Express.js tutorial

Instalacija

Unutar postojećeg NODE.JS projekta (projekt nad kojim smo pokrenuli **npm init** koji nam je kreirao ***package.json*** datoteku) otvorimo terminal i pokrenemo komandu **npm install express**

```
PS C:\Users\dnize\OneDrive - VERN University\__dokumenti\IoT-PzI2\Express.js-1> npm install express

up to date, audited 63 packages in 1s

11 packages are looking for funding
  run `npm fund` for details

found 0 vulnerabilities
npm notice
npm notice New patch version of npm available! 10.2.3 -> 10.2.4
npm notice Changelog: https://github.com/npm/cli/releases/tag/v10.2.4
npm notice Run npm install -g npm@10.2.4 to update!
npm notice
PS C:\Users\dnize\OneDrive - VERN University\__dokumenti\IoT-PzI2\Express.js-1>
```

Hello world web server

Korištenjem express.js možemo u svega nekoliko linija koda napisati i pokrenuti web server.

Za početak trebamo „učitati“ express modul (onaj koji smo u prethodnom poglavlju imstalirali/dodali kao modul u naš projekt)

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

Kreiramo instancu naše web aplikacije

```
const app=express()
```

definiramo port na kojem će web server primati zahtjeve (requests)

```
const port=3000;
```

Definiramo funkciju koja će biti pozvana kada neki posjetite pristupi našem webu (root adresa je /)

```
app.get("/",(request,response, next)=>{
    response.send("Hello world")
})
```

I konačno, pokrenemo aplikaciju, web server

```
app.listen(port, () => {
    console.log(`Example app listening on port ${port}`)
})
```

U konačnici, cijeli program izgleda ovako:

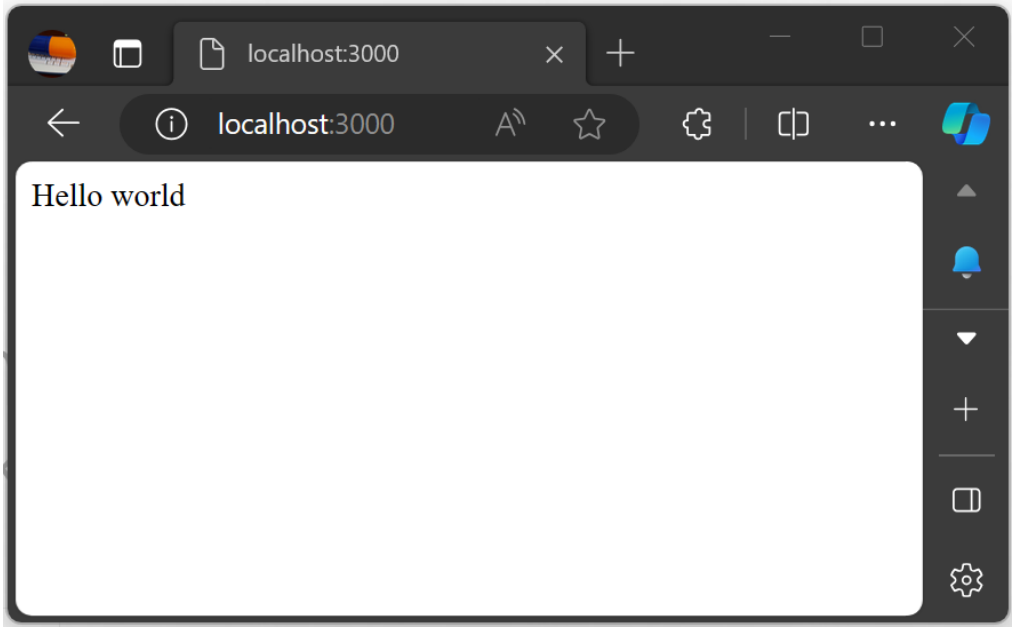
```
const express = require("express")
const app=express()
const port=3000;

app.get("/",(request,response, next)=>{
    response.send("Hello world")
})

app.listen(port, () => {
    console.log(`Example app listening on port ${port}`)
})
```

Sada možemo uterminalu pokrenuti program koji smo upravo napisali

```
PS C:\Users\dnize\OneDrive - VERN University\__dokumenti\IoT-PzI2\Express.js-1> node .\index.js
Example app listening on port 3000
```

Output koji nam se prikazuje u terminalu nam govori o tome da naša aplikacija sluša, očekuje zahtjeve na portu 3000 (defaultni port za web protokol http je 80)

Da bismo pristupili stranici otvaramo browser i ukucavamo adresu
http://localhost:3000/



Output koji vidimo („Hello world“) dolazi iz našeg programa

```
app.get("/",(request,response, next)=>{
    response.send("Hello world")
})
```

app.get dio je definirao da će funkcija biti pozvana ako je zahtjev tipa GET (druge moguće često korištene metode su post, put, delete)

„/“ je definirao da će funkcija biti pozvana kada je korisnik zatražio indexnu (root stranicu) weba.

A

```
response.send("Hello world")
```

je odredio što vraćamo kao sadržaj zatražene web stranice. U ovom slučaju radi se o običnom tekstu

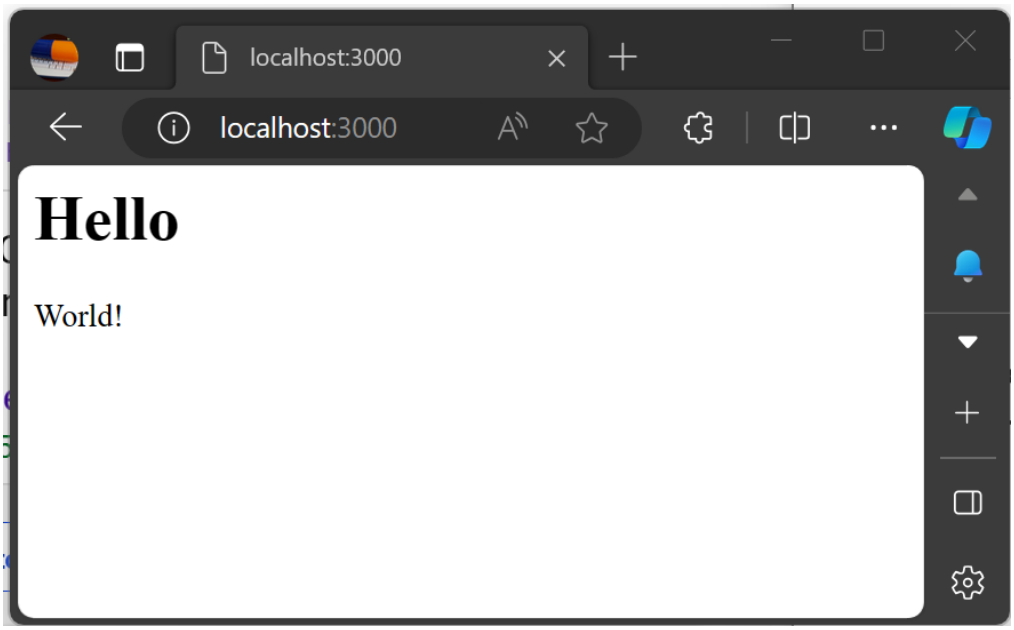
Pokušajmo nešto mrvicu naprednije,

Kao odgovor na zahtjev ćemo vratiti html pa prvo postavljamo content type na text/html a potom šaljemo text koji sadrži html formatiranje.

```
app.get("/",(request,response, next)=>{
    response.set('content-type', 'text/html');
    response.send("<html><body><h1>Hello</h1>World!</body></html>")
})
```

Restartajte program i potom refreshajte stranicu

Naša stranica je sada protumačena kao html

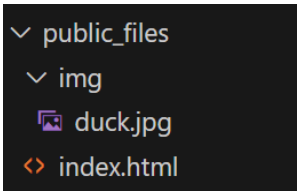


Serviranje statičkih stranica

Da bismo sadržaj nekog direktorija posluživali kao statičke stranice u našem projektu ćemo kreirati direktorij naziva **public_files** a u kodu (prije app.listen linije) dodajemo

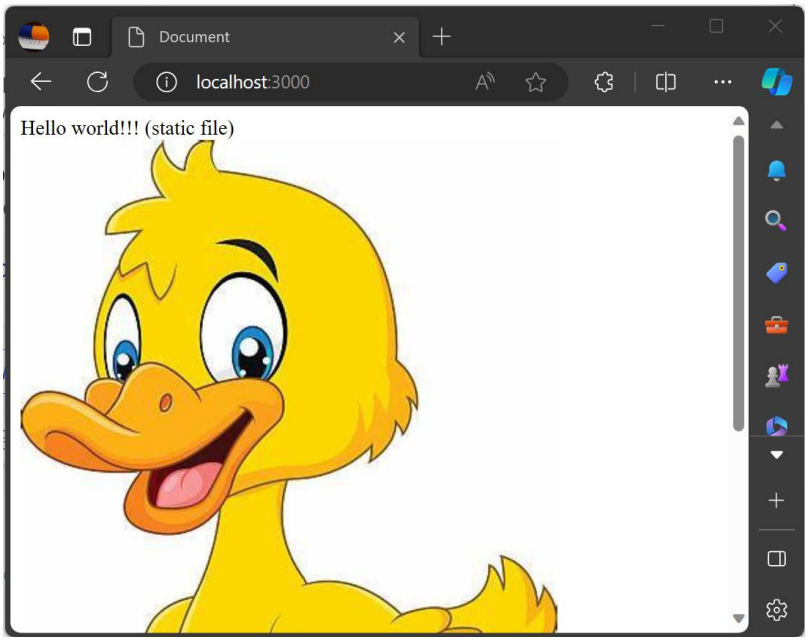
```
// serving static files
app.use(express.static('./public_files'))
```

time definiramo da će sadržaj tog direktorija biti posluživan kao statički sadržaj (statički jer je unaprijed definiran, za razliku od dinamičkog koji program generira „u letu“)



U public_file dodajemo index.html datoteku (koja će služiti kao root datoteka) te folder ***img*** u koji ćemo dmah dodati i sliku (duck.jpg)

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Document</title>
</head>
<body>
    Hello world!!! (static file)<br />
    <img src="./img/duck.jpg" />

</body>
</html>
```



Trebamo imati na umu da, ukoliko smo prvo definirali mapiranje za „/“ a tek potom statično oslućivanje datoteka, prednost će imati mapitanje koje smo prvo postavili

```
app.get("/",(request,response, next)=>{
    response.set('content-type', 'text/html');
    response.send("<html><body><h1>Hello</h1>World!</body></html>")
})
// serving static files
app.use(express.static('./public_files'))
```

ovako postavljen redoslijed će nam dati naš dinamočno generirani hello world (defniran je prvi pa ima prednost)

dok ako im zamijenimo redoslijed dobivamo sadrža statički definirane datoteke

```
// serving static files
app.use(express.static('./public_files'))

app.get("/",(request,response, next)=>{
    response.set('content-type', 'text/html');
    response.send("<html><body><h1>Hello</h1>World!</body></html>")
})
```

Metode

Metode GET, POST, PUT, DELETE predstavljaju osnovu izrade REST API-ja i web aplikacija

```
app.get("/", (request,response, next)=>{
    //...
})
app.post("/", (request,response, next)=>{
    //...
})
app.put("/", (request,response, next)=>{
    //...
})
app.delete("/", (request,response, next)=>{
    //...
})
```

Te četiri metode predstavljaju CRUD (create, read, update, delete) komande

POST nam predstavlja Create

GET predstavlja read metodu

PUT predstavlja update

DELETE briše odgovarajući zapis

Routing/Mapiranje i parametri

Routingom definiramo mapiranje, primjerice želimo mapirati da se pozivom metode GET na adresu /users korisniku vrati popis korisnika aplikacije

Da bismo to definirali zadajemo

```
app.get("/users", (request,response, next)=>{
    //...
})
```

Pa će naša funkcija biti pozvana upravo kada korisnik zatraži adresu /users korištenjem GET metode

Da bismo definirali DELETE metodu bit će potrebno primiti i neki identifikator (id) pomoću kojega možemo jedinstvano identificirati korisnika (njegov zapis) koji je potrebno obrisati

Routing nam omogućuje definiranje adrese stranice koja će sadržavati informaciju o tom identifikatoru

```
app.delete("/users/:id", (request,response, next)=>{
    //...
})
```

Pristupna adrese za ovako definirani DELETE zahtjev bi sadržavala potreban identifikator i za ova primjer bi mogla izgledati ovako (za korisnika sa id=7)

/users/7

Da bismo unutar funkcije pristupili toj vrijednosti koristimo request objekt koji sadrži sve potrebne inofrmacije o zahtjevu koji je upućen serveru

```
app.delete("/users/:id", (request,response, next)=>{
    let id=request.params.id;
    //...
})
```

Query parameters

Ukoliko su parametri proslijeđeni putem query-ja (http://localhost:3000/users?id=7 na raspolaganju nam je query objekt

```
app.get("/users", (request,response, next)=>{
    let id=request.query.id;
    //...
})
```

Form post parameters

Za parametre koje prikupimo putem html forme i proslijedimo na server kao POST trebamo prvo definirati

```
app.use(express.urlencoded());
```

Time aplikaciji unaprijed najavljujemo da može očekivati i tako proslijeđene podatke *(tu liniju treba dodati prije nego kažemo **app.listen**)*a potom možemo podatke forme dohvatiti

```
app.post("/", (request,response, next)=>{
    let ime=request.body.name
    let prezime=request.body.lastname
    //...
})
```

JSON podatci

Za primanje podataka u JSON obliku, najavljujemo košištenje JSON parsera

```
app.use(express.json())
```

A potom možemo u samoj metodi pristupiti podatcima na sljedeći način

```
app.post("/", (request,response, next)=>{
    let data=request.body
    let ime=data.name
    let prezime=data.lastname
    //...
})
```

Routing i modularnost

U slučaju da radimo nešto opsežniju aplikaciju, preporučljivo je aplikaciju razdijeliti u module (umjesto da nam sve mapiranje bude upisano u jednoj datoteci u kojoj će neizbježno nastati zbrka)

Na taj način u našoj index datoteci definiramo samo osnove aplikacije

```
const express = require("express")
const app=express()
const port=3000;

// serving static files
app.use(express.static('./public_files'))

const users = require('./api/users.js')
app.use("/users",users);

const books = require('./api/books.js')
app.use("/books",books);

//...
//...

// starting web server service (and listening for requests on given port)
app.listen(port, () => {
    console.log(`Example app listening on port ${port}`)
})
```

Potom kreiramo folder ***api*** u kojem za svaku cjelinu (users, books, ...) definamo jednu js datoteku u kojoj je sadržana kompletna funkcionalnost koja se bavi tim segmentom aplikacije

Mogli bismo reći da je svaka cjelina aplikacija za sebe pa tako učitavanjem users.js definiramo da će taj modul „živjeti“ u /users domeni

```
const users = require('./api/users.js')
app.use("/users",users);
```

Pa bi tako users.js datoteka (koja se brine o administraciji korisnika) definira routing koji se nadovezuje na routing zadan od strane aplikacije (može biti potpuno nesvjestan toga je je za njega definiran routing /users, jednako tako moga je biti definiran i /korisnici)

Naša users miniaplikacija definira svoj routing (unutar onoj koji joj je dodjeljen od aplikacije)

```
const express = require("express")
const router = express.Router()
let users=[
    {ime:"Dean",prezime:"Nižetić"},
    {ime:"Marko",prezime:"Marić"},
    {ime:"Jure",prezime:"Jurić"},
    {ime:"Ana",prezime:"Ankić"},
]
router.use(express.json())
// READ
router.get("/", (req,res)=>{
    res.send(users);
})
// CREATE
router.post("/", (req,res)=>{
    let dataReceived=req.body;
    users.push(dataReceived)
})
//DELETE
router.delete("/:id",(req,res)=>{
    let id=Number(req.params.id);
    if (users.length>id && id>=0){
        users.splice(id, 1);
    }
})
//UPDATE
router.put("/:id",(req,res)=>{
    let id=Number(req.params.id);
    let dataReceived=req.body;
    if (users.length>id && id>=0){
        users.splice(id, 1, dataReceived);
    }
})
module.exports = router
```

pa je tako stvari routing za delete metodu /users/:id (dio koji je definiran od strane aplikacije plus dio koji je definiran unutar samog users modula)