gotovih *middleware* funkcija za validaciju podataka** u zahtjevima. Biblioteka zahtjeva Node.js 14+ verziju i Express.js 4.17.1+ verziju.

`express-validator` biblioteka kroz svoje *middleware* funkcije nudi dvije vrste provjera:

1. **Validacija** (*eng. Validation*): **provjera ispravnosti podataka** u zahtjevu
2. **Sanitizacija** (*eng. Sanitization*): **čišćenje podataka u zahtjevu** u sigurno stanje

Do sada smo ručno definirali validacije direktno unutar *callback* funkcija ruta ili unutar zasebnih *middleware* funkcija. Takva praksa je u redu za jednostavne aplikacije, ali kako aplikacije rastu, počinjemo ponavljati iste validacije na više mjesta, što dovodi do duplikacije koda i otežava održavanje.

Također, veliki broj validacija podataka koji radimo su uobičajene i često korištene validacije u web aplikacijama, poput provjere ispravnosti emaila, jačine lozinke, formata datuma itd. Iz tog razloga, postoji veliki broj gotovoih biblioteka koje nude skupove unaprijed definiranih validacija i sanitizacija. Međutim, potrebno je razumijeti middleware koncept kako bismo znali ispravno koristiti te biblioteke.

> Napomena: Osim `express-validator`, postoje i druge popularne biblioteke za validaciju podataka u Node.js aplikacijama, poput `Joi`, `Yup`, `Validator.js`, `Vine.js`, `Zod` itd. Za projekte iz kolegija možete odabrati bilo koju od navedenih biblioteka, a mi ćemo u nastavku ove skripte proći kroz `express-validator` kao primjer.

> Slika 2: Express-validator je popularna biblioteka za validaciju podataka u Express.js aplikacijama.

Instalirajmo biblioteku:

```
npm install express-validator
```

2.1 Učitavanje modula

Učitajmo uobičajene funkcije iz `express-validator` biblioteke:

```
// index.js

import { body, validationResult, query, param } from 'express-validator';
```

- `body()` - funkcija koja definira **provjere za tijelo zahtjeva**

- `validationResult(req)` - funkcija koja **izračunava rezultate provjera zahtjeva**
- `query()` - funkcija koja definira **provjere za *query* parametre**
- `param()` - funkcija koja definira **provjere za route parametre**
- `check()` - funkcija koja definira **provjere za bilo koji dio zahtjeva**

Primjerice: definirat ćemo super jednostavni endpoint `GET /hello` koji očekuje *query* parametar `ime`:

```
app.get('/hello', (req, res) => {
    res.send('Hello, ' + req.query.ime);
});
```

Ako pošaljemo zahtjev bez *query* parametra `name`, dobit ćemo odgovor `"Hello, undefined"`.

Validator dodajemo na isti način kao i prethodno manualno definirane *middleware* funkcije, a to je kao drugi argument metode `app.METHOD()`.

- to je zato što su validatori ustvari predefinirane *middleware* funkcije

Koristimo `query` funkciju za provjeru *query* parametra `ime`:

Sintaksa:

```
query('key');
```

U našem slučaju je to:

```
query('ime');
```

➡️ Validator za provjeru da li **vrijednost nije prazna** `notEmpty()`.

Jednostavno vežemo na rezultat funkcije `query()`:

```
query('ime').notEmpty();
```

To je to! Sad ga još samo dodajemo u našu rutu:

```
//index.js

app.get('/hello', [query('ime').notEmpty()], (req, res) => {
    res.send('Hello, ' + req.query.ime);
});
```

Ako pokušate ponovno poslati zahtjev bez, i dalje ćete dobiti odgovor `"Hello, undefined"`.

Razlog tomu je što `express-validator` ne izvještava automatski klijenta o greškama. Dodavanjem dodatnih validatora, moramo ručno definirati strukturu JSON odgovora u slučaju greške.

2.2 Obrada validacijskih grešaka

Kako bismo dobili rezultate provjere, koristimo funkciju `validationResult(req)` koja prima `req` objekt i **vraća rezultate provjere u slučaju da dode do greške**.

```
const errors = validationResult(req); // sprema greške svih validacija koje su provele middleware funkcije, ako ih ima!
```

Dodajemo u našu rutu i ispisom provjeravamo sadržaj:

```
//index.js

app.get('/hello', [query('ime').notEmpty()], (req, res) => {
    const errors = validationResult(req); // spremanje grešaka
    console.log(errors);
    res.send('Hello, ' + req.query.ime);
});
```

Ako nema grešaka, npr. ako pošaljemo zahtjev: `GET http://localhost:3000/hello?ime=Ana`, dobivamo sljedeći ispis:

```
Result { formatter: [Function: formatter], errors: [] }
```

Ako pošaljemo zahtjev bez *query* parametra, npr. `GET http://localhost:3000/hello`, dobivamo detaljan ispis s detaljima o pogrešci:

```
Result {
  formatter: [Function: formatter],
  errors: [
    {
      type: 'field',
      value: '',
      msg: 'Invalid value',
      path: 'ime',
      location: 'query'
    }
  ]
}
```

Kako čitamo ispis? "Greška je nastala u *query* parametru naziva `ime`, jer je njegova vrijednost `value` prazna."

➡️ Funkcijom `isEmpty()` možemo **provjeriti je li vrijednost prazna.**

Ako greške ne postoje (tj. `errors.isEmpty() == true`), šaljemo odgovor `OK` klijentu, u suprotnom šaljemo odgovor s detaljima o grešci koji je dostupan u `errors.array()` uz status `Bad Request`.

```
// index.js

app.get('/hello', [query('ime').notEmpty()], (req, res) => {
    const errors = validationResult(req);
    // ako nema greške
    if (errors.isEmpty()) {
        return res.send('Hello, ' + req.query.ime);
    }
    return res.status(400).json({ errors: errors.array() });
});
```

2.3 Kombiniranje vlastitih *middlewarea* s `express-validator`

Moguće je kombinirati vlastite *middleware* funkcije s `express-validator` validatorima.

- Primjerice, ako želimo provjeriti da li je korisnik s određenim `id`-om pronađen, a zatim provjeriti da li je email ispravan, možemo iskoristiti vlastiti `pretragaKorisnika` *middleware* koji se nalazi u `middleware/korisnici.js`, a ostatak provjere odraditi kroz `express-validator` biblioteku.

Primjer: Nadogradit ćemo rutu `PATCH /korisnici:id` tako da provjerava ispravnost email adrese.

Prvi korak je izbaciti postojeći vlastiti middleware za provjeru email adrese:

```
// routes/korisnici.js

// uklanjamo validacijaEmaila middleware
router.patch('/:id', [pretragaKorisnika], async (req, res) => {
    req.korisnik.email = req.body.email;
    console.log(korisnici);
    return res.status(200).json(req.korisnik);
});
```

Želimo provjeriti sljedeće:

- da li je ključ `email` proslijeđen u **tijelu zahtjeva**, dakle koristimo `body('email')` a ne `query('email')`
- da li je vrijednost ključa `email` ispravno strukturirana

➡️ Funkcijom `isEmail()` možemo brzo provjeriti je li vrijednost email adrese ispravna.

- dodajemo provjeru kao drugi *middleware* u nizu, nakon `pretragaKorisnika`

```
// routes/korisnici.js

router.patch('/:id', [pretragaKorisnika, body('email').isEmail()], async (req, res) => {
    req.korisnik.email = req.body.email;
    console.log(korisnici);
    return res.status(200).json(req.korisnik);
});
```

Na kraju još dodajemo obradu grešaka te vraćamo klijentu odgovarajuće JSON odgovore:

```
// routes/korisnici.js

router.patch('/:id', [pretragaKorisnika, body('email').isEmail()], async (req, res) => {
    const errors = validationResult(req);
    // ako nema greške
    if (errors.isEmpty()) {
        req.korisnik.email = req.body.email;
        console.log(korisnici);
        return res.status(200).json(req.korisnik);
    }
    return res.status(400).json({ errors: errors.array() });
});
```

Primjerice: ako pokušamo proslijediti neispravnu email adresu, npr. `PATCH http://localhost:3000/korisnici/983498356` s tijelom zahtjeva:

```
{
    "email": "sssssanja123gmail.com"
}
```

Dobivamo natrag JSON odgovor s detaljima o grešci:

```
{
    "errors": [
        {
            "type": "field",
            "value": "sssssanja123gmail.com",
            "msg": "Invalid value",
            "path": "email",
            "location": "body"
        }
    ]
}
```

Ako pokušamo definirati pogrešan ključ u tijelu zahtjeva, npr. `PATCH http://localhost:3000/korisnici/983498356` s tijelom zahtjeva:

```
{
    "email123": "sssssanja123gmail.com"
}
```

Dobivamo odgovarajuću grešku i za to:

```
{
    "errors": [
        {
            "type": "field",
            "msg": "Invalid value",
            "path": "email",
            "location": "body"
        }
    ]
}
```

Ako pošaljemo prazno tijelo zahtjeva, također ćemo dobiti grešku u tijelu odgovora.

Navedeno je korisno budući da možemo sve greške proslijediti natrag klijentu u jednom odgovoru, umjesto da se greške obrađuju pojedinačno.

2.4 Validacijski lanac

U `express-validator` biblioteci ima **mnoštvo validatora**, a nudi i mogućnost **kombiniranja više validatora u jedan lanac provjere** (*eng. Validation Chain*), koji se izvršava redom, definiranjem lanca metoda.

- bez obzira što postoji lanac provjera, ovdje se radi o jednoj *middleware* funkciji

Primjerice: želimo osim ispravnosti emaila provjeriti i sadrži li email nastavak `@unipu.hr`.

➡️ Isto možemo postići kombinacijom validatora `isEmail()` i `contains()`

```
// routes/korisnici.js

router.patch('/:id', [pretragaKorisnika, body('email').isEmail().contains('@unipu.hr')],
async (req, res) => {
    const errors = validationResult(req);
    // ako nema grešaka
    if (errors.isEmpty()) {
        req.korisnik.email = req.body.email; // ažuriramo email
        console.log(korisnici);
        return res.status(200).json(req.korisnik);
    }
    return res.status(400).json({ errors: errors.array() });
});
```

Na svaki validator možemo dodati i poruku koja će se prikazati u slučaju greške:

➡️ **Poruku definiramo metodom `withMessage()`**:

```
body('email').isEmail().withMessage('Email adresa nije
ispravna').contains('@unipu.hr').withMessage('Email adresa mora biti s @unipu.hr');
```

2.4.1 Validacija emaila

➡️ **Koristimo `isEmail()` validator**:

```
body('email').isEmail().withMessage('Molimo upišite ispravnu email adresu');
```

2.4.2 Provjera minimalne/maksimalne duljine lozinke

➡️ **Koristimo `isLength()` validator**:

Minimalnu duljinu navodimo kao argument metode, slično kao kod MongoDB upita:

```
body('password').isLength({ min: 6 }).withMessage('Lozinka mora imati minimalno 6 znakova');

// ili

body('password').isLength({ min: 6, max: 20 }).withMessage('Lozinka mora imati između 6 i 20 znakova');
```

2.4.3 Provjera sadržaja

➡️ **`isAlphanumeric()` validator provjerava sadrži li vrijednost samo slova i brojeve**:

```
body('username').isAlphanumeric().withMessage('Korisničko ime mora sadržavati samo slova i brojeve');
```

➡️ **`isAlpha()` validator provjerava sadrži li vrijednost samo slova**:

```
body('name').isAlpha().withMessage('Ime mora sadržavati samo slova');
```

2.4.4 Min/Max vrijednosti

➡️ **Koristimo `isInt()` validator za provjeru je li vrijednost tipa integer**, opcionalno možemo definirati raspon kao i kod `isLength()`:

```
body('age').isInt({ min: 18, max: 99 }).withMessage('Dob mora biti između 18 i 99 godina');
```

➡️ **Koristimo `isFloat()` validator za provjeru je li vrijednost tipa float**:

```
body('price').isFloat({ min: 0 }).withMessage('Cijena mora biti pozitivan broj');
```

2.4.5 Provjera je li vrijednost Boolean

➡️ **Koristimo `isBoolean()` validator**:

```
body('active').isBoolean().withMessage('Aktivnost mora biti tipa boolean');
```

2.4.6 Provjera specifičnih vrijednosti

➡️ **Koristimo `isIn()` validator za provjeru je li vrijednost sadržana u nekom nizu**:

```
body('role').isIn(['admin', 'user']).withMessage('Uloga mora biti admin ili user');
```

2.4.7 Složena provjera lozinke regularnim izrazom

➡️ **Koristimo `matches()` validator**:

- pišemo **regularni izraz** koji definira pravila za lozinku
- npr. lozinka mora sadržavati barem jedno slovo i jedan broj, duljine minimalno 8 znakova

```
body('password')
    .matches(/^(?=.*[A-Za-z])(?=.*\d)[A-Za-z\d]{8,}$/)
    .withMessage('Lozinka mora sadržavati barem jedno slovo i jedan broj');
```

2.4.8 Grananje lanca provjere

➡️ Možemo koristiti i `check()` validator koja će pretražiti parametar definiran prema nazivu na svim mogućim lokacijama u HTTP zahtjevu, uključujući:

- u **tijelu zahtjeva** (`req.body`)
- u ***query*** parametrima (`req.query`)
- u ***route*** parametrima (`req.params`)
- u **zaglavljima** (`req.headers`)
- u **kolačićima** (`req.cookies`)

Ako se naziv parametra ponavlja na više mjesta, npr. parametar `password` postoji i u tijelu zahtjeva i u *query* parametrima (naravno nije dobra praksa), `check()` će svejedno odraditi validaciju za sve te vrijednosti.

Primjer validacijskog grananja za registraciju korisnika gdje želimo provjeriti sljedeće:

- korisnik obavezno mora unijeti ime
- korisnik obavezno mora unijeti ispravnu email adresu
- lozinka mora imati minimalno 6 znakova
- potvrda lozinke mora biti jednaka lozinki

```
const { check, validationResult } = require('express-validator');

app.post(
    '/register',

    [
        // ne navodimo lokaciju jer će check() pretražiti sve parametre
        check('name').notEmpty().withMessage('Ime je obavezno'), // zaseban middleware (1)
        check('email').isEmail().withMessage('Email je u krivom formatu'), // zaseban
middleware (2)
        check('password').isLength({ min: 6 }).withMessage('Lozinka mora imati barem 6
znakova'), // zaseban middleware (3)
        check('confirmPassword') //zaseban middleware (4)
            .custom((value, { req }) => value === req.body.password)
            .withMessage('Lozinke se ne podudaraju!')
    ],

    (req, res) => {
        // callback funkcija
```

```
        const errors = validationResult(req);
        // >>> implementacija registracije ovdje (radit ćemo na WA7) <<<
        // ako nema pogrešaka:
        if (!errors.isEmpty()) {
            return res.status(400).json({ errors: errors.array() });
        }
        res.send('Registracija uspješna!');
    }
);
```

> **Napomena**: U primjeru iznad imamo 4 *middleware* funkcije (bez obzira što imaju ulančane metode). Ukupno je 4 *middlewarea* jer polje gdje se definiraju ima ukupno 4 elementa.

2.4.9 Obrada polja u tijelu zahtjeva

Što ako klijent proslijedi **polje elemenata** u tijelu zahtjeva?

- Stvari ostaju iste! `express-validator` će provjeriti svaki element polja 🚀

Na primjer: klijent pošalje zahtjev s nekim `ID`-evima:

```
{
    "ids": [5, 4, 11, 4, 123]
}
```

Validacija provjerava je li svaki element polja `ids` tipa integer:

```
body('ids').isInt().withMessage('Svaki element polja mora biti tipa integer');
```

Međutim, `express-validator` će sve **dolazne podatke tretirati kao stringove**, samim time, ako proslijedimo string `"123"`, validacija će proći.

- Proslijedimo li niz `[5, 4, 11, 4, "abc"]`, validacija **neće proći**.

2.5 Često korišteni validatori

Validator	Sintaksa	Primjer
Obavezno polje	`notEmpty()`	`check('ime').notEmpty().withMessage('Ime je obavezno')`
Je prazno	`isEmpty()`	`check('kljuc').isEmpty().withMessage('kljuc mora biti prazan')`
Ključ postoji	`exists()`	`check('kljuc').exists().withMessage('kljuc mora postojati')`
Validacija emaila	`isEmail()`	`check('email').isEmail().withMessage('Pogrešan email format')`
Min. duljina	`isLength({ min: X })`	`check('password').isLength({ min: 6 }).withMessage('Mora biti minimalno 6 znakova')`
Max. duljina	`isLength({ max: X })`	`check('username').isLength({ max: 12 }).withMessage('Mora biti maksimalno 12 znakova')`
Alfanumerički znak	`isAlphanumeric()`	`check('username').isAlphanumeric().withMessage('Samo slova i brojevi!')`
Točna duljina	`isLength({ min: X, max: X })`	`check('zip').isLength({ min: 5, max: 5 }).withMessage('Mora biti točno 5 znakova')`
Jednako	`equals('vrijednost')`	`check('role').equals('admin').withMessage('Mora biti admin')`
Min/Max vrijednost	`isInt({ min: X, max: Y })`	`check('age').isInt({ min: 18, max: 65 }).withMessage('Samo vrijednosti između 18 i 65')`
Integer check	`isInt()`	`check('age').isInt().withMessage('Mora biti integer')`
Decimal check	`isDecimal()`	`check('price').isDecimal().withMessage('Mora biti decimalni broj')`
Boolean check	`isBoolean()`	`check('isActive').isBoolean().withMessage('Mora biti Boolean vrijednost')`
String check	`isString()`	`check('name').isString().withMessage('Mora biti string')`
Inclusion	`isIn(['a', 'b'])`	`check('role').isIn(['admin', 'user']).withMessage('Kriva uloga!')`
Sadržavanje podskupa	`contains('nešto')`	`check('username').contains('admin').withMessage('Mora sadržavati admin')`
Exclusion	`not().isIn(['a', 'b'])`	`check('username').not().isIn(['root', 'admin'])`
Custom Regex	`matches(/regex/)`	`check('username').matches(/^[a-zA-Z]+$/).withMessage('Dozvoljena samo velika i mala slova')`
Validacija URL-a	`isURL()`	`check('website').isURL().withMessage('Pogrešan URL!')`
Validacija kreditne kartice	`isCreditCard()`	`check('card').isCreditCard().withMessage('Pogrešan broj kreditne kartice')`
Validacija IBAN-a	`isIBAN()`	`check('iban').isIBAN().withMessage('Pogrešan IBAN')`
ISO Date	`isISO8601()`	`check('date').isISO8601().withMessage('Netočan format datuma')`
Custom Validator	`custom(fn)`	`check('field').custom(value => value > 0).withMessage('Vrijednost mora biti pozitivna')`
Poklapanje lozinke	`custom((value, { req }))`	`check('confirm').custom((lozinka, { req }) => lozinka === req.body.proslijedena_lozinka)`
Trim	`trim()`	`check('username').trim().notEmpty().withMessage('Polje je obavezno!')`
Array check	`isArray()`	`check('roles').isArray().withMessage('Mora biti polje')`
Object check	`isObject()`	`check('user').isObject().withMessage('Mora biti objekt')`

> Sve validatore `express-validator` biblioteke možete pronaći na službenoj dokumentaciji. Naravno, nije ih potrebno sve znati napamet, već ove koji se najčešće koriste kao što su: `notEmpty()`, `isEmail()`, `isLength()`, `isAlphanumeric()`, `isInt()`, `matches()` i `custom()`.

2.6 Sanitizacija podataka

Sanitizacija podataka (*engl. data sanitization*) predstavlja postupak obrade i pročišćavanja ulaznih podataka kojim se oni uklanjanjem neispravnih, nepoželjnih ili potencijalno opasnih elemenata dovode u sigurno i pouzdano stanje za daljnju obradu.

Proces sanitizacije mora biti izveden prije same validacije podataka.

Također, sanitizacija može uključivati i **transformaciju podataka u odgovarajući format** koji je prihvatljiv i siguran za aplikaciju.

> **Napomena:** Ovdje se ne radi o sanitizaciji podataka u značenju trajnog i ireverzibilnog uklanjanja podataka s fizičkih medija za pohranu, poput HDD-a, SSD-a ili USB memorije. Iako se termin u praksi koristi i u tom kontekstu, u ovom slučaju odnosi se isključivo na sanitizaciju podataka unutar poslužiteljskog okruženja web aplikacije.

- *Primjerice*: ako korisnik unese email adresu s velikim slovima, možemo ju pretvoriti u mala slova prije nego krenemo s validacijom

`express-validator` biblioteka nudi **niz *middlewarea*** koji se koriste na isti način kao i validatori.

➡️ **Pretvorba email adrese u mala slova** korištenjem `normalizeEmail()` *middlewarea*:

```
body('email').normalizeEmail(all_lowercase: true);

// npr. email: 'Sanja.sanjic@Gmail.com' -> 'sanja.sanjic@gmail.com'
```

➡️ **Uklanjanje praznih znakova** s početka i kraja stringa koristeći `trim()` *middleware*:

```
body('username').trim();

// npr. '  Sanja  ' -> 'Sanja'
```

➡️ **Pretvorba stringa** u broj koristeći `toInt()` *middleware*:

```
body('age').toInt();

// npr. '25' -> 25
```

➡️ **Brisanje znakova koji nisu definirani** u `whitelist` parametru koristeći `whitelist()` *middleware*:

```
body('username').whitelist('abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ1234567890
');

// npr. 'Sanja123!' -> 'Sanja123'
```

➡️ **Brisanje znakova koji su definirani** u `blacklist` parametru koristeći `blacklist()` *middleware*:

```
body('username').blacklist('!@#$%^&*()_+');

// npr. 'Sanja123!$$$' -> 'Sanja123'
```

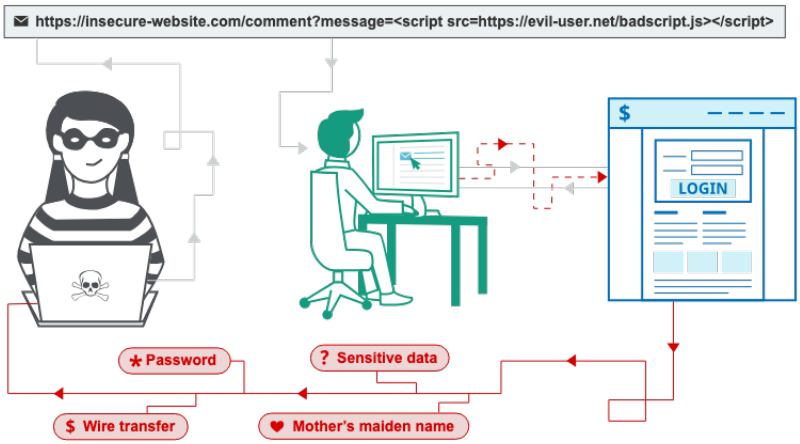
2.7 Sprječavanje reflektiranog XSS napada

XSS (*eng. Cross-Site Scripting*) napadi su vrlo česti i opasni. Postoji više kategorija XSS napada, a jedan od najčešćih je **reflektirani XSS napad** (*eng. Reflected XSS attack*).

Napad izgleda ovako:

- korisnik šalje HTTP zahtjev na poslužitelj s malicioznim JavaScript kodom, najčešće u URL-u
- maliciozni kod, najčešće obuhvaćen u HTML `<script>` tagu, izvršava se na korisničkoj strani
- u usporedbi sa **pohranjenim XSS napadom** (*eng. Stored XSS attack*), reflektirani XSS napad je **jednokratan** i **ne ostavlja tragove u bazi podataka niti na poslužitelju**

Radi se o čestom hakerskom napadu koji iskorištava ranjivosti u web aplikacijama koje ne provjeravaju i ne sanitiziraju ulazne podatke na poslužiteljskoj strani.



> Slika 3: Ilustracija reflektiranog XSS napada - korisnik šalje maliciozni kod u URL-u koji se izvršava na klijentskoj strani.

Uzet ćemo za primjer našu rutu `GET /hello` koja očekuje *query* parametar `ime`.

```
// index.js

app.get('/hello', [query('ime').notEmpty()], (req, res) => {
    const errors = validationResult(req);
    if (!errors.isEmpty()) {
        return res.status(400).json({ errors: errors.array() });
    }
    res.send('Hello, ' + req.query.ime);
});
```

Ako pošaljemo zahtjev: `GET http://localhost:3000/hello?ime=Pero`, dobit ćemo odgovor `"Hello, Pero"`.

- Ako pošaljemo prazan zahtjev, dobit ćemo grešku jer smo to pokrili s `notEmpty()` validatorom.

Možemo nadograditi rutu tako da još sanitiziramo *query* parametar koristeći `trim()` *middleware* kako bi uklonili prazne znakove s početka i kraja stringa te možemo provjeriti je li korisnik poslao samo slova koristeći `isAlpha()` validator.

Sljedeći primjer ima samo 1 *middleware*, međutim možemo ih odvojiti i u zasebne *middleware* funkcije:

```
// 1 middleware
app.get('/hello', [query('ime').notEmpty().trim().isAlpha()], (req, res) => {
    const errors = validationResult(req);
    if (!errors.isEmpty()) {
        return res.status(400).json({ errors: errors.array() });
    }
    res.send('Hello, ' + req.query.ime);
});
```

```
// 3 middlewarea
app.get('/hello', [query('ime').notEmpty(), query('ime').trim(), query('ime').isAlpha()],
(req, res) => {
    const errors = validationResult(req);
    if (!errors.isEmpty()) {
        return res.status(400).json({ errors: errors.array() });
    }
    res.send('Hello, ' + req.query.ime);
});
```

Možemo dodati i odgovarajuće poruke za greške:

```
// index.js

app.get('/hello', [query('ime').notEmpty().withMessage('Ime je obavezno'),
query('ime').trim(), query('ime').isAlpha().withMessage('Ime mora sadržavati samo
slova')], (req, res) => {
    const errors = validationResult(req);
    if (!errors.isEmpty()) {
        return res.status(400).json({ errors: errors.array() });
    }
    res.send('Hello, ' + req.query.ime);
});
```

Međutim, što da nemamo provjeru `isAlpha()` i korisnik pošalje maliciozni kod u *query* parametru? Prisjetite se, *query parametri* nisu obavezni dio definicije rute, stoga ih klijenti mogu slati kako žele, koliko žele i u bilo kojem obliku.

- ***Banalni primjer*: Maliciozni korisnik pošalje HTML skriptni tag u *query* parametru** koji sadrži `alert('Hakirani ste! Molimo da pošaljete novac na adresu...')`:

> Zašto je primjer banalan? Zato što se radi o vrlo jednostavnom XSS napadu koji koristi osnovne HTML i JavaScript elemente kako bi demonstrirao koncept XSS napada. U stvarnim scenarijima, napadi mogu biti mnogo sofisticiraniji gdje korisnik umjesto običnog `alert` prozora može pokušati ukrasti kolačiće ili JWT tokene, preusmjeriti zahtjev korisnika na drugi zlonamjerni poslužitelj ili čak izvršiti zlonamjerni

> kod na klijentskoj strani.

Primjer takvog HTTP zahtjeva izgledao bi ovako:

```
GET http://localhost:3000/hello?ime=<script>alert('Hakirani ste! Molimo da pošaljete novac na adresu...')</script>
```

Ako maknete `isAlpha()` validator, dobit ćete odgovor s "malicioznim kodom", odnosno **skripta će se izvršiti na korisničkoj strani**:

```
// index.js

app.get('/hello', [query('ime').notEmpty().withMessage('Ime je obavezno'),
query('ime').trim()], (req, res) => {
    const errors = validationResult(req);
    if (!errors.isEmpty()) {
        return res.status(400).json({ errors: errors.array() });
    }
    res.send('Hello, ' + req.query.ime);
});
```

Ako pošaljete GET zahtjev u web pregledniku, dobit ćete `alert` poruku.

> Slika 4: Primjer reflektiranog XSS napada - maliciozni kod `<script>alert...</script>` izvršava se na klijentskoj strani.

➡️ Jedan od *middlewarea* koji se može koristiti za sprječavanje reflektiranog XSS napada je **`escape()`** **sanitizacijski *middleware***:

```
query('ime').escape();
```

Ovaj *middleware* će zamijeniti HTML znakove, npr. `<`, `>`, `&`, `'`, `"` s njihovim kodiranim ekvivalentima `<`, `>`, `&`, `&#39;`, `&quot;`.

```
// index.js

app.get('/hello', [query('ime').notEmpty().withMessage('Ime je obavezno'),
query('ime').trim(), query('ime').escape()], (req, res) => {
    const errors = validationResult(req);
    if (!errors.isEmpty()) {
        return res.status(400).json({ errors: errors.array() });
    }
    res.send('Hello, ' + req.query.ime);
});
```

Primjer odgovora (neće se izvršiti skripta i XSS napad je spriječen):

```
Hello, &lt;script&gt;alert(&#x27;Hakirani ste! Molimo da pošaljete novac na
adresu...&#x27;)&lt;&#x2F;script&gt;
```

Samostalni zadatak za Vježbu 6

Izradite novi poslužitelj `movie-server` na proizvoljnom portu te implementirajte sljedeće rute:

1. `GET /movies` - vraća listu filmova u JSON formatu
2. `GET /movies/:id` - vraća podatke o filmu s određenim `id`-om
3. `POST /movies` - dodaje novi film u listu filmova (*in-memory*)
4. `PATCH /movies/:id` - ažurira podatke o filmu s određenim `id`-om
5. `GET /actors` - vraća listu glumaca u JSON formatu
6. `GET /actors/:id` - vraća podatke o glumcu s određenim `id`-om
7. `POST /actors` - dodaje novog glumca u listu glumaca (*in-memory*)
8. `PATCH /actors/:id` - ažurira podatke o glumcu s određenim `id`-om

Podaci za filmove:

```
[
    {
        "id": 4222334,
        "title": "The Shawshank Redemption",
        "year": 1994,
        "genre": "Drama",
        "director": "Frank Darabont"
    },
    {
        "id": 5211223,
        "title": "The Godfather",
        "year": 1972,
        "genre": "Crime",
        "director": "Francis Ford Coppola"
    },
    {
        "id": 4123123,
        "title": "The Dark Knight",
        "year": 2008,
        "genre": "Action",
        "director": "Christopher Nolan"
    }
]
```

Podaci za glumce:

```
[
    {
        "id": 123,
        "name": "Morgan Freeman",
        "birthYear": 1937,
```

```
        "movies": [4222334]
    },
    {
        "id": 234,
        "name": "Marlon Brando",
        "birthYear": 1924,
        "movies": [5211223]
    },
    {
        "id": 345,
        "name": "Al Pacino",
        "birthYear": 1940,
        "movies": [5211223]
    }
]
```

Implementirajte *middleware* koji će se upotrebljavati za pretraživanje filmova i glumaca po `id`-u. Kada korisnik pošalje zahtjev na rutu koja ima route parametar `id` na resursu `/movies`, *middleware* će provjeriti postoji li taj film u listi filmova. Napravite isto i za glumce, dodatnim *middlewareom*. Odvojite rute u zasebne router instance, a implementacije *middlewareova* prebacite u zasebne datoteke unutar `middleware` direktorija.

Dodajte novi *middleware* na **razini cijele Express aplikacije** koji će logirati svaki dolazni zahtjev na konzolu u sljedećm formatu:

```
[naziv_poslužitelja] [trenutni_datum_i_vrijeme] HTTP_metoda URL_zahtjeva
```

Primjer:

```
[movie-server] [2024-06-01 12:00:00] GET /movies
```

Za svaki zahtjev morate logirati:

- naziv aplikacije
- trenutni datum i vrijeme
- HTTP metodu zahtjeva
- URL zahtjeva

Instalirajte `express-validator` biblioteku te implementirajte sljedeće validacije za odgovarajuće rute:

- `POST /movies` - validirajte jesu li poslani `title`, `year`, `genre` i `director`
- `PATCH /movies/:id` - validirajte jesu li poslani `title`, `year`, `genre` ili `director`
- `POST /actors` - validirajte jesu li poslani `name` i `birthYear`
- `PATCH /actors/:id` - validirajte jesu li poslani `name` ili `birthYear`
- `GET /movies/:id` - validirajte je li `id` tipa integer
- `GET /actors/:id` - validirajte je li `id` tipa integer

- `GET /movies` - dodajte 2 *query* parametra `min_year` i `max_year` te validirajte jesu li oba tipa integer. Ako su poslani, provjerite jesu li `min_year` i `max_year` u ispravnom rasponu (npr. `min_year < max_year`). Ako je poslan samo jedan parametar, provjerite je li tipa integer.
- `GET /actors` - dodajte *route* parametar `name` te provjerite je li tipa string. Uklonite prazne znakove s početka i kraja stringa koristeći odgovarajući *middleware*.

Obradite greške za svaku rutu slanjem gotovog objekta s detaljima o greškama koju generira `express-validator` biblioteka.

Osigurajte sve rute od reflektiranog XSS napada koristeći odgovarjauće validatore iz `express-validator` biblioteke.

U prilogu zadaće obavezno uključiti poveznicu na javno Postman okruženje s testovima za definirane rute.