# Okvir za razvoj poslužiteljskih aplikacija u TypeScript jeziku

**Sabo, Mario**

Sveučilište Josipa Jurja Strossmayera u Osijeku

Odjel za matematiku

Sveučilišni diplomski studij matematike
smjer: Matematika i računarstvo

# Okvir za razvoj poslužiteljskih aplikacija u TypeScript jeziku

Diplomski rad

Mentor:

**izv. prof. dr. sc.**
**Domagoj Matijević**

Kandidat:

**Mario Sabo**

Osijek, 2022

# Sadržaj

# 1 | Uvod

Dugi niz godina web aplikacije implementirane su tako da su im klijent i poslužitelj razvijeni kao jedna aplikacija. U moderno vrijeme, a posebno nakon popularizacije pametnih telefona, pojavila se potreba da za isti poslužitelj imamo više različitih klijentskih aplikacija. Iz tog razloga sve je popularnije web aplikacije razvijati na način da su klijent i poslužitelj potpuno različite aplikacije, često implementirane u različitim tehnologijama. Na taj način jednom razvijenu poslužiteljsku aplikaciju možemo koristiti u proizvoljnom broju klijentskih aplikacija koje će tu poslužiteljsku aplikaciju "konzumirati" na različite načine.

Razne aplikacije često obavljaju slične radnje, a i u istim aplikacijama puno toga se obavlja na sličan način. Iz tog razloga koriste se takozvani "okviri za razvoj" koji većinu koda koji se ponavlja te određenog tehnički zahtjevnog koda generiraju u pozadini, a programerima pružaju prijateljsko sučelje i na taj način uvelike ubrzavaju i olakšavaju razvoj aplikacija. U ovom radu ćemo implementirati jedan okvir za razvoj poslužiteljskih aplikacija, a za implementaciju ćemo koristiti TypeScript jezik.

Motiviran nedostatkom strukture u JavaScript jeziku, koja je u većini slučajeva prouzročena nedostatkom snažnog tipiziranja, Microsoft je odlučio razviti programski jezik TypeScript kojeg je prvi puta predstavio 2012. godine. TypeScript je proširenje JavaScripta kojem je osnovna svrha omogućavanje snažnog tipiziranja prilikom razvoja JavaScript aplikacija. On je također pravi objektno orijentirani jezik što nam omogućuje da aplikacije razvijamo na strukturiran i održiv način. Iako je jezik relativno mlad, brzo je stekao veliku popularnost i koristi u velikom broju aplikacija. S vremenom TypeScript se razvio i danas pruža mnoge napredne značajke koje ćemo mi u ovom radu iskoristiti za implementaciju okvira.

Kroz rad ćemo se prvo u poglavljima 2-5 upoznati s tehnologijama i pojmovima potrebnima za implementaciju okvira kako bismo s razumijevanjem mogli proći kroz nju u poglavlju 6.

U poglavlju 2 upoznajemo se s HTTP protokolom i nekim njegovim specifičnostima.

Nakon toga ćemo u poglavlju 3 ukratko opisati JavaScript jezik te Node.js platformu.

U poglavlju 4 upoznat ćemo se s TypeScript jezikom, njegovim prevoditeljem i naprednim značajkama poput dekoratora i refleksije.

Prije implementacije, kroz poglavlje 5 ćemo se još upoznat s Express.js okvirom na kojem će naša implementacija biti bazirana.

Cijela i objedinjena implementacija ovog okvira može se pronaći na `https://github.com/msabo1/expressts`.

# 2 | Hypertext Transfer Protocol

*Hypertext Transfer Protocol* (HTTP) je protokol koji se koristi pri komunikaciji na *World Wide Web* mreži. Baziran je na klijent/poslužitelj arhitekturi, odnosno on definira komunikaciju između čvorova koji šalju zahtjeve (klijenti) i čvorova koji te zahtjeve obrađuju te šalju odgovore (poslužitelji).

## 2.1 Resursi

Ono što poslužitelj vrati nakon uspješno obrađenog zahtjeva nazivamo resurs. Klijenti resursima pristupaju pomoću URI-ja (Uniform Resource Identifier) koji jedinstveno označava taj resurs. URI zapravo predstavlja internetsku adresu na kojoj se taj resurs nalazi. Na primjer *www.primjer.com/resurs* je jedan URI koji jedinstveno označava resurs *resurs* na poslužitelju *www.primjer.com*.

Resursi mogu biti statičke datoteke (npr. slike, videa, HTML datoteke i slično) odnosno one datoteke koje su generirane prije nego je zahtjev upućen, ali oni mogu biti i dinamički generirani u ovisnosti o parametrima zahtjeva.

## 2.2 HTTP metode

Osim što nam omogućuje da dohvaćamo resurse, HTTP nam pruža mogućnost da njima i upravljamo pa ih tako možemo i kreirati, ažurirati, brisati i slično.

Kako bi specificirali što želimo raditi s resursom koristi se HTTP metoda. Ona je dio svakog HTTP zahtjeva i nalazi se na početku HTTP poruke koju klijent šalje. HTTP podržava mnoge akcije s resursima te za svaku akciju postoji definirana HTTP metoda, one koje se najčešće koriste su:

- GET - koristi se za dohvaćanje resursa
- POST - koristi se za kreiranje novog resursa
- PUT i PATCH - koriste se za ažuriranje postojećeg resursa
- DELETE - koristi se za brisanje resursa

## 2.3 HTTP statusi

Kada klijent uputi zahtjev te ga poslužitelj obradi, odgovor ne mora uvijek biti ono što klijent očekuje. Poslužitelj može naići na probleme prilikom obrade jer

ima neke interne probleme, zahtjev može biti loše sastavljen i slično.

Poslužitelj će uz svaki odgovor priložiti HTTP status. To je troznamenkasti broj koji uz sebe veže poruku koja opisuje kako je zahtjev prošao. Postoji mnogo definiranih HTTP statusa, a neki od čestih su:

- 200 - uz sebe veže poruku *OK* i označava da je sve prošlo u redu
- 201 - nosi poruku *CREATED*, odnosno govori da je sve prošlo u redu te da je resurs uspješno kreiran
- 404 - poruka *NOT FOUND*, a označava da resurs nije pronađen na poslužitelju

# 3 | JavaScript

JavaScript je skriptni programski jezik nastao 1995. godine iz potrebe da internetske stranice postanu interaktivne. Kreiran je u sklopu *Netscape Navigator* internetskog preglednika, a danas je dio svih modernih preglednika i bez njega ne možemo zamisliti moderan internet.

## 3.1 Tipovi podataka

U JavaScriptu postoji nekoliko tipova podataka. Sastoji se od primitivnih tipova: broj (*number*), niz znakova (*string*), logička vrijednost (*boolean*) te tipovi *null* i *undefined*. *null* se koristi kada namjerno želimo da neka varijabla nema vrijednost dok *undefined* označava da varijabla nije inicijalizirana. Osim primitivnih tipova, postoji i tip podatka *object* čijim se proširivanjem stvaraju vlastiti tipovi. JavaScript dolazi s nekoliko unaprijed definiranih objekata - polje (*array*), datum (*date*), funkcija (*function*), matematika (*math*) te regularni izraz (*regular expression*). Treba napomenuti kako JavaScript polje nije polje kakvo poznajemo u računalnoj znanosti. Naime u njega možemo pohranjivati podatke različitih tipova te mu dinamički mijenjati veličinu.

JavaScript je dinamički tipiziran jezik, što znači da se tip podatka varijabli dodjeljuje prilikom izvršavanja ovisno o vrijednosti koja je u varijabli pohranjena. Također, varijabla tijekom svog života može promijeniti tip.

## 3.2 Node.js

Node.js je platforma koja omogućuje da se JavaScript kod izvršava izvan preglednika. Za izvršavanje JavaScript-a, Node.js koristi *Google V8 engine* koji je *Google* razvio za potrebe *Google Chrome* preglednika.

Node.js je dodao razne funkcionalnosti za interakciju sa sustavom što nam omogućuje da pomoću JavaScript razvijamo sve vrste aplikacija pa tako i poslužiteljske aplikacije koje će nama biti od interesa kasnije u ovom radu.

### 3.2.1 Paketi

Jezgra Node.js-a je minimalna, odnosno sadrži samo osnovne funkcionalnosti. Ako želimo raditi nešto naprednije (npr. rad s datotekama, mrežna komunikacija i slično) funkcionalnosti moramo uključiti u jezgru pomoću Node paketa koji

se još nazivaju i ovisnostima (*dependency*).

Kako bi se upravljanje paketima olakšalo razvijen je *npm* (*Node Package Manager*). To je alat koji na lak i intuitivan način omogućuje upravljanje cijelim Node.js projektom.

Postoje paketi koji će nam služiti kako bismo aplikaciju lakše razvili, ali sama aplikacija (njeno izvršavanje) o tim paketima neće ovisiti. *npm* nam omogućuje da takve pakete instaliramo na poseban način kako bismo u produkciji na elegantan način mogli instalirati samo one pakete o kojima aplikacija stvarno ovisi. Te ćemo pakete nazivati razvojnim paketima (*development dependency*).

Osim što možemo upravljati paketima unutar Node.js projekta, pomoću *npm-a* pakete možemo instalirati globalno kako bismo ih mogli koristiti bilo kada unutar operacijskog sustava.

### 3.2.2 Node.js projekt

Kako bi upravljanje projektom pomoću *npm-a* bilo moguće, projekt je prvo potrebno inicijalizirati. To možemo učiniti tako da u korijenskom direktoriju aplikacije napravimo *package.json* datoteku. Kao što vidimo iz nastavka datoteke, radi se o *json* konfiguracijskoj datoteci iz koje će *npm* iščitati koje pakete treba instalirati (ako paketi iz nekog razloga nisu instalirani) te će ju ažurirati kada pomoću njega instaliramo neki novi paket.

Umjesto da ručno kreiramo *package.json* datoteku, preporučeni način je da u komandnoj liniji, dok se nalazimo u korijenskom direktoriju, izvršimo naredbu

```
npm init
```

koja će tu datoteku napraviti za nas, a u njoj će se već nalaziti potrebna konfiguracija kako bi aplikaciju mogli pokrenuti.

# 4 | TypeScript

TypeScript je programski jezik koji obogaćuje JavaScript nekim korisnim značajkama. TypeScript omogućuje snažno tipiziranje, odnosno svakoj se varijabli dodjeljuje tip podatka prilikom pisanja koda ili, ako je tip moguće zaključiti, automatski prilikom prevođenja. Nakon što je tip prvi puta određen, on se ne može mijenjati tijekom života varijable. On je također pravi objektno-orijentirani programski jezik što nam omogućuje da aplikaciju koju razvijamo pomoću njega napravimo na strukturiran i održiv način.

TypeScript se ne može izvršavati u preglednicima niti Node.js-u pa ga je potrebno prevesti u JavaScript. To radimo pomoću alata koji se zove TypeScript prevoditelj (*compiler*). Treba imati na umu da tipovi u TypeScriptu služe samo za statičku analizu i hvatanje grešaka prilikom prevođenja. Nakon prevođenja sve informacije o tipovima su izgubljene i ono što se izvršava je običan JavaScript. Zbog toga treba biti dodatno oprezan kada radimo s podacima koje dobivamo dinamički prilikom izvršavanja jer im JavaScript može dodijeliti tip koji mi nismo predvidjeli, a to može dovesti do grešaka u izvršavanju.

## 4.1 TypeScript prevoditelj

TypeScript prevoditelj je aplikacija koja se može instalirati kao *npm* paket. Kao što smo rekli, ono što se izvršava nakon što TypeScript aplikaciju prevedemo je običan JavaScript što znači da TypeScript prevoditelj nije potreban kako bi se aplikacija izvšavala, a to znači da ga možemo instalirati kao razvojni paket. Kako bismo to postigli, u komandnoj liniji je potrebno izvršiti naredbu

```
npm install --save-dev typescript
```

Sada TypeScript prevoditelj možemo koristiti u komandnoj liniji, dok se nalazimo u direktoriju, ili bilo kojem poddirektoriju, projekta u kojem je *typescript* paket instaliran tako da izvršimo naredbu

```
npx tsc
```

Izvršavanjem te naredbe, sve datoteke koje se nalaze u tom direktoriju, i svim poddirektorijima, a koje imaju nastavak *.ts* će se prevesti u JavaScript, a rezultirajući kod nalazit će se u datotekama istog imena kao originalne, ali s nastavkom *.js*. Također, struktura direktorija će biti očuvana.

### 4.1.1 Konfiguracija prevoditelja

Često pretpostavljeno ponašanje prevoditelja nije ono što želimo. Iz tog razloga TypeScript prevoditelj nam omogućuje da to ponašanje promijenimo.

Postoji puno konfiguracijskih postavki. Primjerice, ako ne želimo da se nakon prevođenja rezultirajuće datoteke stvore u istim direktorijima kao originalne, nego ih želimo preusmjeriti u neki drugi direktoriji možemo koristiti opciju *outDir*. Kako bi primijenili neku opciju, prevoditelju ju možemo proslijediti kao zastavicu, odnosno ako želimo primijeniti spomenutu *outDir* opciju izvršit ćemo naredbu

```
npx tsc -outDir dist
```
(4.1)

Time ćemo prevesti sve *.ts* datoteke, ali rezultirajuće datoteke će se nalaziti unutar *dist* direktorija koji će se kreirati u direktoriju iz kojeg smo naredbu izvršili i poštivat će originalnu strukturu.

Osim spomenutog, postoji još jedan način konfiguriranja prevoditelja. U korijenskom direktoriju možemo napraviti *tsconfig.json* datoteku unutar koje napravimo objekt *compilerOptions* te unutar njega definiramo opcije. Ako bi na taj način htjeli postići isto što smo postigli s 4.1, *tsconfig.json* datoteka trebala bi izgledati kao u kodu 4.1.1, nakon čega je dovoljno izvršiti naredbu

```
npx tsc
```

```
1 {
2     "compilerOptions": {
3         "outDir": "dist"
4     }
5 }
```

Kod 4.1: Primjer konfiguracije prevoditelja

Kako ne bismo morali ručnu praviti *tsconfig.json* datoteku, možemo izvršiti naredbu

```
npx tsc --init
```

koja će generirati datoteku sa svim mogućim opcijama koje će biti zapisane u obliku komentara pa ako ih želimo koristiti dovoljno je samo ukloniti oznake komentara i prilagoditi vrijednost opcije.

Prednost korištenja konfiguracijske datoteke je što će prevođenje biti konzistentno neovisno o računalu na kojem ga izvršavamo, a time eliminiramo potencijalne pogreške koje bi bilo teško otkriti.

## 4.2 Deklaracijske datoteke

*npm* paketi moraju biti objavljeni u JavaScriptu pa čak i ako su napisani u TypeScriptu, prije objave se moraju prevesti u JavaScript. To znači da ako aplikaciju

pravimo pomoću TypeScripta i želimo koristiti neki vanjski paket, nećemo imati pristup tipovima podataka objekata koje koristimo iz tog paketa.

Rješenje za taj problem su deklaracijske datoteke koje se mogu isporučivati zajedno s *npm* paketima. One imaju nastavak *.d.ts* i napisane su u posebnom deklarativnom jeziku vrlo sličnom TypeScriptu koji služi za definiranje tipova podataka u JavaScript datotekama.

Ako je paket razvijen u TypeScriptu, prevoditelj se može konfigurirati tako da uz JavaScript datoteke generira i deklaracijske datoteke. Međutim, ako je paket razvijen u JavaScriptu te ako razvojni tim nije priložio deklaracijske datoteke, moramo ih napisati sami, no postoji i mogućnost da je netko drugi te datoteke već napravio.

Na sreću, za većinu popularnih JavaScript paketa postoje definirane deklaracijske datoteke i one su objavljene pod okriljem *@types* organizacije.

Na primjer ako želimo koristiti *express* paket, koji je napisan u JavaScriptu, u TypeScript aplikaciji to možemo napraviti tako da instaliramo samo *express* paket. U tom slučaju nećemo znati ništa o tipovima podataka unutar paketa, a tada nemamo koristi od TypeScripta. Ako ipak želimo koristiti *express* paket zajedno s njegovim tipovima, potrebno je dodatno instalirati *@types/express* paket. Primijetimo kako deklaracijske datoteke zapravo nisu potrebne kako bi se prevedena aplikacija izvršavala pa *@types* pakete možemo instalirati kao razvojne.

## 4.3 Dekoratori

Dekoratori su korisna značajka u TypeScriptu koja nam omogućuje da na elegantan način promijenimo ponašanje klase i njenih članova (*members*) bez da mijenjamo njihovu definiciju te da im jednostavno pridružimo metapodatke.

Dekoratori su zapravo obične funkcije koje će biti pozvane s informacijama o deklaraciji (npr. klasa, metoda klase i slično) koju dekoriramo, a primjenjuju se tako da se navedu neposredno prije deklaracije tako da im se doda prefiks @.

```
1 @MojDekorator
2 class MojaKlasa{}
```

Kod 4.2: Primjena dekoratora na klasu

Dekoratori su u TypeScriptu eksperimentalna značajka te se način njihovog korištenja u budućnosti može promijeniti tako da ne bude kompatibilan sa sadašnjim načinom. Kako bismo toga bili svjesni, razvojni tim TypeScript jezika je odlučio da kako bi se dekoratori koristili, moramo to eksplicitno omogućiti u konfiguraciji prevoditelja. To radimo tako da uključimo opciju *experimentalDecorators*.

### 4.3.1 Dekorator klase

Dekorator klase je funkcija koja prima konstruktor klase kao jedini parametar. Naime u JavaScriptu su klase potpuno opisane svojim konstruktorima koji su zapravo

JavaScript objekti koji za sebe vežu sva svojstva klase. Unutar te funkcije s konstruktorom postupamo kao s bilo kojim JavaScript objektom te mu možemo proizvoljno mijenjati svojstva te na taj način manipulirati definicijom klase. Također, ta funkcija može vratiti novi konstruktor koji će onda zamijeniti stari te na taj način u potpunosti mijenjamo definiciju klase.

```
1  function promijeniDefiniciju(target: any){
2    return class {
3      f(){
4        console.log('nova definicija');
5      }
6    };
7  }
8
9  @promijeniDefiniciju
10 class MojaKlasa{
11   f(){
12     console.log('definicija');
13   }
14 }
15
16 let objekt = new MojaKlasa();
17 objekt.f();
```

Kod 4.3: Definicija i primjena dekoratora klase

U primjeru iz koda 4.3 smo definirali funkciju *promijeniDefiniciju* koju smo potom iskoristili kao dekorator klase *MojaKlasa*. Taj će dekorator zamijeniti definiciju klase novom definicijom te nakon što prevedemo taj kod, rezultat njegovog izvršavanja bit će ispis *nova definicija*.

Ako vraćamo novu definiciju, moramo paziti da sučelje ostane isto jer vraćanje nove definicije neće promijeniti tip. Vratimo se na kod 4.3. U tom slučaju, ako na primjer dodamo metodu $g$ u novoj definiciji, nećemo joj moći pristupiti kroz instance klase *MojaKlasa* jer ona u svom originalnom sučelju nema metodu $g$.

### 4.3.2 Dekorator metode i pristupnih svojstava

JavaScript objekti mogu imati dvije vrste svojstava (*properties*) - podatkovna svojstva (*data properties*) i pristupna svojstva (*accessor properties*). Podatkovna svojstva su obična svojstva koja poznajemo, a njima pripadaju i metode. Pristupna svojstva su posebne metode koje nam omogućuju da na elegantan način dohvaćamo i mijenjamo vrijednosti svojstava. Primjer pristupnog svojstva možemo vidjeti u kodu 4.4 gdje smo koristili dohvaćanje vrijednosti (*getter*). Rezultat izvršavanja tog koda bit će ispis broja 22. Slično se koristi i postavljanje vrijednosti (*setter*).

```
let osoba = {
  godinaRodjenja = 2000;

  get dob(){
    return 2022 - godinaRodjenja;
  }
}
console.log(osoba.dob);
```

Kod 4.4: Pristupna svojstva

Dekorator metode ili pristupnog svojstva je funkcija koja prima prototip klase (ili konstruktor klase ako je metoda statična) kao prvi parametar, ime metode, odnosno pristupnog svojstva, kao drugi parametar te opis svojstva kao treći parametar. Naime, svakom svojstvu neke klase pridružen je poseban opisni objekt (*property descriptor*). To je običan JavaScript objekt koji ima nekoliko svojstava koja nam omogućuju da svojstvo klase konfiguriramo. Pomoću njega možemo, na primjer, promijeniti definiciju same metode. Dekorator može vratiti novi opisni objekt koji će u tom slučaju potpuno zamijeniti originalni, ali promjene možemo raditi i na originalnom objektu te funkcija ne mora vratiti ništa.

U slučaju dekoratora pristupnog svojstva, ukoliko imamo definirano i dohvaćanje i postavljanje vrijednosti za isto svojstvo, dekorirati možemo samo jedan i to onaj koji je prvi naveden. Takvo je pravilo jer će u tom slučaju oba dva svojstva dijeliti isti opisni objekt.

```
function promijeniDefiniciju(target: any, propertyKey: string,
    descriptor: PropertyDescriptor){
  descriptor.value = function() {
    console.log('nova definicija');
  }
}

class MojaKlasa{
  @promijeniDefiniciju
  f(){
    console.log('definicija');
  }
}

let objekt = new MojaKlasa();
objekt.f();
```

Kod 4.5: Definicija i primjena dekoratora metode

Slično kao u i kodu 4.3, u kodu 4.5 smo definirali funkciju *promijeniDefiniciju*, ali smo ju u ovom slučaju primijenili direktno na metodu. U ovom primjeru, rezultat izvršavanja će također biti ispis *nova definicija*.

### 4.3.3 Dekorator svojstva

Dekorator svojstva je funkcija koja prima prototip klase (ili konstruktor klase ako je metoda statična) kao prvi parametar te ime svojstva kao drugi parametar. Iako je za svako svojstvo definiran opis, zbog tehničkih ograničenja, kada se radi o običnim svojstvima (onima koji nisu metode) oni ne mogu biti proslijeđeni dekoratoru. Iz tog razloga svojstva pomoću dekoratora možemo samo promatrati, ne možemo im mijenjati konfiguraciju.

Za prethodne dvije vrste dekoratora intuitivno je jasno kako promjena ponašanja metoda i klasa može biti korisna iz više razloga, no izvan nekog konteksta teško je konstruirati primjer koji će pokazati kako će nam samo promatranje biti korisno. A da je stvarno korisno uvjerit ćemo se u poglavlju 6.

### 4.3.4 Dekorator parametra

Dekorator parametra je funkcija koja prima prototip klase (ili konstruktor klase ako je metoda statična) kao prvi parametar, ime metode kao drugi parametar te indeks parametra u polju parametara metode kao treći parametar. Slično kao dekorator svojstva, ovi se dekoratori također mogu koristiti samo za promatranje.

Mogu se primijeniti na sve parametre metoda klase te na parametre konstruktora klase.

### 4.3.5 Tvornice dekoratora

Dekorator je zapravo bilo koji izraz koji će se evaluirati u jednu od funkcija koje smo prethodno opisali. To znači da dekorator može biti i poziv na funkciju koja vraća funkciju. Takve funkcije zvat ćemo tvornice dekoratora (*decorator factories*). One se koriste kako bi dekorator konfigurirali, odnosno generirali dekorator dinamički.

```
1  function promijeniIspis(tekst: string){
2    return function (target: any, propertyKey: string, descriptor:
       PropertyDescriptor){
3      descriptor.value = function() {
4        console.log(tekst);
5      }
6    }
7  }
8
9  class MojaKlasa{
10   @promijeniIspis('nova definicija')
11   f(){
12     console.log('definicija');
13   }
14 }
15
16 let objekt = new MojaKlasa();
17 objekt.f();
```

Kod 4.6: Definicija i primjena tvornice dekoratora

U primjeru iz koda 4.6 rezultat izvršavanja bit će ispis onog što smo proslijedili dekoratoru *promijeniIspis*, to jest u ovom slučaju će to biti *nova definicija*.

### 4.3.6 Evaluacija dekoratora

Na deklaracije možemo primjenjivati proizvoljan broj dekoratora. Oni će se primjenjivati na deklaraciju na način da se prvo primijeni dekorator najbliže deklaraciji (npr. ako ih navodimo jedan ispod drugog, primjenjivat će se odozdo prema gore), a svaki idući dekorator primjenjuje se na ono što je prethodni vratio. To zovemo kompozicija dekoratora koju zapravo možemo shvatiti kao kompoziciju funkcija iz matematike.

```
1  function promijeniDefiniciju(target: any){
2    return class {
3      f(){
4        target.prototype.f();
5        console.log('nova definicija');
6      }
7    };
8  }
9
10 function promijeniDefiniciju2(target: any){
11   return class {
12     f(){
13       target.prototype.f();
14       console.log('najnovija definicija');
15     }
16   };
17 }
18
19 @promijeniDefiniciju2
20 @promijeniDefiniciju
21 class MojaKlasa{
22   f(){
23     console.log('definicija');
24   }
25 }
26
27 let objekt = new MojaKlasa();
28 objekt.f();
```

Kod 4.7: Kompozicija dekoratora

U primjeru iz koda 4.7 definirali smo dva dekoratora koji će funkciji $f$ nadodati vlastiti ispis. Prvo će se primijeniti dekorator *promijeniDefiniciju* na klasu *MojaKlasa*, a nakon toga dekorator *promijeniDefiniciju2* koji će se primijeniti na klasu koju je vratio dekorator *promijeniDefiniciju*. Rezultat izvršavanja će biti ispis

*definicija*
*nova definicija*
*najnovija definicija*

Također, definiran je redoslijed kojim će se različite vrste dekoratora izvršavati:

1. Dekoratori parametara metode, nakon čega se izvršavaju dekoratori te metode (odnosno pristupnog svojstva ili svojstva), za svaku metodu, pristupno svojstvo i svojstvo
2. Dekoratori parametara statične metode, nakon čega se izvršavaju dekoratori te metode (odnosno pristupnog svojstva ili svojstva), za svaku statičnu metodu, pristupno svojstvo i svojstvo
3. Dekoratori parametara konstruktora
4. Dekoratori klase

### 4.3.7 Dekorator obrazac

Dekorator obrazac (*decorator pattern*) je strukturalni oblikovni obrazac u objektno orijentiranom programiranju koji koristimo kada objektu želimo dodati funkcionalnosti dinamički. On je svojevrsna zamjena za nasljeđivanje te se često koristi kada neki programski jezik nema mogućnost višestrukog nasljeđivanja. Dekorator se primjenjuje na pojedine instance klase i mijenja ponašanje samo tog objekta dok sama klasa ostaje netaknuta. Također nasljeđivanje u nekim slučajevima može biti nepraktično, na primjer ako imamo puno malih funkcionalnosti koje želimo dodati na baznu klasu u svim kombinacijama, tada je potrebno napraviti sve kombinacije klasa unaprijed. U tim slučajevima se također koristi dekorator obrazac jer se pomoću njega funkcionalnosti dodaju dinamički, odnosno potrebni su nam samo dekoratori koji će dodati pojedine funkcionalnosti, a njihove kombinacije generiramo kako su nam potrebne.

Iako dijele isto ime i oboje se koriste za promjenu ponašanja, TypeScript dekoratori **nisu** implementacija dekorator obrasca. TypeScript dekoratore primjenjujemo na klase i njezine članove, dok u dekorator obrascu dekoratore primjenjujemo na instance klase. Također TypeScript dekoratore primjenjujemo jednom, prilikom deklaracije klasa i nemamo mogućnost njima manipulirati tijekom izvršavanja, a dekorator obrazac se upravo i koristi kako bi funkcionalnosti mogli dodavati dinamički.

## 4.4 Refleksija

Metaprogramiranje je tehnika u programiranju u kojoj razvijamo programe koji na neki način manipuliraju drugim programima. Metaprogramiranje je općenito jako širok pojam i obuhvaća puno podtehnika. Jedan čest primjer metaprograma su prevoditelji programskog koda. Oni analiziraju i obrađuju druge programske kodove te iz njih generiraju nove programe.

Mogućnost programskog jezika da koristi tehnike metaprogramiranja na samom sebi zovemo refleksija. Odnosno, refleksija nam omogućuje da program koji pišemo ima *svijest* o sebi - da se može analizirati i modificirati. Na primjer,

nekada će nam biti potrebno da znamo imena svojstava neke klase tijekom izvođenja programa. Te se informacije inače gube nakon prevođenja, a refleksija će nam omogućiti da te informacije sačuvamo i iskoristimo prilikom izvršavanja.

### 4.4.1 Refleksija u TypeScriptu

JavaScript nudi skup metoda koje se nalaze u *Reflect* i *Object* objektima i koje nam omogućuju da dohvatimo neke osnovne informacije o objektima poput imena svojstava i metoda te da svojstva pozivamo dinamički.

Često nam osnovne funkcionalnosti koje JavaScript nudi nisu dovoljne. Iz tog razloga razvijen je *npm* paket *reflect-metadata* koji nam omogućuje da za svaki objekt i njegova svojstva dodatno definiramo prilagođene metapodatke koje onda kasnije možemo dohvatiti i na neki način iskoristiti. On zapravo proširuje *Reflect* objekt s tim dodatnim metodama.

```
1 class MojaKlasa{
2   f(x: any){
3     return x;
4   }
5 }
6 Reflect.defineMetadata('mojkljuc', 'ova je funkcija identiteta',
    MojaKlasa, 'f');
7 console.log(Reflect.getMetadata('mojkljuc', MojaKlasa, 'f');
```

Kod 4.8: Definiranje i dohvaćanje metapodataka

U kodu 4.8 smo definirali metapodatak za metodu *f* klase *MojaKlasa* pod ključem *mojkljuc*, nakon toga smo taj metapodatak dohvatili i ispisali, odnosno rezultat izvršavanja prevedenog koda bit će ispis *ovo je funkcija identiteta*.

Također, paket nudi i dekorator koji možemo primijeniti na klase te njezine metode i svojstva kako bi nad njima elegantnije definirali metapodatke.

Kako bismo taj paket mogli koristiti s TypeScriptom, prevoditelju je potrebno uključiti opciju *emitDecoratorMetadata*. Kada je ta opcija uključena, prevoditelj će automatski dodati metapodatke o tipovima parametara neke metode klase dok god je ona prethodno dekorirana nekim dekoratorom. Pretpostavimo da imamo definiran dekorator *MojDekorator* i primijenimo ga kao u kodu 4.9. Ono što će se zapravo dogoditi je ekvivalentno kao da smo napisali kod 4.10. Naravno, TypeScript neće raditi taj međukorak nego će jednostavno dodati već prevedenu verziju tog koda prilikom prevođenja. Dakle TypeScript će za nas sačuvati informacije o tipovima podataka svojstava koje onda možemo dohvatiti prilikom izvođenja koda koristeći ključeve *design:type*, *design:paramtypes* i *design:returntype* koji redom označavaju tip svojstva, te u slučaju metode tipove podatka njenih parametara i njen povratni tip.

```
class MojTip{}

@MojDekorator
class MojaKlasa{
  f(x: MojTip){
    return x;
  }
}
```

Kod 4.9: Primjena dekoratora

```
class MojTip{}

@MojDekorator
class MojaKlasa{
  @Reflect.metadata('design:type', Function)
  @Reflect.metadata('design:paramtypes', [MojTip])
  @Reflect.metadata('design:returntype', MojTip)
  f(x: MojTip){
    return x;
  }
}
```

Kod 4.10: Primjena dekoratora s *emitDecoratorMetadata*

# 5 | Express.js

Express.js je okvir za razvoj poslužiteljskih aplikacija na Node.js platformi. On je zapravo sloj apstrakcije nad Node.js-ovim HTTP modulom s kojim je inače vrlo teško raditi. Node.js-ov HTTP modul koristi jednu funkciju kako bi obradio sve zahtjeve koje pristignu na poslužitelj pa kada bismo pomoću njega razvijali aplikaciju, zahtjeve bi morali parsirati kako bi odgonetnuli što zapravo moramo raditi te bi takva aplikacija bila teško održiva. Express.js nam omogućuje da u aplikaciju koji razvijamo uvedemo strukturu tako što ćemo za svaki HTTP resurs i svaku metodu moći definirati posebnu funkciju koja će taj zahtjev obraditi, a parsiranje i sva konfiguracija će se odraditi za nas u pozadini.

Express.js je dostupan kao *npm* paket i u projekt ga možemo dodati tako da u komandnoj liniji izvršimo naredbu

```
npm install express
```

```
1  const express = require('express');
2  const app = express();
3  const port = 3000;
4
5  app.get('/resource', (req, res) => {
6    res.send('Hello World!');
7  });
8
9  app.listen(port, () => {
10   console.log(`Listening on port ${port}`);
11 });
```

Kod 5.1: Primjer Express.js aplikacije

U primjeru iz koda 5.1 smo definirali funkciju koja će upravljati dohvaćanjem resursa *resource*. Odnosno kada klijent pošalje *HTTP GET* zahtjev za resurs *resource*, odgovor će biti niz znakova *Hello World!*. Nakon što smo definirali sve funkcije za upravljanje resursima, potrebno je pokrenuti poslužiteljsku aplikaciju. To radimo pozivom na metodu *listen* kojoj prosljeđujemo *port* na kojem će poslužitelj osluškivati nove zahtjeve.

## 5.1 Posrednici

Kada definiramo kako ćemo upravljati pojedinim resursom, funkcija iz Express.js paketa koju pozivamo kao prvi parametar prima rutu, odnosno identifikator resursa te nakon rute prima proizvoljan broj funkcija koje nazivamo posrednici (*middlewares*), posebno, posljednji posrednik u nizu nazivamo upravitelj rute (*route handler*). Posrednici nam omogućuju da neovisno ili ovisno jedan o drugom promatramo i uređujemo zahtjev i odgovor koji ćemo poslati. Na taj način dijelove logike možemo ponovno iskoristiti na više mjesta.

```
1  app.get(
2    '/resource',
3    (req, res, next) => {
4      req.message = 'Hello World!';
5      next();
6    },
7    (req, res) => {
8      res.send(req.message);
9    }
10 );
```

Kod 5.2: Primjena posrednika

U primjeru iz koda 5.2 smo definirali posrednik koji će na *req* objekt dodati svojstvo *message* s vrijednošću *Hello World!*. Na kraju će upravitelj rute taj podatak iskoristiti i vratiti ga kao odgovor.

Nekada želimo primijeniti posrednik na sve zahtjeve na razini aplikacije. Kako ga ne bismo morali dodavati na svaki zahtjev posebno, Express nam nudi način da ga primijenimo globalno. Na primjer, tijelo HTTP zahtjeva poslužitelj prima kao običan tekst, no ono će zapravo biti strukturirano prateći neku specifikaciju. Ako želimo da bude strukturirano kao *json*, tada možemo primijeniti globalni posrednik koji će za nas tijelo zahtjeva parsirati i pretvoriti ga u JavaScript objekt. Za to možemo koristiti već definirani posrednik koji je dio *express* paketa, a to možemo učiniti kao u kodu 5.3, koristeći Express *use* metodu.

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

Kod 5.3: Globalni posrednik

## 5.2 Ruter

Ako upravitelje rutama želimo nekako smisleno grupirati, Express nam nudi ruter (*router*) objekt. Ruter se ponaša kao posebna instanca Express aplikacije, koju onda samo uključimo u stvarnu instancu. To nam omogućuje da posrednike primijenimo "globalno", ali samo za dio aplikacije. Ruter je i sam posrednik pa se u

aplikaciju uključuje kao i svaki drugi.

```
1 const router = express.Router();
2 router.get('/', (req, res) => res.send('Hello World!');
3
4 app.use('/resource', router);
```

Kod 5.4: Primjer rutera

U primjeru iz koda 5.4 smo napravili instancu rutera na kojoj smo definirali upravitelja za *HTTP GET* zahtjev na praznu rutu. Ruter smo uključili u aplikaciju koristeći *use* metodu kojoj smo proslijedili baznu rutu i ruter kao posrednik. Tada će se taj posrednik koristiti samo na rutama koje imaju tu bazu. Kod 5.4 u ovom slučaju radi isto kao kod 5.1.

# 6 | Okvir za razvoj poslužiteljskih aplikacija u TypeScript jeziku

## 6.1 Motivacija

U potpoglavlju 4.2 smo već vidjeli kako za Express postoje definirani tipovi koje možemo koristiti ako aplikaciju razvijamo u TypeScript jeziku. Nameće se pitanje čemu onda izrada dodatnog okvira za razvoj kada je Express potpuno kompatibilan s TypeScriptom? Pogledajmo primjer iz koda 6.1. Pretpostavimo da je funkcija *createPerson* definirana i da smo definirali upravitelj rute za kreiranje novih osoba. HTTP zahtjev će doći s tijelom za koje znamo kako izgleda, ima dva svojstva - *firstName* tipa *string* te *lastName* također tipa *string*. Ono što ćemo vratiti je isto ono što smo dobili. Ako samo pogledamo u kod, teško je shvatiti kako i zašto smo tipove definirali tako kako jesmo. *Request* je generički tip koji prima nekoliko pozicijskih parametara. Ako želimo definirati tip tijela zahtjeva, prije njega moramo definirati tip parametara rute (ono što je u našem slučaju postavljeno na *null* jer ga ne koristimo), nakon toga moramo definirati tip odgovora i tek tada možemo definirati tip tijela zahtjeva. Tip odgovora također moramo definirati i kada definiramo tip *res* objekta. Kada bismo na primjer koristili parametre upita (*query parameters*), oni se nalaze tek na četvrtome mjestu, te bi prije njih morali definirati sve tipove (odnosno postaviti ih na *null* ako ih ne koristimo). Ako su nam neki od tih tipova jednaki, kao što je slučaj u ovom primjeru, poprilično je zbunjujuće i teško za odgonetnuti što predstavlja koji tip. Ako uzmemo u obzir da aplikacije mogu imati stotine upravitelja rutama, jasno je da će na ovaj način tu aplikaciju biti teško za održavati.

```
import * as express from 'express';
const app = express();
app.post(
  '/persons',
  (
    req: express.Request<
      null,
      { firstName: string; lastName: string },
      { firstName: string; lastName: string }
    >,
    res: express.Response<{ firstName: string; lastName: string }>
  ) => {
    createPerson(req.body);
    res.send(req.body);
  }
);
app.listen(3000);
```

Kod 6.1: Express s tipovima podataka

Također, ako želimo iskoristiti sve mogućnosti koje nam TypeScript pruža te aplikaciju razviti na objektno orijentirani način koristeći klase, jasno je da se ne možemo riješiti proceduralne paradigme koju Express nameće. Logiku možemo enkapsulirati unutar klasa, no od tih klasa neće biti velike koristi jer će se na kraju sve morati napraviti na identičan način kao da nismo koristili klase, samo će logika biti premještena na drugo mjesto.

Kako izgleda ono što mi želimo postići možemo vidjeti u kodu 6.2. Želimo, dakle, da upravitelje ruta možemo grupirati u klase. Želimo definirati metode proizvoljnog imena, kako najbolje odgovara kontekstu aplikacije, i pomoću dekoratora označiti da će ta metoda izvršavati funkcionalnost upravitelja *HTTP POST* zahtjeva na definiranu rutu. Parametre rute, upita i tijelo zahtjeva želimo primati kao parametre te metode te želimo da su im imena proizvoljna i na proizvoljnim pozicijama. I za kraj želimo da ono što metoda vrati bude poslano kao odgovor. Kao što vidimo, ovo će nam omogućiti da napravimo jasno strukturiranu objektno orijentiranu aplikaciju koju će biti lako održavati.

```
@Controller('/persons')
class PersonsController {
  @Post()
  create(@Body() person: {firstName: string, lastName: string}): {
   firstName: string, lastName: string} {
    createPerson(person);
    return person;
  }
}
```

Kod 6.2: Što želimo postići

Uz sve to želimo iskoristiti brzinu i pouzdanost Express okvira. Kako bismo to postigli, morat ćemo napraviti dodatan sloj apstrakcije nad Express-om koji će

Express kod generirati u pozadini.

## 6.2 Specifikacija

Prije nego krenemo s implementacijom, specificirat ćemo koje sve mogućnosti želimo da naš okvir pruža i na koji način će se koristiti. Specifikacija ovog okvira uvelike je inspirirana NestJS okvirom [6], jednim od vodećih okvira za razvoj poslužiteljskih aplikacija u TypeScriptu.

### 6.2.1 Korijen aplikacije

Aplikacije obično imaju korijensku datoteku u koju se sve uvozi i koja cijelu aplikaciju objedinjuje. Korijen aplikacija razvijenih ovim okvirom bit će klasa dekorirana *App* dekoratorom. *App* dekorator primat će objekt koji će imati jedno obvezno svojstvo - *port* koje će predstavljati port na kojem će poslužitelj osluškivati nove zahtjeve. Primjer korijena možemo vidjeti u kodu 6.3. Treba napomenuti da je ime klase potpuno proizvoljno, kao što će u nastavku imena svih klasa, metoda i parametara biti proizvoljne, osim ako nije drugačije napomenuto. Također, zbog konzistentnosti svi dekoratori će biti definirani kao tvornice dekoratora, primali neku vrijednost ili ne.

```
1 @App({ port: 3000 })
2 class Application {}
```

Kod 6.3: Korijen aplikacije

### 6.2.2 Kontroleri

Kontrolere (*controllers*) smo već spomenuli u motivaciji. Oni će zapravo predstavljati Express rutere, odnosno pomoću njih ćemo definirati upravitelje rutama grupirane oko jednog resursa.

Kontroler će biti klasa dekorirana *Controller* dekoratorom koji prima jedan obvezni parametar tipa *string* koji će predstavljati baznu rutu.

```
1 @Controller('/hello-world')
2 class HelloWorldController {}
```

Kod 6.4: Primjer kontrolera

Objekt kojeg prima korijenski *App* dekorator imat će opcionalno svojstvo *controllers* koje će biti polje u kojem ćemo navesti sve kontrolere koje koristimo u aplikaciji te ih na taj način registrirati.

```
@App({
  port: 3000,
  controllers: [HelloWorldController],
})
class Application {}
```

Kod 6.5: Registracija kontrolera

### 6.2.3 Upravitelji rutama

Upravitelj rutama bit će metoda unutar kontroler klase dekorirana dekoratorom odgovarajuće HTTP metode. Za svaku metodu bit će definiran istoimeni dekorator koji će primati jedan opcionalni parametar tipa *string* koji će predstavljati nastavak na baznu rutu koja je definirana u kontroleru. Ono što metoda vrati, bit će poslano kao odgovor na zahtjev.

```
@Controller('/hello-world')
class HelloWorldController {
  @Get()
  get(){
    return 'Hello world';
  }
}
```

Kod 6.6: Primjer upravitelja rute

### 6.2.4 Parametri rute i upita te tijelo zahtjeva

Parametri rute i upita te tijelo zahtjeva bit će proslijeđeni parametrima metode koja je upravitelj rute. Definirat ćemo dekoratore parametara *RequestParams*, *RequestQuery* i *RequestBody* kojima ćemo dekorirati parametre u koje želimo da vrijednosti budu proslijeđene.

```
@Controller('/hello-world')
class HelloWorldController {
  @Get('/:id')
  get(@RequestParams() params: {id: string}) {
    return params;
  }
}
```

Kod 6.7: Korištenje parametra rute

### 6.2.5 Objekti zahtjeva i odgovora

Iako želimo izbjeći korištenje Express-ovih objekata zahtjeva i odgovora (objekti koje primaju Express posrednici), nekada ono što ovaj okvir nudi neće biti dovoljno, iako će pokriti većinu slučajeva. Ne želimo ograničavati korisnike okvira pa ćemo zato omogućiti da se pristupi originalnim objektima zahtjeva i odgovora. Definirat ćemo *Request* i *Response* dekoratore parametara, koji će raditi isto kao i dekoratori iz prethodnog potpotpoglavlja.

```
1 @Get('/:id')
2 get(@RequestParams() params: {id: string}, @Response() res: express
      .Response) {
3   res.set(...)
4   return params;
5 }
```

Kod 6.8: Korištenje objekta odgovora

### 6.2.6 Promjena HTTP statusa i postavljanje zaglavlja

Definirat ćemo dekorator *DefaultHttpStatus* koji će primati jedan obvezni parametar tipa *number* i koji ćemo moći primijeniti na klasu kontrolera ili na pojedine upravitelje rutama. Ako ga primijenimo na cijeli kontroler, svi odgovori će imati taj status. Primjena dekoratora na upravitelja rute će imati prednost nad primjenom na kontroleru.

Kako bismo dodali elemente u zaglavlje odgovora, definirat ćemo dekorator *Headers* koji će primati jedan obvezni parametar - objekt čija će svojstva biti ključevi zaglavlja, a vrijednosti tih svojstava vrijednosti na tim ključevima. Taj će se dekorator ponašati identično kao *DefaultHttpStatus* dekorator.

```
1  @Controller('/hello-world')
2  @DefaultHttpStatus(201)
3  @Headers({ hello: 'world' })
4  class HelloWorldController {
5    @Get()
6    @DefaultHttpStatus(200)
7    @Headers({'my-header': 'my-value'})
8    get() {
9      return 'Hello world';
10   }
11
12   @Post()
13   create() {
14     return 'Hello world created';
15   }
16 }
```

Kod 6.9: Promjena pretpostavljenog HTTP statusa

U primjeru iz koda 6.9, ako napravimo *HTTP POST* zahtjev, status će biti *201*, a ako napravimo *HTTP GET* zahtjev, bit će *200*. Oba će odgovora u zaglavlju sadržavati vrijednost *world* pod ključem *hello*, a odgovor na *GET* zahtjev će dodatno sadržavati vrijednost *my-value* pod ključem *my-header*. Uočimo da poredak dekoratora nije bitan jer oni neće modificirati klase i metode, nego će ih samo promatrati i bilježiti metapodatke.

U nekim slučajevima ćemo status i zaglavlje morati promijeniti dinamički, tada nam dekoratori neće biti od koristi. Kako bismo to postigli, jednostavno možemo koristiti Express objekt odgovora i pomoću njega dinamički promijeniti status i zaglavlje.

### 6.2.7 Posrednici

Želimo da ovaj okvir bude kompatibilan sa svim posrednicima iz Express okvira. Posrednike želimo moći primijeniti, kao i u Express-u, globalno, na kontroler te na svaku rutu posebno.

Kako bi primijenili posrednike globalno, objekt kojeg prima *App* dekorator imat će opcionalno svojstvo *useGlobalMiddlewares* koje će biti polje u kojem ćemo navesti sve globalne posrednike.

```
1 @App({
2   port: 3000,
3   controllers: [HelloWorldController],
4   useGlobalMiddlewares: [express.urlencoded()],
5 })
6 class Application {}
```

Kod 6.10: Primjena globalnih posrednika label

Kako bi ih primijenili na razini kontrolera i rute, definirat ćemo dekorator *UseMiddlewares* koji će primati proizvoljan broj parametara u kojima ćemo navesti posrednike.

```
1 @UseMiddlewares(express.urlencoded(), express.text())
2 @Controller('/hello-world')
3 class HelloWorldController {
4   @Get()
5   @UseMiddlewares(express.static(...))
6   get() {
7     return 'Hello world';
8   }
9 }
```

Kod 6.11: Primjena posrednika na kontroler i rutu

### 6.2.8 Ubrizgavanje ovisnosti

Ubrizgavanje ovisnosti (*dependency injection*) je oblikovni obrazac u objektno orijentiranom programiranju prema kojem ovisnosti neke klase (ovisnosti su neke druge klase o kojima klasa ovisi) trebaju biti otprije instancirane te proslijeđene konstruktoru klase, umjesto da se ovisnosti instanciraju unutar konstruktora. Kako bi bilo jasnije pogledajmo kod 6.12 gdje nije korišteno ubrizgavanje ovisnosti te kod 6.13 gdje je.

```
1 class MyClass{
2     private instance1: Dependency1;
3     private instance2: Dependency2;
4     constructor(){
5         this.instance1 = new Dependency1();
6         this.instance2 = new Dependency2();
7     }
8 }
```

Kod 6.12: Klasa bez ubrizgavanja ovisnosti

```
1 class MyClass{
2     private instance1: Dependency1;
3     private instance2: Dependency2;
4     constructor(instance1: Dependency1, instance2: Dependency2){
5         this.instance1 = instance1;
6         this.instance2 = instance2;
7     }
8 }
```

Kod 6.13: Klasa s ubrizgavanjem ovisnosti

Aplikacije koje koriste ubrizgavanje ovisnosti puno je lakše testirati i održavati jer je sama logika instanciranja izdvojena iz pojedinih klasa. Problem s ovim obrascem je očito kada imamo puno ovisnosti koje opet imaju svoje ovisnosti i to može ići vrlo duboko. Ručno instanciranje ovisnosti u tom slučaju je vrlo nepraktično. Na sreću taj se postupak može automatizirati. Većina modernih okvira koji koriste ubrizgavanje ovisnosti koriste takozvana skladišta ovisnosti (*dependency container*) u kojima se nalaze instance svih ovisnosti koje je na neki način prvo potrebno registrirati. Na taj način okvir vodi brigu o instanciranju i ubrizgavanju ovisnosti.

Kao što smo rekli u potpotpoglavlju 4.4.1, TypeScript prevoditelj će sačuvati tipove podataka parametara svih metoda pa tako i konstruktora. Ono što predstavlja taj tip je zapravo konstruktor, što znači da ćemo taj metapodatak moći iskoristiti kako bi instancirali taj parametar. U 4.4.1 smo također rekli da će se tipovi sačuvati samo ako je klasa ili neko njezino svojstvo prethodno dekorirano. Zbog toga, svaka klasa koja će biti dio lanca ubrizgavanja ovisnosti mora biti dekorirana. Kontroleri već imaju svoj dekorator, no želimo biti u mogućnosti ubrizgavati i ovisnosti u klase koje nisu dio strukture ovog okvira. Iz tog razloga ćemo defi-

nirati dekorator *Injectable* koji ćemo moći primijeniti na sve klase za koje nemamo prikladniji dekorator.

```
1  Injectable();
2  class HelloWorldService {
3    constructor(private readonly dependency: Dependency){}
4
5    findMessage(): string {
6      return 'Hello world!';
7    }
8  }
9
10 @Controller('/hello-world')
11 class HelloWorldController {
12   constructor(private readonly helloWorldService: HelloWorldService
       ) {}
13
14   @Get()
15   get() {
16     return this.helloWorldService.findMessage();
17   }
18 }
```

Kod 6.14: Ubrizgavanje ovisnosti u okviru

Kao što vidimo u primjeru iz koda 6.14, ovisnosti je dovoljno samo navesti unutar konstruktora, a okvir će se pobrinuti o njihovom instanciranju i ubrizgavanju.

Pokrili smo slučaj kada je parametar konstruktora neki tip koji također ima konstruktor, no što ako želimo ubrizgati neku vrijednost? Kako bi to omogućili definirat ćemo dekorator parametra *Inject* koji će primati jedan parametar tipa *string*, taj ćemo parametar zvati token koji će na jednistven način određivati ovisnost koju želimo ubrizgati. Kako bi definirali što želimo ubrizgati na mjesto tog parametra, objekt kojeg prima *App* dekorator imat će opcionalno svojstvo *customProviders*, a ono će biti polje objekata koji će imati svojstva *token* odnosno indentifikator i *instance* koje će predstavljati ono što će biti ubrizgano. Primijetimo da kada ubrizgavamo ovisnost s konstruktorom, njezin token implicitno postaje taj konstruktor. Zbog toga možemo definirati da se umjesto te ovisnosti ubrizga nešto drugo. Primjer toga možemo vidjeti u kodu 6.15.

```
@App({
  port: 3000,
  controllers: [HelloWorldController],
  customProviders: [
    { token: 'helloWorldService', instance: new HelloWorldService()
    },
    { token: HelloWorldService, instance: { findMessage: () => '
   Hello world 2!' } },
  ],
})
class Application {}

@Controller('/hello-world')
class HelloWorldController {
  constructor(
    @Inject('helloWorldService') private readonly helloWorldService
   : HelloWorldService,
    private readonly helloWorldService2: HelloWorldService,
  ) {}

  @Get()
  get() {
    return this.helloWorldService.findMessage();
  }

  @Get('/other')
  get2() {
    return this.helloWorldService2.findMessage();
  }
}
```

Kod 6.15: Ubrizgavanje prilagođenih ovisnosti

U primjeru iz koda 6.15, svojstvo *helloWorldService* će biti instanca *HelloWorldService* klase, dok *helloWorldService2*, iako je tako navedeno u konstruktoru, neće biti instanca te klase nego objekt koji smo naveli prilikom registracije ovisnosti. Treba primijetiti da moramo sami paziti da ono što ubrizgavamo prati sučelje originalne klase.

### 6.2.9 Životni ciklus aplikacije

Učitavanje aplikacije razvijene pomoću ovog okvira odvija se u nekoliko koraka ovim redoslijedom:

1. Globalni posrednici se vežu za instancu Express aplikacije
2. Definiraju se ruteri i upravitelji ruta te se vežu za instancu Express aplikacije
3. Instanca Express aplikacije započinje s osluškivanjem

Želimo omogućiti da korisnici okvira mogu izvršiti vlastiti kod prije i poslije svakog od događaja. Kako bismo to omogućili definirat ćemo 6 dekoratora *Befo-*

*reGlobalMiddlewaresBound*, *AfterGlobalMiddlewaresBound*, *BeforeRoutesBound*, *AfterRoutesBound*, *BeforeListenStarted* te *AfterListenStarted*. Ovim dekoratorima dekorirat ćemo metode korijenske klase i zvat ćemo ih kuke (*hooks*).

```
@App({
  port: 3000,
  controllers: [HelloWorldController],
})
class Application {
  constructor(private readonly helloWorldService: HelloWorldService
   ) {}

  @BeforeRoutesBound()
  private beforeRoutes() {
    console.log('I am executed just before routes are bound');
    console.log(this.helloWorldService.findMessage());
  }
}
```

Kod 6.16: Korištenje kuka životnog ciklusa

Kao što vidimo u primjeru iz koda 6.16, u korijensku klasu također možemo ubrizgavati ovisnosti i koristiti ih u kukama.

### 6.2.10 Instanca Express aplikacije

Iako ovaj okvir izrađujemo kako ne bismo koristili Express direktno, on je sloj apstrakcije, a kao takav uvijek će imati određena ograničenja i neće moći postići baš sve što se može korištenjem Express-a direktno. Kako ne bismo ograničavali korisnike okvira, omogućit ćemo korištenje instance Express aplikacije koju okvir koristi u pozadini. Definirat ćemo *ExpressAppInstance* dekorator parametra te ćemo pomoću njega iskoristiti mehanizam za ubrizgavanje ovisnosti kako bismo instancu Express aplikacije ubrizgali bilo gdje u aplikaciji.

```
@App({
  port: 3000,
  controllers: [HelloWorldController],
})
class Application {
  constructor(
    @ExpressAppInstance() private readonly expressApp: any,
  ) {}
  @AfterGlobalMiddlewaresBound()
  private addUrlencodedMiddleware() {
    this.expressApp.use(express.urlencoded());
  }
}
```

Kod 6.17: Korištenje instance Express aplikacije

## 6.3 Implementacija

### 6.3.1 Ubrizgavanje ovisnosti

Implementaciju ćemo početi od ubrizgavanja ovisnosti jer će nam o tome ovisiti implementacija ostalih značajki.

Prvo su nam potrebni dekoratori *Injectable* i *Inject*. *Injectable* će biti prazan dekorator koji neće raditi ništa, on nam potreban isključivo kako bismo klasu dekorirali te time sačuvali tipove podataka parametara konstruktora. Njegovu implementaciju možemo vidjeti u kodu 6.18. Uočimo da tvornica dekoratora ima definiran povratni tip *ClassDecorator* što znači da ćemo ga moći primijeniti samo na klase. TypeScript nam nudi tipove dekoratora kako bismo onemogućili da ih se krivo primijeni.

```
1 function Injectable(): ClassDecorator {
2   return function (target) {};
3 }
```

Kod 6.18: Implementacija *Injectable* dekoratora

*Inject* dekorator bit će nešto složeniji. To je dekorator parametra koji prima jedan parametar tipa *string*, a koji predstavlja token koji želimo ubrizgati. Pomoću *reflect-metadata* paketa definirat ćemo metapodatak na klasi - rječnik koji će za ključeve imati indekse parametara, a vrijednosti će biti tokeni. *Inject* dekorator će taj rječnik prvo dohvatiti te ga ažurirati. Metapodatak će se nalaziti pod ključem *injectTokens*. Implementaciju možemo vidjeti u kodu 6.19. Koristili smo operator || koji će vratiti vrijednost s lijeve strane ukoliko je definirana, inače vraća desnu stranu. Time pokrivamo slučaj prvog parametra i inicijaliziramo rječnik kao prazan.

```
1 function Inject(token: string): ParameterDecorator {
2   return function (target: any, key: string, index: number) {
3     const injectTokens: { [P: number]: string } =
4       Reflect.getMetadata('injectTokens', target, key) || {};
5     injectTokens[index] = token;
6     Reflect.defineMetadata('injectTokens', injectTokens, target,
      key);
7   };
8 }
```

Kod 6.19: Implementacija *Inject* dekoratora

Svu logiku vezanu uz ubrizgavanje ovisnosti smjestit ćemo u klasu *DependencyContainer* koja će sadržavati jedno privatno statično polje *dependencies* koje će imati ulogu skladišta ovisnosti te statične metode za dohvaćanje *get*, razrješavanje *resolve* i registriranje ovisnosti *registerDependency*.

Polje ovisnosti sadržavat će objekte koji će imati dva svojstva - *token* koje će sadržavati token te *instance* koje će sadržavati instancu. Metoda *get* će primati jedan

parametar *token* koji će biti tipa *string* ili neki konstruktor. Pomoću tokena ćemo pretražiti polje ovisnosti te ako ovisnost pronađemo vratit ćemo ju. Ukoliko ju ne pronađemo, razriješit ćemo ju pomoću metode *resolve* te ćemo token i razriješenu instancu pohraniti u polje i nakon toga vratiti. Implementaciju metode *get* možemo vidjeti u kodu 6.20.

```
1  class DependencyContainer {
2    private static readonly dependencies = [];
3
4    static get(token){
5      let dependency = DependencyContainer.dependencies.find(
6        (dependency) => dependency.token === token
7      );
8      if (!dependency) {
9        const instance: T = DependencyContainer.resolve(token);
10       dependency = { token, instance };
11       DependencyContainer.dependencies.push(dependency);
12     }
13     return dependency.instance;
14   }
15 }
```

Kod 6.20: Implementacija metode *get*

Metoda *resolve* primat će jedan parametar *token*, ali ćemo sada pretpostaviti da je token konstruktor. Prvo ćemo pomoću *reflect-metadata* paketa dohvatiti metapodatke o tipovima podataka parametara konstruktora koji se nalaze pod ključem *design:paramtypes* i tokene koje smo postavili pomoću *Inject* dekoratora koji će se nalaziti pod ključem *injectTokens*. Kao što smo naveli u specifikaciji u potpotpoglavlju 6.2.8, prednost pri ubrizgavanju imat će tokeni navedeni pomoću *Inject* dekoratora. Zato ćemo proći kroz sve parametre konstruktora, provjeriti postoji li definiran token pomoću *Inject* dekoratora te ćemo dohvatiti instancu tog tokena pomoću *get* metode. Primijetimo da je ovaj postupak rekurzivan, jer metoda *get* u nekim slučajevima poziva metodu *resolve*. Sve instance ćemo pohraniti u jedno polje koje ćemo iskoristiti kako bismo na kraju instancirali traženi token. Implementaciju metode možemo vidjeti u kodu 6.21. Koristili smo operator raspršivanja (*spread operator*), odnosno operator ... koji će, kako mu i ime kaže, raspršiti polje i bit će kao da smo parametre naveli kao pojedine parametre funkcije.

```
class DependencyContainer {
  static resolve(token) {
    const constructorParamTypes = Reflect.getMetadata(
      'design:paramtypes',
      token
    );
    const injectTokens = Reflect.getMetadata(
      'injectTokens',
      token
    );
    const constructorParamInstances = [];
    for (const i in constructorParamTypes) {
      let injectToken = constructorParamTypes[i];
      if (injectTokens) {
        injectToken = injectTokens[i] || injectToken;
      }
      constructorParamInstances.push(DependencyContainer.get(
    injectToken));
    }
    return new token(...constructorParamInstances);
  }
}
```

Kod 6.21: Implementacija metode *resolve*

Preostala je još metoda *registerDependency* koja će primati token kao prvi parametar te instancu kao drugi. Metoda će jednostavno parametre spojiti u objekt te ih dodati na kraj polja u koje skladištimo ovisnosti. Implementaciju možemo vidjeti u kodu 6.22.

```
class DependencyContainer {
  static registerDependency(token, instance) {
    DependencyContainer.dependencies.push({ token, instance });
  }
}
```

Kod 6.22: Implementacija metode *registerDependency*

### 6.3.2 Korijen aplikacije

Sva logika zadužena za podizanje naše aplikacije i generiranje Express koda nalazit će se u klasi *Bootstrap*. Njen će konstruktor primati konstruktor korijenske klase te objekt koji prosljeđujemo *App* dekoratoru. Sadržavat će jednu javnu metodu *bootstrap* koja će pokrenuti podizanje aplikacije. *App* dekorator će instancirati *Bootstrap* klasu te joj proslijediti navedene parametre. Nakon toga će pozvati metodu *bootstrap* i time pokrenuti aplikaciju. Implementaciju *App* dekoratora možemo vidjeti u kodu 6.23.

```
function App(properties): ClassDecorator {
  return function (target: any) {
    const bootstrap = new Bootstrap(target, properties);
    bootstrap.bootstrap();
  };
}
```

Kod 6.23: Implementacija *App* dekoratora

U konstruktoru *Bootstrap* klase inicijalizirat ćemo Express aplikaciju te ćemo ju pohraniti u privatno svojstvo. U *bootstrap* metodi zasada ćemo samo registrirati prilagođene ovisnosti. Jednostavno ćemo proći kroz sve prilagođene ovisnosti proslijeđene *App* dekoratoru i registrirati ih pomoću *registerDependency* metode koju smo definirali u prethodnom potpotpoglavlju. Implementaciju trenutno opisanog možemo vidjeti u kodu 6.24.

```
class Bootstrap {
  private appInstance: any;
  private readonly expressApp: express.Express;
  constructor(
    private readonly appClass: any,
    private readonly appProperties: any
  ) {
    this.expressApp = express();
  }

  bootstrap() {
    this.appProperties.customProviders?.forEach((provider) =>
    DependencyContainer.registerDependency(provider.token, provider.
    instance));
  }
}
```

Kod 6.24: Implementacija *Bootstrap* klase

### 6.3.3 Kontroleri, upravitelji rutama i parametri upravitelja rute

Implementirajmo prvo *Controller* dekorator. To će biti jednostavan dekorator koji će klasi dodati metapodatak o baznoj ruti pod ključem *route* koja će mu biti proslijeđena kao parametar. Implementaciju možemo vidjeti u kodu 6.25

```
function Controller(route: string): ClassDecorator {
  return function (target: any) {
    Reflect.defineMetadata('route', route, target);
  };
}
```

Kod 6.25: Implementacija *Controller* dekoratora

Sada je potrebo implementirati dekorator za svaku od HTTP metoda. One će sve biti ekvivalente, samo će spremati metapodatak s drugom vrijednosti. Zato ćemo koristiti "tvornicu tvornice" dekoratora. Ono što će svaki od dekoratora na kraju raditi bit će definiranje dva metapodatka - podatak koji će označavati vrstu metode pod ključem *method* te podatak o dodatku na baznu rutu pod ključem *path*. Implementaciju "tvornice tvornica" i kako ju iskoristiti za *HTTP GET* metodu možemo vidjeti u kodu 6.26. Na ekvivalentan način možemo napraviti sve preostale metode.

```
function httpMethodDecoratorFactory(method: string) {
  return function (path?: string): MethodDecorator {
    return function (target: any, key: string) {
      Reflect.defineMetadata('method', method, target, key);
      Reflect.defineMetadata('path', path, target, key);
    };
  };
}

const Get = httpMethodDecoratorFactory('get');
```

Kod 6.26: Implementacija *Get* dekoratora

Preostalo je još definirati dekoratore parametara upravitelja rute, odnosno dekoratore *RequestParams*, *RequestQuery*, *RequestBody*, *Request* i *Response*. To će biti jednostavni dekoratori koji će metodi dodijeliti metapodatak o indeksu parametra pod odgovarajućim ključem. Svi će biti ekvivalentni pa i u ovom slučaju možemo koristiti "tvornicu tvornice". Implementaciju *RequestParams* dekoratora možemo vidjeti u kodu 6.27, a preostali dekoratori se implementiraju na ekvivalentan način.

```
function argumentDecoratorFactory(metadataKey: string) {
  return function (): ParameterDecorator {
    return function (target: any, key: string, index: number) {
      Reflect.defineMetadata(metadataKey, index, target, key);
    };
  };
}

const RequestParams = argumentDecoratorFactory('requestParams');
```

Kod 6.27: Implementacija *RequestParams* dekoratora

Sada imamo sve metapodatke potrebne za implementaciju osnovnih mogućnosti kontrolera. U klasi *Bootstrap* ćemo implementirati privatnu metodu *registerController* koja će primati klasu kontrolera (odnosno njegov konstruktor) kao jedini parametar. Kontroler ćemo instancirati tako da metodi *get* iz *DependencyContainer* klase proslijedimo taj konstruktor, koja će sve što je potrebo instancirati i ubrizgati za nas. Nakon toga ćemo napraviti instancu Express rutera. Proći ćemo kroz sva svojstva kontrolera (koja se nalaze u *prototype* objektu konstruktora) i

za svako provjeriti jeli na njemu definirana neka od HTTP metoda. Ukoliko je, provjerit ćemo jeli definiran nastavak na baznu rutu te definirati Express upravitelj rute koji će unutar svog tijela pozvati tu metodu i ubrizgati joj parametre na odgovarajuće indekse. Nakon toga ćemo dodati taj upravitelj rute u instancu Express rutera. Nakon što smo prošli kroz sva svojstva, dohvatit ćemo metapodatak o baznoj ruti te dodati ruter u instancu Express aplikacije. Implementaciju *registerController* metode možemo vidjeti u kodu 6.28. Prisjetimo se, dekoratori metoda pozivaju se na prototipu klase, zato kada dohvaćamo metapodatke metode, dohvaćamo ih pomoću prototipa. Uočimo metodu *bind* koju koristimo u liniji 8. Zbog načina na koji JavaScript veže *this* pokazivač uz kontekst izvršavanja, kada metodu pozivamo dinamički, moramo eksplicitno navesti na što da *this* pokazuje, a to radimo pomoću *bind* metode.

```
class Bootstrap {
  private registerController(controllerClass) {
    const controller = DependencyContainer.get(controllerClass);
    const router = express.Router();
    Object.keys(controllerClass.prototype).forEach(key => {
      const httpMethod = Reflect.getMetadata('method',
  controllerClass.prototype);
      if (httpMethod) {
        const handler = controller[key].bind(controller);
        const expressHandler = (req, res) => {
          const args = [];
          const paramsIndex = Reflect.getMetadata('requestParams',
  contollerClass.prototype, key);
          if(paramsIndex != null){
            args[paramsIndex] = req.params;
          }
          const queryIndex = Reflect.getMetadata('requestQuery',
  contollerClass.prototype, key);
          if(queryIndex != null){
            args[queryIndex] = req.query;
          }
          const bodyIndex = Reflect.getMetadata('requestBody',
  contollerClass.prototype, key);
          if(bodyIndex != null){
            args[bodyIndex] = req.body;
          }
          const reqIndex = Reflect.getMetadata('request',
  contollerClass.prototype, key);
          if(reqIndex != null){
            args[reqIndex] = req;
          }
          const resIndex = Reflect.getMetadata('response',
  contollerClass.prototype, key);
          if(resIndex != null){
            args[resIndex] = res;
          }
          const response = handler(...args);
          res.send(response)
        };
        const path = Reflect.getMetadata('path', contollerClass.
  prototype, key) || '/';
        router[method](path, expressHandler);
      }
    });
    const route = Reflect.getMetadata('route', controllerClass);
    this.expressApp.use(route, router);
  }
}
```

Kod 6.28: Implementacija *registerController* metode

Sada je još potrebno proći kroz sve kontrolere navedene u *App* dekoratoru i na njima pozvati *registerController* metodu. To ćemo učiniti unutar *bootstrap* metode. Nakon toga možemo pozvati metodu *listen* na instanci Express aplikacije i time

započeti osluškivanje na zadanom portu. Kako trenutno treba izgledati *bootstrap* metoda možemo vidjeti u kodu *controllerBootstrap*.

```
1 class Bootstrap {
2   bootstrap() {
3     this.appProperties.customProviders?.forEach((provider) =>
      DependencyContainer.registerDependency(provider.token, provider.
      instance));
4     this.appProperties.controllers?.forEach((controller) => this.
      registerController(controller));
5     this.expressApp.listen(this.appProperties.port);
6   }
7 }
```

Kod 6.29: Implementacija *bootstrap* metode

### 6.3.4 Promjena HTTP statusa i postavljanje zaglavlja

Prvo ćemo implementirati *DefaultHttpStatus* dekorator. To će biti dekorator koji će se moći primijeniti i na klase i na metode. Kako bismo odgonetnuli na što je dekorator primijenjen, provjerit ćemo jeli definiran parametar *key*. Ukoliko je, očito se radi o dekoratoru metode, inače ćemo znati da je dekorirana klasa. U oba slučaja definirat ćemo metapodatak pod ključem *httpStatus*, a vrijednost će mu biti HTTP status kod. Implementaciju možemo vidjeti u kodu 6.30.

```
1 function DefaultHttpStatus(code: number): ClassDecorator &
    MethodDecorator {
2   return function (target: any, key?: string) {
3     if (key) {
4       Reflect.defineMetadata('httpStatus', code, target, key);
5     } else {
6       Reflect.defineMetadata('httpStatus', code, target);
7     }
8   };
9 }
```

Kod 6.30: Implementacija *DefaultHttpStatus* dekoratora

*Headers* dekorator implementira se ekvivalentno, samo s ključem *headers*.

Kako bismo iskoristili ove metapodatke, potrebno je modificirati metodu *registerControllers* u *Bootstrap* klasi. Unutar te metode definirali smo Express upravitelj rute koji na kraju vraća odgovor koristeći objekt odgovora, odnosno njegovu *send* metodu. Potrebno je unutar tog upravitelja, prije poziva *send* metode dohvatiti metapodatke o HTTP statusu i zaglavlju te ih postaviti koristeći metode objekta odgovora *status* odnosno *set*. Kako sada treba izgledati implementacija Express upravitelja rute možemo vidjeti u kodu 6.31. Uočimo da je ovo samo dio implementacije *expressHandler* funkcije koja je dio *registerController* metode.

```
class Bootstrap{
  private registerController(controllerClass){
    // ...
      const expressHandler = (req, res) => {
        //...
        const controllerHttpStatus = Reflect.getMetadata('
    httpStatus', controllerClass);
        const methodHttpStatus = Reflect.getMetadata('httpStatus',
    controllerClass.prototype, key);
        const httpStatus = methodHttpStatus || controllerHttpStatus
    ;
        if(httpStatus){
          res.status(httpStatus);
        }
        const controllerHeaders = Reflect.getMetadata('headers',
    controllerClass);
        const methodHeaders = Reflect.getMetadata('headers',
    controllerClass.prototype, key);
        const headers = {...controllerHeaders, ...methodHeaders};
        res.set(headers);
        const response = handler(...args);
        res.send(response)
      };
    //...
  }
}
```

Kod 6.31: Implementacija Express upravitelja rute

### 6.3.5 Posrednici

Globalni posrednici bit će proslijeđeni *App* dekoratoru i kako bi ih postavili globalno jednostavno ćemo ih samo dodati u instancu Express aplikacije na početku *bootstrap* metode u *Bootstrap* klasi. Implementaciju možemo vidjeti u kodu 6.32.

```
class Bootstrap {
  bootstrap() {
    this.appProperties.customProviders?.forEach((provider) =>
    DependencyContainer.registerDependency(provider.token, provider.
    instance));
    this.expressApp.use(this.appProperties.globalMiddlewares);
    this.appProperties.controllers?.forEach((controller) => this.
    registerController(controller));
    this.expressApp.listen(this.appProperties.port);
  }
}
```

Kod 6.32: Dodavanje globalnih posrednika

Za posrednike na razini kontrolera i upravitelja rutama, implementirat ćemo *UseMiddlewares* dekorator koji će se moći primijeniti na klase i metode. Imple-

mentacija će biti slična kao i *DefaultHttpStatus* odnosno *Headers* dekoratora, a možemo ju vidjeti u kodu 6.33. Kako smo naveli u specifikaciji u potpotpoglavlju 6.2.7, *UseMiddlewares* dekorator primat će proizvoljan broj parametara, mi ćemo pomoću operatora raspršivanja te parametre dohvatiti u obliku polja.

```
function UseMiddlewares(...middlewares): ClassDecorator &
    MethodDecorator {
  return function (target: any, key?: string) {
    if (key) {
      Reflect.defineMetadata('useMiddlewares', middlewares, target,
    key);
    } else {
      Reflect.defineMetadata('useMiddlewares', middlewares, target)
    ;
    }
  };
}
```

Kod 6.33: Implementacija *UseMiddlewares* dekoratora

Sada je potrebno modificirati *registerController* metodu u *Bootstrap* klasi kako bismo te posrednike uključili u Express ruter, odnosno upravitelja ruta. Implementaciju možemo vidjeti kodu 6.34.

```
class Bootstrap{
  private registerController(controllerClass){
    const controller = DependencyContainer.get(controllerClass);
    const router = express.Router();
    const controllerMiddlewares = Reflect.getMetadata('
   useMiddlewares', controllerClass);
    if(controllerMiddlewares){
      router.use(controllerMiddlewares);
    }
    // ...
      const expressHandler = (req, res) => {/*...*/};
      const methodMiddlewares = Reflect.getMetadata('useMiddlewares
   ', controllerClass, key) || [];
      const middlewares = [...methodMiddlewares, expressHandler];
      router[method](path, ...middlewares);
    //...
  }
}
```

Kod 6.34: Dodavanje posrednika u ruter i upravitelja rute

### 6.3.6 Kuke životnog ciklusa

Prvo ćemo implementirati dekoratore kuka životnog ciklusa. Kako će one sve biti ekvivalentne i u ovom ćemo slučaju koristiti "tvornicu tvornice". Dekoratori će definirati metapodatak na klasi, s ključem koji će označavati kuku, a vrijednost

će biti ime metode. Implementaciju *BeforeMiddlewaresBound* dekoratora možemo vidjeti u kodu 6.35, a ostali dekoratori implementiraju se na isti način.

```
function lifecycleHookDecoratorFactory(metadataKey: string):
    Function {
  return function (): MethodDecorator {
    return function (target: any, key: string) {
      Reflect.defineMetadata(metadataKey, key, target);
    };
  };
}

const BeforeMiddlewaresBound = lifecycleHookDecoratorFactory('
    beforeMiddlewaresBound');
```

Kod 6.35: Implementacija *BeforeMiddlewaresBound* dekoratora

Konstruktoru *Bootstrap* klase proslijedili smo konstruktor korijenske klase, ali ga još nismo iskoristili. Sada ćemo u *bootstrap* pomoću *DependenyContainer* klase instancirati korijensku klasu, te ćemo dohvatiti metode dekorirane kukama životnog ciklusa koje ćemo pozvati na odgovarajućim mjestima. Implementaciju za prve dvije kuke možemo vidjeti u kodu 6.36, a ostatak je ekvivalentan. Uočimo da korijensku klasu moramo instancirati nakon što smo registrirali prilagođene ovisnosti kako bismo ih i u nju mogli ubrizgavati.

```
class Bootstrap {
  bootstrap() {
    this.appProperties.customProviders?.forEach((provider) =>
    DependencyContainer.registerDependency(provider.token, provider.
    instance));
    const appInstance = DependencyContainer.get(this.appClass);
    const beforeMiddlewaresBoundMethodKey = Reflect.getMetadata('
    beforeMiddlewaresBound', this.appClass.prototype);
    if(beforeMiddlewaresBoundMethodKey){
      appInstance[beforeMiddlewaresBoundMethodKey]();
    }
    this.expressApp.use(this.appProperties.globalMiddlewares);
    const afterMiddlewaresBoundMethodKey = Reflect.getMetadata('
    afterMiddlewaresBound', this.appClass.prototype);
    if(afterMiddlewaresBoundMethodKey){
      appInstance[afterMiddlewaresBoundMethodKey]();
    }
    //...
  }
}
```

Kod 6.36: Dodavanje globalnih posrednika

### 6.3.7 Instanca Express aplikacije

*ExpressAppInstance* dekorator će biti samo tvornica *Inject* dekoratora s unaprijed zadanim tokenom. Implementaciju možemo vidjeti u kodu 6.37.

```
function ExpressAppInstance(): ParameterDecorator {
  return Inject('expressAppInstance');
}
```

Kod 6.37: Implementacija *ExpressAppInstance* dekoratora

Preostalo je još samo registrirati instancu Express aplikacije kao prilagođenu ovisnost s tokenom *expressAppInstance*. To ćemo učiniti tako da tu prilagođenu ovisnost dodamo u polje prilagođenih ovisnosti unutar *bootstrap* metode *Bootstrap* klase prije nego započnemo registraciju ovisnosti. Implementaciju možemo vidjeti u kodu 6.38.

```
class Bootstrap {
  bootstrap() {
    this.appProperties.customProviders = [
      ...(this.appProperties.customProviders || []),
      { token: 'expressAppInstance', instance: this.expressApp },
    ];
    //...
  }
}
```

Kod 6.38: Registracije instance Express aplikacije

# 7 | Zaključak

Motivirani nedostatkom strukture u JavaScript jeziku implementirali smo okvir za razvoj poslužiteljskih aplikacija u TypeScriptu koji omogućuje lakši razvoj strukturiranih i održivih aplikacija. Kod koji se često ponavlja generirat će se u pozadini što korisnicima okvira omogućuje da se usredotoče na glavnu svrhu njihove aplikacije. Također smo razvili mehanizam za automatsko ubrizgavanje ovisnosti što uvelike doprinosi održivosti aplikacija razvijenih pomoću ovog okvira, a uz to olakšava njihovo testiranje.

Iako je aplikacije moguće razvijati i bez ovog ili nekog sličnog okvira, prava snaga dodatnog sloja apstrakcije pokazat će se kada se ukaže potreba za nekom promjenom ili dodavanjem nove značajke, no nedostatak ovakvih okvira je manjak fleksibilnosti u odnosu na sloj oko kojeg je okvir razvijen. Iz tog razloga, prije odabira tehnologije potrebo je dobro proučiti što nam ona pruža i hoće li to biti dostatno za razvoj aplikacije.

# Literatura


[1] G. Erich, *Design patterns : elements of reusable object-oriented software*, Addison-Wesley, 1995.

[2] D. Gourley, B. Totty, M. Sayer, A. Aggarwal, S. Reddy, *HTTP: The Definitive Guide*, O'Reilly Media, Inc., Sebastopol, 2002.

[3] M. Haverbeke, *Eloquent JavaScript*, No Starch Press, 2018.

[4] A. Mardan, *Pro Express.js*, Apress, 2014.

[5] TypeScript dokumentacija dostupna na https://www.typescriptlang.org/

[6] Dokumentacija NestJS okvira dostupna na https://docs.nestjs.com/

[7] Dokumentacija Express.js okvira dostupna na https://expressjs.com/

# Sažetak

U ovom radu opisujemo tehnologije i pojmove potrebne za implementaciju okvira za izradu poslužiteljskih aplikacija u TypeScript jeziku te taj okvir implementiramo.

## Ključne riječi

okvir za razvoj, poslužiteljske aplikacije, TypeScript, JavaScript, HTTP, Express.js

# Backend framework in TypeScript


## Summary

In this paper, we describe technologies and terms necessary to implement backand framework in TypeScript and we implement it.



## Keywords

framework, backend, TypeScript, JavaScript, HTTP, Express.js

# Životopis

Rođen sam 1998. godine u Našicama. Srednju školu sam pohađao u Elektrotehničkoj i prometnoj školi Osijek do 2017. godine, kada sam upisao preddiplomski studij matematike i računarstva na Odjelu za matematiku Sveučilišta Josipa Jurja Strossmayera u Osijeku. Nakon završetka preddiplomskog studija 2020. godine, upisao sam se na diplomski studiji matematike; smjer Matematika i Računarstvo na istom odjelu.