UNIVERZITET METROPOLITAN BEOGRAD

IT355 - WEB SISTEMI 2

Spring IOC kontejner

Lekcija 02

PRIRUČNIK ZA STUDENTE

IT355 - WEB SISTEMI 2

Lekcija 02

SPRING IOC KONTEJNER

UVOD

Lekcija pokriva osnove centralne Spring komponente

Lekcija pokriva osnove centralne *Spring* komponente - *Spring IoC kontejnera*. Posebno će biti obrađene sledeće teme:

- *Spring IoC kontejner* (eng.*Spring IoC Container);*
- *inverzija kontrole* (eng. *Inversion of Control*);
- *umetanje zavisnosti* (eng. *Dependency Injection*):
- umetanje zavisnosti preko konstruktora, polja i *setter* metoda;
- automatsko povezivanje zavisnosti (eng. *Autowiring*);

Savladavanjem ove lekcije student će razumeti osnove Spring IoC kontejnera bez kojih nije moguće upustiti se u savladavanje tema koje slede. Materijal pokriva novije koncepte podešavanja Spring aplikacija koje su uvedene verzijom radnog okvira *Spring 4.*

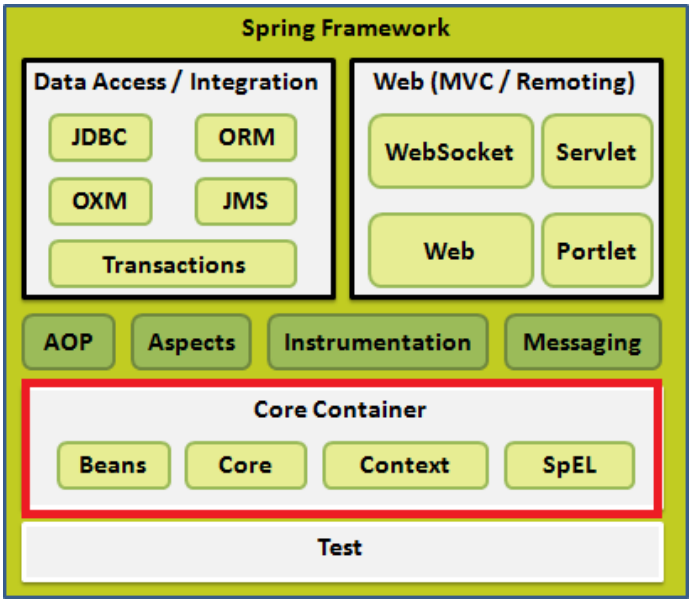
Poglavlje 1

Inverzija kontrole

POZICIONIRANJE IOC KONTEJNERA

Na samom početku biće prikazana organizacija Spring okvira

Spring okvir se može posmatrati kao jedna potpuno snabdevena prodavnica proizvodima neophodnim za razvoj JAVA veb aplikacija. Posebno treba istaći da je Spring modularan i da u zavisnosti od tipa aplikacije i primene, programeri biraju odgovarajuće module i integrišu ih u postojeće projekte. U ovom delu lekcije neophodno je dati prikaz svih modula koji su dostupni u Spring okviru. Spring sadrži oko 20 modula koji mogu biti upotrebljeni u zavisnosti od zahteva aplikacije. Sledećom slikom su prikazani Spring moduli, a posebno je istaknut *Core kontejner*.



Slika 1.1 Moduli Spring okvira [izvor: Spring]

Core kontejner ili IoC kontejner je centralni deo Spring okvira. Izgrađen je od: jezgra, zrna, konteksta aplikacije i jezika izraza. Jezgrom su određene osnovne funkcionalnosti Spring okvira, poput inverzije kontrole (IoC) i umetanja zavisnosti. Modul zrna obezbeđuje produkciju zrna u formi sofisticiranih šablona. Kontekst aplikacije je takođe ključni Spring koncept o kojem će posebno biti govora u temama koje slede. Spring omogućava pisanje izraza na sofisticiran način primenom vlastitoj jezika izraza - Spring SpEL.

Posebno grupisani moduli, odnose se na rad sa bazama podataka, na upravljanje transakcijama, rad sa porukama i brojnim drugim naprednim Spring funkcionalnostima - JDBC, ORM, OXM, JMS i Transactions.

Na kraju, Spring je u potpunosti u službi razvoja JAVA web aplikacija i, iz tog razloga, snabdeven je modulima: WebSocket, Servlet, Web i Portlet. Web modul obezbeđuje osnovne funkcionalnosti za rad na razvoju web aplikacija. Spring MVC sadrži implementaciju model - view - controler za web aplikacije. WebSocket je snabdeven alatima za razvoj klijent - server web aplikacija. Portlet modul je od značaja za razvoj specifičnih portlet aplikacija o kojima će biti govora u narednim lekcijama.

Posebno, Spring poseduje module koji su od velikog značaja za testiranje vebaplikacija, a posebno za razvoj aplikacija zasnovanih na najmlađoj paradigmi razvoja softvera: aspektno - orijentisanom programiranju (AOP)

KONCEPT INVERZIJE KONTROLE

Inverzija kontrole je osnovni Spring koncept

Inverzija kontrole (eng. Inversion of Control) predstavlja princip softverskog inženjerstva prema kojem je kontrola programskih objekata pomerena ka kontejneru ili radnom okviru (eng. framework). Ovaj koncept je specifičan za kontekst savremenih objektno - orijentisanih jezika.

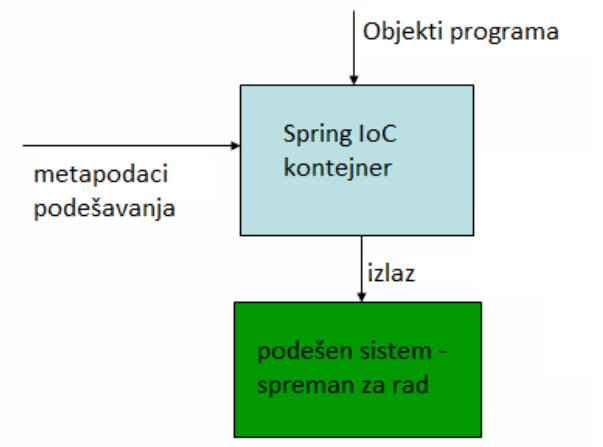
U suprotnosti sa tradicionalnim načinima programiranja i razvoja softvera, po kojima kreirani kod vrši pozive ka bibliotekama, inverzija kontrole (skraćeno IoC) omogućava okviru da preuzme kontrolu nad tokom programa i da vrši pozive u okviru kreiranog koda. Da bi navedeno bilo moguće, radni okvir koristi ugrađene apstrakcijedopunskog ponašanja, nekarakterističnog za standardni programski jezik nad kojim je podignut radni okvir. Ukoliko je cilj dodavanje novog ponašanja u programu, neophodno je proširiti (naslediti, eng *extends*) odgovarajuće klase radnog okvira ili dodataka (eng. *plugin*) treće strane u klasama kreiranog koda.

U našem slučaju, navedene zadatke će obavljati osnovni kontejner Spring okvira poznat pod nazivom Spring IoC kontejner.

Navedeno je moguće ilustrovati sledećom slikom.

Prednosti primene navedenog pristupa razvoju softvera su brojne. Sledi navođenje nekih najznačajnijih prednosti ovakve arhitekture:

- razdvajanje izvršavanja zadataka od njihove implementacije;
- lakši prelasci sa jedne implementacije na drugu;
- veći stepen modularnosti programa;
- olakšano testiranje programa usled mogućnost izolovanja komponenata, ili simuliranja (eng. *mocking*) njihovih zavisnosti, kao i njihovog komuniciranja preko ugovora (eng. *contracts*).



Slika 1.2 IoC - ilustracija [izvor: autor]

Inverzija kontrole je dostupna, u Spring radnom okviru, kroz različite mehanizme. Najznačajniji mehanizam za ovaj deo izlaganja je umetanje zavisnosti (eng. *dependency injection*).

Poglavlje 2

Umetanje zavisnosti

ŠABLON UMETANJA ZAVISNOSTI

Umetanje zavisnosti je šablon implementacije inverzije kontrole.

Umetanje zavisnosti (eng. Dependency injection - DI) predstavlja šablon (eng. *pattern*) kroz koji se implementira IoC, gde se inverzija kontrole dešava preko podešavanja zavisnosti objekata.

Čin povezivanja objekata sa drugim objektima, ili kako se to u programerskom žargonu kaže "umetanje objekata u druge objekte", prepušteno je kontejneru umesto samim objektima.

U nastavku, da bi problematika bila lakše shvaćena, biće napravljena paralela između kreiranja objektne zavisnosti na tradicionalni način i pomoću umetanja zavisnosti. Sledeći listing pokazuje kreiranje zavisnosti objekata na tradicionalni objektno - orijentisani način.

```
public class Store {
    private Item item;

    public Store() {
        item = new ItemImpl1();
    }
}
```

Moguće je primetiti, u primeru iznad, da je bilo neophodno instanciranje implementacione klase interfejsa *Item*.

Neka je sada fokus na primeni umetanja zavisnosti za kreiranje zavisnosti između objekata. Prethodni primer dobija sledeći oblik:

```
public class Store {
    private Item item;
    public Store(Item item) {
        this.item = item;
    }
}
```

Kao što je moguće primetiti primenom *DI* nije bilo potrebno instanciranje implementacione klase interfejsa *Item*.

U narednim izlaganjima biće pokazano kako je moguće obezbediti implementaciju navedenog interfejsa kroz metapodatke.

Gledano iz perspektive razvoja, *IoC* i *DI* predstavljaju jednostavne koncepte. Međutim, ovi koncepti ostavljaju duboke posledice za strukturu projektovanog softverskog sistema. Otuda je od velikog značaja njihovo kvalitetno razumevanje.

Poglavlje 3

DI i IoC - pokazni primer

KREIRANJE INTERFEJSA I IMPLEMENTACIONIH KLASA

Uvodi se primer za razumevanje osnova IoC i DI u Springu.

Problematika koja je "načeta" u prethodnom izlaganju, a tiče se koncepata IoC i DI, biće ilustrovana kroz konkretan primer, a u svrhu lakšeg razumevanja.

Kreira se jednostavna Spring aplikacija zasnovana na jednostavnoj razmeni poruka. Za početak, dat je listing jednostavnog interfejsa:

```
// Interface HelloWorld
public interface HelloWorld {
    public void sayHello();
}
```

Da bi aplikacija mogla da koristi zrna koja redefinišu metode kreiranog interfejsa, neophodno je kreirati jednu ili više implementacionih klasa ovog interfejsa. Za početak neka to bude klasa pod nazivom *SpringHelloWorld1.*

```
// Prva implementaciona klasa
public class SpringHelloWorld implements HelloWorld {
   public void sayHello()  {
           System.out.println("Prva klasa kaže Hello!");
   }
}
```

U nastavku kreira se još jedna klasa pod nazivom *SpringHelloWorld2*. Sledi njen listing.

```
// Druga implementaciona klasa
public class SpringHelloWorld2 implements HelloWorld {
   public void sayHello()  {
           System.out.println("Druga klasa kaže Hello!");
   }
}
```

KREIRANJE SERVISNE KLASE I ANALIZA

Kreira se klasa koja upravlja kreiranjem zrna.

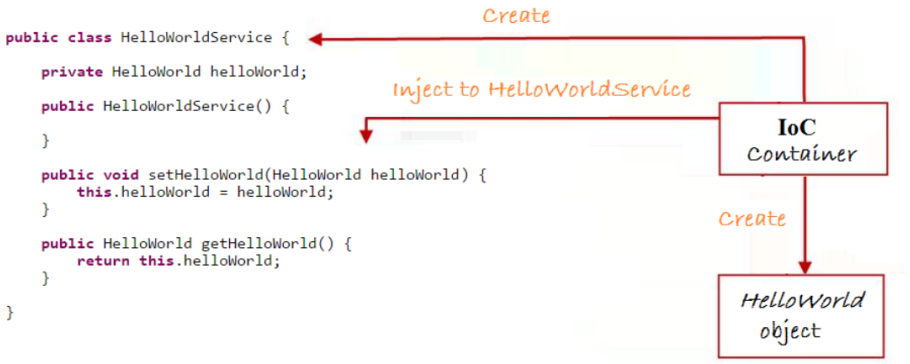
Nakon što je kreiran interfejs i njegove implementacione klase, pristupa se kreiranju klase čiji je zadatak da upravlja procesom kreiranja *HelloWorld* objekata. Ovakva klasa ima često oblik servisa i njen listing može imati sledeći oblik.

```
// Servisna klasa
public class HelloWorldService {

     // Polje tipa HelloWorld
     private HelloWorld helloWorld;

     // Konstruktor HelloWorldService
     // Inicijalizuje vrednosti za polje 'helloWorld'
     public HelloWorldService()  {
           this.helloWorld = new SpringHelloWorld2();
     }

}
```

Na osnovu priloženog koda je očigledno da klasa *HelloWorldService* upravlja procesom kreiranja objekata izvedenih iz interfejsa *HelloWorld*. Šta se zapravo dešava? Objekat klase *HelloWorldService* će biti kreiran angažovanjem njenog konstruktora (linija koda 9). *HelloWorld* objekat će takođe biti kreiran. Za njegovo kreiranje je upotrebljen konstruktor implementacione klase *SpringHelloWorld2* (linija koda 10).

Sledeći problem može biti kreiranje objekta servisne klase *HelloWorldService* sa istovremenim kreiranjem još jednog objekta, za interfejs *HelloWorld*, koji mora biti tipa *SpringHelloWorld1*.

Dakle, *HelloWorldService* kontroliše kreiranje objekata za *HelloWorld.* Da li se ovo dešava baš ovde ili na nekom drugom mestu? Tu dolazimo do suštine koncepta *inverzija kontrole (IoC)*. Spring IoC kontejner ima ulogu menadžera tj. on će obaviti kreiranje oba objekta.



Slika 3.1 DI i IoC na delu [izvor: Spring]

Spring IoC kontejner kreira *HelloWorldService* objekat, a zatim prosleđuje objekat *HelloWorld* u *HelloWorldService* preko *setter* metode. Tada *Spring IoC container* radi ono što je nazvano "umetanje zavisnosti" u objekte tipa *HelloWorldService*. U ovom kontekstu, zavisnost podrazumeva međusobnu zavisnost između objekata: *HelloWorldService* i *HelloWorld*.

Nakon ovog izlaganja, još jednom je podvučeno šta su i kako deluju koncepti inverzije kontrole i umetanja zavisnosti. U nastavku će biti više detalja u vezi sa njihovom primenom i funkcionisanjem.

Poglavlje 4

Spring IoC Container

INSTANCIRANJE IOC KONTEJNERA

U razvojnom okviru Spring, IoC kontejner je reprezentovan interfejsom ApplicationContext.

IoC kontejner je opšta karakteristika razvojnih okvira, pa tako i okvira Spring, za implementiranje koncepta IoC.

U razvojnom okviru *Spring*, *IoC kontejner* je reprezentovan interfejsom ApplicationContext. Spring kontejner je direktno odgovoran za instanciranje, podešavanje i povezivanje objekata, koji se u *Java EE* (*Enterprise Edition*) i *Spring* žargonu nazivaju zrnima (eng. *beans*). Ovaj kontejner, takođe, koristi se i za upravljanje životnim ciklusima ovih specifičnih Java objekata.

Spring okvir obezbeđuje nekoliko implementacija interfejsa *ApplicationContext:*

- *ClassPathXmlApplicationContext ,*
- *FileSystemXmlApplicationContext*
- *WebApplicationContext.*

Prve dve klase su korisne za razvoj konzolnih Java aplikacija, dok je treća značajna prilikom razvoja veb aplikacija. Navedene klase učitavaju podešavanja aplikacije iz odgovarajućih XML konfiguracionih fajlova. U najnovijim verzijama Spring okvira insistira se na Java konfiguracijama umesto navedenih XML datoteka. Ovde se koriste konfiguracione Java klase obeležene anotacijom *@Configuration.* Za učitavanje ovakvih klasa zadužena je implementaciona klasa, interfejsa *ApplicationContext*, pod nazivom *AnnotationConfigApplicationContext.*

Sa ciljem učitavanja zrna, koja se koriste u aplikaciji, kontejner koristi konfiguracione metapodatke koji, kao što je rečeno, u obliku XML konfiguracija ili Java anotacija.

Na sledeći način je moguće ručno podesiti IoC kontejner. Metapodaci se čitaju iz datoteke pod nazivom *applicationConfig.xml.*

```
ApplicationContext context
  = new ClassPathXmlApplicationContext("applicationContext.xml");
```

Ovaj zadatak je moguće obaviti na savremeniji način čitajući metapodatke iz konfiguracione klase *ApplicationConfig.java* primenom Java anotacija *@Configuration* i *@Bean,* na sledeći način:

```
ApplicationContext context =
       new AnnotationConfigApplicationContext(ApplicationConfig.class);
```

Poglavlje 5

Umetanje zavisnosti u Springu

TIPOVI UMETANJA ZAVISNOSTI U SPRING OKVIRU

Umetanje zavisnosti je moguće obaviti na više načina u Spring okviru.

Diskusija će biti nastavljena na prethodno uvedenom primeru. Za podešavanje atributa *item* moguće je koristiti metapodatke, kao što je to već navedeno u prethodnom izlaganju. Spring IoC kontejner čita navedene podatke, iz XML ili Java konfiguracija, i koristi ih za angažovanje zrna tokom vremena izvršavanja programa.

Umetanje zavisnosti u Springu moguće je obaviti na više načina:

- preko konstruktora;
- preko polja;
- preko setter metoda.

UMETANJE ZAVISNOSTI PREKO KONSTRUKTORA

Kontejner poziva konstruktor čiji argumenti reprezentuju zavisnosti koje je neophodno podesiti.

U slučaju umetanja zavisnosti primenom konstruktora,*Spring IoC kontejner* će pozvati konstruktor čiji argumenti predstavljaju zavisnosti koje moraju biti podešene.

Spring analizira svaki argument primarno po tipu, zatim po nazivu atributa i vrednosti indeksa. U sledećem listingu, data je konfiguracija zrna i njegovih zavisnosti primenom anotacija:

```
@Configuration
public class ApplicationConfig {

    @Bean
    public Item item1() {
        return new ItemImpl1();
    }

    @Bean
    public Store store() {
        return new Store(item1());
    }
}
```

Neophodno je primetiti da je navedena klasa pozivana u prethodnom izlaganju prilikom instanciranja IoC kontejnera.

Anotacija *@Configuration* ukazuje da je kreirana klasa izvor definicija zrna. Takođe, u okviru Spring projekta moguće je postojanje više konfiguracionih klasa.

Anotacija *@Bean* se koristi za metode kojima se definiše samo zrno. Ukoliko se ne specificira eksplicitno naziv zrna on će podrazumevano odgovarati nazivu odgovarajuće metode.

Za zrna čija je oblast važenja *singleton*, Spring će prvo proveriti da li već postoji keširana instanca zrna i tek kada se uveri da ne postoji, kreiraće novu instancu zrna. Ukoliko zrno pripada oblasti *prototype*, kontejner će vratiti novu instancu zrna za svaki novi poziv metode.

Kao što je već bilo govora, drugi način za kreiranje konfiguracije zrna jeste primena XML konfiguracionih datoteka. Prethodnoj klasi je ekvivalentan sledeći XML kod:

```
<bean id="item1" class="com.metropolitan.store.ItemImpl1" />
<bean id="store" class="com.metropolitan.store.Store">
    <constructor-arg type="ItemImpl1" index="0" name="item" ref="item1" />
</bean>
```

Moguće je primetiti, iz priloženog XML listinga, da nazivi klasa zrna moraju da se navedu pod punim imenom, zajedno sa paketom kojem pripadaju.

Pored navedenih singleton i prototype oblasti zrna, često je savljaju i oblasti: request (pokriva definiciju zrna tokom jednog HTTP zahteva), session (pokriva definiciju zrna tokom jedne HTTP sesije) i global-session(pokriva definiciju zrna tokomglobalne HTTP sesije). Ove oblasti su karakteristične za Spring veb aplikacije.

UMETANJE ZAVISNOSTI PREKO SETTER METODE

Kontejner poziva setter metodu odgovarajuće klase.

Kada se primenjuje umetanje zavisnosti bazirano na pozivu *setter* metode, kontejner će pozvati *setter* metode kreirane klase, nakon pozivanja podrazumevanog (bez argumenata) konstruktora ili statičkih (bez argumenata) produkcionih (eng. *factory*) metoda kojima se kreiraju instance zrna. Primenom anotacija moguće je na sledeći način kreirati odgovarajuću konfiguraciju zrna, unutar konfiguracione klase:

```
@Bean
public Store store() {
    Store store = new Store();
    store.setItem(item1());
    return store;
}
```

Po analogiji, moguće je koristiti i stariji pristup. Odgovarajuća XML konfiguracija može biti napisana na sledeći način:

```
<bean id="store" class="com.metropolitan.store.Store">
    <property name="item" ref="item1" />
</bean>
```

> Umetanje zavisnoti putem konstruktora ili primenom setter metoda moguće je kombinovati unutar istog zrna. Spring dokumentacija preporučuje primenu umetanja zavisnoti putem konstruktora za glavne zavisnosti, a primenu umetanja zavisnoti putem setter metoda za opcionalne.

UMETANJE ZAVISNOSTI PREKO POLJA KLASE

Objekat koji se injektuje jednostavno se obeležava anotacijom @Autowired.

U slučaju kada se dešava umetanje zavisnosti preko polja, objekat koji se injektuje i koji odgovara polju klase, jednostavno se obeležava anotacijom @Autowired. Ovaj princip umetanja se još naziva i automatskim povezivanjem zavisnosti.

Sledećim listingom je pokazana primena *@Autowired* anotacije u okviru klase *Store*.

```
public class Store {
    @Autowired
    private Item item;
}
```

Šta se, zapravo, dešava u ovom slučaju?

Prilikom kreiranja objekta klase *Store*, ukoliko ne postoji konstruktor ili *setter* metoda za umetanje zrna, kontejner će primeniti refleksiju za umetanje Item instance u objekat klase *Store*.

Takođe, automatsko povezivanje je moguće obaviti i primenom XML anotacija, a detaljnije o ovome je moguće pročitati na sledećem linku: http://websystique.com/spring/spring-beans-auto-wiring-example-using-xml-configuration/.

Iako ovaj pristup izgleda jednostavnije i čistije nego prethodna dva, preporučuje se da se koristi sa velikim stepenom opreznosti. Postoje dva ključna razloga zbog čega se to preporučuje:

- Ovaj metod koristi refleksiju za umetanje zavisnosti, a to je sistemski zahtevnije nego što je slučaj sa pozivom konstruktora ili setter metode;
- Veoma je lako dodati višestruke zavisnosti primenom ovog pristupa. Ukoliko su pre toga korišćeni konstruktori sa više argumenata, u ovom slučaju može se učiniti da klasa obavlja više dužnosti, a to se kosi sa principom jednostruke dužnosti klase (*Single Responsibility Principle*).

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

AUTOMATSKO POVEZIVANJE ZAVISNOSTI

Spring kontejner može automatski da poveže zrna po nekoliko scenarija.

Povezivanje (eng. *wiring*) omogućava *Spring* kontejneru da automatski reši (eng. *resolve*) zavisnosti između zrna koja sarađuju putem provere definisanih zrna.

Automatsko povezivanje zavisnosti postoji i u starijim verzijama Spring radnog okvira. Bilo je dostupno i putem XML konfiguracija pri čemu su se jasno izdvojili sledeći tipovi automatskog povezivanja:

- *no* - ovo je podrazumevana vrednost. Ovo znači da se automatsko povezivanje zrna ne koristi i da programer mora jasno da navede zavisnosti;
- *byName* - automatsko povezivanje se obavlja prepoznavanjem naziva osobine. Iz navedenog razloga Spring kontejner će tražiti zrno sa istim nazivom kao što je naziv osobine koja mora da bude podešena;
- *byType* - automatsko povezivanje se obavlja prepoznavanjem tipa osobine. Iz navedenog razloga Spring kontejner će tražiti zrno sa istim tipom kao što je tip osobine koja mora da bude podešena;
- *contructor* - automatsko povezivanje zavisnosti se oslanja na argumente konstruktora. Ovo praktično znači da će Spring kontejner tražiti zrna čiji tip podataka odgovara tipovima argumenata konstruktora.

Sada je moguće nastaviti demonstraciju primenom postojećeg primera za automatsko povezivanje *item1* zrna unutar zrna klase *Store*.

```
@Bean(autowire = Autowire.BY_TYPE)
public class Store {

    private Item item;

    public setItem(Item item){
        this.item = item;
    }
}
```

Automatsko povezivanje zavisnosti je takođe moguće primeniti i direktno na umetanje zrna:

```
public class Store {

    @Autowired
    private Item item;
}
```

Ukoliko postoji više zrna istog tipa, koji su kandidati za automatsko povezivanje, mogući konflikt je moguće rešiti primenom kvalifikatora. To znači da zrno koje se povezuje biće obeleženo anotacijom *@Qualifier,* a to praktično odgovara povezivanju po imenu (sledeći listing).

```
public class Store {

    @Autowired
    @Qualifier("item1")
    private Item item;
}
```

AUTOMATSKO POVEZIVANJE ZAVISNOSTI - XML KONFIGURACIJE

Brojne aplikacije, izgrađene pre, uvođenja podrške anotacijama koriste XML konfiguracije.

U prethodnim primerima je analizirano i prikazano automatsko povezivanje zavisnosti kroz Java konfiguracije. To svakako predstavlja aktuelniji i preporučeni pristup u savremenoj primeni Spring okvira. Međutim, postoje brojne aplikacije izgrađene pre uvođenja podrške anotacijama i koje je neophodno održati funkcionalnim još izvesno vreme. Otuda je veoma korisno ovladati i starijim Spring pristupom koji podrazumeva korišćenje XML konfiguracija za upravljanje Spring IoC kontejnerom.

Sledećim listingom je prikazano automatsko povezivanje po tipu primenom XML konfiguracija.

```
<bean id="store" class="com.metropolitan.store.Store" autowire="byType"> </bean>
```

Sada je moguće obaviti i automatsko povezivanje zrna po nazivu primenom XML konfiguracija.

```
<bean id="item" class="com.metropolitan.store.ItemImpl1" />

<bean id="store" class="com.metropolitan.store.Store" autowire="byName">
</bean>
```

U ovom slučaju dešava se umetanje zrna *item* u istoimenu osobinu zrna *store*.

Takođe, moguće je redefinisati primenjeno automatsko povezivanje putem definisanja zavisnosti eksplicitno preko argumenata konstruktora ili setter metoda.

PRIMENA LENJIH ZRNA

Ovim pristupom skraćeno je vreme neophodno za učitavanje aplikacije.

Po osnovnim podešavanjima, *Spring IoC kontejner* kreira i podešava sva singleton zrna tokom procesa inicijalizacije aplikacije. Da bi ovo bilo sprečeno, moguće je koristiti atribut *lazy-init* sa vrednošću *true* u konfiguraciji zrna. Navedeno je moguće prikazati sledećim XML kodom:

```
<bean id="item1" class="com.metropolitan.store.ItemImpl1" lazy-init="true" />
```

Navedeno ima za posledicu da će zrno *item1* biti inicijalizovano tek onda kada se realizuje prvi zahtev za ovim zrnom, a nikako tokom samog pokretanja aplikacije. Prednost ovoga je skraćeno vreme neophodno za učitavanje aplikacije. Međutim, moguće je uočiti i izvesne nedostatke ovog pristupa. Na ovaj način, konfiguracione greške je moguće otkriti tek kada se ukaže potreba za zrnom, a to može biti i satima ili danima nakon puštanja same aplikacije u rad.

Ovaj pristup je podržan i unapređen primenom anotacije *@Lazy* u okviru konfiguracione klase.

```
@Lazy
@Configuration
public class ApplicationConfig {

    @Bean
    public Item item1() {
        return new ItemImpl1();
    }

    @Bean
    public Store store() {
        return new Store(item1());
    }
}
```

Poglavlje 6

Umetanje zavisnosti u Springu - pokazni primer

KREIRANJE I PODEŠAVANJE MAVEN PROJEKTA

U Spring softverskim rešenjima često koristimo Maven alat za upravljanje zavisnostima.

O Maven alatu za upravljanje zavisnostima je bilo reči u vežbama prethodne lekcije. Takođe, ovde ćemo iskoristiti sve pogodnosti ovog alata i u okviru njegove konfiguracione datoteke *pom.xml*, navešćemo sve neophodne *JAR* (skraćeno od *Java Archive*) datoteke koje odgovaraju bibliotekama čije ćemo klase koristiti u primeru koji će ovde biti analiziran.

Budući da je razvojno okruženje NetBeans IDE u potpunosti savladano, a daje i podršku za razvoj *Maven* projekata, upravo će ono biti i korišćeno za razvoj pokaznog primera kojim će biti zaokruženo izlaganje vezano za ovu lekciju. Nakon svega rečenog, kreiran je *Maven* projekat pod nazivom *Lekcija2Primer1*.

Prvi korak koji programer čini, kada koristi Maven alat za razvoj vlastite aplikacije, jeste dodavanje zavisnosti (eng. *dependencies*) u specifičnu Maven datoteku pod nazivom *pom.xml*. Zavisnosti podrazumevaju učitavanje poslednjih verzija JAR biblioteka neophodnih za rad u *Spring* radnom okviru.

Sledi listing datoteke pom.xml.

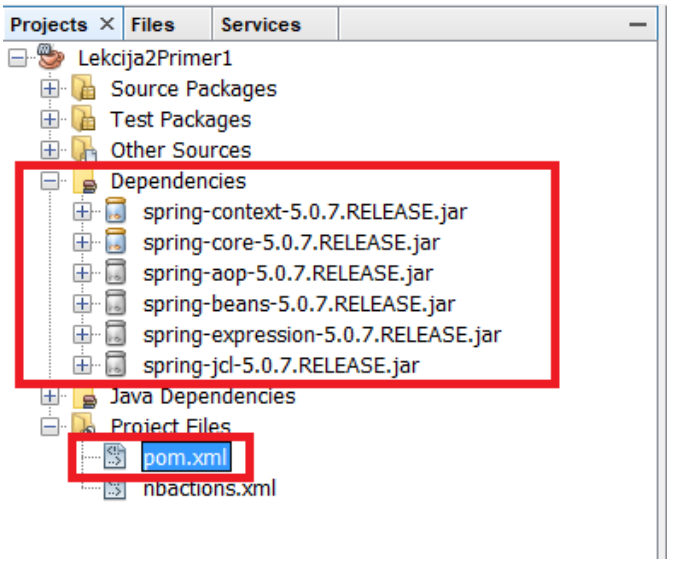
```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/
2001/XMLSchema-instance" xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>
    <groupId>com.metropolitan</groupId>
    <artifactId>Lekcija2Primer1</artifactId>
    <version>1.0-SNAPSHOT</version>
    <packaging>jar</packaging>
    <dependencies>
        <dependency>
            <groupId>org.springframework</groupId>
            <artifactId>spring-core</artifactId>
            <version>5.0.7.RELEASE</version>
        </dependency>
        <dependency>
            <groupId>org.springframework</groupId>
            <artifactId>spring-context</artifactId>
```

```
            <version>5.0.7.RELEASE</version>
        </dependency>
    </dependencies>
    <properties>
        <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
        <maven.compiler.source>1.8</maven.compiler.source>
        <maven.compiler.target>1.8</maven.compiler.target>
    </properties>
</project>
```

PRIKAZ STRUKTURE SPRING MAVEN PROJEKTA

Kreiranjem pom.xml datoteke učitavaju se neophodne biblioteke.

U prethodnom kodu je prikazan listing datoteke *pom.xml* u koju su ugrađene zavisnosti spring-core i spring-context koje nam omogućavaju rad sa osnovnim Spring funkcionalnostima. Čim se završi dodavanje navedenih zavisnosti, u okviru XML tagova *<dependency>...<dependency>* i kada se obavi snimanje kreirane datoteke, automatski počinje preuzimanje svih JAR datoteka koje pripadaju *spring-core* i *spring-context* zavisnostima. Ove datoteke zauzimaju svoje mesto u strukturi projekta kao što je prikazano sledećom slikom.



Slika 6.1 Spring datoteke neophodne za razvoj projekta [izvor: autor]

Sada je sve spremno za razvoj konkretne programske logike primenom Spring radnog okvira, a sve zahvaljujući jednostavnim podešavanjima koja su obavljena na pokazani način. Sledećom slikom je izolovan kod dokumenta *pom.xml* koji je omogućio učitavanje pomenutih datoteka iz *Spring repozitorijuma*.

```
<dependencies>
    <dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring-core</artifactId>
        <version>5.0.7.RELEASE</version>
    </dependency>
    <dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring-context</artifactId>
        <version>5.0.7.RELEASE</version>
    </dependency>
</dependencies>
```

Slika 6.2 Osnovne Spring zavisnosti (izvor: autor)

ZAHTEVI PROJEKTA

Sledi definisanje zahteva rešenja koje odgovara kreiranom projektu

Cilj ovog jednostavnog Spring primera jeste:

- korišćenje koncepta inverzije kontrole;
- korišćenje koncepta umetanja zavisnosti (na načine obrađene u teorijskom delu lekcije);
- korišćenje i objašnjenje često korišćenih specifičnih Spring anotacija *@Service*. *@Component*, *@Repository*;
- konfigurisanje Spring aplikacije primenom Java konfiguracija, a to je specifično za sve novije verzije Spring okvira 4.x i 5.x.

Ideja je jednostavna. Na konzoli se pokazuju određene pozdravne poruke u zavisnosti od željenog jezika. Poruke su za dobrodošlicu i kraj programa, a mogu biti na Srpskom ili Engleskom jeziku.

Otuda, prvi korak jeste definisanje metoda pomoću kojih će navedene poruke biti prikazane na ekranu. Da bi više različitih objekata (zrna) moglo da koristi ove poruke, biće razvijen interfejs kojeg će da implementiraju određene klase.

KREIRANJE INTERFEJSA

Biće razvijen interfejs kojeg će da implementiraju određene implementacione klase.

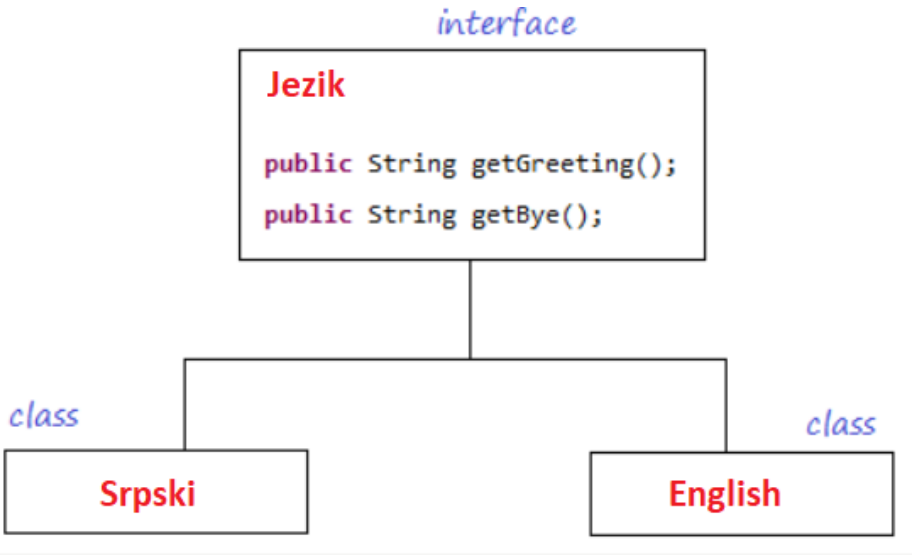
Kao što je istaknuto u prethodnom izlaganju, prvi korak jeste definisanje metoda pomoću kojih će navedene poruke biti prikazane na ekranu. Da bi više različitih objekata (zrna) moglo da koristi ove poruke, biće razvijen interfejs kojeg će da implementiraju određene klase, koje upravo i nazivamo implementacionim klasama interfejsa.

Interfejs će dobiti naziv *Jezik.java* i biće sačuvan u odgovarajućem paketu kao što je prikazano sledećim listingom:

```
package com.metropolitan.lekcija2primer1.interfejs;

/**
 *
 * @author Vlada
 */
public interface Jezik {
    // poruka dobrodošlice
    public String getGreeting();

    // poruka za kraj
    public String getBye();
}
```

Iz priloženog koda je moguće primetiti da interfejs *Jezik* definiše dve apstraktne metode *getGreeting()* i *getBye()* koje će biti redefinisane naknadno u implementacionim klasama ovog interfejsa.

Sledećom slikom je grafički prikazana implementacija kreiranog interfejsa u okviru tekućeg Spring projekta.



Slika 6.3 Implementacija metoda interfejsa Jezik [izvor: autor]

Iz priložene slike se jasno vidi da sledeći korak u razvoju navedene aplikacije predstavlja kreiranje klasa *Srpski.java* i *English.java.* Ove klase će implementirati interfejs *Jezik*.

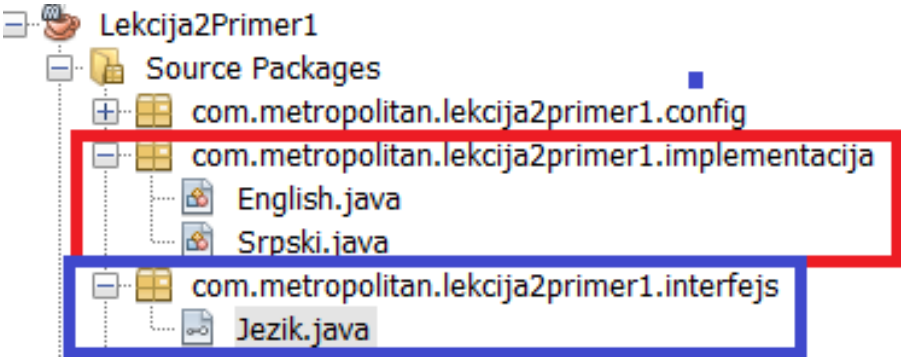
IMPLEMENTACIONE KLASE INTERFEJSA

Definisane metode interfejsa moraju da dobiju konkretnu implementaciju.

U prethodnom izlaganju je dat grafički opis zamišljene implementacije kreiranog interfejsa *Jezik*. Dalja ideja jeste da se omogući prikazivanje poruka na dva jezika: Sprskom i Engleskom jeziku. Otuda će biti neophodno izvršiti kreiranje dveju klasa, koje će na različite načine implementirati kreirani interfejs. Kreira se klasa:

- *Sprski.java* - objekti (zrna) ove klase će pozivati metode za prikazivanje poruka na Srpskom jeziku;
- *English.java* - objekti (zrna) ove klase će pozivati metode za prikazivanje poruka na Engleskom jeziku;

Da bi struktura programa bila lepa i pregledna, implementacione klase će biti sačuvane u posebnom paketu, razdvojeno od interfejsa kojeg implementiraju. Ovo je prikazano sledećom slikom:



Slika 6.4 Razdvojena programska logika interfejsa i implementacije [izvor: autor]

Sledi listing klase *Srpski.java*:

```
public class Srpski implements Jezik{

    @Override
    public String getGreeting() {
       return "Dobro jutro!!!";
    }

    @Override
    public String getBye() {
        return "Doviđenja!!!";
    }

}
```

Sledi listing klase *English.java*:

```
public class English implements Jezik{

    @Override
    public String getGreeting() {
        return "Good morning!!!";
    }
```

```
    @Override
    public String getBye() {
        return "Good bye!!!";
    }

}
```

PRIMENA SPECIFIČNIH SPRING ANOTACIJA

Opis upotrebe anotacija @Component, @Repository, @Service i @Controller

Pre nego što se nastavi sa daljim razvojem aplikacije, neophodno je da se navedu i objasne Spring anotacije koje će u daljem radu biti korišćene:

- *@Component,*
- *@Repository,*
- *@Service* i
- *@Controller.*

Prvo će biti objašnjena anotacija *@Component.* Ovom anotacijom se obeležava Java klasa kao zrno i na taj način je omogućen mehanizam skeniranja komponenata (eng. *component scanning*) koji će u strukturi projekta pronaći zrno, preuzeti ga i učitati u kontekst aplikacije. Primer korišćena anotacije *@Component* može biti prikazan sledećim delimičnim listingom:

```
@Component
public class MojaKomponenta {

   ****

}
```

Iako je upotreba anotacije *@Component* jako dobra, moguće je koristiti i pogodniju anotaciju koja obezbeđuje dopunske funkcionalnosti pogotovo ako se koristi sa objektima koji manipulišu podacima (*DAO - Data Access Objects*) - anotaciju *@Repository*. Primer korišćena anotacije *@Repository* može biti prikazan sledećim delimičnim listingom:

```
@Repository
public class MojRepozitorijum {
***
}
```

Anotacija *@Service* takođe predstavlja specijalizaciju anotacije *@Component*. Ova anotacija praktično ne obezbeđuje nikakvo dodatno ponašanje, u odnosu na anotaciju *@Component*, klase na koju je primenjena ali je jako dobra za isticanje servisnog nivoa projekta jer na taj način bolje opisuje šta je namera klase. Primer korišćena anotacije *@Service* može biti prikazan sledećim delimičnim listingom:

```
@Service
public class GreetingService {
    ***
}
```

O poslednje navedenoj anotaciji će više biti govora kada se bude obrađivala tema *Spring MVC.* Anotacijom *@Controller* se obeležava klasa kao *Spring Web MVC* kontroler (eng. *controller*). Ovo je takođe specijalni oblik anotacije *@Component* tako da zrna obeležena ovom anotacijom mogu biti automatski učitana u *IoC kontejner*.

KREIRANJE SERVISNE KLASE

Primena anotacije @Service na klasu projekta.

U prethodnom izlaganju je dat pregled primene i značaja često korišćenih Spring anotacija. Većinu od njih moguće je naći i u tekućem primeru. Pa, neka izlaganje započne primenom anotacije *@Service.* Ovom anotacijom (videti priloženi listing) je obeležena klasa *GreetingService.java*. Takođe, opet je zadržana dobra praksa razdvajanja različitih nivoa projekta po različitim paketima. Tako je i ova klasa spakovana u paket rezervisan za servise.

Prvo što je moguće primetiti, kada se uđe u kod klase, jeste da je izvršeno umetanje (eng. *injection*) objekta implementacione klase interfejsa *Jezik*. Budući da je ovo polje klase *GreetingService.java* obeleženo anotacijom *@Autowired*, desiće se automatsko povezivanje zrna jezik, sa zrnom tipa *GreetingService*.

U nastavku je definisan podrazumevani konstruktor i metoda *sayGreeting()* koju će u odgovarajućem trenutku izvršavanja programa pozvati zrno ove klase.

> @Service je anotacija koja je u ovom slučaju upotrebljena da obeleži klasu i da obezbedi Spring IoC kontejenru informaciju da je klasa Spring zrno.

Nakon detaljne analize, u nastavku je moguće priložiti i listing kreirane klase iz servisnog nivoa aplikacije.

```
package com.metropolitan.lekcija2primer1.servis;

import com.metropolitan.lekcija2primer1.interfejs.Jezik;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;

/**
 *
 * @author Vlada
 */
@Service
public class GreetingService {
    @Autowired
   private Jezik jezik;
```

```
   public GreetingService() {

   }

   public void sayGreeting() {

       String greeting = jezik.getGreeting();

       System.out.println("Pozdrav: " + greeting);
   }
}
```

REPOZITORIJUM APLIKACIJE

Primena anotacije @Repository na klasu projekta.

Kao što je već bilo govora, moguće je koristiti veoma pogodnu anotaciju koja obezbeđuje dopunske funkcionalnosti pogotovo ako se koristi sa objektima koji manipulišu podacima (*DAO - Data Access Objects*) - anotaciju *@Repository*. Za početak, ovom anotacijom je obeležena klasa pod nazivom *MojRepozitorijum.java*. Zadatak ove klase je da njeni objekti obezbede dva tipa informacija:

- String - daje predefinisani naziv aplikacije u trenutku poziva metode *getAppName();*
- Date - daje tekuće vreme u trenutku poziva metode *getSystemDateTime();*

> @Repositoryje anotacija koja je u ovom slučaju upotrebljena da obeleži klasu i da obezbedi Spring IoC kontejenru informaciju da je klasa Spring zrno.

Nakon analize ovog dela aktuelnog Spring projekta, u nastavku je moguće priložiti i listing kreirane klase iz repozitorijumskog nivoa aplikacije.

```
package com.metropolitan.lekcija2primer1.repozitorijum;

import java.util.Date;
import org.springframework.stereotype.Repository;

/**
 *
 * @author Vlada
 */
@Repository
public class MojRepozitorijum {

    public String getAppName() {
        return "Prvi Spring primer";
    }
```

```
    public Date getSystemDateTime() {
        return new Date();
    }
}
```

KREIRANJE I PRETRAGA KOMPONENATA SPRING APLIKACIJE

Primena anotacije @Component na klasu projekta.

U nastavku izlaganja, posebna pažnja će biti posvećena opštoj anotaciji *@Component*. Budući da ova anotacija ima veliki značaj u Spring aplikacijama neophodno je još jednom posebno istaći da se ovomanotacijom obeležava Java klasa kao zrno i na taj način je omogućen mehanizam skeniranja komponenata (eng. *component scanning*) koji će u strukturi projekta pronaći zrno, preuzeti ga i učitati u kontekst aplikacije.

Java klasa projekta, obeležena navedenom anotacijom, dobila je naziv *MojaKomponenta.java*. Kada "zavirimo" u njen listing možemo da primetimo primenu automatskog povezivanja zavisnosti objekata klasa *MojRepozitorijum* i *MojaKomponenta*.

U odgovarajućem trenutku izvršavanja aplikacije, zrno *MojaKomponenta* će iskoristiti zrno *MojRepozitorijum* za pribavljanje informacija koje su rezultat poziva metode *showAppInfo()*.

> Takođe, @Component je osnovna anotacija koja je u ovom slučaju upotrebljena da obeleži klasu i da obezbedi Spring IoC kontejenru informaciju da je klasa Spring zrno.

Nakon analize ovog dela aktuelnog Spring projekta, u nastavku je moguće priložiti i listing kreirane klase iz ovog nivoa aplikacije.

```
package com.metropolitan.lekcija2primer1.komponenta;

import com.metropolitan.lekcija2primer1.repozitorijum.MojRepozitorijum;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Component;

/**
 *
 * @author Vlada
 */
@Component
public class MojaKomponenta {

    @Autowired
    private MojRepozitorijum repository;

    public void showAppInfo() {
        System.out.println("Trenutno vreme: "+ repository.getSystemDateTime());
        System.out.println("Naziv aplikacije: "+ repository.getAppName());
```

```
    }

}
```

Kao što je bilo moguće primetiti, nema suštinske razlike u primeni anotacija @Service, @Componenti @Repository u opštem smislu. One se koriste za obeležavanje klasa i ukazivanje njihove uloge u kontekstu aplikacije.

SPRING (JAVA) KONFIGURISANJE KONTEKSTA APLIKACIJE

Savremene Spring verzije insistiraju na Java konfiguracijama umesto XML.

Ključni deo Spring aplikacije zauzimaju konfiguracije. Konfiguracije mogu biti realizovane putem XML konfiguracionih datoteka ili, kako je to sada preporučljivo, primenom Java konfiguracionih klasa. U jednoj aplikaciji može biti više konfiguracija. Njihov ključni zadatak jeste definisanje konteksta aplikacije, tj. konteksta po kojem *Spring IoC kontejner* realizuje koncepte inverzije kontrole i umetanja zavisnosti. Ovde će akcenat biti na konfiguracionoj klasi.

Klasa je nazvana *AppConfiguration.java* i izolovana je od ostatka aplikacije u okviru vlastitog paketa. Obavezno, klasa mora da bude obeležena anotacijom *@Configuration*.

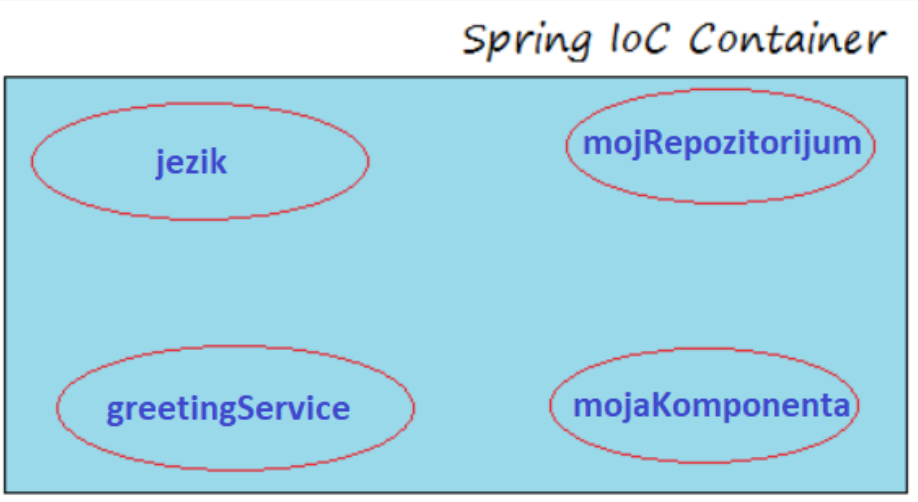
Dalje, klasa je obeležena još jednom anotacijom *@ComponentScan*. Ova anotacija dozvoljava kontekstu aplikacije da pretražuje zrna unutar zadatog paketa (linija koda 17).

```
package com.metropolitan.lekcija2primer1.config;

import com.metropolitan.lekcija2primer1.implementacija.Srpski;
import com.metropolitan.lekcija2primer1.interfejs.Jezik;
import com.metropolitan.lekcija2primer1.komponenta.MojaKomponenta;
import com.metropolitan.lekcija2primer1.repozitorijum.MojRepozitorijum;
import com.metropolitan.lekcija2primer1.servis.GreetingService;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.ComponentScan;
import org.springframework.context.annotation.Configuration;

/**
 *
 * @author Vlada
 */
@Configuration
@ComponentScan({"com.metropolitan.lekcija2primer1.komponenta"})
public class AppConfiguration {

    @Bean(name ="jezik")
```

```
    public Jezik getJezik() {

        return new Srpski();
    }

    @Bean
    public MojRepozitorijum getRepozitorijum (){

        return new MojRepozitorijum();
    }

    @Bean
    public MojaKomponenta getKomponenta (){

        return new MojaKomponenta();
    }

    @Bean(name ="greetingService")
    public GreetingService getGreetingService (){

        return new GreetingService();
    }

}
```

Sada je moguće detaljnije se pozabaviti kodom ove konfiguracione klase. U okviru klase su inicijalizovana četiri zrna. Kreiranje navedenih objekata je prepušteno metodama koje su obeležene anotacijom *@Bean*. Podrazumevani naziv zrna odgovara nazivu klase ali, kao i svaki objekat, njegov naziv počinje malim slovom (linije koda 26 i 32). Ukoliko programer želi da koristi drugi naziv za zrno, unutar anotacije *@Bean* neophodno je dodeliti odgovarajuću String vrednost parametru name (linije koda 20 i 38).

Kreirana Spring zrna su sada dostupna i biće upravljana *Spring IoC kontejnerom*. Grafički, navedeno je moguće ilustrovati sledećom slikom.



Slika 6.5 Kreirana zrna u IoC kontejneru [izvor: autor]

UČITAVANJE KONTEKSTA APLIKACIJE

Instanca izvedena iz ApplicationContext interfejsa učitava konfiguracije iz prethodne klase.

Da bi primer bio u potpunosti funkcionalan i da bi mogao da prikaže umetanje zavisnosti i inverziju kontrole na delu, neophodno je kreirati glavnu klasu aplikacije, koja će zapravo izvršiti svoju dobro poznatu statičku metodu *main().*

Ako se obrati pažnja na liniju koda 20, moguće je primetiti kreiranje objekta pod nazivom context. Objekat je kreiran pozivom konstruktora *AnnotationConfigApplicationContext()* istoimene implementacione klase interfejsa *ApplicationContext.* Kao argument ovog konstruktora, javlja se klasa *AppConfiguration.java* iz koje se učitava kontekst aplikacije, odnosno sva zrna koja su prethodno u njoj kreirana.

Redom, jedno po jedno, zrna se učitavaju u kontekst aplikacije (linije koda 23, 30, 37) pozivom metode *getBean()* objekta konteksta aplikacije. Sada su zrna dostupna za izvršavanje vlastitih metoda unutar *Main.java* klase.

Kreiranjem ove klase je završen razvoj ovog projekta i u nastavku je neophodno izvršiti njegovo prevođenje i pokretanje sa ciljem demonstracije izlaza kojeg proizvodi kreirana aplikacija.

> Povratni tip metode getBean() je java.lang.Object tako da je neophodno konvertovati ga u željeni tip podataka pre njegove upotrebe.

Sledi listing klase *Main.java*.

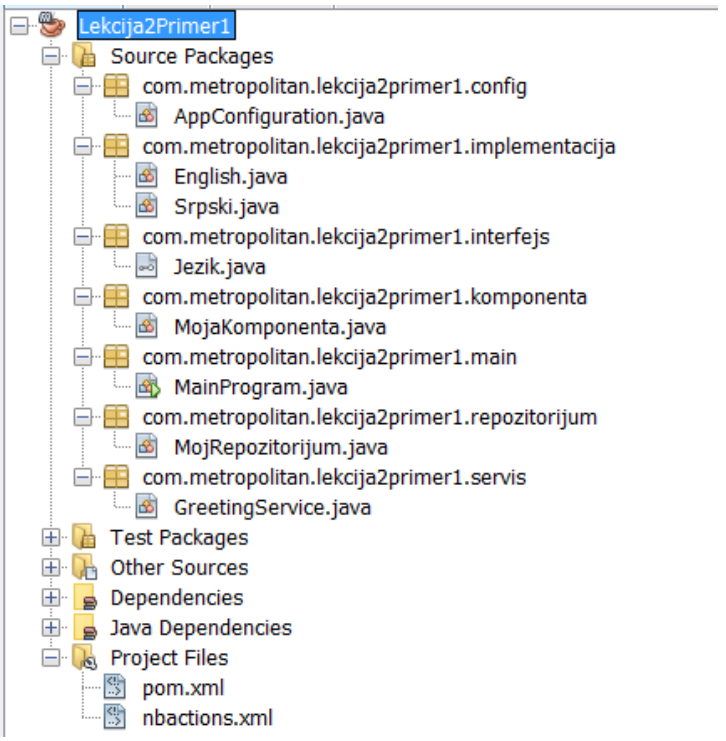
```
package com.metropolitan.lekcija2primer1.main;

import com.metropolitan.lekcija2primer1.config.AppConfiguration;
import com.metropolitan.lekcija2primer1.interfejs.Jezik;
import com.metropolitan.lekcija2primer1.komponenta.MojaKomponenta;
import com.metropolitan.lekcija2primer1.servis.GreetingService;
import org.springframework.context.ApplicationContext;
import org.springframework.context.annotation.AnnotationConfigApplicationContext;

/**
 *
 * @author Vlada
 */
public class MainProgram {

   public static void main(String[] args) {

       // Kreiranje konteksta aplikacije čitanjem
       // konfiguracija definisanih u klasi 'AppConfiguration'.
       ApplicationContext context = new
AnnotationConfigApplicationContext(AppConfiguration.class);
```

```
        System.out.println("----------");
        Jezik language = (Jezik) context.getBean("jezik");

        System.out.println("Zrno jezika: "+ language);
        System.out.println("Pozivam language.sayBye(): "+ language.getBye());

        System.out.println("----------");

        GreetingService service = (GreetingService)
context.getBean("greetingService");


        service.sayGreeting();

        System.out.println("----------");

        MojaKomponenta myComponent = (MojaKomponenta)
context.getBean("mojaKomponenta");

        myComponent.showAppInfo();

    }

}
```

PREVOĐENJE I POKRETANJE KREIRANOG SPRING PROJEKTA

Demonstacija izlaza kojeg produkuje izvršavanje aplikacije.

Pošto je primer u potpunosti zaokružen moguće je prikazati njegovu konačnu strukturu. Sledećom slikom je prikazan kreirani interfejs, njegove implementacione klase, servis, komponenta i repozitorijum u okviru vlastitih paketa. Prikazan je i folder (čvor projekta) u kojem se nalazi zavisnosti iz *pom.xml* datoteke.

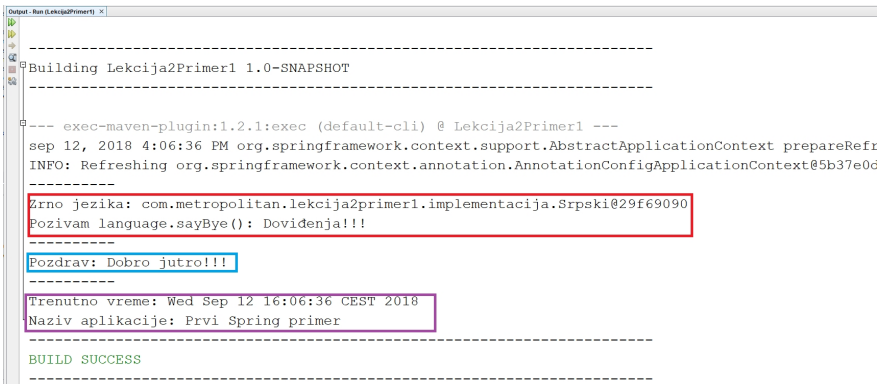


Slika 6.6 Konačna struktura projekta [izvor: autor]

Klikom na dugme *Run Project*, u okviru razvojnog okruženja *NetBeans IDE*, vrši se prevođenje aplikacije, a zatim njeno pokretanje. Dešava se sledeće:

- učitava se zrno jezika (Srpski) koje poziva metode kojima prikazuje naziv klase zrna, a nakon toga i poruka za kraj;
- učitava se zrno servisa koje umeće zrno jezika i poziva metodu koja prikazuje pozdravnu poruku;
- učitava se zrno tipa MojaKomponenta i poziva metodu koja prikazuje informacije o trenutnom vremenu i predefinisanom nazivu aplikacije.

Kreirani izlaz, koji predstavlja rezultat izvršavanja kreirane aplikacije, prikazan je sledećom slikom.



Slika 6.7 Izvršavanje aplikacije [izvor: autor]

Poglavlje 7

Kompatibilnost unazad - XML konfigurisanje

INSTANCIRANJE SPRING IOC KONTEJNERA

Neophodno je instancirati Spring IoC kontejner za kreiranje zrna instanci kroz čitanje podataka iz odgovarajućeg konfiguracionog fajla.

U ovom slučaju problem je definisan na sledeći način: *Neophodno je instancirati* Spring IoC kontejner *za kreiranje zrna instanci kroz čitanje podataka iz odgovarajućeg* XML konfiguracionog *fajla. Tek nakon toga, moguće je koristiti* zrna *iz* ***Spring IoC kontejnera*** .

Rešenje problema moguće je tražiti na dva načina. Osnovno rešenje se naziva bean factory , a napredniji pristup podrazumeva kontekst aplikacije. Kontekst aplikacije je kompatibilno proširenje koncepta *bean factory*. Posebno, trebalo bi imati na umu da su konfiguracioni fajlovi zrna, za oba tipa *Spring IoC kontejnera*, u potpunosti identični.

NAPOMENA: U cilju prevođenja Spring koda, koji će biti priložen u narednim izlaganjima, neophodno je dodati Spring zavisnosti (eng. *dependencies*) u odgovarajući projekat (*classpath*). Za to je neophodno koristiti neko rešenje za upravljanje prevođenjem poput: Maven, Apache Ant ili Ivy.Ako se koristi *Maven* neophodno je dodati zavisnosti na način prikazan sledećim kodom u kreirani *Maven* projekat. Posebno biće korišćena notacija *${spring.version}* koja će ukazivati na verziju Spring okvira.

```
<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-aop</artifactId>
    <version>${spring.version}</version>
</dependency>
<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-web</artifactId>
    <version>${spring.version}</version>
</dependency>
<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-context-support</artifactId>
    <version>${spring.version}</version>
</dependency>
```

INSTANCIRANJE KONTEKSTA APLIKACIJE I PREUZIMANJE ZRNA

Nakon instanciranja konteksta aplikacije zrno se koristi kao i bilo koji drugi objekat.

ApplicationContext predstavlja interfejs čija implementacija mora da bude instancirana. Implementacija *ClassPathXmlApplicationContext* gradi kontekst aplikacije učitavanjem iz XML datoteke koja odgovara konfiguracionom fajlu. Ovde je, takođe, moguće definisati više različitih konfiguracionih fajlova. Kontekst aplikacije moguće je instancirati na sledeći način:

```
ApplicationContext context = new ClassPathXmlApplicationContext("beans.xml");
```

Pored ***ClassPathXmlApplicationContext*** , u Springu postoji još nekoliko implementacija za *ApplicationContext* interfejs . Na primer, *FileSystemXmlApplicationContext* se koristi za učitavanje XML konfiguracionog fajla iz fajl sistema ili sa odgovarajućeg URL. Zatim, *XmlWebApplicationContext* i *XmlPortletApplicationContext* se upotrebljavaju isključivo u veb i portal Spring aplikacijama.

Za preuzimanje definisanog zrna iz konteksta aplikacije (ili *bean* factory) neophodno je izvršiti poziv metode *getBean()* uz prosleđivanje jedinstvenog naziva za zrno. Povratni tip metode *getBean()* je *java.lang.Object* tako da je neophodno konvertovati ga u željeni tip podataka pre njegove upotrebe.

```
// ovo je deo koda primera koji će biti
// analiziran u nastavku
SequenceGenerator generator =
                (SequenceGenerator) context.getBean("sequenceGenerator");
```

Nakon ovog koraka, moguće je koristiti zrno kao i bilo koji drugi objekat kreiran konstruktorom. Kompletiran kod, ovog jednostavnog primera aplikacije nazvane ***SequenceGenerator*** , prikazan je kroz sledeću Main klasu:

```
package com.metropolitan.lekcija2primer2.main;

import com.metropolitan.lekcija2primer2.sequence.SequenceGenerator;
import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;
/**
 *
 * @author Vlada
 */
public class Main {

    /**
     * @param args the command line arguments
     */
    public static void main(String[] args) {
        ApplicationContext appContext = new
```

```
ClassPathXmlApplicationContext("classpath:/META-INF/beansConfig.xml");

        SequenceGenerator generator;
        generator = (SequenceGenerator) appContext.getBean("sequenceGenerator");
        System.out.println(generator.getSequence());
        System.out.println(generator.getSequence());
    }
}
```

Ukoliko je prevođenje i pokretanje programa prošlo u najboljem redu, korisnik može da vidi izlaz koji odgovara broju sekvence (kao npr na sledećoj slici) zajedno sa izvesnim porukama logovanja.

```
INFO: Loading XML bean definitions from class path resource [META-INF/beansConfig.xml]
30100000A
30100001A
------------------------------------------------------------------------
BUILD SUCCESS
```

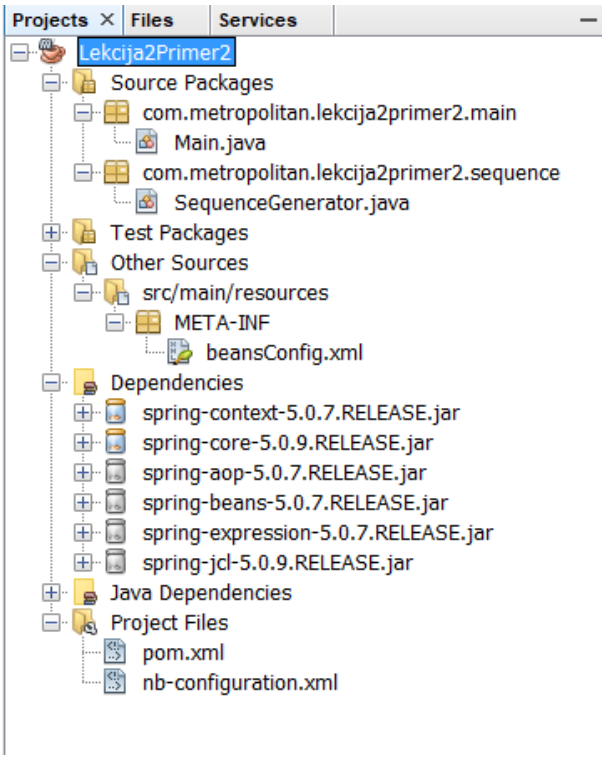
Slika 7.1 Rezultat izvršavanja pokrenutog programa [izvor: autor]

Detaljno rešenje ovog zadatka sledi u nastavku.

STRUKTURA PROJEKTA I MAVEN POM.XML

Primer za demonstraciju instanciranja konteksta i preuzimanje zrna iz XML konfiguracije

Primer koji j je u prethodnoj analizi delimično izložen sada će biti prikazan detaljno. Kreiran je *Maven Java Spring* projekat pod nazivom *Lekcija2Primer2*. Sledećom slikom je prikazana struktura kreiranog projekta:



Slika 7.2 Hijerarhija kreiranog Maven Spring projekta [izvor: autor]

Iz priložene strukture moguće je primetiti da je nekoliko ključnih delova aplikacije istaknuto:

- sekcija sa ***pom.xml*** Maven konfiguracionom datotekom;
- sekcija *src/main/resources* koja sadrži konfiguracionu datoteku zrna koje instancira klasu generatora sekvenci;
- sekcija sa Java klasama.

Za kreiranje i prevođenje programa je bilo neophodno u datoteci *pom.xml* izvršiti dodavanje svih potrebnih zavisnosti koje će automatski biti preuzete iz *Spring repozitorijuma* i dodate u *Dependencies* folder projekta. Sledi listing ove datoteke:

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/
2001/XMLSchema-instance" xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>
    <groupId>com.metropolitan</groupId>
    <artifactId>Lekcija2Primer2</artifactId>
    <version>1.0-SNAPSHOT</version>
    <packaging>jar</packaging>
    <dependencies>
        <dependency>
            <groupId>org.springframework</groupId>
            <artifactId>spring-core</artifactId>
            <version>5.0.9.RELEASE</version>
        </dependency>
         <dependency>
            <groupId>org.springframework</groupId>
```

```
            <artifactId>spring-context</artifactId>
            <version>5.0.7.RELEASE</version>
             <type>jar</type>
         </dependency>
    </dependencies>
    <properties>
        <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
        <maven.compiler.source>1.8</maven.compiler.source>
        <maven.compiler.target>1.8</maven.compiler.target>
    </properties>
</project>
```

JAVA KLASE PROJEKTA

Konkretna logika je prepuštena Java klasama

Logika programa biće realizovana pomoću dve java klase. Prva klasa će da definiše generatore sekvenci sa atributima: *prefix, suffix, initial* i *counter* i poslužiće kao šablon za kreiranje zrna (instanci) ove klase. Sledi listing klase *SequenceGenerator.java*:

```
package com.metropolitan.lekcija2primer2.sequence;

/**
 *
 * @author Vlada
 */
public class SequenceGenerator {

    private String prefix;
    private String suffix;
    private int initial;
    private int counter;

    public SequenceGenerator() {
    }

    public SequenceGenerator(String prefix, String suffix, int initial) {
        this.prefix = prefix;
        this.suffix = suffix;
        this.initial = initial;
    }

    public void setPrefix(String prefix) {
        this.prefix = prefix;
    }

    public void setSuffix(String suffix) {
        this.suffix = suffix;
    }
```

```
    public void setInitial(int initial) {
        this.initial = initial;
    }

    public synchronized String getSequence() {
        StringBuilder buffer = new StringBuilder();
        buffer.append(prefix);
        buffer.append(initial + counter++);
        buffer.append(suffix);
        return buffer.toString();
    }
}
```

Druga klasa je klasična klasa Java aplikacije, sa *main()* metodom za pokretanje samog programa. U njoj je instanciran kontekst aplikacije i ukazano je da je datoteka *beansConfig.xml* konfiguraciona datoteka zrna. Sledi listing klase Main.java.

```
package com.metropolitan.lekcija2primer2.main;

import com.metropolitan.lekcija2primer2.sequence.SequenceGenerator;
import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;
/**
 *
 * @author Vlada
 */
public class Main {

    /**
     * @param args the command line arguments
     */
    public static void main(String[] args) {
        ApplicationContext appContext = new
ClassPathXmlApplicationContext("classpath:/META-INF/beansConfig.xml");

        SequenceGenerator generator;
        generator = (SequenceGenerator) appContext.getBean("sequenceGenerator");
        System.out.println(generator.getSequence());
        System.out.println(generator.getSequence());
    }
}
```

KONFIGURACIJA ZRNA

Demonstracija kreiranja Spring zrna.

Instanciranje zrna u Spring programu moguće je obaviti kroz konfiguracionu datoteku zrna, u ovom slučaju nosi naziv beansConfig.xml. Datoteka, po osnovnim Spring podešavanjima, trebalo bi da se nalazi na lokaciji src/main/resources. Datoteka je priložena sledećim listingom:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:c="http://www.springframework.org/schema/c"
       xmlns:context="http://www.springframework.org/schema/context"
       xmlns:lang="http://www.springframework.org/schema/lang"
       xmlns:p="http://www.springframework.org/schema/p"
       xmlns:util="http://www.springframework.org/schema/util"

       xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.xsd
          http://www.springframework.org/schema/context
http://www.springframework.org/schema/context/spring-context-5.0.xsd
          http://www.springframework.org/schema/lang http://www.springframework.org/
schema/lang/spring-lang-5.0.xsd
          http://www.springframework.org/schema/util http://www.springframework.org/
schema/util/spring-util-5.0.xsd
">
<bean id="sequenceGenerator"
          class="com.metropolitan.lekcija2primer2.sequence.SequenceGenerator">
        <property name="prefix">
            <value>30</value>
        </property>
        <property name="suffix">
            <value>A</value>
        </property>
        <property name="initial">
            <value>100000</value>
        </property>
    </bean>
</beans>
```

Aplikacija se pokreće i u glavnoj klasi se dva puta poziva metoda *getSequence()*. Učitavaju se podaci iz konfiguracionog fajla zrna i generišu se sekvence. Rezultat izvršavanja je prikazan sledećom slikom.

```
------------------------------------------------------------------------

--- exec-maven-plugin:1.2.1:exec (default-cli) @ Lekcija2Primer2 ---
sep 12, 2018 6:39:53 PM org.springframework.context.support.AbstractAppl
INFO: Refreshing org.springframework.context.support.ClassPathXmlApplica
sep 12, 2018 6:39:53 PM org.springframework.beans.factory.xml.XmlBeanDef
INFO: Loading XML bean definitions from class path resource [META-INF/be
30100000A
30100001A
------------------------------------------------------------------------
BUILD SUCCESS
------------------------------------------------------------------------
Total time: 1.480s
Finished at: Wed Sep 12 18:39:53 CEST 2018
Final Memory: 6M/135M
------------------------------------------------------------------------
```

Slika 7.3 Generisane sekvence [izvor: autor]

Prilikom testiranja programa moguće je pojavljivanje greške. Razlog je taj što je razvojno okruženje kao prostor naziva (*name space*) u *beansConfig.xml* za *xsi:schemaLocation* dodao kao jednu od vrednosti .../*spring-beans-3.2.3.xsd.* Usled nekompatibilnosti sa verzijama 4.x i

5.x Spring razvojnog okvira odgovarajuća klasa nije pronađena. Trebalo bi zameniti vrednost spring-beans-3.2.3.xsd novijom ili jednostavno obrisati -3.2.3 (kao na slici dole) i aplikacija će funkcionisati ispravno.

Slika 7.4 Modifikacija u beansConfig.xm [izvor: autor]

Poglavlje 8

Lekcija 2 - Pokazne vežbe

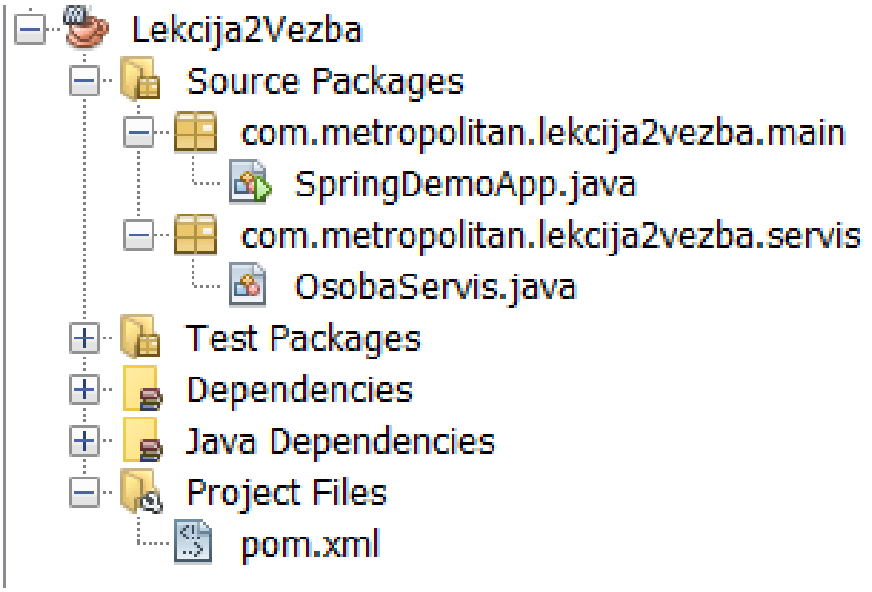
PRVA SPRING 4.X, 5.X APLIKACIJA (45 MINUTA)

Kreirajmo prvu Spring aplikaciju na računaru

Koristeći razvojno okruženje *NetBeans IDE*, kreirati prvu *Spring* aplikaciju po sledećim zahtevima:

1. Kreirati *Maven* projekat aplikacije pod nazivom *Lekcija2Vezba2*;
2. Dodati u *pom.xml* sledeće zavisnosti: *spring-core* i *spring-context*;
3. Kreirati servisnu klasu *Osoba.java* čiji će zadatak biti da u *Log*-u prikaže podatke o osobi (ime i prezime);
4. Kreirati konfiguracionu klasu koja inicijalizuje zrno *Osoba*;
5. Kreirati glavnu klasu koja umeće zrno osoba, učitava kontekst aplikacije, učitava podatke o osobi, predaje ih zrnu *Osoba* i traži od zrna da pozove svoju metodu za prikazivanje podataka.
6. Dozvoljeno je da korake 4 i 5 obuhvati jedna klasa (konfiguraciona i glavna istovremeno). Neka se zove *SpringDemoApp.java*.

Sledećom sliko je prikazana predložena struktura Maven projekta.



Slika 8.1 Struktura projekta Lekcija2Vezba2 [izvor: autor]

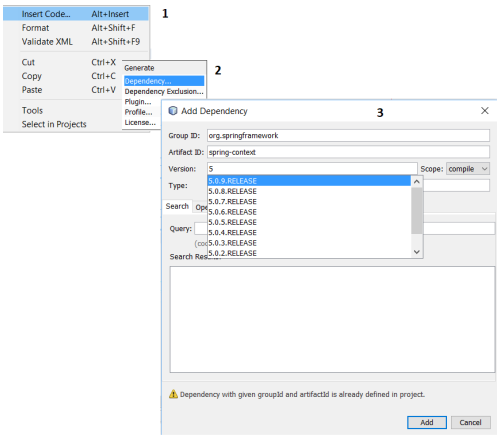
KONFIGURISANJE MAVEN SPRING PROJEKTA

Dodajemo Spring zavisnosti neophodne za kreiranje aplikacije.

Kao što je definisano, prvi korak jeste dodavanje *spring-core* i *spring-context* zavisnosti u projekat. Neophodno je otvoriti datoteku *pom.xml* i desnim klikom iznad taga *<properties>,* a ispod *<packaging>*, izabrati opciju "*Insert code*" iz otvorenog menija. Izborom ove opcije, otvara se sledeći meni iz kojeg je neophodno izabrati opciju "*Dependency*". Konačno, otvara se prozor (čarobnjak) za dodavanje zavisnosti u *pom.xml*.

Dodaju se sledeće vrednosti:

1. naziv paketa (groupId);
2. naziv klase - zavisnosti (artifactId);
3. verzija (version).



Slika 8.2 Čarobnjak za dodavanje zavisnosti [izvor: autor]

Nakon dodavanja zavisnosti, datoteka pom.xml ima sledeći sadržaj:

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/
2001/XMLSchema-instance" xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>
    <groupId>com.metropolitan</groupId>
    <artifactId>Lekcija2Vezba</artifactId>
    <version>1.0-SNAPSHOT</version>
    <packaging>jar</packaging>
    <dependencies>
        <dependency>
            <groupId>org.springframework</groupId>
            <artifactId>spring-core</artifactId>
            <version>5.0.9.RELEASE</version>
        </dependency>
        <dependency>
            <groupId>org.springframework</groupId>
```

```
            <artifactId>spring-context</artifactId>
            <version>5.0.7.RELEASE</version>
            <type>jar</type>
        </dependency>
    </dependencies>
    <properties>
        <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
        <maven.compiler.source>1.8</maven.compiler.source>
        <maven.compiler.target>1.8</maven.compiler.target>
    </properties>
</project>
```

KREIRANJE SERVISNE KLASE

Kreira se klasa obeležena anotacijom @Service

U sledećem koraku kreira se klasa obeležena anotacijom *@Service.* To znači da će ova klasa predstavljati zrno koje će prikazivati podatke o nekoj osobi. Podaci će biti uneti sa tastature kada se program pokrene.

Sledećim listingom je dato rešenje za ovu klasu.

```
package com.metropolitan.lekcija2vezba.servis;

/**
 *
 * @author Vlada
 */
import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;
import org.springframework.stereotype.Service;

@Service
public class OsobaServis {
    public static final Log LOG = LogFactory.getLog(OsobaServis.class);

    public void prikaziPodatke(String a, String b) {
        LOG.info("Podaci o osobi: " + a + " " + b);
    }
}
```

KONFIGURACIONA I GLAVNA KLASA

Sledeći zadatak je instanciranje zrna, konfigurisanje i pokretanje apliakcije.

Zadatak će biti rešen korišćenjem jedne klase. Obeležićemo je anotacijama *@Configuration* i *@ComponentScan* da bi dobila ulogu konfiguracione klase i sposobnost pretrage komponenata u aplikaciji.

Dalje, u klasu će biti dodat sledeći kod sa ciljem inicijalizacije zrna klase *OsobaServis.java*:

```
@Bean
    public OsobaServis getOsoba() {
        return new OsobaServis();
    }
```

U nastavku ova klasa će preuzeti i ulogu glavne (*Main*) klase. Vršimo automatsko umetanje zrna klase *OsobaServis,* a zatim u *main()* metodi kreiramo i učitavamo kontekst aplikacije (klasa učitava samu sebe jer je istovremeno i konfiguraciona). Zatim se kreira i zrno ove klase koje poziva metodu *runApp()*. Metoda koristi umetnuto zrno *osoba* koje poziva metodu *prikaziPodatke()* sa parametrima ime i prezime dobijenih unosom preko *InputDialog* obejkata. Izvršavanjem ove metode u log-u se dobijaju uneti podaci o osobi.

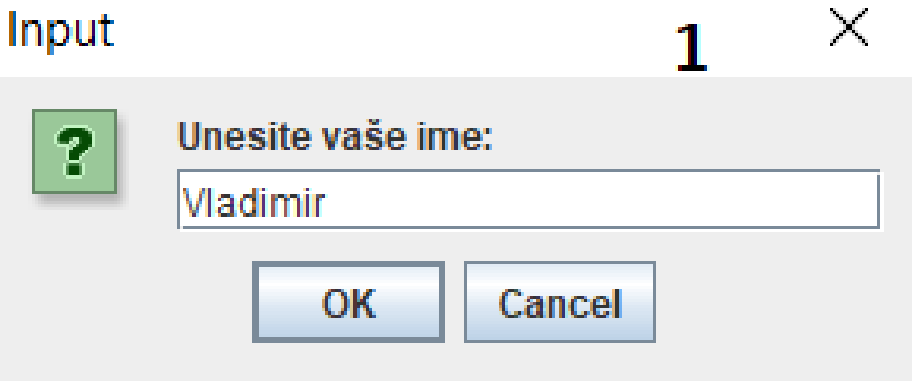
```
package com.metropolitan.lekcija2vezba.main;

import com.metropolitan.lekcija2vezba.servis.OsobaServis;
import javax.swing.JOptionPane;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.context.ApplicationContext;
import org.springframework.context.annotation.AnnotationConfigApplicationContext;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.ComponentScan;
import org.springframework.context.annotation.Configuration;

/**
 *
 * @author Vlada
 */
@Configuration
@ComponentScan
public class SpringDemoApp {

    /**
     * @param args the command line arguments
     */
    // This will be created and injected by spring IoC
    @Autowired
    private OsobaServis osoba;

    String ime = JOptionPane.showInputDialog("Unesite vaše ime: ");
    String prezime = JOptionPane.showInputDialog("Unesite vaše prezime: ");

    public static void main(String[] args) {
        ApplicationContext context = getAppContext();

        SpringDemoApp app = context.getBean("springDemoApp", SpringDemoApp.class);
```

```
        app.runApp();
    }

    public void runApp() {
        osoba.prikaziPodatke(ime, prezime);
    }

    public static ApplicationContext getAppContext() {
        ApplicationContext context = new
AnnotationConfigApplicationContext(SpringDemoApp.class);
        return context;
    }

    @Bean
    public OsobaServis getOsoba() {
        return new OsobaServis();
    }

}
```

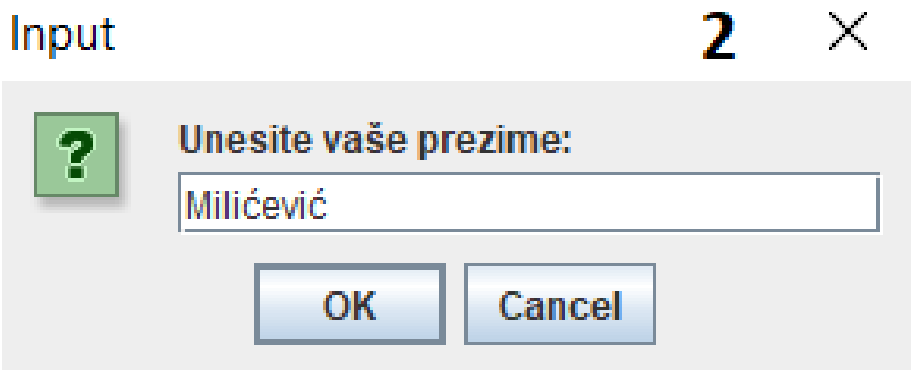
IZVRŠAVANJE APLIKACIJE

Aplikacija se prevodi i pokreće.

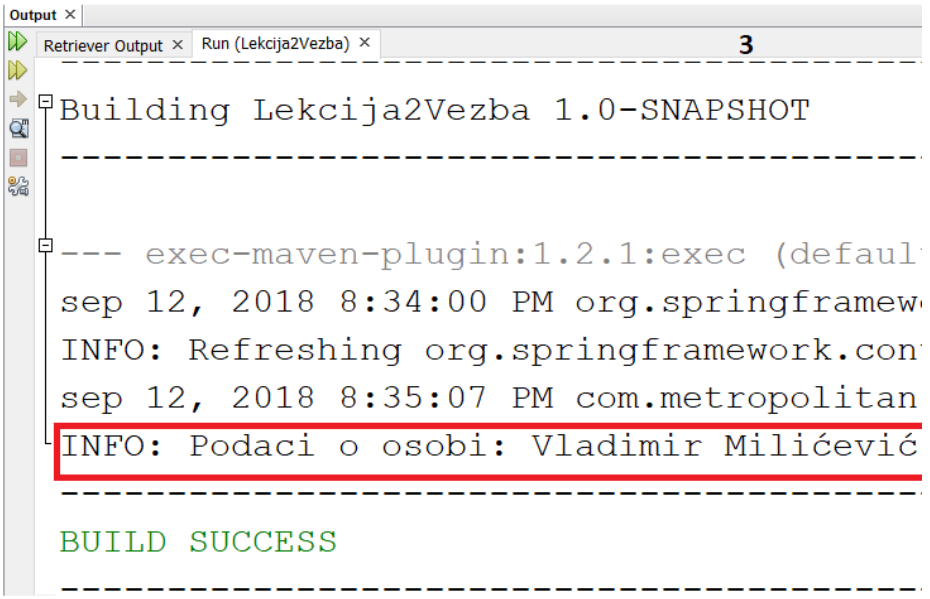
Aplikacija se prevodi i pokreće, a rezultati izvršavanja aplikacije su prikazani sledećim slikama.



Slika 8.3 Unos imena [izvor: autor]



Slika 8.4 Unos Prezimena [izvor: autor]



Slika 8.5 Podaci o osobi [izvor: autor]

Poglavlje 9

Lekcija 2 - Individualne vežbe

ZADATAK ZA SAMOSTALNI RAD 1 (45 MIN)

Napravite samostalno aplikaciju modifikovanjem prethodnog primera

1. Kreirajte nov projekat;
2. Projekat ima identične zahteve kao pokazna vežba;
3. Nije dozvoljeno da ulogu konfiguracione i glavne klase obavlja ista klasa;
4. Podelite navedene uloge u dve klase;

ZADATAK ZA SAMOSTALNI RAD 2(45 MIN)

Vežbajte i XML konfiguracije

1. Kreirajte nov projekat;
2. Projekat ima identične zahteve kao pokazna vežba i Zadatak1;
3. Kreirajte XML konfiguracioni dokument za upravljanje kontekstom aplikacije;
4. Kreirajte glavnu klasu za učitavanje konteksta aplikacije, zrna iz konteksta aplikacije i demonstraciju traženog ponašanja.

Poglavlje 10

Domaći zadatak 2

ZADATAK 1

Uradite samostalno domaći zadatak 1

Koristeći razvojno okruženje *NetBeans IDE*, kreirati prvu *Spring* aplikaciju po sledećim zahtevima:

1. Kreirati *Maven* projekat aplikacije pod nazivom *DomaciZadatak2*;
2. Dodati u *pom.xml* sledeće zavisnosti: *spring-core* i *spring-context*;
3. Kreirati interfejs *Zivotinja.java* sa metodama *nacinOglasavanja()* i *nahraniZivotinju()*;
4. Kreirati implementacione klase *Pas.java*, *Kokoska.java*, *Ovca.java*.
5. Napraviti servise po uzoru na materijale sa predavanja;
6. Kreirati konfiguracionu klasu koja inicijalizuje zrna *kreiranih* klasa;
7. Kreirati glavnu klasu koja kreira i učitava kontekst aplikacije iz konfiguracione klase, učitava podatke o životinjama i prikazuje informaciju kako se životinja oglašava i da li je gladna.

ZADATAK 2

Uradite samostalno domaći zadatak 2

Prethodni zadatak uraditi koristeći XML konfiguracije.

Nakon urađenih obaveznih zadataka, studenti dobijaju od predmetnog asistenta različite primere na email.

Poglavlje 11

Zaključak

ZAKLJUČAK

Lekcija se bavila savladavanjem osnovnog Spring IoC kontejnera

Lekcija je stavila poseban akcenat na pokrivanje osnova centralne *Spring* komponente - *Spring IoC kontejnera*. Sa posebnom pažnjom su obrađene sledeće teme:

- *Spring IoC kontejner* (eng.*Spring IoC Container);*
- *inverzija kontrole* (eng. *Inversion of Control*);
- *umetanje zavisnosti* (eng. *Dependency Injection*):
- umetanje zavisnosti preko konstruktora, polja i *setter* metoda;
- automatsko povezivanje zavisnosti (eng. *Autowiring*);

Savladavanjem ove lekcije student će razumeti osnove Spring IoC kontejnera bez kojih nije moguće upustiti se u savladavanje tema koje slede. Materijal pokriva novije koncepte podešavanja Spring aplikacija koje su uvedene verzijom radnog okvira Spring 4.

LITERATURA

Prilikom pripremanja lekcije korišćena je najnovija pisana i veb literatura

Za pripremu lekcije korišćena je najnovija pisana i elektronska literatura:
1. Gary Mak, Josh Long, and Daniel Rubio, Spring Recipes Third Edition, Apress
2. Spring Framework Reference Documentation - https://docs.spring.io/spring-framework/docs/5.2.1.RELEASE/spring-framework-reference/
3. Craig Walls, Spring in Action, Manning
4. Craig Walls, Spring Boot in Action, Manning
5. Ranga Karanam, Naučite Spring 5, Kompjuter biblioteka

6. https://www.baeldung.com/inversion-control-and-dependency-injection-in-spring

7. https://o7planning.org/en/10127/spring-tutorial-for-beginners

Dopunska literatura:
1. http://www.javacodegeeks.com/tutorials/java-tutorials/enterprise-java-tutorials/spring-tutorials/
2. http://www.tutorialspoint.com/spring/
3. http://www.javatpoint.com/spring-tutorial