UNIVERZITET METROPOLITAN BEOGRAD

IT255 - VEB SISTEMI 1

Umetanje zavisnosti

Lekcija 08

PRIRUČNIK ZA STUDENTE

IT255 - VEB SISTEMI 1

Lekcija 08

UMETANJE ZAVISNOSTI

UVOD

Lekcija će se fokusirati na veoma korišćeni koncept umetanja zavisnosti u savremenim aplikacijama.

Lekcija će se fokusirati na veoma korišćeni koncept umetanja zavisnosti u savremenim aplikacijama. O ovom konceptu će uskoro biti detaljno diskutovano.

Kako napreduje gradivo ovog predmeta i kako se aktuelnim lekcijama uvode novi koncepti, programi, koji prate izlaganje, biće sve veći i kompleksniji. Tako će se ubrzo javiti potreba da komponente komuniciraju sa ostalim modulima. U situaciji kada je za izvršavanje modula A neophodan modul B, kaže se da je B zavisnost za A.

Jedan od opštih načina za pristupanje zavisnostima predstavlja importovanje odgovarajuće datoteke. Na primer, ovo je deo listinga nekog modula *A.ts*:

```
//deo koda klase A.ts

import {B} from 'B'; // ovo je zavisnost

B.funkcija(); // koristi B
```

U brojnim slučajevima dovoljan je prikazani uvoz koda. Međutim, postoje i situacije kada je neophodno obezbediti zavisnosti na sofisticiraniji način. Na primer, možda je cilj da se u aplikaciji primeni neki od sledećih scenarija:

- zamena klase B klasom *MockB* tokom izvođenja testiranja;
- deljenje jedinstvene instance klase B preko cele aplikacije (*Singleton šablon - pattern*);
- kreiranje nove instance klase B svaki put kada se koristi (*Produkcioni šablon - Factory pattern*).

Navedene probleme je moguće rešiti primenom koncepta umetanja zavisnosti. Umetanje zavisnosti (dependency injection - DI) prestavlja sistem koji omogućava da delovi nekog programa budu dostupni ostalim delovima tog programa, a programeri podešavaju načine kako se to izvodi.

Izraz "umetanje zavisnosti" se istovremeno upotrebljava za opisivanje dizajn šablona (kojeg koriste brojni radni okviri) i specifične implementacije za *DI* ugrađene u *Angular* okvir.

Glavna korist primene umetanja zavisnosti ogleda se u činjenici da klijent komponenta ne mora da bude upućena u način kreiranja same zavisnosti. Sve što klijent komponenta mora da zna jeste kako da komunicira sa zavisnošću.

Za sada, studentima navedeno može biti apstraktno i nerazumljivo. Međutim, brzo će biti, kao i u ostalim lekcijama, uveden pokazni primer za lakše razumevanje ove važne problematike veb programiranja.

OPŠTI POGLED NA UMETANJE ZAVISNOSTI

Izlaganje u vezi sa umetanjem zavisnosti može biti apstraktno i nerazumljivo - neophodno ga je približiti na jednostavniji način.

Upravo iz razloga što studentima navedeno izlaganje u vezi sa umetanjem zavisnosti može biti apstraktno i nerazumljivo i pre uvođenja pokaznog primera za lakše razumevanje ove važne problematike veb programiranja, moguće je napraviti paralele sa realnim životnim situacijama.

Kako bi *DI* bilo primenjeno u stvarnom životu?

Uzmimo za primer da ste neko ko voli da putuje i za to angažujete agenciju, ili da ste na poziciji menadžera. Da biste organizovali putanje imali biste otprilike sledeće korake:

- kontaktirali agenciju za prodaju avio karata, tj. avio - prevoznika i rezervisali karte;
- da biste kupili karte morate fizički da odete do agenciju (u slučaju da nemaju online prodaju) i podignete karte;
- ukoliko agencija ima online prodaju, karte rezervišete, platite i onda ih morate preuzeti (osim ako ih nema na aerodromu);
- pozvali biste taksi i rezervisali vožnju do aerodroma u određeno vreme.

Naravno, sve ove korake morali biste da uradite i za koleginice, odnosno kolege koje putuju sa vama što je vremenski zahtevan posao.

Sa druge strane sve ove obaveze može za vas da uradi posebno odeljenje. Odeljenje koje unapred zna termine putovanja i vi dobijete kupljenu kartu na sto.

Oba ova scenarije su interakcija između vas kao klijenta i agencije kao pružaoca usluge, jedino što u drugom scenariju vas i vašu kompaniju mnogo manje košta vremena i novca a rezultat je isti. Vi imate zakazano putovanje i kupljene karte.

Ovim primerom želeli smo da predstavimo koncept umetanja zavisnosti u realnom životu, jer je karta koja je vama potrebna kreirana eksterno i vama dostavljena bez da ste vi pokrenuli neku akciju.

Šta je onda DI u softverskom domenu?

Skraćena verzija odgovora: davanje (ili dodeljivanje) instance objekta varijabli nekog drugog objekta. I to je to!

Ništa više od toga, ništa manje.

Poglavlje 1

Uvodni primer – PriceService

PRICESERVICE - DEFINICIJA KLASE

Izlaganje u vezi sa umetanjem zavisnosti može biti apstraktno i nerazumljivo - uvodi se primer.

Kao što je istaknuto u uvodnom izlaganju, studentima prikazano izlaganje u vezi sa umetanjem zavisnosti može biti apstraktno i nerazumljivo. Međutim, ovde će biti, kao i u ostalim lekcijama, uveden pokazni primer za lakše razumevanje ove važne problematike veb programiranja.

Na primer, kreira se Angular aplikacija prodavnice koja poseduje proizvode (*Products*) i jedan od njenih zadataka je računanje konačne cene proizvoda nakon uračunate takse. Sa ciljem računanja pune cene proizvoda neophodna je klasa *PriceService* koja poseduje adekvatnu metodu koja kao ulaz uzima:

- osnovnu cenu proizvoda (objekta *Product*);
- državu (*state*) u koju se proizvod prodaje;

a nakon toga vraća rezultat u formi konačne cene, računajući i taksu.

Sledećim listingom je prikazan kod ove klase. Potpuno urađen i proveren primer možete preuzeti na kraju izlaganja lekcije u aktivnosti *Shared Resources*. U urađenom primeru, ovaj listing se nalazi na lokaciji: *src/app/price-service-demo/price.service.1.ts*

```
export class PriceService {
  constructor() { }

  calculateTotalPrice(basePrice: number, state: string) {
    // npr. zamislite da se u realnoj aplikaciji pristupa bazi podataka
    // sa podacima o iznostu poreza
    const tax = Math.random();

    return basePrice + tax;
  }

}
```

U kreiranoj servisnoj klasi, funkcija *calculateTotalPrice()* preuzima osnovnu cenu (parametar *basePrice*) proizvoda i državu (parametar *state*) u koju se kupljeni proizvod šalje i vraća ukupnu cenu proizvoda.

PRIMENA SERVISA U MODELU

Servis je neophodno primeniti na konkretan objekat.

Servis, sam po sebi, ne radi ništa ukoliko nije povezan sa konkretnim objektom koji ga primenjuje. U konkretnom slučaju, kreirani servis se primenjuje na objekte modelske klase Product. Za početak neophodno je pokazati kako je to moguće uraditi bez umetanja zavisnosti. Sledi listing iz datoteke primera: *src/app/price-service-demo/product.model.1.ts:*

```
import { PriceService } from './price.service';

export class Product {
  service: PriceService;
  basePrice: number;

  constructor(basePrice: number) {
    this.service = new PriceService(); // <-- direktno kreirano ("hardcoded")
    this.basePrice = basePrice;
  }

  totalPrice(state: string) {
    return this.service.calculateTotalPrice(this.basePrice, state);
  }
}
```

Neka je sledeća zamisao da je potrebno kreirati test za upravo demonstriranu klasu i to je moguće ilustrovati sledećim listingom:

```
import { Product } from './product.model';

describe('Product', () => {
  let product;

  beforeEach(() => {
    product = new Product(11);
  });

  describe('price', () => {
    it('izračunato na osnovu osnovne cene i države', () => {
      expect(product.totalPrice('RS')).toBe(11.66); // -> ???
    });
  });
});
```

Problem ovog testa je u tome što ne postoji saznanje o tačnoj vrednosti takse za slanje kupljenog proizvoda ka Republici Srbiji ("RS"). Čak i ako *PriceService* bude implementiran na realan način API pozivom ili pozivom baze podataka, postojaće problem:

- *API* mora da bude dostupan ili pokrenuta baza podataka;
- Neophodna je informacija o tačnom iznosu takse za slanje kupljenog proizvoda ka Republici Srbiji u trenutku pisanja testa.

KREIRANJE INTERFEJSA

Ukoliko se testira metoda iz spoljašnjeg resursa moguće je simulirati (mock-ovati) servis.

U ovom delu izlaganja postavlja se ključno pitanje: Šta je neophodno učiniti ukoliko se testira metoda *product.totalPrice()* oslanjajući se na spoljašnji resurs? U ovakvim situacijama simulira se (*mock*) *PriceService.* Ukoliko je poznat interfejs za *PriceService,* moguće je napisati klasu *MockPriceService* koja će uvek obaviti predvidljivo izračunavanje i neće se oslanjati na bazu podataka ili API.

Neka se traženi interfejs naziva *IPriceService* i neka je njegova lokacija u priloženom urađenom projektu: *src/app/price-service-demo/price-service.interface.ts.* Sledi njegov listing:

```
export interface IPriceService {
  calculateTotalPrice(basePrice: number, state: string): number;
}
```

Posmatrani interfejs definiše samo jednu funkciju pod nazivom *calculateTotalPrice().* Sada je moguće kreirati pomenutu klasu *MockPriceService* koja je zapravo implementaciona klasa kreiranog interfejsa. Sledi njen listing koji se u projektu nalazi u datoteci: *src/app/price-service-demo/price.service.mock.ts.*

```
import { IPriceService } from './price-service.interface';

export class MockPriceService implements IPriceService {
  calculateTotalPrice(basePrice: number, state: string) {
    if (state === 'RS') {
      return basePrice + 0.66; // uvek je 66 centi!
    }

    return basePrice;
  }
}
```

Klasom je konkretizovana metoda i definisana je visina takse za slanje proizvoda koji je kupljen.

U nastavku izlaganja je neophodno posebnu pažnju posvetiti modifikaciji modelske klase *Product*.

MODIFIKOVANJE MODELA

Da bi Product objekti mogli da koriste inoviran servis, neophodno je modifiovati ovu klasu.

Na programu je neophodan dodatni rad. Iako je uspešno kreirana klasa *MockPriceService* klasa *Product* je još uvek ne koristi. Da bi *Product* objekti mogli da koriste ovakav servis, neophodno je obaviti modifikaciju modelske klase *Product*. Sledi modifikovan listing navedene klase iz datoteke *src/app/price-service-demo/product.model.ts*:

```
import { IPriceService } from './price-service.interface';

export class Product {
  service: IPriceService;
  basePrice: number;

  constructor(service: IPriceService, basePrice: number) {
    this.service = service; // <-- prosleđeno kao argument!
    this.basePrice = basePrice;
  }

  totalPrice(state: string) {
    return this.service.calculateTotalPrice(this.basePrice, state);
  }
}
```

Sada, prilikom kreiranja objekta *Product*, klijent koristeći klasu *Product* postaje odgovoran za odlučivanje koja konkretna implementacija servisa *PriceService* će biti data novoj instanci.

Nakon izvršene modifikacije modela, moguće je sada popraviti test i osloboditi se zavisnosti od nepredvidljivog *PriceService*-a (datoteka: */src/app/price-service-demo/product.spec.ts*):

```
import { Product } from './product.model';
import { MockPriceService } from './price.service.mock';

describe('Product', () => {
  let product;

  beforeEach(() => {
    const service = new MockPriceService();
    product = new Product(service, 11.00);
  });

  describe('price', () => {
    it('izračunato na osnovu osnovne cene i države', () => {
      expect(product.totalPrice('RS')).toBe(11.66);
    });
  });
});
```

U nastavku izlaganje, akcenat se stavlja na izučavanje novog modela po kojem se implementacija zavisnosti može menjati tokom vremena izvršavanja aplikacije. U literaturi, ovaj model je poznat pod nazivom model sistema Angular umetanja zavisnosti.

MODEL SISTEMA ANGULAR DI

Po ovom modelu implementacija zavisnosti može menjati tokom vremena izvršavanja aplikacije.

Ako se dodatno posveti pažnja prethodnom izlaganju moguće je zaključiti da sigurno postoje dobre strane testiranja modelskih klasa u izolaciji. To, pre svega, znači da sa sigurnošću može da se tvrdi da će klasa pravilno funkcionisati sa predvidljivim zavisnostima.

Iako je predvidljivost, u određenim situacijama, sasvim u redu, postavlja se nov problem. Šta se dešava sa prosleđivanjem konkretne implementacije servisa svaki put kada se kreira nov objekat klase *Product*? Angular DI biblioteka u velikoj meri pomaže prilikom rešavanja navedenog problema.

Unutar Angular DI sistema, umesto direktnog uvoza (*import*) i kreiranja nove instance klase, moguće je postupiti na sledeći način:

- registrovanje zavisnosti pomoću *Angular*-a;
- opisivanje kako će zavisnost biti umetnuta;
- umetanje zavisnosti.

Jedna prednost ovog modela je ta što se implementacija zavisnosti može menjati tokom vremena izvršavanja aplikacije (kao u prethodnom slučaju sa *mock* klasom). Međutim, značajnija prednost primene ovog modela je u činjenici da programer sam može da podesi kako će zavisnost biti kreirana.

Ovo je veoma čest slučaj u programima koji su servisno orijentisani - pri čemu se insistira na postojanju jedne instance koja je označena kao *Singleton* (jedinac). Primenom umetanja zavisnosti ovakvi objekti se lako podešavaju.

Treći slučaj primene umetanja zavisnosti predstavlja podešavanje aplikacije ili specifičnih promenljivih okruženja. Na primer, moguće je definisati *API_URL* kao konstantu, a potom obaviti umetanje različitih vrednosti u različitim fazama životnog ciklusa razvoja aplikacije: produkcija (*production*) ili razvoj (*development*).

Nakon obavljenog uvoda, najvažniji zadatak jeste da studenti nauče da kreiraju vlastite servise i različite načine kako da obave njihovo umetanje. Rešenje se nalazi u nastavku lekcije.

Poglavlje 2

Segmenti umetanja zavisnosti

TOKEN I DELOVI ZAVISNOSTI

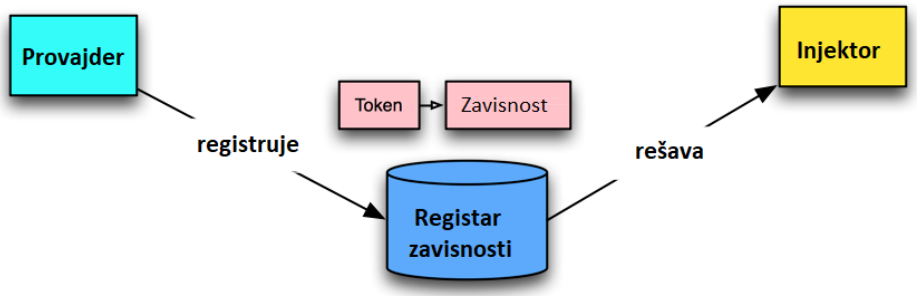
Za registrovanje zavisnosti neophodno je povezati zavisnost sa objektom koji će je identifikovati.

U Angular okviru, za registrovanje zavisnosti neophodno je povezati zavisnost sa nekim objektom koji će je identifikovati. Ova identifikacija je poznata kao token zavisnosti (*dependency token*). Na primer, neophodno je registrovati URL za API i tada je moguće koristiti string *API_URL* kao token. Na sličan način, ukoliko se registruje klasa moguće je koristiti upravo klasu kao vlastiti token.

Umetanje zavisnosti u Angular - u čine tri dela:

- *provajder (provider)* - povezuje token (koji može biti string ili klasa) sa listom zavisnosti. Navodi Angular kako da kreira objekat na osnovu tokena;
- *umetač ili injektor (injector)* - sadrži skup povezivanja (binding) i odgovoran je za rešavanje zavisnosti i njihovo umetanje prilikom kreiranja objekata;
- *zavisnost (dependency)* - vrednost koja je umetnuta.

Sledećom slikom je moguće ilustrovati uloge svakog navedenog dela umetanja zavisnosti u *Angular* okviru.



Slika 2.1.1 Angular - umetanje zavisnosti

Slika daje sledeće tumačenje: kada se podešava umetanje zavisnosti, specificira se šta je umetnuto (*inject*) i kako će biti rešeno (*resolve*).

Značaj i primenu umetača za implementaciju koncepta umetanja zavisnosti u *Angular* aplikacijama je posebno potrebno skenirati u narednom izlaganju.

2.1 Rad sa umetačima (Injector)

ULOGA UMETAČA

Angular koristi injektore za rešavanje zavisnosti i kreiranje promenljivih.

Umetač ili injektor (injector) je veoma važan koncept Angular *DI* modela kojeg je neophodno detaljno izložiti u narednom izlaganju.

U prethodnoj analizi, prilikom rada sa klasama *Product* i *PriceService*, instanca za *PriceService* je manuelno kreirana primenom operatora *new*. Šta Angular, zapravo, radi? Angular koristi injektore za rešavanje zavisnosti i kreiranje promenljivih. Ovo se sve dešava u pozadini. Međutim, za vežbu je odlično istražiti i pokazati šta se dešava. U tom kontekstu, moguće je obaviti manuelnu primenu injektora sa ciljem pokazivanja šta to, zapravo, Angular radi u pozadini.

Proširuje se tekući primer kreiranjem nove komponente *services*. Komponenta će posedovati funkcionalnosti koje omogućavaju manuelnu upotrebu injektora za rešavanje (izvršavanje) i kreiranje servisa.

Jedan od opštih načina primene servisa podrazumeva postojanje globalne jedinac (singleton) instance. Na primer, postroji klasa *UserService* koja sadrži informaciju u vezi sa trenutno prijavljenim korisnikom. Veliki broj različitih komponenata može da zahteva logiku baziranu na tekućem korisniku. Zbog navedenog, ovo je odličan primer servisa.

Sledećim listingom je data jednostavna *UserService* klasa koja poseduje objekat *user* kao osobinu. U projektu, klasa se nalazi na lokaciji */src/app/services/user.service.ts*:

```
import { Injectable } from '@angular/core';

@Injectable()
export class UserService {
  user: any;

  setUser(newUser) {
    this.user = newUser;
  }

  getUser(): any {
    return this.user;
  }
}
```

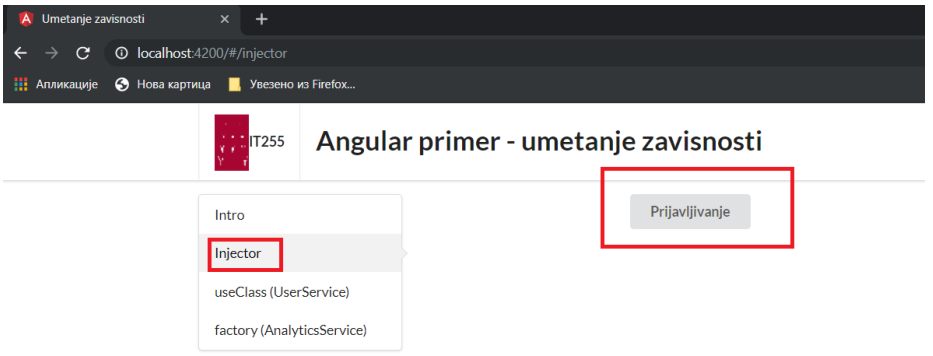
PRIMENA SERVISA U DRUGOJ KOMPONENTI

Proširuje se tekući primer kreiranjem nove komponente user-demo.

U drugoj komponenti (*user-demo*) neophodno je kreirati i odgovarajući šablon koji će omogućiti rad sa servisima korisnika. Neka šablon odgovara najjednostavnijoj mogućoj formi za prijavljivanje korisnika (videti sliku) koja sadrži samo jedno dugme "*Prijavljivanje*". Sledećim listingom je dat HTML kod opisanog šablona (datoteka: *src/app/user-demo/user-demo.component.html*):

```
<div>
  <p
    *ngIf="userName"
    class="welcome">
    Dobrodošli: {{ userName }}!
  </p>
  <button
    (click)="signIn()"
    class="ui button">
    Prijavljivanje
  </button>
</div>
```

Iz listinga je moguće primetiti da će klikom na dugme "Prijavljivanje" biti omogućeno prikazivanje pozdravne poruke za korisnika, baš kao na sledećoj slici:



Slika 2.2.1 Jednostavna forma za prijavljenog korisnika

DIREKTNA PRIMENA INJEKTORA

Funkcionalnost servisa je neophodno implementirati direktnom primenom injektora.

Funkcionalnost, prikazana prethodnom slikom, još uvek nije postignuta u trenutnom stanju primera, koji prati izlaganje lekcije. Traženu funkcionalnost je neophodno implementirati direktnom primenom injektora. Za tekuću komponentu *user-demo* kreira se klasa komponente *UserDemoInjectorComponent* na lokaciji u projektu: *src/app/user-demo/user-demo.injector.component.ts:*

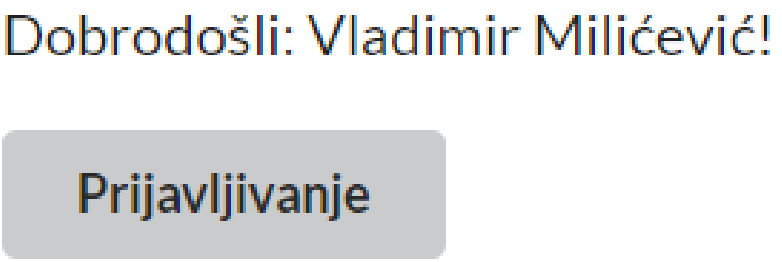
```
import {
  Component,
  ReflectiveInjector
 } from '@angular/core';
```

```
 import { UserService } from '../services/user.service';

@Component({
  selector: 'app-injector-demo',
  templateUrl: './user-demo.component.html',
  styleUrls: ['./user-demo.component.css']
})
export class UserDemoInjectorComponent {
  userName: string;
  userService: UserService;

  constructor() {
    // kreira injektor za kreiranje i rešavanje UserService
    const injector: any = ReflectiveInjector.resolveAndCreate([UserService]);

    // koristi injektor za dobijanje UserService instance
    this.userService = injector.get(UserService);
  }

  signIn(): void {
    // sa prijavljivanjem, podešava se korisnik

    this.userService.setUser({
      name: 'Vladimir Milićević'
    });

    // čitanje korisničkog imena iz servisa
    this.userName = this.userService.getUser().name;
    console.log('Korisničko ime je: ', this.userName);
  }
}
```

Kod započinje kao kod osnovne komponente: tu je selektor, šablon i CSS. Zatim, postoje dve osobine: *userName*, koja čuva naziv trenutno prijavljenog korisnika i *userService* koja čuva referencu ka *UserService*.

Posebni pažnju bi trebalo posvetiti konstruktoru. Iz koda se primećuje da on sadrži statičku metodu *resolveAndCreate()* klase *ReflectiveInjector*. Ova metoda je odgovorna za kreiranje novog injektora. Parametar koji se metodi prosleđuje predstavlja informacije koje injektor koristi za umetanje zavisnosti. U konkretnom slučaju, ukazuje se injektoru na klasu *UserService*.

Klasa *ReflectiveInjector* je konkretna implementacija apstraktne klase *Injector* koja koristi refleksiju za pretragu odgovarajućih tipova parametara.



Slika 2.2.2 Poruka prijavljenom korisniku

Poglavlje 3

Obezbeđivanje zavisnosti sa ngModule

TIPIČAN NAČIN PRIMENE INJEKTORA

Direktna primena injektora ne predstavlja tipičan način njihove primene u Angular okviru.

U prethodnom izlaganju da direktna primena injektora može biti veoma zanimljiva. Pa, ipak, ovo ne predstavlja tipičan način njihove primene u Angular okviru. Umesto toga, obično se radi sledeće:

- koristi se NgModule za registrovanje šta će biti umetnuto;
- koriste se dekoratori (obično za konstruktore) za specifikaciju šta će biti umetnuto.

Izvođenjem navedenih koraka Angular upravlja kreiranjem injektora i rešavanjem zavisnosti. Neophodno je krenuti od prvog koraka, pretpostavka je da je klasa *UserService* konvertovana u oblik da može biti umetnuta kao jedinac (singleton) na više pogodnih mesta u aplikaciji. Zbog toga je potrebno dodati je u *NgModule* klasu pod ključem providers. To je prikazano sledećim listingom (datoteka: */src/app/user-demo/user-demo.module.ts*)

```
import { NgModule } from '@angular/core';
import { CommonModule } from '@angular/common';

// uvde se uvozi UserService
import { UserService } from '../services/user.service';

@NgModule({
  imports: [
    CommonModule
  ],
  providers: [
    UserService // <-- registrovan pod ključem providers
  ],
  declarations: []
})
export class UserDemoModule { }
```

Sada je moguće umetnuti servis *UserService* u komponentu na pogodan način. To je prikazano sledećim listingom (datoteka: */src/app/user-demo/user-demo.component.ts*):

```
import { Component, OnInit } from '@angular/core';

```

```
import { UserService } from '../services/user.service';

@Component({
  selector: 'app-user-demo',
  templateUrl: './user-demo.component.html',
  styleUrls: ['./user-demo.component.css']
})
export class UserDemoComponent {
  userName: string;
  // uklonjen `userService` zbog dole skraćene def. konstruktora

  // Angular će umetnuti singleton instancu `UserService` ovde.
  // Biće podešena kao osobina `private`.
  constructor(private userService: UserService) {
    // prazno jer ovde nije ništa potrebno!
  }

  // ostaje isto...
  signIn(): void {
    // prilikom prijavljivanja, podešava se user
    this.userService.setUser({
      name: 'Vladimir Milićević'
    });

    // čitanje korisničkog imena iz servisa
    this.userName = this.userService.getUser().name;
    console.log('Korisničko ime je: ', this.userName);
  }
}
```

Prvo što je moguće primetiti iz listinga jeste da je unutar konstruktora klase *UserDemoComponent*, kao argument, dodata vrednost *userService: UserService.* Kada se ova komponenta kreira u odgovarajućoj stranici, Agular će rešiti i umetnuti *UserService singleton* objekat. Ono što je odlično, Angular upravlja instancom, nije potrebno da to uradi sam programer. Svaka klasa koja ubrizgava *UserService* dobiće isti singlton.

PROVAJDERI KAO KLJUČEVI

UserService je registrovan pod ključem providers u klasi NgModules.

Važno je znati da kada se dodaje *UserService* u konstruktor komponente *UserDemoComponent*, *Angular* će znati šta i kako treba da umetne. Ovo je moguće iz razloga što je *UserService* registrovan pod ključem providers u klasi *NgModules*.

Angular ne obavlja umetanje proizvoljnih klasa. Veoma je bitno podešavanje *NgModule* klase za omogućavanje funkcionisanja umetanja zavisnosti. Iako je već bilo dosta govora o singleton servisima neophodno je imati na umu da je njihovo umetanje u komponente *Angular* aplikacije moguće obaviti na više različitih načina. Upravi će o tome biti reči u nastavku lekcije.

Poglavlje 4

Provajderi

NAČINI REŠAVANJA UMETNUTIH ZAVISNOSTI

Postoji nekoliko načina na koje je moguće podesiti rešavanje umetnutih zavisnosti u radnom okviru Angular.

Postoji nekoliko načina na koje je moguće podesiti rešavanje umetnutih zavisnosti u radnom okviru *Angular*. Na primer, moguće je sledeće:

- umetnuti (*singleton*) instancu klase (pokazano je u prethodnom izlaganju);
- umetnuti vrednost;
- obaviti poziv funkcije i umetnuti vrednost posmatrane funkcije.

Neophodno je sagledati detalje za svaku prehodno navedenu stavku.

PRIMENA KLASE

Umetanje singleton instance klase je verovatno najčešći tip umetanja zavisnosti

Kao što je već diskutovano, umetanje singleton instance klase je verovatno najčešći tip umetanja zavisnosti. Tada se klasa dodaje u listu provajdera. To je moguće uraditi na sledeći način:

```
providers: [ UserService ]
```

Na ovaj način je ukazano Angular radnom okviru da je neophodno obezbediti singleton instancu klase *UserService* svaki put kada je ovaj servis injektovan. Zbog toga što je ovaj šablon opšti, klasa sama po sebi predstavlja skraćenu notaciju za sledeće, ekvivalentno podešavanje:

```
providers: [
 { provide: UserService, useClass: UserService }
  ]
```

Ono što je važno napomenuti jeste da podešavanje objekta, u drugom slučaju, zahteva primenu dva ključa: *provide* i *useClass*. Ključem *provide* je označen *token* koji se koristi za identifikovanje umetanja zavisnosti, a drugim ključem, *useClass*, određeno je kako i šta bi trebalo umetnuti.

U konkretnom slučaju vrši se mapiranje klase *UserService* sa tokenom *UserService*. Kao što se vidi, u ovom slučaju, token i naziv klase savršeno odgovaraju jedno drugom. Ove je opšti slučaj, ali je takođe bitno napomenuti da token i umetnuta klasa ne moraju, nužno, da imaju iste nazive.

Kao što je već diskutovano, injektor će kreirati singleton objekat u pozadini i vratiti istu instancu svaki put kada se vrši njeno umetanje. Naravno, kada je prvi put umetnuta, singleton instanca još nije kreirana. Zbog navedenog, kada se prvi put kreira *UserService* instanca, Angular DI sistem će pokrenuti konstruktor klase.

PRIMENA VREDNOSTI

Sledeći način umetanja zavisnosti predstavlja obezbeđivanje vrednosti

Kao što je već diskutovano, umetanje singleton instance klase je verovatno najčešći tip umetanja zavisnosti. Sledeći način, vredan diskusije, umetanja zavisnosti predstavlja *obezbeđivanje vrednosti*. Navedeno je veoma slično upotrebi globalnih promenljivih. Na primer, moguće je podešavanje URL-a krajnje API tačke (*API Endpoint URL*) u zavisnosti od okruženja. Da bi navedeno bilo realizovano, pod globalnim *NgModule* ključem *providers*, koristi se ključ *useValue,* na primer, na sledeći način:

```
providers: [
  { provide: 'API_URL', useValue: 'http://my.api.com/v1' }
  ]
```

Iz priloženog koda se primećuje da je obezbeđen token u formi stringa API_URL. Ukoliko se koristi string za obezbeđivanje vrednosti, Angular ne može da odluči koju zavisnost je neophodno rešiti po tipu podataka. Na primer, nije dozvoljena upotreba ovakvog koda.

```
// ovo ne radi - anti-primer
2 export class AnalyticsDemoComponent {
3 constructor(apiUrl: 'API_URL') { // <--- ovo nije tip podataka, samo običan string
4 // ako stavimo `string` to je dvosmisleno
5 }
6 }
```

Postavlja se pitanje: Šta se čini u ovakvim situacijama? Odgovor leži u primeni dekoratora *@Inject(),* na primer, na sledeći način:

```
import { Inject } from '@angular/core';

export class AnalyticsDemoComponent {
constructor(@Inject('API_URL') apiUrl: string) {
// radi! uradi nešto w/ apiUrl
 }
}
```

Sada, pošto je naučeno upravljanje jednostavnim vrednostima preko *useValue* i singleton klasama preko *useClass*, moguće je posvetiti se naprednim konceptima. Jedan od njih je, svakako, pisanje podesivih servisa primenom fabrikacije ili produkcije (*factory*).

PODESIVI SERVISI

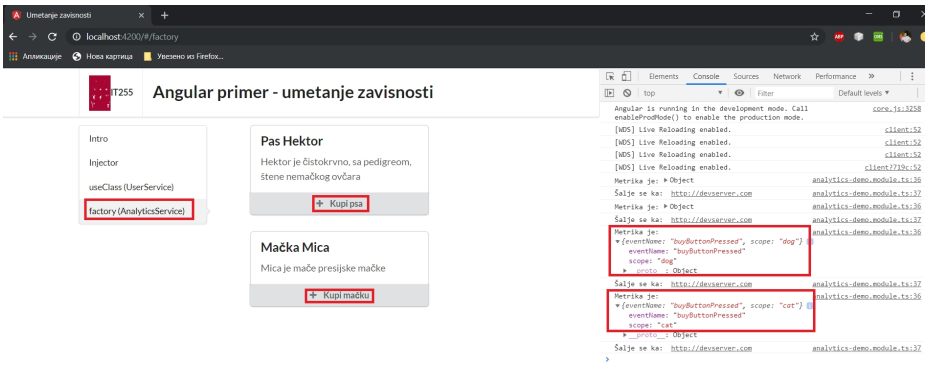
Podesivi servisi predstavljaju napredan koncept kojem je neophodno posvetiti posebnu pažnju.

Podesivi servisi (*Configurable Services*), kao što je istaknuto u prethodnom izlaganju, predstavljaju napredan koncept kojem je neophodno posvetiti posebnu pažnju.

U slučaju servisa *UserService*, nijedan argument nije potreban konstruktoru. Međutim, šta se dešava ukoliko konstruktor servisa zahteva neke argumente? Ovaj problem je moguće rešiti primenom produkcione funkcije (fabrikacije) koja predstavlja funkciju koja vraća bilo koji umetnuti objekat.

Na primer, kreira se biblioteka za beleženje analitike korisnika (čuvanje zapisa o događajima akcija koje je korisnik obavio na stranici). U ovom slučaju, neophodno je kreiranje servisne klase *AnalyticsService* koja definiše interfejs za čuvanje događaja ali ne i implementaciju za rukovanje događajima.

Sledećom slikom je prikazana funkcionalnost aktuelnog primera za praćenje analitike događaja



Slika 4.1 Praćenje analitike događaja

Korisnici aplikacije bi možda želeli da šalju i čuvaju vlastite metrike na Google Analytics (https://analytics.google.com/analytics/web/provision/?authuser=0#/provision) ili da koriste platformu poput *Optimizely* (www.optimizely.com/) ili neku drugu soluciju.

SERVIS ZA PRAĆENJE ANALITIKE

Servisne metode će biti definisane preko dva konkretna interfejsa koje će konkretizovati klasa servisa AnalyticsService.

U nastavku cilj je kreiranje servisa kojeg će biti moguće umetnuti i koji će definisati metode koje će konkretizovati implementaciona klasa servisa AnalyticsService. Servisne metode će biti definisane preko dva konkretna interfejsa, od kojih i započinje izlaganje ovog dela lekcije.

Prvo sledi definicija interfejsa *Metric* (datoteka: */src/app/analytics-demo/analytics-demo.interface.ts*):

```
/**
 * Definiše jednostavnu metriku
 **/
export interface Metric {
  eventName: string;
  scope: string;
}
```

Kreirani interfejs će poslužiti za čuvanje osobina *eventName* i *scope.* Za primer može da posluži sledeće, kada se korisnik "Vlada" poveže na aplikaciju, osobina *eventName* može da dobije vrednost *loggedIn*, a osobina scope vrednost "Vlada".

```
// ovo je samo primer
  let metric: Metric = {
  eventName: 'loggedIn',
  scope: 'Vlada'
  }
```

Na ovaj način je moguće, na primer, prebrojati korisnička povezivanja na aplikaciju preko osobine*eventNeme* i vrednosti *"loggedIn"*, kao i povezivanje specifičnih korisnika (vrednost "Vlada") na aplikaciju.

U nastavku je, takođe, potrebno definisati implementaciju analitike putem interfejsa *AnalyticsImplementation* (datoteka:*/src/app/analytics-demo/analytics-demo.interface.ts*):

```
/**
 * Definiše šta bi implementacija sačuvane metrike trebalo da bude
 **/
 export interface AnalyticsImplementation {
   recordEvent(metric: Metric): void;
 }
```

KLASA SERVISA ZA PRAĆENJE ANALITIKE

Neophodno je obaviti kreiranje implementacione klase za interfejs koja će koristiti koncept umetanja zavisnosti.

U prethodnom izlaganju je kreiran interfejs *AnalyticsImplementation* koji definiše jednu funkciju *recordEvent()*. Navedena funkcija uzima objekat tipa *Metrics* kao argument. U nastavku je neophodno kreiranoj metodi dati konkretno zaduženje. To se postiže kreiranjem implementacione klase za interfejskoja će koristiti koncept umetanja zavisnosti.

Klasa se naziva *AnalyticsService* i u priloženom projektu se nalazi na lokaciji */src/app/services/analytics.service.ts*:

```
import { Injectable } from '@angular/core';
import {
  Metric,
  AnalyticsImplementation
} from '../analytics-demo/analytics-demo.interface';

@Injectable()
export class AnalyticsService {
  constructor(private implementation: AnalyticsImplementation) {
  }

  record(metric: Metric): void {
    this.implementation.recordEvent(metric);
  }
}
```

Klasa definiše jednu metodu *record()* koja kao parametar uzima objekat tipa *Metric* i prosleđuje ga objektu tipa *AnalyticsImplementation*.

Takođe, zanimljivo je za primetiti kako konstruktor uzima izraz kao parametar. Ukoliko bi bilo pokušana regularna primena mehanizma umetanja *useClass*, u veb pregledaču bi se pojavila greška poput:

Cannot resolve all parameters for AnalyticsService.

Ovo se dešava iz razloga što nije obezbeđen injektor sa implementacijom neophodnom za konstruktor. Sa ciljem rešavanja navedenog problema, neophodno je podesiti provajdera da koristi fabrikaciju (factory).

PRIMENA FABRIKACIJE

Da bi servis AnaliticsService mogao da bude korišćen neophodno ga je registrovati u listi provajdera preko produkcione metode useFactory()

Ukratko rečeno, da bi servis *AnaliticsService* mogao da bude korišćen, neophodno je uraditi sledeće:

- kreirati implementaciju koja odgovara interfejsu *AnalyticsImplementation*;
- dodati servis u listu provajdera primenom produkcione metode (fabrikacije) *useFactory()*.

Sledećim listingom je prikazano kako se to radi (datoteka: *src/app/analytics-demo/analytics-demo.module.1.ts*):

```
import { NgModule } from '@angular/core';
import { CommonModule } from '@angular/common';
import {
  Metric,
```

```
  AnalyticsImplementation
} from './analytics-demo.interface';
import { AnalyticsService } from '../services/analytics.service';

@NgModule({
  imports: [
    CommonModule
  ],
  providers: [
    {
      // `AnalyticsService` je token koji se koristi za umetanje
      // tapamtite,  token je klasa, ali se koristi kao identifikator!
      provide: AnalyticsService,

      // useFactory je funkcija - svaki vraćeni rezultat funkcije
      // biće umetnut
      useFactory() {

        // kreiranje implementacije za log prikazivaneje događaja
        const loggingImplementation: AnalyticsImplementation = {
          recordEvent: (metric: Metric): void => {
            console.log('Metrika je:', metric);
          }
        };

        // kreiranje novog `AnalyticsService` sa implementacijom
        return new AnalyticsService(loggingImplementation);
      }
    }
  ],
  declarations: [ ]
})
export class AnalyticsDemoModule { }
```

PRIMENA FABRIKACIJE - DODATNO O SINTAKSI PROVAJDERA

Neophodno je izolovati jedan detalj sintakse provajdera.

Ako se pogleda detaljno prethodni listing, neophodno je izolovati jedan detalj sintakse provajdera.

```
providers: [
  { provide: AnalyticsService, useFactory: () => ... }
  ]
```

useFactory kao ključ preuzima funkciju i šta god da ta funkcija vrati biće injektovano.

Takođe, primećuje se da je obezbeđena i primena klase *AnalyticsService.* Ponovo, kada se koristi ključ provide na ovaj način, klasa *AnalyticsService* se koristi kao token za identifikaciju šta će biti ubrizgano (umetnuto ili injektovano).

Primenom *useFactory* kreira se objekat tipa *AnalyticsImplementation* koji poseduje jednu funkciju: *recordEvent()*. Preko ovo funkcije moguće je, na primer, podesiti šta će se desiti kada je događaj sačuvan.

Još jednom je moguće napomenuti da bi ovakva aplikacija verovatno, u realnim uslovima, poslala događaj na platformu Google Analytics ili neki drugi softver za vođenje i obradu evidencije događaja (*event logging software*).

Konačno, obavlja se instanciranje i vraćanje *AnalyticsService* objekta.

ZAVISNOSTI FABRIKACIJE

Ponekad će za funkcionisanje produkcionih metoda morati da postoje dodatne zavisnosti.

Primena produkcionih metoda (fabrikacija) je moćan način za implementaciju koncepta umetanja zavisnosti u *Angular* okviru jer je veoma širok spektar funkcionalnosti koje mogu da budu ugrađene u ovu funkciju. Ponekad će za funkcionisanje produkcionih metoda morati da postoje dodatne zavisnosti. Na primer, neophodno je podesiti klasu Ponekad će za funkcionisanje produkcionih metoda morati da postoje *AnalyticsImplementation* da upućuje HTTP zahteve za određeni URL. Sa ciljem realizovanja navedenog, neophodno je posedovati sledeće:

- Angulat Http klijent;
- Konkretan URL.

U klasi *AnalyticsDemoModule,* koja se čuva u tekućem projektu na lokaciji */src/app/analytics-demo/analytics-demo.module,* moguće je obaviti sledeća podešavanja u skladu sa navedenim smernicama:

```
import { NgModule } from '@angular/core';
import { CommonModule } from '@angular/common';
import {
  Metric,
  AnalyticsImplementation
} from './analytics-demo.interface';
import { AnalyticsService } from '../services/analytics.service';

// dodato je sledeće ->
import {
  HttpModule,
  Http
} from '@angular/http';

@NgModule({
  imports: [
```

```
    CommonModule,
    HttpModule, // <-- dodato
  ],
  providers: [
    // dodaje API_URL provajdera
    { provide: 'API_URL', useValue: 'http://devserver.com' },
    {
      provide: AnalyticsService,

      // dodaje zavisnosti za specifikaciju produkcionih (factory) zavisnosti
      deps: [ Http, 'API_URL' ],

      // ovde su dodati argumenti
      // redosled odgovara redosledu zavisnosti
      useFactory(http: Http, apiUrl: string) {

        // kreiranje implementacije za prikazivanje događaja u logu
        const loggingImplementation: AnalyticsImplementation = {
          recordEvent: (metric: Metric): void => {
            console.log('Metrika je:', metric);
            console.log('Šalje se ka: ', apiUrl);
            // ... slanje metrike primenom http  ...
          }
        };

        // kreiranje nove `AnalyticsService` sa implementacijom
        return new AnalyticsService(loggingImplementation);
      }
    },
  ],
  declarations: [ ]
})
export class AnalyticsDemoModule { }
```

U priloženom listingu je obavljeno dodavanje *HttpModule*, istovremeno putem *ES6* import naredbe (omogućava primene konstanti klase) u *NgModul* dekoratoru (omogućava da ova klasa bude dostupna za umetanje zavisnosti).

Dalje, kao provajder je dodat *API_URL*, a potom i *AnalyticsService.* Posebna pažnja bi trebalo da se posveti drugom navedenom provajderu. Ovde je dodat nov ključ: *depts* koji predstavlja niz tokena umetanja zavisnosti koji će biti rešeni (upareni) i prosleđeni kao argumenti produkcionim funkcijama (fabrikacijama).

VIDEO MATERIJAL 1

Angular Dependency Injection - Understanding Providers and Injection Tokens - trajanje 9:34

Ova lekcija sadrži video materijal. Ukoliko želite da pogledate ovaj video morate da otvorite LAMS lekciju.

Poglavlje 5

Umetanje zavisnoti u Angular aplikaciji

KORACI IZVOĐENJA UMETANJA ZAVISNOSTI

Postoje tri koraka koje je neophodno preći sa ciljem izvođenja umetanja zavisnosti.

Ako se detaljno prođe kroz objekte učenja ove lekcije, moguće je izvući sledeći zadatak - postoje tri koraka koje je neophodno preći sa ciljem izvođenja umetanja zavisnosti u Angular radnom okviru:

1. kreiranje zavisnosti (na primer, kreiranje servisne klase);
2. podešavanje umetanja zavisnosti (registrovanje u Angularu preko NgModule);
3. deklarisanje zavisnosti u komponenti koja je primenjuje.

Prva stvar koju je neophodno učiniti jeste kreiranje servisne klase. Ova klasa ispoljava neko ponašanje koje je neophodno upotrebiti u ostalim komponentama programa. Kreirana servisna klasa je obeležena dekoratorom *@Injectable()* zato što predstavlja resurs koji je dostupan ostalim komponentama aplikacije preko umetanja zavisnosti.

U zavisnosti od terminologije, provajderi obezbeđuju (kreiraju, instanciraju i tako dalje) servise za umetanje zavisnosti. U *Angular* okviru kada je neophodno pristupiti servisu, obavlja se umetanje zavisnosti u neku funkciju (često konstruktor) nakon čega će *Angular DI okvir* locirati i obezbediti traženu zavisnost.

Konačno, kao rezime lekcije, moguće je istaći da koncept umetanja zavisnosti obezbeđuje moćan način upravljanja zavisnostima u Angular veb aplikacijama.

Konačno, možete preuzeti i testirati prateći primer. Potpuno urađen i proveren primer možete preuzeti na kraju ovog objekta učenja u aktivnosti *Shared Resources* .

DODATNI POGLED NA KONCEPT UMETANJA ZAVISNOSTI

Angular preporučuje da komponente sadrže samo podatke specifične za šablone.

Angular preporučuje da komponente sadrže samo podatke specifične za šablone. Biznis logika i obrada podataka treba da se nalazi u servisima. U ovom slučaju komponente su korisnici servisa i služe da proslede šablonu podatke iz servisa za prikaz. Na neki način komponente samo delegiraju posao servisima i koriste njihove usluge. Da bi se klasa definisala kao servis ona mora da bude opisana dekoratorom @Injectable *()*. Servisi mogu da imaju zavisnost i od drugih servisa, funkcija ili vrednosti pa je tako Česta upotreba servisa koji zavise od servisa HttpClient koji služi za slanje zahteva ka zadatim adresama. Kada je kreirana instanca neke klase, obezbeđivanje svih elemenata od kojih ta klasa zavisi se zove umetanje zavisnosti (eng. *dependecy injection*).

U Angularu - u, "umetač" (eng. *injector*) održava skladište servisa i podataka od kojih zavisi funkcionisanje svakog pojedinačnog servisa. Ako neka komponenta zahteva korišćenje nekog servisa, umetač prvo proverava da li postoji instanca tog servisa i ako ne postoji kreira je, a zatim instancu servisa prosleđuje kroz provajdere (eng. *providers*) nazad komponenti na korišćenje. Umetač na osnovu konstruktora komponente određuje koji servisi su toj komponenti potrebni i u koju promenljivu da ih stavi, jer se servisi koji se koriste poslеduju kao argumenti konstruktora. U tom slučaju bi kreiranje instance komponente zavisilo i od servisa.

Ovim bi bilo završeno teorijsko izlaganje na temu primene i značaja koncepta umetanja zavisnosti u *Angular* aplikacijama i fokus se prebacuje na izradu konkretnih zadataka kroz pokazne i individualne vežbe.

VIDEO MATERIJAL 2

Angular 8 Tutorial - 18 - Dependency Injection - trajanje 9:24

Ova lekcija sadrži video materijal. Ukoliko želite da pogledate ovaj video morate da otvorite LAMS lekciju.

VIDEO MATERIJAL 3

EP 10.6 - Angular / Dependency Injection & Providers /Providers and viewProviders - 15:01

Ova lekcija sadrži video materijal. Ukoliko želite da pogledate ovaj video morate da otvorite LAMS lekciju.

Poglavlje 6

Vežbe 8

UMETANJE ZAVISNOSTI - POKAZNA VEŽBA

Cilj vežbe je da upozna studenta kreiranjem servisa i primenom umetanja zavisnosti.

Cilj ove pokazne vežbe jeste da se obuka studenata za korišćenje koncepta umetanja zavisnosti (*dependency injection*) zaokruži nakon teorijskog izlaganja prezentovanih u prethodnim objektima učenja. Izradom individualne vežbe, a posebno domaćeg zadatka, student će steći osnove za samostalno rukovanje ovom veoma važnom programerskom problematikom.

Vežba započinje kreiranjem odgovarajuće komponente koja će poslužiti kao osnov za implementaciju koncepta umetanja zavisnosti u pokaznoj vežbi.

Koristeći Angular CLI i sledeću komandu kreiraćemo *YoutubeService* unutar foldera *services* postojećeg projekta:

- *ng g service services/youtube*

Trebalo bi da sadržaj fajla *youtube.service.ts* bude kao u slučaju sledećeg listinga (slika 1):

src > app > services > TS youtube.service.ts > YoutubeService

```
1   import { Injectable } from '@angular/core';
2
3   @Injectable({
4     providedIn: 'root'
5   })
6   export class YoutubeService {
7
8     constructor() { }
9   }
10
```

Slika 6.1 Sadržaj fajla youtube.service.ts

Unutar *YouTube* servisa kreiraćemo pomoćnu funkciju za izvlačenje *ID*-a videa, kako bismo isti mogli ugraditi unutar *HTML*-a i tu logiku izmestili iz same komponente.

Neophodno je da naš servis bude dostupan na nivou čitave aplikacije, a to ćemo učiniti kroz *app.module.ts* gde ćemo unutar provider bloka dodati *YoutubeService* na sledeći način:

```
providers: [
    YoutubeService
  ],
```

DATOTEKA APP.MODULE.TS

Neophodno je izvesti dodatne korekcije nad glavnim modulom aplikacije.

Da bi sve bilo podešeno na odgovarajući način, neophodno je u nastavku prebaciti akcenat na glavni modul aplikacije - klasu *AppModule*.

Nad navedenom klasom je neophodno izvesti dodatne korekcije nad glavnim modulom aplikacije. Trenutni sadržaj datoteke *app.module.ts* odgovara sledećem listingu:

```
import { BrowserModule } from '@angular/platform-browser';
import { NgModule } from '@angular/core';

import { AppRoutingModule } from './app-routing.module';
import { AppComponent } from './app.component';
import { VideoBoxComponent } from './components/video-box/video-box.component';
import { FilterPipe } from './helpers/filter-pipe.pipe';
import { AddVideoComponent } from './components/add-video/add-video.component';
import { FormsModule, ReactiveFormsModule } from '@angular/forms';
import { YoutubeService } from './services/youtube.service';

@NgModule({
  declarations: [
    AppComponent,
    VideoBoxComponent,
    FilterPipe,
    AddVideoComponent
  ],
  imports: [
    BrowserModule,
    AppRoutingModule,
    FormsModule,
    ReactiveFormsModule
  ],
  providers: [
    YoutubeService
  ],
  bootstrap: [AppComponent]
})
export class AppModule { }
```

DATOTEKA YOUTUBESERVICE.TS

Neophodno je obaviti kreiranje novog servisa YoutubeService.

Ključni deo rada na implementiraju koncepta umetanja zavisnosti predstavlja kreiranje odgovarajućeg servisa.

Upravo iz navedenog razloga, biće obavljeno kreiranje novog servisa pod nazivom *YoutubeService*. Za početak je identifikovana i kodirana metoda *getEmbedLink()* sa sledećim sadržajem:

```
public getEmbedLink(link) {
    let ID = '';
    link =
link.replace(/(>|<)/gi,'').split(/(vi\/|v=|\/v\/|youtu\.be\/|\/embed\/)/);
    if(link[2] !== undefined) {
      ID = link[2].split(/[^0-9a-z_\-]/i);
      ID = ID[0];
    }
    else {
      ID = link;
    }
    return 'https://www.youtube.com/embed/' + ID;
  }
```

Veoma važno, u ovom delu analize jeste prikazivanje i sagledavanje servisne klase kao celine. Pa, kreiranjem prethodne metode zaokružuje se definicija posmatranog servisa i sledećim listingom je moguće dati njegov konačan izgled:

```
import { Injectable } from '@angular/core';

@Injectable({
  providedIn: 'root'
})
export class YoutubeService {

  constructor() { }

  public getEmbedLink(link) {
    let ID = '';
    link =
link.replace(/(>|<)/gi,'').split(/(vi\/|v=|\/v\/|youtu\.be\/|\/embed\/)/);
    if(link[2] !== undefined) {
      ID = link[2].split(/[^0-9a-z_\-]/i);
      ID = ID[0];
    }
    else {
      ID = link;
    }
    return 'https://www.youtube.com/embed/' + ID;
```

```
  }
}
```

INICIJALIZACIJA SERVISA YOUTUBESERVICE

Neophodno da ovaj servis inicijalizujemo unutar VideoBox komponente.

Konačno, neophodno je obaviti konačna podešavanja aplikacije, sa ciljem da umetanje zavisnosti u potpunosti da očekivane rezultate. Kada se ovo obavi neophodno je ponovo prevesti, pokrenuti aplikaciju i pratiti njeno izvršavanje.

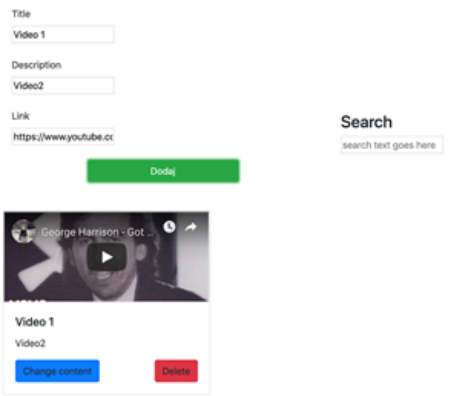
U sledećem koraku je neophodno da ovaj servis inicijalizujemo unutar *VideoBox* komponente na sledeći način:

```
constructor(private _sanitizer: DomSanitizer, private _youtubeService:
YoutubeService) {
```

A potom je neophodno i da našu funkciju *embedUrl()* modifikujemo na sledeći način:

```
public embedUrl() {
    this.link =
this._sanitizer.bypassSecurityTrustResourceUrl(this._youtubeService.getEmbedLink(thi
s.video.link));
  }
```

Rezultat izvršavanja aplikacije nakon popunjavanja odgovarajuće forme prikazan je na sledećoj slici:



Slika 6.2 Rezultat izvršavanja aplikacije

Kod vežbe je dostupan na sledećem linku: https://github.com/markorajevic/it255-v05/commit/e3dfc6005b9d7fca178c5942c4c8b919b426e1ed

INDIVIDUALNA VEŽBA

Na osnovu date ideje samostalno kreirajte i pokrenite Angular DI aplikaciju.

Za izradu individualne vežbe može poslužiti primer urađene aplikacije koja je demonstrirana na sledećem linku: https://www.codingame.com/playgrounds/8003/angular-dependency-injection-tutorial. Idejno rešenje aplikacije, može biti prikazano sledećom slikom:

Slika 6.3 Idejno rešenje aplikacije

Zadatak za samostalni rad:

1. **Istražite rešenje koje ne primenjuje koncept umetanja zavisnosti;**
2. **Kreirajterešenje koje prilikom kodiranja u potpunostiprimenjuje i uvažava koncept umetanja zavisnosti;**
3. **Napravite paralele i obrazložite ukratko prednosti drugog pristupa**.

INDIVIDUALNA VEŽBA - PRIMER BEZ UMETANJA ZAVISNOSTI

Doradite aplikaciju u skladu sa gradivom obrađenim u predavanjima i pokaznoj vežbi

Na osnovu inicijalnog primera koji ne uvažava koncept umetanja zavisnosti, napravite korekcije i doradite aplikaciju u skladu sa gradivom obrađenim u predavanjima i pokaznoj vežbi.

Zadatak:

Kompjuter čine sledeće komponente:

- *Monitor;*
- *CPU;*
- *KeyBoard*.

Da bi kompjuter softverski bio realizovan, neophodno je kreirati sledeće klase:

- *Computer;*
- *Monitor;*
- *CPU;*
- *Keyboard*.

Vaš prvi zadatak je kreiranje navedenih modelskih klasa na odgovarajućim lokacijama u projektu (*src/ app* direktorijum).

Predlog:

1. Klasa *Monitor* (datoteka *Monitor.ts*) ima jednu javnu osobinu koja može inicijalno biti podešena na sledeći način:

public monitorNo = 2;

2. Klasa *CPU* (datoteka *CPU.ts*) ima jednu javnu osobinu koja može inicijalno biti podešena na sledeći način:

public cpuNo = 3;

3. Klasa *Keyboard* (datoteka *Keyboard.ts*) ima jednu javnu osobinu koja može inicijalno biti podešena na sledeći način:
public keyboardNo = 1;

INDIVIDUALNA VEŽBA - PREDLOG IZGLEDA KLASE COMPUTER

Klasa Computer je agregatna klasa koja sadrži objekte prethodnih klasa.

Klasa *Computer* (datoteka: *Computer.ds*) je agregatna klasa koja sadrži objekte prethodnih klasa. Njen konstruktor može da ima sledeći oblik:

```
constructor() {
    this.monitor = new Monitor();
    this.cpu = new CPU();
    this.keyboard = new Keyboard();
  }
```

Predlaže se dodavanje sledeće metode klase *Computer* odmah iza konstruktora.

```
public description = 'This Matrix computer';
  complete() {
    return `${this.description} has ` +
      `${this.monitor.monitorNo} monitors, ${this.cpu.cpuNo} cpus and,
${this.keyboard.keyboardNo} keyboard.`;
  }
```

Za kompletiranje komponente *Computer* neophodno je importovanje kreiranih klasa *Monitor*, *CPU* i *KeyBoard* sa ciljem kreiranja potpuno kompletnog *Matrix* računara. Na osnovu konstruktora klase *Computer*, koji je priložen prvim listingom, vidi se da su kreirane tri instance klasa čiji je predlog dat u prethodnoj sekciji.

Ovde je bitno navesti da, po ovom scenariju, klasa Computer će biti u potpunosti zavisna od ostalih predloženih klasa: *Monitor*, *CPU* i *KeyBoard.* Budući da će kreiranje objekata biti obavljeno unutar konstruktora, objekti klasa: *Monitor*, *CPU* i *KeyBoard* ni na koji način nisu razdvojeni od klase *Compute*r.

Sledi predlog inicijalne strukture klase glavne komponente i njenog šablona.

Vaš zadatak: **Modifikovati predlog i primeniti umetanje zavisnosti, a to znači da je cilj da se kreiranje prikazanih instanci obavi na drugom mestu, a ne u konstruktoru klase Computer,**

Predlaže se sledeći sadržaj datoteke **app.component.ts:**

```
import { Component } from '@angular/core';
import { Computer } from './Computer';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {
    public computer: Computer;

    constructor(){
        this.computer = new Computer();
    }
  makeComputer(){
    return this.computer.complete();
  }
}
```

Predlaže se sledeći sadržaj datoteke **app.component.html**:

```
<div style="text-align:center">
  <h1>
      {{makeComputer()}}!!
  </h1>
</div>
```

Poglavlje 7

Domaći zadatak 8

DOMAĆI ZADATAK

Samostalna izrada domaćeg zadatka.

Na projekat nastavljen prethodnim domaćim zadatkom dodati sledeće funkcionalnosti koje uključuju DI:

1. dodati *RoomService* unutar kog ćete definisati dodatnu metodu *getPrice(numberOfNights: number)* i
2. izračunati cenu na osnovu broja noći.
3. koristiti koncept umetanja zavisnosti prilikom izrade DZ.

Domaći zadatak dodati na *Github* pod "*commitom*" *IT255-DZ08* i poslati obaveštenje predmetnom asistentu o postavljenom domaćem zadatku.

Poglavlje 8

Zaključak

ZAKLJUČAK

Lekcija se bavila konceptom umetanja zavisnosti u savremenim veb aplikacijama.

Cilj lekcije je bio fokusiranje na veoma korišćeni koncept umetanja zavisnosti u savremenim aplikacijama. O ovom konceptu je detaljno diskutovano u objektima učenja ove lekcije.

Posebno je istaknuto, a kasnije i u lekciji pokazano, kako napreduje gradivo ovog predmeta i kako se aktuelnim lekcijama uvode novi koncepti, programi, koji prate izlaganje, postaju sve veći i kompleksniji. U tom svetlu se i javila potreba da komponente komuniciraju sa ostalim modulima programa. U situaciji kada je za izvršavanje modula A neophodan modul B, kaže se da je B zavisnost za A.

Jedan od opštih načina za pristupanje zavisnostima predstavlja importovanje odgovarajuće datoteke, što je u brojnim slučajevima i dovoljno. Međutim, postoje i situacije kada je neophodno obezbediti zavisnosti na sofisticiraniji način. U lekciji je istaknuto da to može biti neki od sledećih scenarija:

- zamena klase B klasom *MockB* tokom izvođenja testiranja;
- deljenje jedinstvene instance klase B preko cele aplikacije (*Singleton šablon - pattern*);
- kreiranje nove instance klase B svaki put kada se koristi (*Produkcioni šablon - Factory pattern*).

Lekcija je identifikovala rešenje. Navedene probleme je moguće rešiti primenom koncepta umetanja zavisnosti. DI može biti definisano na sledeći način - Umetanje zavisnosti (dependency injection - DI) prestavlja sistem koji omogućava da delovi nekog programa budu dostupni ostalim delovima tog programa, a programeri podešavaju načine kako se to izvodi.

Posebno je pokazano da umetanje zavisnosti može da ima neki od sledeća dva oblika:

- opisivanje dizajn šablona (kojeg koriste brojni radni okviri) i
- specifična implementacije za *DI* ugrađena u *Angular* okvir.

Glavna korist primene umetanja zavisnosti ogleda se u činjenici da klijent komponenta ne mora da bude upućena u način kreiranja same zavisnosti. Sve što klijent komponenta mora da zna jeste kako da komunicira sa zavisnošću.

Za demonstraciju gradiva ove lekcije je pažljivo izabran i uveden pokazni primer za lakše razumevanje diskusije koja je, u početku, mogla studentima da bude apstraktna i nerazumljiva.

ZAKLJUČAK - DODATNA RAZMATRANJA

Kratak rezime primene servisa i koncepta umetanja zavisnosti.

Kratak rezime primene servisa i koncepta umetanja zavisnosti:

- Za podatke ili logiku koja nije povezana sa određenim prikazom, a koju želimo da podelimo sa više različitih komponenti, kreiramo servisnu klasu (*servis*);
- Umetanje zavisnosti (*Dependency injection*) je proces u kome jedna komponenta definiše zavisnost od drugih komponenata;
- Definiciji servisne klase uvek prethodi dekorator *@Injectable( )*;
- Dekorator pruža metapodatke koji omogućavaju servisu da ubaci u neku komponentu klijenta zavisnost (npr. komponenti koja pristupa veb serveru pomoću HTTP request, ubaciti HTTP servise u komponentu) .
- Injektiranje zavisnosti definiše i dinamički injektira objekat zavisnosti u drugi objekat, pa sve funkcije koje obezbeđuje objekat zavisnosti postaju dostupne.
- Angular obezbeđuje injektiranje zavisnosti pomoću *provajdera* i *servisa* injektora.
- Da bi se koristilo injektiranje zavisnosti na drugoj direktivi ili komponenti, potrebno je da se u okviru modula za aplikaciju doda naziv klase direktive ili komponente u metapodatke *declarations* u dekoratoru *@NgModule*, čime se niz direktiva uvozi u aplikaciju.

- Da bi se u *Angularu* uneli podaci u drugu direktivu ili komponentu, potrebno je da se uveze dekorator *Input* iz paketa *@angular/core*:

```
import {Component, Input} from '@angular/core';
```

- Kada se uveze ovaj dekorator, možemo da počnemo da definišemo podatke koje ćemo da unesemo u direktivu. Definisanje dekoratora se radi korišćenjem *@input()*, koji koristi string podatak kao parametar.

```
@Input('name') personName: string;
```

- U *Angular* okviru, za registrovanje zavisnosti neophodno je povezati zavisnost sa nekim objektom koji će je identifikovati. Ova identifikacija je poznata kao token zavisnosti (*dependency token*).
- U *Angularu* - u, "*umetač*" (eng. *injector*) održava skladište servisa i podataka od kojih zavisi funkcionisanje svakog pojedinačnog servisa.
- Provajder (*provider*) - povezuje token (koji može biti string ili klasa) sa listom zavisnosti. Navodi Angular kako da kreira objekat na osnovu tokena;
- *Umetač ili injektor (injector)* - sadrži skup povezivanja (binding) i odgovoran je za rešavanje zavisnosti i njihovo umetanje prilikom kreiranja objekata;
- Konačno, zavisnost (*dependency*) je vrednost koja je umetnuta.

LITERATURA

Za pripremu lekcije korišćena je aktuelna pisana i elektronska literatura.

Pisana literatura:

1. Nate Murray, Felipe Coury, Ari Lerner, Carlos Taborda, ng-Book – The Complete Book on Angular 6, Fullstack.io, 2018
2. Cody Lindley, Frontend Handbook – 2017, Frontend masters, 2017
3. Sandeep Panda, AngularJS – From Novice to Ninja, SitePoint Pty. Ltd, 2014

Elektronska literatura:

4. https://angular.io/
5. https://angular.io/tutorial
6. https://www.w3schools.com/angular/
7. https://www.tutorialspoint.com/angular4/
8. https://nodejs.org/en/
9. https://code.visualstudio.com/
10. https://www.w3schools.com/whatis/whatis_htmldom.asp
11. https://angular.io/guide/dependency-injection
12. http://www.ucim-programiranje.com/2013/02/dependency-injection-inversion-of-control/
13. https://www.codingame.com/playgrounds/8003/angular-dependency-injection-tutorial