Spring

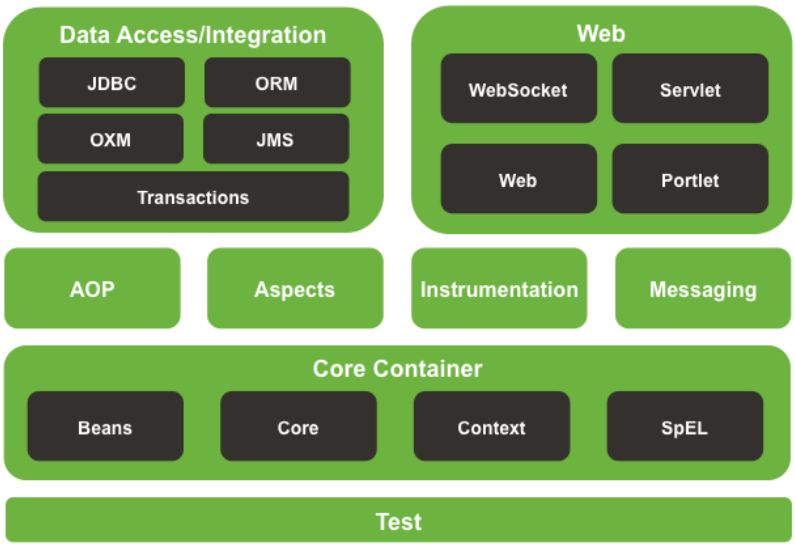
Autori:
Goran Savić
Milan Segedinac

1. Spring

Spring je skup biblioteka koje predstavljaju radni okvir (*eng*. framework) za jednostavniji razvoj poslovnih (eng. *enterprise*) aplikacija u Javi. Spring je već niz godina najpopularniji radni okvir ove vrste za programski jezik Java. Dizajniran je kao alternativa Java EE platformi. Java EE (Java Platform, Enterprise Edition) predstavlja proširenje standardne Java platforme (Java SE – Java Platform, Standard Edition). Java EE sadrži specifikaciju tehnologija koje se standardno koriste u poslovnim aplikacijama, npr. podrška za perzistenciju podataka kroz objektno-relaciono mapiranje , upravljanje transakcijama, komunikacija sa udaljenim računarima itd. Za jedan broj tehnologija postoje različita rešenja u Springu u odnosu na Java EE, dok se neke Java EE tehnologije koriste i u radu sa Spring radnim okvirom (npr. Java Persistence API (JPA) specifikacija za objektno-relaciono mapiranje).

Glavne karakteristike Spring radnog okvira su da omogućuje kreiranje poslovnih Java aplikacija na jednostavniji način i da sadrži infrastrukturu za izgradnju Java aplikacija. Spring upravlja infrastrukturom tako da programer može da se fokusira na domenske probleme umesto na tehnologiju i realizaciju. Dizajniran je tako da se standardne funkcionalnosti realizuju brzo i lako. Iako postoji ugrađena podrška da se svaka standardna funkcionalnost brzo realizuje, količina ugrađenih koncepata čini Spring prilično kompleksnom tehnologijom koja nije laka za razumevanje. Takođe, Spring je prošao kroz nekoliko verzija koje iste stvari realizuju na različite načine, što takođe komplikuje korišćenje ovog radnog okvira.

Spring radni okvir se sastoji iz većeg broja modula. Struktura Spring radnog okvira je prikazana na slici.



U ovom tekstu fokusirani smo na Spring Core modul (*Core Container*) u kojem se nalaze funkcionalnosti koje predstavljaju jezgro Spring radnog okvira. Svi ostali moduli koriste neke od funkcionalnosti iz ovog modula. Modul za pristup podacima (*Data Access/ Integration*) apstrahuje vezu ka različitim izvorima podataka. Veb modul omogućuje programiranje veb aplikacija. Delovima pomenutih modula ćemo se baviti u narednim lekcijama kada se budemo bavili implementacijom veb orijentisanih informacionih sistema putem Spring radnog okvira. U ovom tekstu, pored funkcionalnosti sadržanih u *Core*

modulu, pokrićemo i koncepte aspektno orijentisanog programiranja, kao i načine implementacije ovih koncepata u Springu (AOP, *Aspects*), obzirom da različiti moduli koriste ove koncepte. Pored samog Spring radnog okvira razvijen je veliki broj drugih okvira i biblioteka koji se koriste u okviru Spring aplikacija. U narednim lekcijama bavićemo se nekim od tih projekata. Konkretno, koristićemo SpringSecurity za bezbednost u aplikaciji, SpringData za upravljanje podacima, a SpringBoot za jednostavnije korišćenje samog Spring okvira.

Spring predstavlja skup biblioteka, tako da bi se Spring koristio u aplikaciji, potrebno je dobaviti ove biblioteke. Ukoliko se koristi maven za izgradnju projekta, potrebno je navesti Spring u listi zavisnosti projekta. Primer konfiguracije maven *pom.xml* je dat u nastavku.

```
<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>vp.spring</groupId>
  <artifactId>example</artifactId>
  <version>0.0.1-SNAPSHOT</version>
  <packaging>jar</packaging>

  <name>javaconfig</name>

  <dependencies>
      <dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring-context</artifactId>
        <version>4.3.0.RELEASE</version>
    </dependency>
  </dependencies>
</project>
```

Vidimo da u spisku zavisnosti stoji artefakt org.springframework.spring-context.

2. Spring kontejner

Glavna komponenta *core* modula je Spring kontejner. To je program čija je odgovornost da upravlja objektima u aplikaciji. Osnovna ideja je da se programer oslobodi obaveze da brine o životnom ciklusu objekata (kreiranje, inicijalizacija, konfiguracija, upotreba, uništenje). Umesto toga, Spring kontejner objekte kreira i evidentira, tako da različiti delovi koda mogu da dobiju željeni objekat na zahtev. Kontejner sadrži fabriku za kreiranje objekata (*eng. object factory*).

Objekti u Spring aplikaciji se nalaze u Spring kontejneru koji kreira objekte, povezuje objekte, konfiguriše objekte i upravlja objektima kroz njihov ceo životni ciklus. Postoje različite implementacija kontejnera. Svaka implementacija predstavlja jedan *application context*. Kontekst je zadužen da na specifičan način upravlja konfiguracijom aplikacije. Kontekst tumači značenje konfiguracionih fajlova i programskog koda. Od različitih konteksta koji postoje u Springu, ovde ćemo navesti dva:

- *AnnotationConfigApplicationContext* - učitava kontekst na osnovu konfiguracije definisane u okviru Java klasa
- *ClassPathXmlApplicationContext* - učitava kontekst na osnovu konfiguracije definisane u XML fajlovima

U primerima ćemo koristiti prvi od dva navedena konteksta.

Kontekst je zadužen da po potrebi dobavlja aplikaciji objekte kojima kontejner upravlja. Objekat kojim Spring kontejner upravlja se naziva Bean. U konfiguraciji se za objekat proglašava da je to Spring Bean objekata, čime je specificirano da će Spring brinuti o životnom ciklusu tog objekta. Takođe, ako je drugim objektima potreban Bean objekat, Spring će na sebe preuzeti dobavljanje objekta.

Pogledajmo primer definisanja Spring Bean objekta.

```
@Configuration
public class AppConfig {

	@Bean
	public CountryRepository countryRepository() {
		return new CountryRepository();
	}
}
```

U primeru je izabrano da se Bean objekat definiše kroz konfiguraciju u Java klasi. Vidimo da je definisana klasa *AppConfig* (ime je proizvoljno) koja je anotirana Spring anotacijom *@Configuration*. Ova anotacija označava da se unutar klase definiše jedan ili više Spring Bean objekata. U prikazanom primeru je definisano da objekti klase *CountryRepository* budu Spring Bean objekti.

Kao što smo objasnili, Spring kontekst je zadužen za evidenciju i dostupnost Bean objekata. U nastavku je prikazan kod za eksplicitno preuzimanje Spring Beana iz konteksta.

```
ApplicationContext context =
                new AnnotationConfigApplicationContext(AppConfig.class);
CountryRepository repo = context.getBean(CountryRepository.class);
```

Vidimo da je kontekst predstavljan objektom klase AnnotationConfigApplicationContext, jer smo za konfiguraciju konteksta koristili Java klasu. Java klasa u kojoj je konfiguracija konteksta se prosleđuje pri kreiranju konteksta.

3. Dependency injection (DI)

Kada se u aplikaciji referencira objekat koji je Spring Bean, Spring će preuzeti dobavljanje objekta klase. Ideja je da svaki objekat samo definiše svoje zavisnosti (tj. objekte koje koristi pri izvršavanju), a da Spring kontejner **injektuje** te zavisnosti kada se objekat kreira. Da bi kontejner ovo mogao da uradi, kontejner mora da upravlja objektima koji predstavljaju zavisnosti, tj. zavisnosti se moraju definisati kao Spring Bean objekti.

Dakle, kontejner koristi *dependency injection* mehanizam da poveže objekte tako što formira veze asocijacije između njih. Ovim sami objekti imaju čitljiviji i kraći kod i jednostaviji su za razumevanje.

Pomenuti princip u kojem objekat sam ne obezbeđuje zavisnosti, nego se one injektuju u njega se naziva **inverzija kontrole** (eng. *Inversion of Control - IoC*) ili **injekcija zavisnosti** (eng. *dependency injection - DI*).

Podrazumevano ponašanje je da će Spring kreirati samo jednu instancu za određeni Spring Bean i da će svi delovi koda koji referenciraju taj Bean dobijati tu istu instancu. Dakle, Spring Bean objekat je podrazumevano *singleton*. Moguće je konfigurisati i da ne bude *singleton, tj.* da se pri svakom referenciranju dobija nova instanca objekta. Pogledajmo u narednom primeru kako Spring automatski dobavlja Spring objekat kada se on referencira.

```
@Bean
public CountryService countryService(CountryRepository countryRepository) {
      return new CountryService(countryRepository);
}
```

U primeru je definisan novi Bean objekat tipa *CountryService*. Vidimo da je za instanciranje ovog Bean objekta, neophodan objekat tipa *CountryRepository*, koji je ranije definisan kao Bean. To znači da će Spring automatski obezbediti taj objekat na osnovu koda u ranije prikazanoj metodi *countryRepository()*.

Obzirom da ta metoda definiše Spring Bean objekat, čak i eksplicitni poziv te metode Spring presreće, tako da će Spring svaki put vratiti isti Bean objekat (jer je Bean podrazumevano *singleton*). Pogledajmo ovo na narednom primeru.

```
CountryRepository cr1 = countryRepository();
CountryRepository cr2 = countryRepository();
System.out.println(cr1 == cr2);
```

Prikazani kod dva puta eksplicitno poziva metodu countryRepository(), čiji kod kreira novu instancu klase *CountryRepository*. Ipak, prikazani deo koda će ispisati true, jer će Spring presresti poziv ove metode i neće se kreirati dva objekta klase *CountryRepository*, već će promenljive *cr1* i *cr2* dobiti referencu na isti Spring Bean objekat.

Vratimo se na kratko na prvi primer sa definisanjem *CountryRepository* Bean objekta da bismo pomenuli da samo ime metode koja definiše bean nije važno (npr. *countryRepository()* i *countryService()* u prethodnim primerima). Važan je tip metode i Spring će preuzeti dobavljanje objekata tog tipa ako se referenciraju u ostatku koda. Ime metode postaje važno tek ako ima više definisanih Spring Bean objekata istog tipa. Tada se promenljiva koja referencira Bean mora u ostatku koda nazvati u skladu sa nazivom metoda koja je definisala Bean.

4. Automatska konfiguracija

Moguće je konfigurisati Spring kontejner tako da se Bean objekti automatski registruju u kontekst i dobavljaju po potrebi iz konteksta.

Potrebno je klasu Bean objekta anotirati anotacijom @Component, kao u narednom primeru.

```
@Component
```

```
public class CountryRepository {

...

}
```

Ovako definisani Bean objekti mogu biti automatski pronađeni i učitani u Spring kontekst korišćenjem anotacije @ComponentScan u konfiguracionoj klasi. Dat je primer ovakve konfiguracione klase.

```
@Configuration
@ComponentScan("vp.spring.autocountry") // paket od kojeg krece skeniranje
public class AppConfig {
}
```

Vidimo da je kao i u ranijem primeru konfiguraciona klasa označena anotacijom @Config. Vrednost koja se prosleđuje anotaciji @ComponentScan specificira početni paket od kojeg pretraga Bean objekata kreće. Pretraga će počevši od tog paketa pretražiti i sve njegove potpakete. Ako se početni paket ne naznači, pretraga kreće od paketa u kojem se nalazi ova konfiguraciona klasa. Obzirom da su Bean objekti definisani u klasama anotacijom @Component, nema potrebe da se u konfiguracionoj klasi specificiraju Bean objekti, tako da je telo konfiguracone klase u ovom primeru prazno. Naravno, u opštem slučaju klasa ne mora biti prazna i može da sadrži konfiguraciju različitih aspekata Spring aplikacije (npr. konfiguraciju prava pristupa).

Obzirom da se u ovoj varijanti realizacije Bean objekti automatski specificiraju, drugačije se vrši i injekcija zavisnosti u Bean. Neki Spring Bean može biti automatski injektovan u drugi Bean korišćenjem anotacije @Autowired. Ova anotacija je ilustrovana na primeru injektovanja atributa tipa CountryRepository u objekat tipa CountryService.

```
@Component
public class CountryService {
	@Autowired
	CountryRepository countryRepository;

	...

}
```

Obzirom da je CountryService Spring Bean (zbog anotacije @Component), Spring će izvršiti instanciranje ove klase. Pri instanciranju će inicijalizovati vrednost atributa *countryRepository* tako da referencira ranije kreirani Bean objekat klase *CountryRepository*.

Osim nad atributom, anotacija @Autowired može biti postavljena i nad konstruktorom, kao i nad set metodom. U sva tri slučaja cilj anotacije je isti – da injektuje vrednost u određeni objekat. Dat je primer injektovanja Bean objekta kroz set metodu.

```
@Component
public class CountryService {
   CountryRepository countryRepository;

   @Autowired
   public void setCountryRepository(CountryRepository countryRepository) {
      this.countryRepository = countryRepository;

   }
```

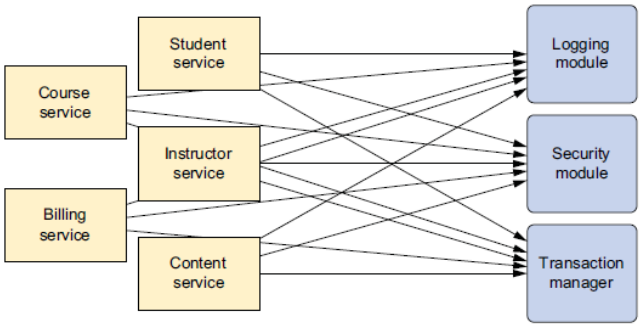
```
    ...
}
```

Kao i kod varijante sa direktnim injektovanjem vrednosti atributa, Spring će automatski pozvati ovu set metodu i time postaviti vrednost atributa. Slično, ova vrednost može biti postavljena i u konstruktoru, kao što je prikazano u narednom primeru.

```
@Component
public class CountryService {
    CountryRepository countryRepository;

    @Autowired
    public CountryService(CountryRepository countryRepository) {
        this.countryRepository = countryRepository;
    }
    ...
}
```

5. Aspektno orijentisano programiranje

Određene funkcionalnosti prožimaju većinu drugih funkcionalnosti u aplikaciji. Npr. evidencija događaja (logging), upravljanje transakcijama, kontrola pristupa, itd. Ako ovaj tip funkcionalnosti implementiramo na klasičan način isti kod će se ponavljati ili pozivati u svim delovima aplikacije, a glavna funkcionalnost će biti zaprljana kodom te dodatne funkcionalnosti. Ovakav pristup ilustrovan je na slici.



Vidimo da sve komponente aplikacije moraju da koriste module za logovanje, bezbednost i upravljanje transakcijama.

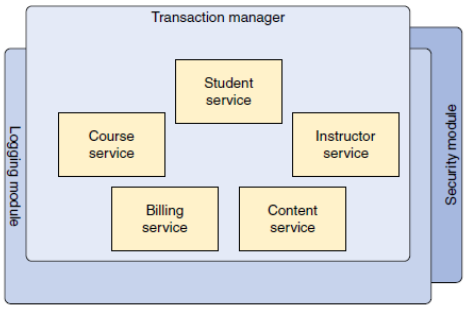
Kao primer, pogledajmo kako bi izgledala implementacija evidencije događaja u aplikaciji na klasičan način. Ako bismo želeli da ispišemo u konzolu informaciju da je određena metoda pozvana, to bismo mogli učiniti na način prikazan u sledećem kodu.

```
public class CountryService {
	@Autowired
	CountryRepository countryRepository;

	public Country findOne(int id) {
		System.out.println("Method findOne called.");
		return countryRepository.findOne(id);
	}

	public List<Country> findAll() {
		System.out.println("Method findAll called.");
```

```
            return countryRepository.findAll();
      }

      public void save(Country country) {
            System.out.println("Method save called.");
            countryRepository.save(country);
      }

      ...

}
```

Vidimo da je pored svoje glave funkcionalnosti svaka metoda morala biti proširena vrlo sličnim kodom koji evidentira poziv metode. Osim ponavljanja koda, nedostatak ovakvog pristupa je i smanjena čitljivost koda, jer je teže fokusirati se na glavnu funkcionalnost pri analizi koda.

Aspektno orijentisano programiranje (AOP) omogućuje organizovanje često korišćenih funkcionalnosti u komponente koje se mogu višekratno koristiti. AOP omogućuje primenu ovih komponenata na druge delove aplikacije. Primena se vrši na deklarativan način, što znači da nije potrebno eksplicitno pozivati naredbe programa, već se samo specificira da određeni aspekt treba biti primenjen na određenom mestu u aplikaciji. Ovim poslovna logika onda ne mora eksplicitno da se bavi funkcionalnostima koje su obezbeđene kroz aspekte, pa je glavna funkcionalnost fokusirana na svoju glavnu funkciju, a dodatne funkcije se deklarativno primenjuju kroz aspekte. To dovodi do čitkijeg koda koji sadrži samo logiku glavne funkcionalnosti. Aspektno orijentisan pristup u izradi aplikacije je ilustrovan na narednoj slici.



Vidimo da sada upravljanje transakcijama, evidencija događaja i bezbednost „prožimaju“ sve glavne funkcionalnosti aplikacije.

Kao što smo ranije pomenuli, Spring sadrži podršku za AOP. Za svaki aspekt potrebno je definisati kod koji aspekt izvršava, kao i tačku u aplikaciji gde se aspekt primenjuje. Aspekt se u Springu definiše u metodi proizvoljne klase, koja mora biti anotirana anotacijom @Aspect. Takođe, obzirom da Spring treba da preuzme upravljanje objektima ove klase, klasa mora biti Spring Bean, što se postiže anotacijom @Component. Sadržaj metode predstavlja kod koji će biti izvršen pri primeni aspekta. Pored toga, različitim anotacijama se specificira kada se aspekt primenjuje. Podržane su sledeće tri glavne anotacije:

- @Before – aspekt će se izvršiti pre određene metode
- @After - aspekt će se izvršiti nakon određene metode

- @Around – deo aspekta će se izvršiti pre, a deo posle određene metode

Vidimo da je u sve tri varijante neophodno naznačiti na koju metodu se aspekt odnosi (pre, posle ili oko koje metode će kod aspekta biti izvršen). Tačka primene aspekta se u terminologiji AOP naziva pointcut, pa se sintaksa u kojoj se ovo specificira u Springu naziva Poincut Expression i preuzeta je iz AspectJ projekta.

Pogledajmo kako bi u Springu izgledala implementacija evidentiranja događaja u aplikaciji primenom AOP.

```
@Component
@Aspect
public class LogAspect {

	@Before("execution(* vp.spring.CountryService.*(..))")
	public void logEvent () {
		System.out.println("CountryService method called.");
	}
}
```

Obzirom da za implementaciju aspekata koristimo AspectJ projekat, neophodno je da u spisku zavisnosti budu dve biblioteke iz ovog projekta. To su `org.aspectj.aspectjrt` i `org.aspectj.aspectjweaver`.

U primeru vidimo da je klasa anotirana da definiše aspekte. Važno je naglasiti da je neophodno u konfiguracionoj klasi dodati anotaciju @EnableAspectJAutoProxy, kako bi Spring kontejner mogao pri skeniranju komponenti da pronađe i AspectJ aspekte. Aspekt treba da se izvrši svaki put pre izvršavanja neke metode klase CountryService. Ovo je definisano @Before anotacijom i *pointcut* izrazom koji joj je prosleđen. Putem pointcut izraza moguće je specificirati povratni tip, naziv paketa, naziv klase, naziv metode i parametre metode na koju će se aspekt odnositi. U prikazanom primeru, *pointcut* se odnosi na sve metode bilo kojeg naziva, parametara i povratnog tipa (definisano džoker znakom * i oznakom .. kod parametara), iz paketa vp.spring, iz klase CountryService.

Na sličan način moguće je definisati aspekt koji se izvršava nakon poziva neke metode. Korišćenje anotacije @Around je nešto specifičnije, pa će biti objašnjeno dodatnim primerom. Pogledajmo kako bi izgledao aspekt koji deo koda treba da izvrši pre, a deo koda posle poziva metode.

```
@Around("execution(* vp.spring. CountryService.*(..))")
public Object logEvent(ProceedingJoinPoint jp) throws Throwable {
	System.out.println("Before method " + jp.getSignature());
	Object result = jp.proceed();
	System.out.println("After method: " + jp.getSignature());
	return result;
}
```

U ovom slučaju, aspekt je zadužen da inicira poziv metode oko koje se izvršava. Objekat *jp* klase *ProceedingJoinPoint* koji aspekt dobija kao parametar, sadrži podatke o metodi oko koje se aspekt izvršava. Pozivom metode *proceed* inicira se poziv ove metode. Vidimo da pre i nakon poziva metode, aspekt vrši ispis događaja. Metoda *getSignature* vraća detalje o metodi oko koje se aspekt poziva (povratni tip, paket, klasa, naziv i parametri metode). Obzirom da metoda vraća rezultat, aspekt je zadužen da rezultat prosledi pozivaocu metode. Iz tog razloga metoda *logEvent* u kojoj je aspekt definisan je ovog puta tipa Object i na kraju izvršavanja vraća rezultat.

Pomenuti način definisanja aspekata omogućuje centralizovano definisanje koda koji može biti izvršen u različitim delovima aplikacije. Mana ovakvog pristupa je što smanjuje čitljivost koda jer se uvidom u metodu ne vidi koji aspekt se izvršava pri pozivu metode. Da bi se to saznalo potrebno je analizirati *pointcut* izraze u svim aspektima u aplikaciji. Zato je dobro definisati aspekt tako da se aspekt primenjuje na mestima u kojima je to eksplicitno označeno.

Pogledajmo kako bi izgledala realizacija ovakvog pristupa ukoliko bismo ranije definisani aspekt za evidenciju događaja hteli da primenimo isključivo na metode eksplicitno anotirane novom anotacijom @LogEvent. Najpre je neophodno definisati ovu novu anotaciju. Nove anotacije se u Javi zadaju na sledeći način.

```
@Retention(RetentionPolicy.RUNTIME)
public @interface LogEvent {
}
```

Primetimo da je anotacija dodatno anotirana anotacijom @Retention koja označava da anotacija mora biti dostupna u Java virtuelnoj mašini u toku izvršavanja kako bi Spring kontejner mogao da pronađe sve metode anotirane ovom anotacijom.

Sada je potrebno anotirati ovom anotacijom sve metode nad kojima želimo da se aspekt izvrši. Primer postavljanja anotacije nad jednom metodom klase CountryRepository je dat u primeru.

```
@Component
public class CountryRepository {
	@LogEvent
	public List<Country> findAll() {
		...
	}
}
```

Vidimo da je metode *findAll* anotirana anotacijom @LogEvent. Sada možemo definisati aspekt koji bi bio primenjen pre poziva svake metode koja ima ovu anotaciju.

```
@Before("@annotation(vp.spring.aspect.LogEvent)")
public void logAnnotatedMethod() {
	System.out.println("LogEvent: before method");
}
```

Vidimo da je pointcut izraz definisan tako da se odnosi na metode koje imaju anotaciju *LogEvent* definisanu u paketu vp.spring.aspect.

6. Spring Expression Language (SpEL)

Spring definiše poseban jezik pod nazivom Spring Expression Language (SpEL) za preuzimanje objekata u toku izvršavanja aplikacije. Za razumevanje svrhe ovog jezika možemo iskoristiti analogiju sa bazama podataka. Dok SQL jezik omogućuje preuzimanje podataka uskladištenih u bazi podataka, SpEL omogućuje preuzimanje svih podataka kojima kontejner ima pristup u toku izvršavanja aplikacije. Ovo se odnosi na Bean objekte, ali i na različite konfiguracione vrednosti.

U ovom tekstu nećemo ulaziti u sve detalje SpEL jezika koji je dosta kompleksan. Samo ćemo na jednom primeru ilustrovati kako se jezik može koristiti. Pogledajmo primer kako dužina liste uskladištene u jednom Bean objektu može biti injektovana u atribut drugog objekta.

```
@Value("#{countryRepository.getCountries().size()}")
private int numberOfCountries;
```

Vrednost se injektuje anotacijom @Value čiji je parametar SpEL izraz. Ovaj izraz nad Bean objektom pod nazivom *countryRepository* poziva metodu *getCountries* koja vraća listu država, nad kojom dalje poziva metodu *size()* koja preuzima veličinu ove liste. Treba naglasiti da se vrednost injektuje samo jednom i to pri instanciranju objekta. Osim pristupa grafu objekata, SpEL podržava i aritmetičke i logičke operatore, kao i još niz dodatnih funkcija, koje ovde nećemo objašnjavati zbog obimnosti.