Univerzitet Singidunum

Miodrag Živković
Nebojša Bačanin Džakula
Eva Tuba

PROGRAMSKI JEZICI

VEB SERVISI I NAPREDNO JAVA PROGRAMIRANJE U SPRING OKRUŽENJU

Beograd, 2022.

UNIVERZITET SINGIDUNUM

Miodrag Živković
Nebojša Bačanin Džakula
Eva Tuba

PROGRAMSKI JEZICI

Veb servisi i napredno Java programiranje u Spring okruženju

Drugo izdanje

Beograd, 2022.

PROGRAMSKI JEZICI
Veb servisi i napredno Java programiranje u Spring okruženju

Autori:
Prof. dr Miodrag Živković
Prof. dr Nebojša Bačanin Džakula
Doc. dr Eva Tuba

Recenzenti:
Prof. dr Mladen Veinović
Prof. dr Dejan Živković
Prof. dr Boško Nikolić

Izdavač:
UNIVERZITET SINGIDUNUM
Beograd, Danijelova 32
www.singidunum.ac.rs

Za izdavača:
Prof. dr Milovan Stanišić

Priprema za štampu:
Jelena Petrović

Dizajn korica:
Aleksandar Mihajlović, MA

Godina izdanja:
2022.

Tiraž:
700 primeraka

Štampa:
Caligraph, Beograd

ISBN: 978-86-7912-698-6

Copyright:
© 2022. Univerzitet Singidunum
Izdavač zadržava sva prava.
Reprodukcija pojedinih delova ili celine ove publikacije nije dozvoljena.

SADRŽAJ

Lista slika VII
Predgovor XII

I - Uvod u mrežno programiranje 1
1.1. Računarska mreža 1
1.2. Mrežni slojevi 2
1.3. Mrežni i transportni sloj - IP, TCP i UDP 5
1.4. IP adrese i domenska imena 6
1.5. Aplikacioni sloj i HTTP 7
1.5.1. Funkcionisanje HTTP protokola 8
1.5.2. HTTP request 9
1.5.3. HTTP response 10
1.5.4. Performanse HTTP protokola 11
1.5.5. HTTP keširanje i proxy 12

II - Niti 15
2.1. Prednosti višenitnog programiranja 16
2.2. Osnovni pojmovi 18
2.3. Interfejs Runnable 19
2.4. Klasa Thread 21
2.5. Kreiranje i pokretanje niti 23
2.6. Naredba join 27
2.7. Lambda izrazi sa Runnable interfejsom 28
2.8. Sinhronizacija niti 31
2.8.1. Mrtva petlja 36
2.9. Saradnja niti 42
2.10. Semafori 46
2.11. Zadaci za vežbu 51

III - Klijent-server programiranje 107
3.1. Klijent-server 109
3.2. Jednostavni klijent-server programi 112
3.3. Višenitno mrežno programiranje 115
3.4. Zadaci za vežbu 121

IV - Uvod u Spring okruženje 156
4.1. POJO 157
4.2. Injektovanje zavisnosti (DI) 157
4.3. Primena aspekata 163
4.4. Eliminisanje viška koda upotrebom šablona 169
4.5. Spring arhitektura okruženja 170
4.6. Osnovno konfigurisanje Spring projekta 173
4.7. Prvi Spring projekat 176

V - Konfiguracija zrna u Spring okruženju 182
5.1. Zrna i njihov životni ciklus 184
5.2. Deklaracija zrna 186
5.3. Osnovna konfiguracija zrna 187
5.4. Injekcija zavisnosti - DI 195
5.5. Metode init i destroy 200
5.6. Opseg zrna 203
5.7. Injektovanje kolekcija 204
5.8. Injektovanje liste zrna 212
5.9. Zadaci za vežbu 216

VI - Spring - Automatsko ožičavanje zrna 235
6.1. Automatsko ožičavanje po imenu 236
6.2. Automatsko ožičavanje po tipu 241
6.3. Automatsko ožičavanje po konstruktoru 244
6.4. Podrazumevano automatsko ožičavanje 247
6.5. Automatsko ožičavanje pomoću anotacija 248
6.6. Kvalifikatori 253
6.7. Inject anotacija 254
6.8. Automatsko otkrivanje zrna 256
6.9. Zadaci za vežbu 260

VII - Aspektno orijentisano programiranje u Springu 273
7.1. Koncept AOP i osnovni pojmovi 273
7.2. Podrška za AOP u Springu 276
7.3. Definisanje tačke preseka i programiranje aspekta (XML) 277
7.4. Around savet, XML 289
7.5. Aspekti sa anotacijama 293
7.6. Definisanje tačaka preseka sa džokerima 296
7.7. Zadaci za vežbu 297

VIII - Osnove rada sa bazom podataka u Spring okruženju 308
8.1. Filozofija pristupa podacima u Springu 308
8.2. Obrada SQL izuzetaka u Springu 310
8.3. Šabloni za pristup podacima 311
8.4. Implementacija pristupa bazi podataka 313
8.5. Uvođenje imenovanih parametara 323

IX - Uvod u veb servise i osnovna terminologija 331
9.1. Pojam, vrste, arhitektura, osnovne karakteristike i prednosti veb servisa 334
9.1.1. Vrste i osnovne komponente veb servisa 336
9.1.2. Prednosti veb servisa 339
9.1.3. Arhitektura veb servisa 340
9.1.4. Osnovne karakteristike veb servisa 341
9.2. SOAP veb servisi 342
9.3. Primer SOAP veb servisa u Visual Studio okruženju 349

X - RESTful veb servisi 357
10.1. Ključni elementi RESTful veb servisa 357
10.2. RESTful metode 358
10.3. Razlozi za sve veću upotrebu RESTful veb servisa 360
10.4. RESTful arhitektura, principi i ograničenja 361

XI - Kratak uvod u Spring Boot Framework 364
11.1. Konfigurisanje Spring Boot aplikacije 366
11.2. Spring Boot Starter projekti 366
11.3. Osnovne Spring Boot anotacije 368

XII - Primeri i implementacije RESTful veb servisa 370
12.1. Primer 1: Implementacija projekta RESTful veb servisa pomoću Spring Initializr alata u Eclipse razvojnom okruženju 371
12.1.1. Implementacija pokaznog HelloWorld primera 376
12.1.2. Implementacija RESTful veb servisa sa listom korisnika 380
12.2. Primer 2: Jednostavni veb servisi za prodavnicu knjiga i video igara 387
12.3. Primer 3: Jednostavan REST API za veb blog postove u IntelliJ IDEA okruženju 404
12.3.1. Implementacija veb servisa bez baze podataka 405
12.3.2. Implementacija Blog veb servisa sa bazom podataka 423

Literatura 439

LISTA SLIKA

Slika 1.1. Uporedni prikaz TCP/IP i OSI modela 3
Slika 1.2. Horizontalna i vertikalna komunikacija kroz TCP/IP slojeve 4
Slika 1.3. Struktura IP datagrama (slika preuzeta sa https://www.w3.org/) 5
Slika 1.4. Funkcionisanje HTTP protokola (izvor: http://etutorials.org) 8
Slika 1.5. Format HTTP request poruke (izvor: developer.mozilla.org) 9
Slika 1.6. Format HTTP response poruke (izvor: developer.mozilla.org) 10
Slika 1.7. HTTP keširanje 13
Slika 1.8. HTTP proksi 13
Slika 2.1. Dijagram nasleđivanja klase Thread 19
Slika 2.2. Mehanizam katanca za sinhronizaciju 36
Slika 2.3. Mrtva petlja u Javi 36
Slika 2.4. Problem filozofa koji večeraju 37
Slika 2.5. Dijagram rada semafora 47
Slika 3.1. Unos IP adrese u DatumKlijent programu 115
Slika 3.2. Prikaz podataka pristiglih sa DatumServer programa 115
Slika 3.3. Pozdravna poruka servera 120
Slika 3.4. Unos teksta na klijentskoj strani 120
Slika 3.5. Prikaz kapitalizovanog stringa koji je stigao od server 121
Slika 4.1. Pozivi sistemskih servisa u modulima gde im to nije osnovna funkcionalnost 164
Slika 4.2. AOP i primena servisa na komponente na koje trebaju da imaju uticaj 165
Slika 4.3. Spring moduli (izvor: https://docs.spring.io/spring/docs/5.0.0.RC3/spring-framework-reference/overview.html) 171
Slika 4.4. Eclipse Marketplace za dodavanje biblioteka 173
Slika 4.5. Instalacija Maven preko Eclipse Marketplace 174
Slika 4.6. Maven Repositories 175
Slika 4.7. Dodavanje referenci kroz pom.xml 176
Slika 4.8. Dodavanje Spring Tools kroz Marketplace 176
Slika 4.9. Kreiranje Maven projekta 177

Slika 4.10. Osnovna podešavanja Maven projekta 177
Slika 4.11. Podešavanje Group Id i Artefact Id 178
Slika 4.12. Prikaz projekta nakon kreiranja u Project Explorer tabu 178
Slika 4.13. Pregledni prikaz pom.xml fajla 179
Slika 4.14. Dodavanje zavisnosti u pom.xml 179
Slika 4.15. Prikaz dodatih referenci u tabu Project Explorer 180
Slika 5.1. Spring kontejner koji kreira i ožičava objekte 182
Slika 5.2. Proces kreiranja zrna 184
Slika 5.3. Tipičan životni ciklus zrna u Spring okruženju (slika preuzeta sa https://docs.spring.io) 185
Slika 5.4. Dodavanje konfiguracionog fajla zrna, korak 1 188
Slika 5.5. Dodavanje konfiguracionog fajla zrna, korak 2 188
Slika 5.6. Dodavanje konfiguracionog fajla zrna, korak 3 188
Slika 5.7. Beans.xml editor 189
Slika 5.8. Meni za dodavanje novog zrna 189
Slika 5.9. Postavljanje id zrna i povezivanje sa klasom 190
Slika 5.10. Odabir klase sa kojom se povezuje zrno sa Id osoba 190
Slika 5.11. Potvrda kreiranja zrna osoba 190
Slika 5.12. Dodavanje argumenta u konstruktor 193
Slika 5.13. Dodavanje parametra tipa String u konstruktor 194
Slika 5.14. Dodavanje parametra tipa long u konstruktor 194
Slika 5.15. Dodavanje zrna klase Adresa 197
Slika 5.16. Injekcija parametara klase Adresa kroz konstruktor 197
Slika 5.17. Injekcija polja u klasu Osoba 198
Slika 5.18. Injekcija primitivne vrednosti kroz set metodu 198
Slika 5.19. Injekcija objekta kroz set metodu 199
Slika 5.20. Dodavanje init i destroy metoda 202
Slika 5.21. Dodavanje podrazumevanih init i destroy metoda 203
Slika 5.22. Opseg zrna 204
Slika 5.23. Konfigurisanje zrna za korpu sa voćem 207
Slika 5.24. Injektovanje liste stringova kroz konstruktor, korak 1 207
Slika 5.25. Injektovanje liste stringova kroz konstruktor, korak 2 207
Slika 5.26. Injektovanje liste stringova kroz konstruktor, korak 3 208
Slika 5.27. Injektovanje liste stringova kroz konstruktor, korak 4 208
Slika 5.28. Injektovanje liste zrna 214
Slika 5.29. Beans Graph tab u editoru 221

Slika 6.1. Zrna u programu Loger 239
Slika 6.2. Podešavanje automatskog ožičavanja po imenu u editoru 239
Slika 6.3. Podešavanje automatskog ožičavanja po tipu u editoru 241
Slika 6.4. Podešavanje automatskog ožičavanja po konstruktoru u editoru 246
Slika 6.5. Postavljanje podrazumevanog ožičavanja u editoru 247
Slika 6.6. Uključivanje context imenskog prostora pomoću editora 248
Slika 6.7. Dodavanje podrške za anotacije pomoću editora 249
Slika 6.8. Dodavanje podrške za automatsko skeniranje zrna pomoću editora 257
Slika 6.9. Postavljanje početnog paketa za automatsko skeniranje 257
Slika 7.1. Aktivnosti koje se protežu preko više modula u aplikaciji 274
Slika 7.2. Aspekt se primenjuje na jednoj ili više tačaka primene za vreme izvršavanja programa. 275
Slika 7.3. Omovatanje zrna proksi klasom prilikom vezivanja aspekata 277
Slika 7.4. Dodavanje aop imenskog prostora u editoru 282
Slika 7.5. Dodavanje korenog aop elementa 283
Slika 7.6. Dodavanje elementa koji definiše tačku preseka 283
Slika 7.7. Definisanje tačke preseka na izvršavanju metode Telefon.pozovi() 284
Slika 7.8. Definisanje tačke preseka na izvršavanju metode Telefon.posaljiPoruku() 284
Slika 7.9. Dodavanje aspekta unutar <aop:config> elementa 285
Slika 7.10. Podešavanje aspekta i dodavanje reference na zrno koje će pružiti funkcionalnost 285
Slika 7.11. Podešavanje kada će se savet aspekta izvršiti 286
Slika 7.12. Podešavanje *before* saveta 286
Slika 7.13. Podešavanje after saveta 288
Slika 7.14. Podešavanje around saveta u editoru 291
Slika 7.15. Konfigurisanje aspectj-autoproxy elementa 293
Slika 8.1. Delegiranje pristupa podacima repozitorijumu 309
Slika 8.2. Hijerarhija DataAccessException izuzetaka 311
Slika 8.3. Klasa šablon i callback mehanizam 312
Slika 8.4. Pregled jedne tabele u bazi u okviru alata MySQL Workbench 314
Slika 8.5. Potrebne zavisnosti za konektore ka MySQL i Oracle bazama podataka 314
Slika 9.1. Generisanje JAR-a i import u drugu aplikaciju 332
Slika 9.2. Generisanje JAR-a i import u drugu aplikaciju 332
Slika 9.3. Definicija servisa 334
Slika 9.4. Uprošćena arhitektura veb servisa 335
Slika 9.5. Jednostavan primer komunikacije preko SOAP protokola 337

Slika 9.6. Interakcija između klijenta, provajdera i registra usluge 341
Slika 9.7. Gradivni blokovi SOAP poruke 344
Slika 9.8. Primer SOAP poruke 345
Slika 9.9. Primer RPC komunikacije 348
Slika 9.10. Primer SOAP komunikacije 349
Slika 9.11. Kreiranje SOAP veb servisa – Korak 1 350
Slika 9.12. Kreiranje SOAP veb servisa – Korak 2a 351
Slika 9.13. Kreiranje SOAP veb servisa – Korak 2b 351
Slika 9.14. Kreiranje SOAP veb servisa – Korak 3a 352
Slika 9.15. Kreiranje SOAP veb servisa – Korak 3b 352
Slika 9.16. Kreiranje SOAP veb servisa – Korak 3c 353
Slika 9.17. Kod fajla SingidunumService.asmx – Korak 4 353
Slika 9.18. Interfejs veb servisa 354
Slika 9.19. Odgovor veb servisa 354
Slika 9.20. SOAP zahtev 1.1 355
Slika 9.21. SOAP zahtev 1.2 355
Slika 9.22. SOAP odgovor 1.1 355
Slika 9.23. SOAP odgovor 1.2 355
Slika 10.1. Primeri upotrebe GET, POST, PUT i DELETE metoda 359
Slika 10.2. Veb aplikacija i komunikacija sa Facebook, Twitter i Google servisima 360
Slika 10.3. RESTful klijent – server 362
Slika 10.4. Mehanizam keširanja 363
Slika 11.1. Arhitektura aplikacije za online kupovinu bez mikro servisa (slika preuzeta sa: http://amilasilva88.blogspot.com) 364
Slika 11.2. Arhitektura aplikacije za online kupovinu sa mikro servisima (slika preuzeta sa: http://amilasilva88.blogspot.com) 365
Slika 12.1. Konfigurisanje Spring Initializr projekta 372
Slika 12.2. Import Maven projekta u Eclipse (1) 373
Slika 12.3. Import Maven projekta u Eclipse (2) 373
Slika 12.4. Import Maven projekta u Eclipse (3) 374
Slika 12.5. Pokretanje veb servisa 374
Slika 12.6. Pozdravna HelloWorld poruka 378
Slika 12.7. Prikaz HelloWorldBean instance u JSON formatu 378
Slika 12.8. Poziv HelloWorldBean sa promenljivom putanje 379
Slika 12.9. HTTP GET zahtev koji prikazuje listu svih korisnika u JSON format 385
Slika 12.10. HTTP GET zahtev koji prikazuje korisnika sa id brojem 2 u JSON format 385

Slika 12.11. HTTP POST zahtev kojim se kreira novi korisnik 386
Slika 12.12. Promena vrednosti podrazumevanog Tomcat porta 386
Slika 12.13. Dodavanje biblioteka u IntelliJ projekat 399
Slika 12.14. GUI klijentske aplikacije 404
Slika 12.15. *Screenshot* – kreiranje novog projekta u IntelliJ IDEA okruženju 405
Slika 12.16. *Screenshot* – izbor opcije za Maven projekat u IntelliiJ IDEA okruženju 406
Slika 12.17. *Screenshot* – osnovne informacije o novom Maven projekat u IntelliJ IDEA okruženju 406
Slika 12.18. *Screenshot* – novi Maven projekat sa pom.xml fajlom 407
Slika 12.19. *Screenshot* – ažuriranje Maven repozitorijuma 408
Slika 12.20. *Screenshot* – kreiranje paketa (levo) i kreiranje klase (desno) 409
Slika 12.21. *Screenshot* – http://localhost:8080 410
Slika 12.22. *Screenshot* – http://localhost:8080 411
Slika 12.23. Atributi klase Blog i Blog tabela u bazi podataka 411
Slika 12.24. Metode klase BlogData 413
Slika 12.25. Prikaz svih blog postova u JSON format pomoću Postman aplikacije 420
Slika 12.26. Prikaz jednog bloga u JSON format u Postman aplikaciji 421
Slika 12.27. Kreiranje novog bloga pomoću POST zahteva u Postman aplikaciji 422
Slika 12.28. Ažuriranje postojećeg bloga pomoću PUT zahteva u Postman aplikaciji 422
Slika 12.29. Brisanje bloga pomoću DELETE zahteva u Postman aplikaciji 423
Slika 12.30. PHPMyAdmin veb bazirani interfejs za upravljanje MySQL bazom 424
Slika 12.31. Struktura tabele Blog 424
Slika 12.32. Baza restws i tabela blog 425
Slika 12.33. Kreiranje fajla application.properties 429
Slika 12.34. Primer 1: kreiranje novog bloga 437
Slika 12.35. Primer 2: prikaz svih blogova iz baze 437
Slika 12.36. Primer 3: prikaz blogova koji zadovoljavaju određen kriterijume pretrage 438
Slika 12.37. Primer 4: prikaz jednog bloga iz baze 438

PREDGOVOR

Programski jezik Java, kao jedan od najpopularnijih programskih jezika u svetu, svoju popularnost duguje pre svega eleganciji, savremenim konceptima i jednostavnom objektno orijentisanom modelu. Programi napisani u Javi se mogu pronaći svuda, od Android mobilnih uređaja, preko klasičnih desktop aplikacija, pa sve do Internet strana i veb servisa. Zbog sve veće popularnosti veb aplikacija, fokus programiranja u Javi se pomera sa tradicionalnih desktop aplikacija na mrežno programiranje i servisno orijentisano programiranje, što je ujedno i primarni fokus ovog udžbenika.

Kako bi se razumeo koncept servisno orijentisanog programiranja, potrebno je poznavati paradigme na kojima je zasnovan, uključujući objektno orijentisano programiranje (OOP), klijent – server programiranje i modele komponenti. OOP se bazira na premisi da se problem koji se rešava može modelovati objektima u domenu problema. OOP ima svoje specifične karakteristike, poput nasleđivanja, enkapsulacije i polimorfizma. Servisno orijentisano programiranje se nadgrađuje na OOP, dodavanjem premise da se problem koji se rešava može modelovati u obliku servisa koji objekat pruža ili koristi.

Jedan od najpopularnijih programskih okvira za pisanje veb aplikacija je Spring Framework, koji spada u kategoriju platformi otvorenog koda. Okvir se može koristiti za bilo koje Java aplikacije, ali osnovna primena je u implementaciji veb aplikacija nad Java EE platformom. Spring pored standardnih koncepata OOP uvodi i intenzivno korišćenje šablona injekcije zavisnosti i aspektno orijentisanog programiranja. Predstavlja jedno od standardnih izbora za pisanje serverskih komponenti koda u Javi. Pošto je u pitanju veoma često korišćeni programski okvir, veliki deo ove knjige se bazira na njemu.

Obim i sadržaj udžbenika prilagođeni su nastavnom programu istoimenog predmeta na Fakultetu za informatiku i računarstvo Univerziteta Singidunum u Beogradu. Podrazumeva se da čitaocima knjige ovo nije prvi susret sa programiranjem u Javi, tj. očekuje se osnovno poznavanje programskog jezika Java, kao i principa objektno orijentisanog programiranja.

Tekst je propraćen velikim brojem slika i detaljno objašnjenih primera pomoću kojih su ilustrovani novouvedeni pojmovi. Primeri su odabrani na takav način da budu što jednostavniji za razumevanje, ali sa druge strane i što realističniji i interesantniji, sa krajnjim ciljem da podstaknu čitaoce na dublje samostalno učenje i budu početna tačka za dalje eksperimentisanje. Iako je knjiga pisana kao udžbenik, ambicija autora jeste da ona posluži svima koji se prvi put susreću sa servisno orijentisanim programiranjem u programskom jeziku Java. Na kraju većine poglavlja data su pitanja koja mogu pomoći u utvrđivanju gradiva.

Sve primedbe, komentari, preporuke, pohvale i eventualno uočene greške se mogu poslati na adresu mzivkovic@singidunum.ac.rs, nbacanin@singidunum.ac.rs i etuba@singidunum.ac.rs.

Beograd,
Srbija, 2018.

Autori

I
Uvod u mrežno programiranje

Cilj ovog poglavlja jeste da uvede osnovne koncepte i pojmove mrežnog programiranja koje je neophodno poznavati pre pisanja bilo koje vrste mrežnih programa, kako u programskom jeziku Java, tako i u bilo kom drugom programu. Počinje se od opštih pojmova, poput računarske mreže, sve do specifičnih protokola koji se koriste u umrežavanju.

1.1. Računarska mreža

Računarska mreža je kolekcija računara i drugih uređaja koji mogu međusobno da šalju i primaju podatke u realnom vremenu. Sama fizička povezanost računara može se postići različitim sredstvima, počev od žičanih kablova, preko optičkih kablova, bežičnih veza itd. U teoriji, podaci bi mogli da se šalju i putem računara koji rade na ugalj i šalju međusobno dimne signale (mada bi vreme odziva i uticaj na životnu sredinu bili vrlo problematični). Što se tiče samog medijuma za prenos podataka, ništa nije fiksirano ili zapisano u kamenu – što se nas tiče, bitno je da postoji hardverska povezanost. Ali, pored hardverske povezanosti, neophodna je i njihova softverska povezanost, odnosno za komunikaciju dva računara u mreži neophodno je da postoje programi u njima koji zapravo razmenjuju podatke.

Svaki uređaj u mreži se naziva čvor. To su uglavnom računari, ali mogu biti i ruteri, štampači i drugi komunikacioni i umreženi periferni uređaji. Svaki čvor u mreži koji je potpuno funkcionalni računar se naziva *host*. Pošto je neophodno jednoznačno odrediti svaki čvor u mreži, njemu se dodeljuje adresa. Adrese se dodeljuju na različite načine u različitim tipovima mreža, ali to prevazilazi okvire ove knjige.

Sve moderne računarske mreže se zasnivaju na komutaciji paketa (engl. *packet-switched*), što u praksi znači da se podaci pre slanja kroz mrežu dele u manje jedinice koje se nazivaju paketi, pri čemu se svaki paket kroz mrežu šalje nezavisno. Svaki paket u sebi nosi informaciju o tome ko ga je poslao, i kuda treba da ide. Paketi su označeni brojevima kako bi se na mestu odredišta mogla rekonstruisati poruka (pošto putuju različitim rutama kroz mrežu, paketi mogu stići u izmešanom redosledu).

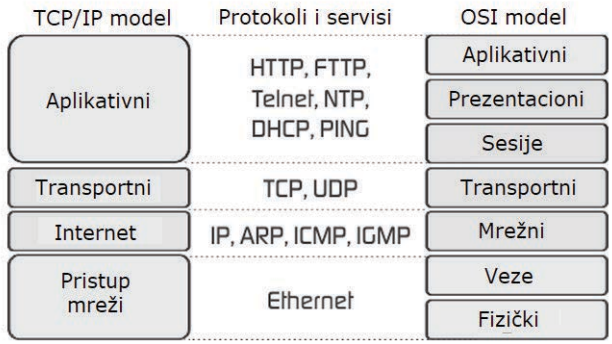
Za uspešnu komunikaciju između računara potreban je još jedan deo slagalice – precizan skup pravila koja definišu kako računari komuniciraju međusobno – format

adrese, kako se podaci dele u pakete, koji se sigurnosni mehanizmi koriste i slično. Ovaj skup pravila se naziva protokol. Postoji veliki broj protokola koji definišu različite aspekte mrežne komunikacije. Na primer, jedan od nama najznačajnijih protokola HTTP (engl. *Hypertext Transfer Protocol*) definiše način komunikacije između veb čitača i servera. Sa druge strane, IEEE 802.3 standard na primer definiše kako se bitovi predstavljaju kao električni signali na određenom tipu žice. Drugi javni protokoli i standardi omogućavaju da softverske i hardverske komponente različitih proizvođača mogu da međusobno komuniciraju. U opštem slučaju, veb čitač ne mora da ima informaciju da li se određeni server izvršava na računaru pod Linux ili Windows operativnim sistemom – server i veb čitač koriste isti HTTP protokol nezavisno od platforme koja se koristi.

1.2. Mrežni slojevi

Prenos podataka kroz mrežu je veoma kompleksna operacija, koja zavisi kako od fizičkih karakteristika samog mrežnog medijuma, tako i od tipa podataka koji se prenose. Softver koji šalje podatke kroz mrežu mora da zna kako da pretvori digitalne podatke u analogne signale, izbegne kolizije između paketa, detektuje i ispravi greške, rutira pojedinačne pakete od jednog hosta do drugog, ispravi pogrešan redosled pristiglih paketa itd. Kako bi se kompletan proces pojednostavio i držao pod kontrolom, komunikacija se posmatra kroz slojeve. Različiti aspekti mreže su podeljeni po različitim nivoima modela. Svaki nivo predstavlja drugačiji nivo apstrakcije sa striktno određenom funkcijom. Na primer, jedan sloj je zadužen za rutiranje paketa, dok je sloj iznad njega zadužen za otkrivanje grešaka i zahtevanje ponovnog slanja oštećenih ili izgubljenih paketa. Svaki sloj direktno komunicira samo sa slojem direktno iznad, i slojem direktno ispod njega. Na ovaj način se većina operacija sakriva kako od programera, tako i od krajnjeg korisnika.

Postoji nekoliko modela mrežne komunikacije. Referentni model je OSI model (engl. *Open Systems Interconnection, OSI*), koji mrežnu komunikaciju deli na 7 slojeva. OSI model je možda i previše detaljan, pa se često ne koristi direktno zbog složenosti. U praksi se koristi nešto jednostavniji model koji integriše određene slojeve, i koji je poznat pod nazivom TCP/IP model koji se zasniva na dva osnovna protokola, TCP (engl. *Transmission Control Protocol*) i IP (engl. *Internet Protocol*). Uporedni prikaz TCP/IP i OSI modela, zajedno sa definisanim slojevima i protokolima koji važe na odgovarajućim slojevima prikazan je na slici 1.1.



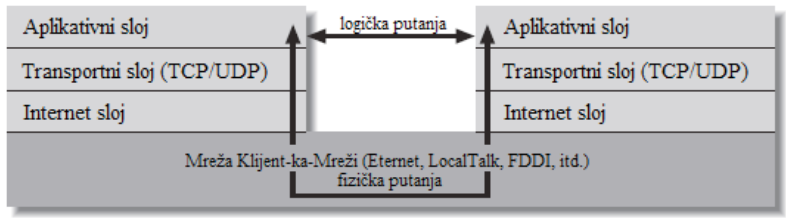
Slika 1.1. Uporedni prikaz TCP/IP i OSI modela

Neki od najbitnijih protokola TCP/IP modela su:

- TCP – uspostavljanje konekcije, kontrola toka podataka, kontrola redosleda paketa, zahtevanje ponovnog slanja izgubljenih paketa itd.
- IP – rutiranje paketa kroz mrežu.
- ARP (engl. *Address Resolution Protocol*) – konverzija Internet (IP) adresa u adrese koji sloj mrežnog interfejsa može da razume.
- SMTP (engl. *Simple Mail Transfer Protocol*) – definiše proceduru razmene mejlova između dva sistema povezana na Internet.
- HTTP – prenos elemenata poput teksta ili slika preko Interneta.

Kao primer, možemo da posmatramo prosto učitavanje stranice u veb čitač. Niz operacija koje se izvršavaju kad veb server šalje stranicu klijentu je sledeći:

- Veb server šalje HTML fajl klijentu koristeći HTTP protokol.
- Sloj aplikacije gde se izvršava HTTP program zahteva od TCP sloja da uspostavi komunikaciju i pošalje fajl.
- TCP stek deli fajlove u pakete, numeriše ih i pojedinačno ih prosleđuje IP sloju na slanje.
- Iako svaki od paketa ima iste IP adrese izvora i odredišta, paketi mogu biti poslati različitim rutama.
- TCP program na klijentskom računaru čeka da svi paketi stignu, šalje potvrdu za pakete koji su primljeni, zahteva ponovno slanje paketa koji nisu stigli (na osnovu brojeva paketa koji fale). Nakon toga sklapa sve pakete u fajl i prosleđuje fajl veb čitaču na strani klijenta (koji se izvršava na sloju aplikacije).



Slika 1.2. Horizontalna i vertikalna komunikacija kroz TCP/IP slojeve

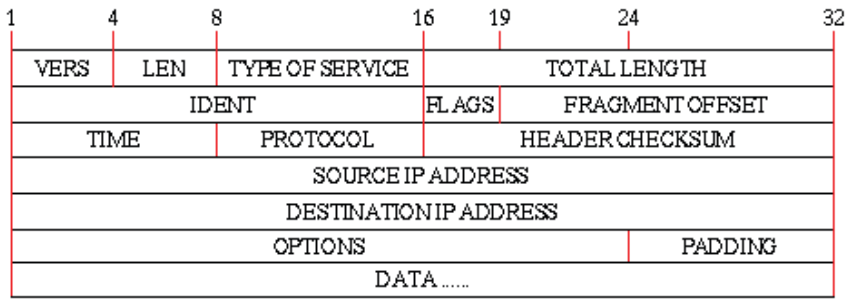
Na slici 1.2. prikazana su dva osnovna vida komunikacije kroz TCP/IP model. Vertikalna komunikacija se vrši na dole ili na gore kroz stek protokola svaki put kada se nešto šalje preko mreže (na dole), ili prima (na gore). Viši nivoi modela su implementirani kao logičke funkcije u softveru – ne postoji stvarna fizička konekcija. Prilikom slanja podataka, viši nivoi pakuju podatke i šalju ih nižim nivoima koji će ih poslati preko mreže. Zapravo, samo slanje podataka kroz mrežu se dešava na najnižem sloju. Na suprotnoj strani koja prima podatke, proces je obrnut. Podaci putuju na gore ka višim slojevima na odredišnom računaru.

Posmatrajmo sada detaljnije kako ovi slojevi komuniciraju koristeći protokole. Prvo, podsetimo se da je svaki sloj, osim sloja na dnu (fizičkog sloja), zapravo program ili algoritam koji se izvršava na računaru. Ne postoji mogućnost da veb čitač i veb server direktno komuniciraju međusobno – oni su samo programi koji se izvršavaju na fizički razdvojenim računarima. Veb čitač i veb server su procesi koji se izvršavaju na nivou aplikacije u TCP/IP modelu, i oni uspostavljaju logičku komunikaciju međusobno upotrebom softvera, procedura i protokola na nižim slojevima. Pošto su računari fizički povezani samo na fizičkom sloju, kako bi protokol četvrtog sloja (aplikativni sloj) uopšte funkcionisao, podaci se prilikom slanja moraju spustiti kroz sve slojeve sve do fizičkog sloja. Podaci se zatim prenose preko fizičke konekcije do prvog sloja na računaru koji prima podatke, i zatim prenose skroz gore do četvrtog sloja – sloja aplikacije. Na taj način su ova dva računara logički povezana na četvrtom sloju, iako na tom sloju ne postoji stvarna fizička konekcija. Može se zaključiti da, osim u slučaju stvarne fizičke konekcije na prvom sloju, sva horizontalna komunikacija takođe zahteva i vertikalnu komunikaciju – spuštanje kroz stek protokola na jednom računaru, a zatim penjanje kroz stek protokola na drugom računaru.

Na sloju aplikacije, procesu sa jedne strane izgleda kao da komunicira direktno sa procesom na sloju aplikacije sa druge strane – mreža kreira logičku putanju između dva sloja aplikacije. Na našu sreću, Java programeri se nalaze prilično visoko u mrežnom lancu ishrane. Većina stvari se dešava ispod naših radara. Preciznije govoreći, fizički sloj nam uopšte nije od interesa.

1.3. Mrežni i transportni sloj - IP, TCP i UDP

IP, odnosno Internet protokol, je prvi protokol od interesa za Java programere. Nalazi se na drugom sloju TCP/IP modela, na mrežnom sloju. IP je praktično jedini protokol koji se koristi na ovom sloju. Podaci se šalju na mrežnom sloju u obliku paketa (datagrama). Kao ilustraciju, prikazaćemo IPv4 datagram. Svaki IP datagram sadrži zaglavlje dužine između 20 i 60 bajtova, i tovar koji može biti do 65 515 bajtova podataka. Struktura datagrama je prikazana na slici 1.3.



Slika 1.3. Struktura IP datagrama (slika preuzeta sa https://www.w3.org/)

Jedna od najbitnijih stvari koja se može uočiti sa strukture IP datagrama jeste IP adresa. Svaki datagram ima IP adrese pošiljaoca i primaoca. Kako bi dva računara mogla da komuniciraju, pored korišćenja protokola moraju se koristiti i adrese koje jednoznačno određuju računare u mreži. Dakle, svaki računar u mreži ima jedinstvenu IP adresu po kojoj se razlikuje od svih ostalih računara. Sa strukture IP datagrama možemo uočiti da je IP adresa 32-bitni podatak. To je zapravo 32-bitni ceo broj, koji se u praksi piše u obliku četiri broja odvojena tačkama, na primer 192.168.1.1. Svaki od ova četiri pojedinačna broja predstavlja 8-bitni ceo broj iz intervala 0 – 255. Ovo su tradicionalne adrese koje se koriste u IPv4, koji je starija verzija IP protokola (i dalje u upotrebi). Sa IPv4 standardom moguće je jednoznačno adresirati nešto više od 4 milijarde različitih uređaja. Zbog nedostatka adresnog prostora, najnovija verzija IP protokola IPv6 koristi 128-bitne brojeve za adrese. Java programeri ne moraju detaljno da poznaju strukturu i funkcionisanje IP sloja, ali moraju da poseduju osnovno znanje o adresama.

Sirovi datagrami imaju određene mane. Kao prvo, ne postoji garancija da će zaista biti isporučeni. Čak i ako se isporuče, mogu biti oštećeni ili da ne stignu u odgovarajućem redosledu. Na primer, ukoliko se datagram A pošalje pre datagrama B, ne postoji garancija da će A zaista stići pre B na odredište, pošto datagrami ne moraju ići istom putanjom kroz mrežu.

Transportni sloj je odgovoran da se svi paketi stignu u redosledu kako su i poslati, uz garancije da nema izgubljenih ili oštećenih podataka. Ukoliko je paket izgubljen, transportni sloj može da zahteva da pošiljalac ponovo pošalje sporni paket. Na transportnom sloju postoje dva primarna protokola, koji koriste IP protokol za uspostavljanje logičke konekcije između dva računara. Prvi je TCP protokol, koji obezbeđuje retransmisiju izgubljenih ili oštećenih podataka i garantuje isporuku paketa u redosledu u kome su poslati. U nekim specifičnim slučajevima, umesto TCP protokola može se koristiti i drugi protokol, koji se zove UDP protokol (engl. *User Datagram Protocol*). UDP je efikasniji i brži od TCP protokola, ali ne garantuje da će svi paketi biti isporučeni u ispravnom redosledu, a može doći i do izgubljenih paketa. Zbog toga se za TCP obično kaže da je pouzdan protokol, dok se za UDP kaže da je nepouzdan. Iako na prvi pogled možda izgleda čudno, UDP je ipak neophodan u nekim situacijama.

Ukoliko se šalju neki osetljivi podaci, poput mejla, fajlova, bankovnih izveštaja i slično, logično je koristiti TCP protokol, pošto bi paketi koji eventualno fale učinili da sadržaj bude oštećen i nepristupačan. Međutim, mehanizam koji obezbeđuje pouzdanost podrazumeva u nekim slučajevima i veliki broj retransmisija paketa koji fale. To kao rezultat često ima kašnjenje, pošto je potrebno određeno vreme od slanja zahteva za retransmisijom dok ponovno poslati paket ne stigne na odredište. Dakle, ukoliko je primarno da transfer bude pouzdan (bez grešaka), a brzina nije toliko bitna, koristi se TCP protokol.

Sa druge strane, ukoliko je primarno da transfer bude brz, odnosno da nema kašnjenja, koristi se UDP protokol. Na primer, prenos glasa funkcioniše po ovom principu (VoIP – engl. *Voice over IP*). Kada razgovaramo sa nekim putem neke VoIP aplikacije (Skype, Viber i slično), nije nam bitno da li su zaista svi paketi preneti, pošto se glas kvalitetno rekonstruiše i ukoliko nisu svi paketi stigli. Međutim, da se u ovom slučaju koristi TCP protokol, svaki izgubljeni paket bi zahtevao retransmisiju, odnosno glas bi kočio i drastično kasnio, a razgovor između dve strane bio praktično nemoguć.

1.4. IP adrese i domenska imena

Kao što smo već naveli u prethodnom poglavlju, svaki računar poseduje svoju IP adresu. Ukoliko se koristi IPv4, adresa je broj dužine četiri bajta. Uobičajen način pisanja je u obliku 192.168.1.1, koji se još na engleskom naziva *dotted quad* format. U slučaju IPv6 adresa koje su dužine 128 bita, one se obično pišu u osam blokova od po četiri heksadecimalne cifre razdvojenih dvotačkom, na primer jedna moguća adresa bi bila: FEDC:BA98:7654:3210:FEDC:BA98:7654:3210. Vodeće nule u svakom bloku se ne moraju pisati. U programskom jeziku Java, IPv6 adrese su podržane od Java verzije 1.4.

Računari veoma lako rade sa brojevima, međutim, sa ljudima je potpuno druga priča. Jako je teško zapamtiti adrese u ovom obliku. Ljudi, sa druge strane, jako lako pamte smislena imena, poput www.google.com ili fir.singidunum.ac.rs. Ova domenska imena su jedinstvena, ali nisu predviđena da zamene IP adrese, već samo da ljudima olakšaju rad sa računarima u mreži. Zbog toga je razvijen sistem domenskih imena poznat pod skraćenicom DNS (engl. *Domain Name System*). DNS prevodi domenska imena koja ljudi lakše pamte, poput www.google.com, u numeričke IP adrese u obliku 172.217.168.206. Dakle, ukoliko se koristi domensko ime radi uspostavljanja veze sa udaljenim računarom, ovo domensko ime se najpre prevodi u IP adresu, i tek nakon toga se šalje zahtev na odgovarajuću IP adresu. Ovaj zadatak obavljaju posebni računari u mreži koji se nazivaju DNS serveri. Svaki računar u mreži može imati više IP adresa i više domenskih imena. Jedna od ovih IP adresa je 127.0.0.1, koja se koristi kada program komunicira sa drugim programom koji se nalazi na istom računaru, i naziva se adresa petlje (engl. *loopback address*), za koju je vezano domensko ime *localhost*.

Kada Java program pristupa mreži, mora da bude u stanju da obrađuje kako numeričke adrese, tako i odgovarajuća domenska imena. Metode koje omogućavaju rad sa adresama i domenskim imenima se nalaze u klasi java.net.InetAddress.

Osim adresa, neophodno je poznavati i broj porta. Adresa bi bila dovoljna ukoliko bi računar radio samo jednu stvar u svakom trenutku. Moderni računari rade više stvari u paraleli, pa je u mrežnom saobraćaju neophodno razdvojiti na primer mejl od FTP zahteva. To se postiže upotrebom portova – svaki računar koji ima IP adresu poseduje nekoliko desetina hiljada logičkih portova (65 535 da budemo precizniji). Svakom portu je dodeljen jedan broj iz intervala 1 – 65 535, i mogu se koristiti za različite servise. Portovi u intervalu između 1 i 1023 su portovi rezervisani za poznate servise, dok se ostali mogu slobodno koristiti.

1.5. Aplikacioni sloj i HTTP

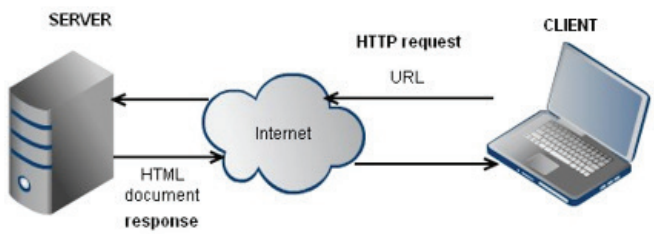
Aplikacioni sloj je četvrti sloj TCP/IP modela, i služi da prikaže podatke korisniku. Sva tri sloja ispod aplikacionog sloja rade zajedno na definisanju kako će se podaci preneti sa jednog računara na drugi. Aplikacioni sloj odlučuje šta dalje raditi sa podacima nakon što su preneti. Na primer, HTTP protokol aplikacionog nivoa obezbeđuje da veb čitač može da prikaže grafičku sliku kao liku, a ne kao dugačak niz brojeva. Postoji čitava abeceda raznoraznih protokola aplikacionog sloja. Pored već pomenutog HTTP koji se koristi za WWW (engl. *World Wide Web*), postoje SMTP, POP i IMAP koji se koriste za mejl, FTP i SFTP za prenos fajlova, NFS za pristup fajlovima i još mnogo drugih. Dodatno, ukoliko je potrebno, i programeri mogu definisati protokole aplikacionog sloja za svoje programe. HTTP je protokol koji nam je od najvećeg interesa, tako da će u nastavku ovog poglavlja biti detaljno objašnjen.

HTTP (engl. *HyperText Transfer Protocol*) je mrežni protokol koji pripada sloju aplikacije i predstavlja glavni i najčešće korišćeni protokol za prenos informacija na vebu. Glavna namena ovog protokola je isporučivanje HTML dokumenata, tj. veb stranica. Standardizaciju i razvoj HTTP protokola nadgledaju W3C (engl. *World Wide Web Consortium*) i IETF (engl. *Internet Engineering Task Force*). W3C je međunarodna organizacija koja se bavi kreiranjem internet standarda (https://www.w3.org/).

1.5.1. Funkcionisanje HTTP protokola

HTTP protokol funkcioniše kao *request-response* protokol u modelu klijent-server. Klijent inicira zahtev (engl. *request*), dok server šalje odgovor (engl. *response*). Klijent je najčešće veb čitač (engl. *web browser*), dok je server veb sajt ili veb server. Klijent serveru šalje HTTP request poruku, na koju server odgovara sa HTTP response porukom koja može da sadrži HTML fajl. Klijent obrađuje response poruku koju je dobio i kao posledicu može da generiše nove HTTP request poruke da bi upotpunio celu komunikaciju.

HTTP je protokol koji pripada aplikacionom sloju i oslanja se na transportni sloj TCP/IP modela. Transportni sloj najčešće koristi TCP, a može da koristi i UDP. Različiti HTTP resursi identifikuju se preko URL i URI identifikatora. Identifikatori i hiperlinkovi u HTML dokumentima formiraju mrežu povezanih dokumenata. Vizuelni prikaz rada HTTP protokola dat je na slici 1.4.



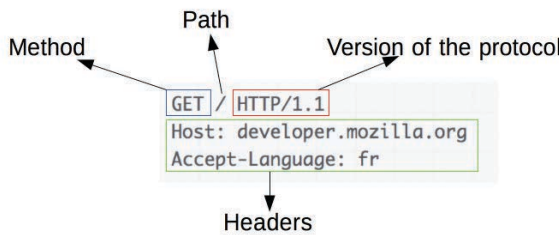
Slika 1.4. Funkcionisanje HTTP protokola (izvor: http://etutorials.org)

HTTP klijent inicira komunikaciju preko TCP konekcije na određeni port. Uglavnom je to port 80 ili port 8080, na kome HTTP server sluša i čeka request poruke. Nakon što je obradio poruku koju je dobio od klijenta, HTTP server vraća status poruke i sadržaj poruke klijentu. HTTP protokol ne vodi računa o sesiji i spada u grupu STATELESS protokola. STATELESS protokoli su protokoli koji ne „pamte stanja". Neki serveri sesiju održavaju pomoću skrivenih varijabli ili preko HTTP kolačića (engl. *cookies*). HTTP kolačići predstavljaju malu količinu podataka koja se čuva na klijentu i razmenjuje se u komunikaciji sa serverom. Za kriptovane HTTP sesije koriste se HTTPS (HTTP Secure) preko porta 443.

Osnovni model HTTP autentifikacije je *basic access authentication*. Kompleksniji i sigurniji model HTTP autentifikacije je *Digest access authentication*. Koristi se najčešće MD5 (Message Digest 5) heš. HTTP podržava okruženje za kontrolu pristupa i autentifikaciju.

1.5.2. HTTP request

Request poruka koju šalje klijent ka serveru uključuje u prvoj liniji poruke, metod koji je potrebno primeniti na resurs, identifikator resursa i verziju protokola koji se koristi, kao što je prikazano na slici 1.5.



Slika 1.5. Format HTTP request poruke (izvor: developer.mozilla.org)

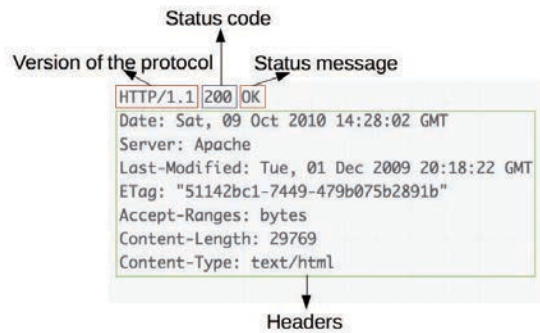
HTTP metod definiše operaciju koju klijent želi da izvrši. Najčešće, klijent želi da dohvati resurs (pomoću GET) ili da pošalje vrednosti neke HTML forme (pomoću POST), ali postoje i dodatne operacije. Potpuna lista metoda sa kratkim opisom svake data je u nastavku:

- GET – čita i vraća sadržaj veb dokumenta (preuzimanje dokumenta).
- HEAD – čita zaglavlje veb dokumenta.
- POST – dodaje podatke veb dokumentu (slanje podataka).
- PUT – skladišti veb dokument.
- DELETE – uklanja veb dokument.
- TRACE – prikazuje dolazni zahtev.
- CONNECT – veza kroz proksi server.
- OPTIONS – parametri zahteva
- PATCH – samo delimična izmena resursa

Metode GET, HEAD, TRACE i OPTIONS ne menjaju stanje resursa i zato se smatraju da su „sigurne“ metode. Postoje i dodatna zaglavlja koja nose dodatne informacije koje su eventualno potrebne serveru.

1.5.3. HTTP response

HTTP response je odgovor koji server šalje klijentu, u formatu prikazanom na slici 1.6. Odgovor se sastoji od verzije protokola koji se koristi, statusnog koda, statusne poruke i HTTP zaglavlja (slično kao i u request poruci). Dodatno, u telu odgovora se može nalaziti dohvaćeni sadržaj (na primer HTML stranica).



Slika 1.6. Format HTTP response poruke (izvor: developer.mozilla.org)

HTTP odgovor koristi kodove, kako bi se olakšala razmena informacija. U tabeli 1.1 prikazani su HTTP statusni kodovi, zajedno sa kratkim opisima.

Tabela 1.1. HTTP response statusni kodovi

Kod	Značenje koda	Primer
1xx	Informacija	100 – server prihvata zahtev
2xx	Uspeh	200 – zahtev je uspeo
3xx	Preusmeravanje	301 – dokument pomeren
4xx	Klijentska greška	403 – zabranjen pristup 404 – nema dokumenta
5xx	Serverska greška	500 – interna logička greška

Format response poruke najbolje se vidi na primeru:

```
HTTP/1.1 200 OK
     Date: Mon, 23 May 2005 22:38:34 GMT
     Content-Type: text/html; charset=UTF-8
     Content-Encoding: UTF-8
```

```
Content-Length: 138
Last-Modified: Wed, 08 Jan 2003 23:11:55 GMT
Server: Apache/1.3.3.7 (Unix) (Red-Hat/Linux)
ETag: “3f80f-1b6-3e1cb03b”
Accept-Ranges: bytes
Connection: close
  <html>
  <head>
  <title>Test Page</title>
  </head>
  <body>
  This is very simple HTML document
  </body>
  </html>
```

U HTTP paketima zahteva i odgovora nalaze se dodatna zaglavlja. U tabeli 1.2 prikazana su ova zaglavlja sa primerima.

Tabela 1.2. HTTP dodatna zaglavlja

Funkcija i smer slanja	Primeri zaglavlja
Info o pregledaču (klijent -> server)	User-Agent,Accept, Accept-Charset, Accept-Encoding, Accept-Language
Info o keširanju (oba smera)	If-Modified-Since,If-None-Match, Date, Last-Modified, Expires, Cache-Control, ETag
Info o pregledača (klijent -> server)	Cookie, Referer, Authorization, Host
Tip sadržaja (server -> klijent)	Content-Encoding,Content-Length, Content-Type, Content-Language, Content-Range, Set-Cookie

1.5.4. Performanse HTTP protokola

Ključni indikator performansi HTTP protokola je PLT (engl. *Page Load Time*), tj. vreme učitavanja dokumenta. PLT odnosi se na vreme koje protekne od trenutka kada korisnik klikne na link, do trenutka otvaranja dokumenta kojem se pristupa preko tog linka. Čak i malo povećanje PLT ima veliki uticaj na smanjivanje zadovoljstva korisnika. PLT zavisi od nekoliko faktora, kao na primer:

- Strukture dokumenta kojem se pristupa.
- Verzije HTTP i TCP protokola.
- Mrežnog protokola i RTT (engl. *Round Trip Time*).

Postoji više načina za smanjenje PLT-a, kako bi se poboljšale performanse HTTP protokola. Načini za smanjenje PLT su:

- Smanjivanje veličine poslatog sadržaja (manje slike, kompresija podataka i slično).
- Prilagođavanje HTTP za bolje korišćenje protoka komunikacionog kanala.
- Izbegavanje ponavljanja istih HTTP zahteva (keširanje i proksi serveri).
- Pomeranje sadržaja da bude bliži korisniku (CDN – engl. *Content Delivery Networks*).

Paralelne TCP veze koriste se kao način za poboljšanje PLT-a. Funkcionisanje paralelne TCP veze:

- Veb pregledač na primer istovremeno radi sa 10 HTTP konteksta (svaki kontekst pravi odvojenu TCP vezu).
- Samo klijent (veb pregledač) zahteva promenu, server ne.
- 10 HTTP veza može bolje da iskoristi dostupan protok.
- Sve rade istovremeno, pa se ne gomilaju kašnjenja zbog uspostavljanja TCP veza.

Trajne (višestruko upotrebljive veze) su takođe način za poboljšanje PLT:

- Kreira se jedna TCP veza za sve resurse na jednom serveru.
- Ova veza se onda deli nezavisnim HTTP zahtevima ka tom serveru.

1.5.5. HTTP keširanje i proxy

Obzirom da se sadržaj nekih poruka ne menja, klijent može zbog optimizacije da koristi staru kopiju poruke i da ne pita stalno server za novi sadržaj. Ovaj proces naziva se keširanje. Keširanje omogućava višestruku upotrebu istog sadržaja. Poruke (HTML stranice) koje se retko menjaju mogu da se serviraju klijentu i preko *proxy* servera.

Korisnici putem klijentskog softvera (veb pregledača) često posećuju jednu veb stranicu više puta. Ovo omogućava čuvanje i ponovnu upotrebu lokalne kopije te stranice (keširanje). Kao jedno od pitanja koje se postavlja jeste kada je ispravno koristiti lokalnu kopiju? Lokalna kopija poruke se smatra ispravnom ako nije istekla. Prilikom dostavljanja sadržaja, server klijentu dostavlja i vreme isticanja sadržaja u vidu dodatnog polja, koje se obično naziva *expires*. Ako je kopija ispravna (nije istekla), koristi se u toj formi, u suprotnom slučaju, sadržaj se ponovo preuzima sa servera.

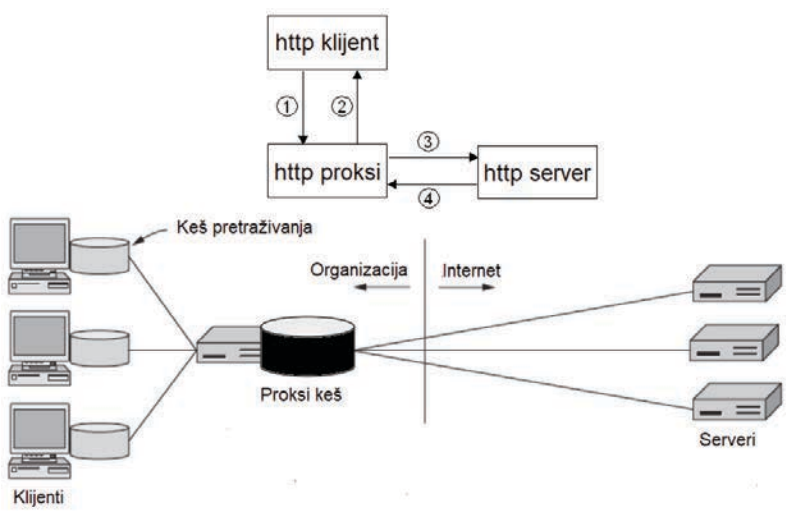
Ako lokalna kopija nije ispravna, mora da se izvrši provera ispravnosti poruke na serveru. Klijent u tom slučaju serveru šalje uslovni GET zahtev (GET request), u kom su navedene informacije o lokalnoj kopiji. Na osnovu informacija o lokalnoj kopiji server proverava da li je lokalna kopija zastarela u skladu sa tim šalje ili ne šalje

ažurirani/novi sadržaj. Takođe je moguća i provera lokalne kopije zasnovana na sadržaju upotrebom Etag zaglavlja (neki vid heš koda). Vizuelni prikaz procesa HTTP keširanja dat je na slici 1.7.



Slika 1.7. HTTP keširanje

HTTP proksiji (eng. *proxy*) se kao posrednici između grupe klijenta i veb servera takođe mogu koristiti za keširanje. Različiti klijenti mogu da koriste iste sadržaj, tako da ima smisla da se sadržaj kešira i za grupu klijenata, a ne samo za individualne klijente. U scenarijima gde se koriste proksi serveri, klijent prvo kontaktira proksi, koji zatim u ime klijenta kontaktira veb server. Bez korišćenja proksi servera, klijent direktno kontaktira veb server. Takođe, proksiji imaju i dodatne mehanizme, kao što su zabrana i filtriranje sadržaja (content filter), itd., gde se klijentima određuje sadržaj kojem mogu da pristupe. Ove funkcionalnosti su pre svega korisne za organizacije, provajdere internet usluga (engl. *Internet Service Provider – ISP*), itd. Princip rada HTTP proksija prikazan je na slici 1.8.



Slika 1.8. HTTP proksi

Pitanja

1. Šta je računarska mreža?
2. Kako se šalju podaci kroz računarsku mrežu?
3. Šta je protokol? Navesti neke protokole.
4. Koji je referentni model mrežne komunikacije i koji slojeve prepoznaje?
5. Koje slojeve prepoznaje TCP/IP model mrežne komunikacije?
6. Navesti i opisati neke od najvažnijih TCP/IP protokola.
7. Na kom sloju se dešava slanje paketa?
8. Da li su viši slojevi mrežnih modela fizički povezani?
9. Čemu služi IP adresa? Koje dve verzije IP protokola postoje?
10. Koji sloj je zadužen da se postara da svi paketi stignu u ispravnom redosledu?
11. Koja su dva primarna protokola na transportnom sloju koji koriste IP protokol za uspostavljanje logičke konekcije između dva računara?
12. Kada se koristi TCP protokol a kada UDP protokol?
13. Čemu služi sistem domenskih imena (DNS)?
14. Da li svaki računar ima jedinstveno domensko ime i IP adresu?
15. Zašto je pored IP adrese potrebno poznavati i broj porta?
16. Šta radi aplikacioni sloj modela računarskih mreža?
17. Navesti neke protokole aplikacionog sloja i kada se oni koriste.
18. Opisati rad HTTP protokola.
19. Navesti neke operacije HTTP metode.
20. Navesti neke HTTP statusne kodove.
21. Koji je ključni indikator performansi HTTP protokola i od čega on zavisi?
22. Šta je keširanje i šta omogućava?
23. Opisati princip rada HTTP proksija.

II
Niti

U dalekoj prošlosti, kad računari nisu imali ni operativni sistem, izvršavali su jedan program od početka do kraja. Taj program je imao direktan pristup svim resursima mašine. Ako se zanemari činjenica da je bilo jako teško napisati program koji se izvršava praktično na golom metalu, izvršavanje samo jednog programa u bilo kom datom trenutku je bilo izuzetno neefikasno iz ugla upotrebe skupih i ograničenih resursa.

Operativni sistemi su omogućili izvršavanje više od jednog programa u jednom trenutku, tako što se svaki pojedinačni program izvršavao u okviru procesa. Programi su se izvršavali izolovano i nezavisno, a operativni sistem im je dodeljivao resurse poput memorije, pristupa fajlovima i slično. Ukoliko je potrebno, procesi su mogli da komuniciraju međusobno upotrebom jednog od komunikacionih mehanizama poput soketa, deljene memorije, semafora i slično.

Postoji nekoliko glavnih razloga koji su doveli do razvoja operativnih sistema koji su omogućili simultano izvršavanje više programa. Prvi razlog je iskorišćenost resursa. Programi su često morali da čekaju na vremenski duge ulazno-izlazne operacije, a dok su čekali nisu mogli da rade ništa korisno, pri čemu su držali resurse zauzetim. Mnogo efikasnije bi bilo da se za vreme tog čekanja dozvoli drugom programu da se izvršava i da koristi resurse. Drugi razlog je fer izvršavanje programa (engl. *fairness*), koji se odnosi na činjenicu da više različitih korisnika i programa mogu imati podjednake i ravnopravne zahteve za sistemskim resursima. Poštenije je dozvoliti programima da dele računar i njegove resurse primenom deljenja vremena računara, nego da se pusti jedan program da se izvrši od početka do kraja, pre pokretanja sledećeg programa. I, na kraju, kao treći razlog se izdvaja prosto činjenica da je ponekad lakše i poželjnije napisati nekoliko manjih programa od kojih je svaki zadužen za jedan zadatak i koji međusobno koordinišu, neo da se napravi jedan veliki program koji izvršava sve potrebne zadatke.

Sekvencijalno programiranje je intuitivno i prirodno, pošto modeluje ljudsko ponašanje – mi radimo po jednu stvar u trenutku, najčešće u sekvenci. Ustajemo iz kreveta, oblačimo garderobu, idemo u kupatilo, idemo u kuhinju, sipamo vodu u džezvu za kafu i stavljamo je na šporet. Poslednji korak, stavljanje džezve na šporet i čekanje da voda proključa, daje nam izbor – možemo prosto čekati i gledati u džezvu dok voda ne proključa, ili da paralelno uradimo još neke stvari dok čekamo da se voda zagreje, poput stavljanja tosta u toster, dohvatanja i prelistavanja novina, dok

smo sve vreme svesni da će naša pažnja uskoro biti ponovo usmerena na džezvu kada voda proključa. Pronalaženje pravog balansa između sekvencijalnog izvršavanja i asinhronog paralelnog izvršavanja je često karakteristika kako efikasnih ljudi, tako i efikasnih programa.

Kao i ljudi, računari pored sekvencijalnog izvršavanja programa, mogu obavljati i više različitih zadataka u isto vreme. Ova sposobnost računara se zove po engleskoj reči *multitasking*. U modernim računarima sa više procesora nije teško zamisliti da se različiti zadaci izvršavaju paralelno na fizički razdvojenim procesorima. Čak i u slučaju da računar ima samo jedan procesor, moguće je raditi multitasking, tako što se paralelnost postiže naizmeničnim izvršavanjem nekoliko zadataka svakog po malo.

Na isti način, zadatak jednog programa u programskom jeziku Java se može podeliti na više manjih zadataka koji se paralelno izvršavaju. Svaki od ovih pojedinačnih zadataka koji se paralelno izvršava sa drugim zadacima istog programa se naziva nit izvršavanja, programska nit ili prosto kraće rečeno nit (engl. *thread*). Ovaj mehanizam omogućava postojanje više paralelnih niti kontrole toka u jednom programu. Ove niti dele programske resurse poput memorije ili fajlova, ali svaka nit na primer ima svoj zasebni programski brojač i lokalne promenljive. Niti na prirodan način iskorišćavaju hardverski paralelizam u vidu multiprocesorskih sistema, pošto se više niti istog programa može izvršavati paralelno na različitim procesorima. Ukoliko ne postoji eksplicitna koordinacija između niti, one se izvršavaju simultano i asinhrono u odnosu na druge niti programa. Pošto niti dele memorijski adresni prostor procesa odnosno programa kome pripadaju, sve niti imaju pristup zajedničkim resursima poput promenljivih i kreiraju objekte na istom hipu, koji olakšava deljenje podataka i komunikaciju između različitih niti. Međutim, bez eksplicitne sinhronizacije koja bi kontrolisala pristup deljenim podacima, jedna nit bi mogla da promeni promenljivu koju druga nit upravo koristi, sa nepredviđenim rezultatima. Zbog toga je višenitno (paralelno) programiranje znatno teže od običnog sekvencijalnog programiranja, jer kada se više niti paralelno izvršava, njihov uticaj na stanje programa se teže prati i dolazi do potpuno novih vrsta grešaka.

Potrebno je naglasiti da se svaki Java program izvršava u bar jednoj niti – *main* nit. Kada se program pokrene, Java virtuelna mašina kreira main nit i poziva main() metodu programa u okviru te niti. Java virtuelna mašina takođe kreira i druge niti, koje nisu vidljive korisniku – na primer niti povezane sa sakupljanjem đubreta (engl. *Garbage Collection*), zatim finalizaciju objekata, kao i za druge zadatke koji se izvršavaju u pozadini.

2.1. Prednosti višenitnog programiranja

Kada se koriste na ispravan način, niti mogu smanjiti ukupne troškove razvoja i održavanja softvera, i poboljšati performanse kompleksnih aplikacija. Pomoću niti se lakše modeluje kako ljudi izvršavaju zadatke i vrše interakcije, pretvaranjem

asinhronih tokova u sekvencijalne. Niti su veoma korisne u aplikacijama sa grafičkim korisničkim interfejsom (engl. *Graphical User Interface – GUI*), pošto se u opštem slučaju poboljšava brzina odziva (engl. *responsiveness*) korisničkog interfejsa. Na primer, prilikom razvoja mobilnih aplikacija za Android platformu, kompletan grafički korisnički interfejs se nalazi u posebnoj korisničkoj niti (engl. *User Interface thread*). Sa druge strane, i serverske aplikacije se oslanjaju na niti kako bi se poboljšalo iskorišćenje resursa i propusna moć programa. Takođe, većina netrivijalnih Java aplikacija se u nekom stepenu oslanja na niti za organizaciju izvršavanja svojih zadataka.

Osim očigledne prednosti kada se višenitni program izvršava na multiprocesorskom sistemu, čak i u slučaju izvršavanja višenitnog programa na jednom procesoru se mogu dobiti efikasniji rezultati. Na primer, ukoliko je program napisan samo u jednoj niti, procesor je blokiran dok program čeka završetak sinhrone ulazno-izlazne operacije. U slučaju više niti, dok prva nit čeka rezultate ulazno-izlazne operacije, procesor se može dati drugoj niti, i na taj način omogućiti programu da radi uprkos blokirajućoj ulazno-izlaznoj operaciji (poput čitanja novina dok čekamo da voda za kafu provri, umesto da čekamo da voda provri pre nego što uopšte počnemo čitanje novina).

Druga prednost je činjenica da je mnogo lakše pisati program koji izvršava jedan tip zadatka nego više različitih tipova zadataka. Opet, ukoliko se napravi paralela sa ljudskim ponašanjem, lakše radimo jedan tip zadatka (na primer ispravka većeg broja bagova) nego kada moramo da radimo više različitih tipova zadataka (ispravka bagova, intervjui sa kandidatima za posao, priprema sastanka, pisanje dokumentacije, planiranje sprinta, implementacija nove funkcionalnosti i priprema prezentacije za klijenta). Kada radimo samo sa jednim tipom zadatka, prosto rečeno počinjemo sa vrha gomile i radimo dok se ta gomila ne isprazni – ne ulažemo vreme u razmišljanje šta raditi sledeće, i ne gubimo vreme na mentalno prebacivanje na novi tip zadatka. U opštem slučaju, program koji izvršava jedan tip zadatka sekvencijalno se lakše piše, manje je podložan greškama i lakše testira od programa koji izvršava više različitih tipova zadataka istovremeno. Ukoliko se svakom tipu zadatka programa dodeli nit, stvara se iluzija sekvencijalnosti, a kompleksni i asinhroni tok programa se može dekomponovati u određen broj jednostavnijih, sinhronih tokova koji se izvršavaju u posebnim nitima, i vrše interakciju samo na određenim sinhronizacionim tačkama. Ova logika je iskorišćena u različitim programskim okvirima, poput servleta ili RMI (engl. *Remote Method Invocation*). Sam programski okvir (engl. *framework*) je zadužen za rukovanje menadžmentom zahteva, kreiranje niti, balansiranje opterećenja i funkciju dispečera – preusmerava delove obrade zahteva na odgovarajuće komponente sistema. Programer koji piše servlet ne mora da brine koliko će drugih zahteva biti obrađeno u isto vreme, ili da li su ulazni i izlazni tokovi soketa blokirani ili ne. Na ovaj način se značajno pojednostavljuje programiranje komponenti. RMI, sa druge strane, se odnosi na poziv udaljene metode na objektu koji se izvršava na drugoj JVM (Java Virtuelna Mašina). Kada se udaljena metoda pozove, argumenti poziva

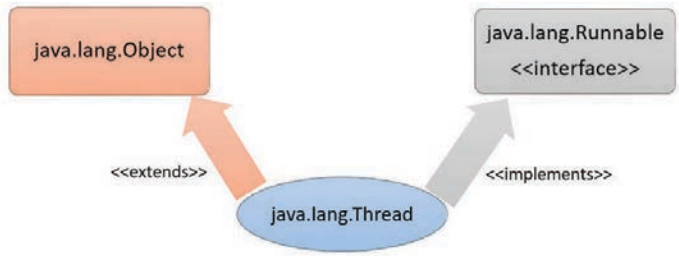
metode se pakuju u strim bajtova i šalju preko mreže do udaljene JVM, gde se vrši njihovo raspakivanje i prosleđivanje udaljenoj metodi. Udaljeni objekat se poziva u niti koju kontroliše RMI. Slično kao kod servleta, RMI objekti moraju biti sposobni da odgovore na više paralelnih poziva udaljene metode, i da obezbede ispravan rad sa višestrukim nitima.

Još jedna prednost upotrebe niti je olakšano rukovanje asinhronim događajima. Serverska aplikacija koja preko soketa prihvata zahteve za konekcijom većeg broja udaljenih klijenata se mnogo lakše razvija ukoliko se svaka konekcija obrađuje u zasebnoj niti. Ukoliko bi serverska komponenta bila napisana kao jedna nit, prilikom pokušaja čitanja iz soketa u kome nema dostupnih podataka, read operacija bi bila blokirana sve dok neki podaci ne budu dostupni. U slučaju jedne niti, ovo znači da ne samo da bi obrada trenutnog zahteva bila blokirana, već bi obrada i svih ostalih zahteva bila blokirana. Kada je svaki zahtev u zasebnoj niti, blokiranje jednog zahteva nema uticaja na obradu drugih zahteva.

2.2. Osnovni pojmovi

Pod zadatkom se posmatra jedna programska celina koja se izvršava nezavisno od drugih delova programa. Nit predstavlja tok izvršavanja jednog zadatka, od njegovog početka do kraja, u okviru programa. Nit zapravo predstavlja programski mehanizam pomoću kojeg se obezbeđuje da se zadatak nezavisno izvrši u okviru programa. Moguće je u jednom programu pokrenuti proizvoljan broj niti koje se paralelno izvršavaju. Svakako, kao što je već ranije navedeno, program se izvršava u bar jednoj niti – main nit, koju pokreće JVM pozivom main metode programa. Iz ove početne niti se mogu dalje pokretati nove niti izvršavanja.

Jedna nit izvršavanja u Javi se predstavlja objektom klase Thread (ili bilo koje klase koja nasleđuje ovu klasu). Klasa Thread se nalazi u paketu java.lang, odnosno predstavlja jednu od osnovnih funkcionalnosti Java programa, pošto se u paketu java.lang nalaze definicije klasa koje su neophodne za osnovni rad programa, a sam paket se podrazumevano implicitno uvozi u sve korisnički definisane Java klase. Objekat klase Thread ima jednu svrhu – da izvrši zadatak koji se nalazi u jednoj njegovoj specijalnoj metodi. Ta metoda je definisana u interfejsu Runnable, koji klasa Thread implementira. Osnovni dijagram nasleđivanja je dat na slici 2.1 Kada se izvršavanje ove metode završi, regularno ili nasilno usled nekog izbačenog izuzetka, završava se i izvršavanje niti. Kada se jednom završi, nit se ne može ponovo aktivirati, a ni njen objekat se ne može iskoristiti za pokretanje nove niti.



Slika 2.1. Dijagram nasleđivanja klase Thread

Postoji nekoliko načina za kreiranje niti u Java program. Osim glavne niti koju pokreće JVM, nove niti se mogu kreirati putem konstruktora klase Thread, ili instanciranjem bilo koje klase koja je izvedena iz klase Thread. Drugim rečima, bilo koja Java nit može kreirati novu nit ili direktnim instanciranjem objekta klase Thread, ili objekta klase koja proširuje klasu Thread. Ukoliko se koristi direktno klasa Thread, može se u konstruktoru proslediti zadatak koji se izvršava u niti, a koji mora pripadati klasi koja implementira interfejs Runnable.

2.3. Interfejs Runnable

Za definisanje jednog zadatka koji treba da se paralelno izvršava mora se definisati klasa tog zadatka. Svaki zadatak koji je potrebno da se paralelno izvršava u Javi jeste, kao i sve ostalo, objekat, ali njegova klasa mora implementirati interfejs Runnable, kako bi se zadatak mogao paralelno izvršavati u niti. Sam interfejs Runnable je veoma jednostavan, pošto sadrži samo jedan metod – run ().

```
public interface Runnable {
       void run();
}
```

Runnable je funkcionalni interfejs, odnosno interfejs sa samo jednom metodom, tako da je veoma pogodan za upotrebu lambda izraza (uvedeni od Jave verzija 8), o čemu će biti reči kasnije. Pošto svaka klasa zadatka mora implementirati interfejs Runnable kako bi mogla da se izvršava u posebnoj niti, ta klasa mora dati i definiciju run () metode. Na taj način se ukazuje da se zadatak izvršava u odvojenoj niti, i definiše se sama funkcija zadatka. Osnovni šablon za kreiranje klase zadatka koji se izvršava u zasebnoj niti je dat u nastavku.

```
public class Zadatak implements Runnable {

	// polja i druge metode

	// konstruktor za inicijalizaciju objekta
	public Zadatak (...) {...}

	// obavezna implementacija metode run() nasledjene iz Runnable
	public void run(){

		// kod koji predstavlja implementaciju zadatka
		// koji treba da se izvrsi u novoj niti

	}
}
```

Nakon definisanja klase zadatka, konkretan objekat (zadatak) se instancira na standardan način:

```
Zadatak zadatak = new Zadatak(...);
```

Nakon instanciranja klase zadatka, potrebno je taj zadatak izvršiti u posebnoj niti. Za konstruisanje niti koristi se klasa Thread. U ovoj klasi, osim konstruktora za konstruisanje novih niti, postoje i mnoge druge korisne metode za rad sa nitima, o čemu će biti reči kasnije. Konstrukcija niti na osnovu prethodno definisanog zadatka se može pisati na sledeći način:

```
Thread nit = new Thread (zadatak);
```

Samim konstruisanjem objekta klase Thread ne započinje automatski izvršavanje datog zadatka. Nakon konstrukcije niti, za pokretanje izvršavanja datog zadatka se mora eksplicitno pozvati metoda start () koja se nalazi u klasi Thread. To se postiže na sledeći način:

```
nit.start();
```

Dakle, nit se ne pokreće direktnim pozivom metode run()! Ako bismo pozvali metodu run direktno u programu, zadatak bi se izvršio u istoj niti odakle je pozvan, dakle ne bi se kreirala nova nit izvršavanja. Pozivom metode start() zapravo JVM implicitno poziva metodu run() iz klase zadatka i izvršava se u novoj niti.

2.4. Klasa Thread

Kao što smo videli u prethodnom poglavlju, klasa Thread se koristi za kreiranje i pokretanje niti. Nit se pokreće pozivom metode start(), koja implicitno poziva metodu run() odgovarajućeg zadatka. Pored metode start(), klasa Thread sadrži i brojne druge korisne metode za kontrolu izvršavanja neke niti. Precizno govoreći, sama klasa Thread implementira interfejs Runnable, tako da je moguće zajedno definisati nit i zadatak koji treba da se izvršava u njoj. To je drugi način programiranja niti. U tom slučaju, potrebno je definisati klasu koja je izvedena iz klase Thread, i istovremeno implementirati metodu run() u novoj klasi (i samim tim definisati zadatak koji treba da se izvrši u posebnoj niti). Ukoliko se koristi ovaj drugi način programiranja, šablon je dat u nastavku:

```
public class KorisnickaNit extends Thread {

	// polja i druge metode

	// konstruktor za inicijalizaciju objekta
	public KorisnickaNit (...) {...}

	// obavezna implementacija metode run() nasledjene iz Runnable
	public void run(){

		// kod koji predstavlja implementaciju zadatka
		// koji treba da se izvrsi u novoj niti

	}
}
```

Pošto klasa Thread implementira interfejs Runnable, i dalje smo u obavezi da implementiramo metodu run(), u kojoj se definiše funkcionalnost zadatka. Nasleđena verzija metode run() iz klase Thread je prazna, dakle ne radi ništa i odmah se vraća na mesto poziva. Ukoliko je objekat klase Thread kreiran sa već definisanim objektom zadatka koji implementira Runnable, kao u prethodnom poglavlju, poziva se metoda run() tog Runnable objekta.

Nit definisana na ovaj način, sa zadatkom koji je direktno implementiran u klasi izvedenoj iz klase Thread, se koristi na sledeći način:

```
KorisnickaNit nit = new KorisnickaNit(...);
// startovanje niti
nit.start();
```

Kao što smo videli, postoje dva osnovna načina programiranja niti. Koji je bolji? Nema nekih striktnih pravila, koriste se oba načina, uz nekoliko komentara. Drugi način programiranja niti na prvi pogled izgleda kompaktniji i jednostavniji, ali u ovom pristupu se meša pojam zadatka sa mehanizmom koji izvršava taj zadatak. Zbog toga, obično je bolje razdvojiti definiciju zadatka od mehanizma, odnosno niti u kojoj se izvršava. Treba takođe biti svestan da ukoliko se koristi drugi način, klasa koja nasleđuje klasu Thread više ne može da proširi nijednu drugu klasu, zbog ograničenja u modelu nasleđivanja programskog jezika Java. U Javi je moguće direktno naslediti tačno jednu klasu (ili eksplicitno, upotrebom ključne reči extends i definisanjem klase koja se nasleđuje, ili podrazumevano nasleđivanje klase Object, ukoliko nije ništa navedeno eksplicitno). Takođe, u Javi je moguće naslediti proizvoljan broj interfejsa, tako da ako se koristi prvi način, uz implementaciju interfejsa Runnable može se implementirati proizvoljan broj dodatnih interfejsa, a slot za nasleđivanje ostaje slobodan. Iz tih razloga, možda se može reći da je često u praksi praktičnije koristiti prvi način programiranja niti sa implementacijom interfejsa Runnable.

Kao što je već ranije napomenuto, u klasi Thread se nalaze i druge korisne metode za kontrolu izvršavanja niti. Nepotpun spisak metoda klase Thread je:

- boolean isAlive() – provera statusa izvršavanja neke niti. Ukoliko postoji objekat t koji je tipa Thread, poziv t.isAlive() kao rezultat vraća true ili false vrednost prema toma da li je nit t živa. Za nit se kaže da je živa od trenutka pokretanja, do trenutka njenog završetka. Nakon toga se nit smatra mrtvom.
- static void sleep (long miliseconds) – nit u kojoj se metoda izvršava prelazi u stanje spavanja za navedeni broj milisekundi. Nit koja spava je živa, ali je blokirana. Za to vreme računar može da izvršava druge niti. Treba imati na umu da ova metoda može izbaciti izuzetak tipa InterruptedException. Ovaj izuzetak je proveravani izuzetak, pa je obavezno rukovanje. Ova metoda se zbog toga obićno piše unutar try – catch bloka gde se hvata pomenuti izuzetak.
- void interrupt () – jedna nit može poslati signal prekida drugoj niti i da uradi nešto sasvim drugo. Ukoliko je t objekat tipa Thread, jedna nit šalje signal prekida drugoj niti t pozivom t.interrupt(). Ovaj signal na primer može probuditi nit koja je u stanju spavanja ili je blokirana.
- void join () – ponekad je neophodno da jedna nit sačeka da se druga nit završi, i da tek onda nastavi izvršavanje. Ukoliko je t objekat tipa Thread, i u nekoj drugoj niti napišemo t.join(), druga nit prelazi u stanje spavanja dok se nit t kompletno ne završi (ukoliko je t već završena, poziv t.join() nema efekta). Poziv metode join() može dovesti do izbacivanja proveravanog izuzetka tipa InterruptedException, čije je rukovanje, kao što je već navedeno, obavezno u programu.
- void setPriority (int prioritet) – JVM svakoj niti dodeljuje prioritet sa kojim se izvršava. Ovaj prioritet se može menjati pozivom metode setPriority, a trenutni prioritet se može dobiti pozivom metode getPriority(). Prioritet je celobrojna

vrednost u intervalu 1 – 10. Klasa Thread sadrži konstante MIN_PRIORITY, NORM_PRIORITY i MAX_PRIORITY sa vrednostima 1, 5 i 10 respektivno. Glavna nit programa podrazumevano dobija Thread.NORM_PRIORITY. Od svih niti koje su spremne za izvršavanje, JVM će odabrati onu koja ima najviši prioritet. Ukoliko su sve niti koje su spremne za izvršavanje istog prioriteta, a ne postoje niti višeg prioriteta, ove niti dobijaju procesor po kružnom principu.

2.5. Kreiranje i pokretanje niti

U prethodnom poglavlju je pokazano da postoje dva načina programiranja niti. Neka se prvo posmatra primer kada nova klasa zadatka nasleđuje klasu Thread. U novoj klasi se nadjačava metoda run (). U sledećem primeru se kreiraju i startuju dve niti, pri čemu svaka štampa dve linije na standardnom izlazu System.out. Neka je prva nit definisana na sledeći način:

```
public class NitSlova extends Thread {

        @Override
        public void run() {
            System.out.println(“A”);
            System.out.println(“B”);
        }
}
```

Druga nit je definisana sa:

```
public class NitBrojevi extends Thread {

        @Override
        public void run() {
            System.out.println(“1”);
            System.out.println(“2”);
        }
}
```

Glavni program je takođe veoma jednostavan, u main metodi se kreira po jedna instanca svake klase, i zatim pokrene odgovarajuća nit:

```
public class DveNiti {

public static void main(String[] args) {
        //kreiranje niti
        NitSlova nit1 = new NitSlova();
        NitBrojevi nit2 = new NitBrojevi();
        //pokretanje niti
        nit1.start();
        nit2.start();
    }
}
```

Nakon kreiranja instanci klasa NitSlova i NitBrojevi, niti se pokreću pozivom odgovarajućih start () metoda. Kao što smo ranije objasnili, poziv metode start() implicitno poziva odgovarajuću run () metodu, i to je jedini ispravan način pokretanja niti.

Do sada smo uglavnom radili samo sa sekvencijalnim programima, gde je bilo potpuno jasno kojim redosledom se naredbe izvršavaju. Kada se radi sa nitima, to nije slučaj. Konkretno, u datom primeru ne možemo znati kojim redom će se naredbe izvršiti. Jedino što sigurno znamo jeste da će broj 1 biti odštampan pre broja 2, i da će slovo A biti odštampano pre slova B. Izlaz programa može biti bilo šta od sledećih šest kombinacija:

- 1 2 A B
- 1 A 2 B
- 1 A B 2
- A 1 2 B
- A 1 B 2
- A B 1 2

Ne samo da rezultati variraju od računara do računara, već čak i izvršavanje istog programa više puta na jednom računaru može dovesti do različitih rezultata. Nikada ne treba nasumično pretpostaviti da će neka nit da uradi neku stvar pre druge niti, ukoliko se ne koristi neki vid sinhronizacije koji će osigurati određeni redosled izvršavanja.

Naravno, moguće je koristiti i drugi opisani način programiranja niti, odnosno implementaciju interfejsa Runnable. Neka se sada posmatra primer sa tri paralelne niti. Potrebno je da postoje tri zadatka, koji se izvršavaju paralelno u svojim nitima:

- prvi zadatak prikazuje tekst Java određen broj puta,
- drugi zadatak prikazuje tekst Spring određen broj puta,
- treći zadatak prikazuje brojeve od 1 do zadatog broja.

Možemo uočiti da prva dva zadatka praktično imaju istu funkcionalnost, stoga su i predstavljeni jednom klasom PrikazTeksta. Pošto klasa implementira interfejs Runnable, mora se definisati metoda run () koja će prikazati zadati tekst određen broj puta.

```
public class PrikazTeksta implements Runnable {

    private String tekst; // tekst za stampu
    private int brojPonavljanja;

    public PrikazTeksta(String tekst, int brojPonavljanja){
        this.tekst = tekst;
        this.brojPonavljanja = brojPonavljanja;
    }

    @Override
    public void run() {
        //implementacija zadatka koji se izvrsava u odvojenoj niti
        for (int i = 0; i < brojPonavljanja; i++){
            System.out.print(tekst + " ");
        }
    }
}
```

Zadatak koji služi za prikaz brojeva je predstavljen uz pomoć klase PrikazBrojeva. Ova klasa takođe implementira interfejs Runnable i definiše metodu run (). Brojevi se prikazuju od broja 1 do zadatog broja.

```
public class PrikazBrojeva implements Runnable {

    private int n;

    public PrikazBrojeva (int n){
        this.n = n;
    }

    @Override
    public void run() {
        for (int i = 1; i <= n; i++){
            System.out.print (i + " ");
        }
    }
}
```

Glavni program koji kreira i pokreće tri niti je dat sa:

```
public class PrimerTriNiti {

    public static void main(String[] args) {

        Runnable java = new PrikazTeksta (“Java”, 20);
        Runnable spring = new PrikazTeksta (“Spring”, 20);
        Runnable brojevi = new PrikazBrojeva (10);

        Thread nitJava = new Thread(java);
        Thread nitSpring = new Thread(spring);
        Thread nitBrojevi = new Thread(brojevi);

        nitJava.start();
        nitSpring.start();
        nitBrojevi.start();

    }

}
```

Isto kao i u prethodnom primeru, nije moguće unapred znati tačan redosled izvršavanja naredbi. Jedan moguć redosled izvršavanja je dat u sledećem listingu.

```
Spring Java Java 1 Java Spring Spring Spring Spring Spring Spring
Spring Java 2 Java Java Java 3 Spring 4 Java 5 Spring 6 Java 7
Spring 8 Java 9 Spring 10 Java Java Java Java Spring Spring Java
Spring Java Spring Spring Spring Spring Spring Java Java Java Java
```

Ponovno pokretanje istog koda na istom računaru bi dalo drugačiji redosled izvršavanja, kao što je prikazano u narednom listingu.

```
Spring Spring Spring Spring Spring Spring Spring Spring Spring Java
Java Java Java Java Java Java Java Java Java 1 Java Spring
Java Java Java Java Java Java Java 2 3 4 Spring 5 Spring Spring
Spring Spring 6 Spring Spring Spring Spring Spring 7 8 9 10
```

2.6. Naredba join

Jedna od bitnih metoda iz klase Thread je metoda join. Ova metoda se koristi kada je potrebno da jedna nit sačeka drugu da završi sa radom pre nego što nastavi svoje izvršavanje. Ako se posmatraju dve niti, a i b, koje su objekti tipa Thread, ukoliko se u niti a pozove b.join(), onda će nit a preći u stanje spavanja sve dok se nit b ne završi. Potrebno je imati na umu da ova metoda može izbaciti izuzetak tipa InterruptedException, kojim se mora rukovati.

Kao primer, neka se ponovo posmatra program sa tri niti iz prethodnog poglavlja. Sve tri pomenute niti se pokreću iz četvrte niti, u kojoj se izvršava main metoda glavnog programa. Odmah nakon toga se glavna nit završava, dok preostale tri niti nastavljaju svoje izvršavanje. Ukoliko bismo hteli da, na primer, izmerimo ukupno vreme izvršavanja sve tri niti, potrebno je da nateramo glavnu nit da sačeka završetak rada ostalih niti, tako što ćemo pozvati metodu join nad objektima sve tri pokrenute niti, kao što je prikazano u nastavku:

```
public class PrimerNitiUvod {

    public static void main(String[] args) {

        Runnable java = new PrikazTeksta (“Java”, 20);
        Runnable spring = new PrikazTeksta (“Spring”, 20);
        Runnable brojevi = new PrikazBrojeva (10);

        Thread nitJava = new Thread(java);
        Thread nitSpring = new Thread(spring);
        Thread nitBrojevi = new Thread(brojevi);

        long pocetak = System.currentTimeMillis();

        nitJava.start();
        nitSpring.start();
        nitBrojevi.start();

        //cekamo da se sve niti zavrse
        try {
            nitJava.join();
            nitSpring.join();
            nitBrojevi.join();
        } catch (InterruptedException e){
            // rukovanje izuzetkom
        }
```

```
        //u ovom trenutku u sve niti zavrsene
        long kraj = System.currentTimeMillis();
       System.out.println ("Vreme izvrsavanja " + (kraj-pocetak));

    }

}
```

Potrebno je naglasiti da izvršavanje ovog programa nije ispravno ukoliko dođe do izbacivanja izuzetka tipa InterruptedException u bilo kojoj od join metoda. Za ispravno ponašanje programa, bilo bi potrebno izmeniti kod i obezbediti registrovanje završetka niti čak i u slučaju izuzetka.

Osim pomenute varijante metode join, postoje još dve preopterećene varijante. Jedna od njih je void join(long millis), koja čeka maksimalno zadati broj milisekundi da se nit nad kojom je pozvana završi. Ukoliko se vrednost milisekundi postavi na nulu, čeka se neograničeno.

2.7. Lambda izrazi sa Runnable interfejsom

Lambda izrazi su jedna od najznačajnijih funkcionalnosti koje su uvedene Java SE 8 platformom. Koriste se za kratko i jasno predstavljanje interfejsa sa jednom metodom, još poznatih kao funkcionalni interfejsi. U opštem obliku, lambda izraz je dat u sledećoj formi:

```
(lista argumenata) -> telo metode
```

Pre Java verzije 8, za kreiranje niti se mogla kreirati anonimna klasa koja implementira Runnable interfejs. Primer sa anonimnom klasom je dat u nastavku.

```
Runnable zadatak1 = new Runnable(){

    @Override
    public void run(){
        System.out.println("Zadatak #1 se izvrsava");
    }
};

Thread nit1 = new Thread(zadatak1);
nit1.start();
```

Anonimne klase su lokalne klase bez imena, koje se koriste u slučajevima kada je potrebno konstruisati samo jedan objekat te klase. Ovakav oblik operatora new se može koristiti bilo gde u kodu gde bi mogao da se koristi i običan new operator. Sama natklasa ili interfejs se proširuje tako što se dodaju implementacije svih potrebnih metoda, što je u slučaju funkcionalnog interfejsa implementacija tačno jedne metode. Ime nadtipa (osnovne klase ili interfejsa) se koristi za anonimnu klasu (pošto nema svoj tip).

Anonimna klasa se može i direktno proslediti konstruktoru klase Thread, kao u nastavku:

```
Thread nit2 = new Thread(new Runnable() {
    @Override
    public void run(){
        System.out.println(“Zadatak #1 se izvrsava”);
    }
});

nit2.start();
```

Sa lambda izrazima, prethodni kod se može napisati još konciznije i jasnije. Upotrebom lambda izraza, eliminiše se višak koda, u ovom slučaju deklaracije anonimne klase i run() metode.

```
// Lambda Runnable
Runnable zadatak3 = () -> { System.out.println(“Zadatak #3 se
izvrsava”); };

// pokretanje niti
new Thread(zadatak3).start();
```

U nastavku je dat kompletan primer kreiranja niti sa oba klasična načina upotrebom anonimnih klasa, kao i novi način pomoću lambda izraza.

```
public class RunnableLambdaPrimer {
    public static void main(String[] args) {
          System.out.println(Thread.currentThread().getName() + “:
RunnableTest”);

        // Runnable kao anonimna klasa
```

```
        Runnable zadatak1 = new Runnable(){

          @Override
          public void run(){
             System.out.println(Thread.currentThread().getName() +
“ se izvrsava”);
          }
        };

        // Prosledjivanje Runnable pri kreiranju nove niti
        Thread nit2 = new Thread(new Runnable() {
            @Override
            public void run(){
               System.out.println(Thread.currentThread().getName()
+ “ se izvrsava”);
            }
        });

        // Lambda Runnable
        Runnable zadatak3 = () -> {
             System.out.println(Thread.currentThread().getName() +
“ se izvrsava”);
        };

        Thread nit1 = new Thread(zadatak1);
        // pokretanje sve tri niti
        nit1.start();
        nit2.start();
        new Thread(zadatak3).start();

    }
}
```

Ponovo, zbog prirode niti, nije moguće tačno predvideti redosled izvršavanja ove tri niti. Dva od više mogućih izlaza programa su data u nastavku.

```
main: RunnableTest
Thread-1 se izvrsava
Thread-0 se izvrsava
Thread-2 se izvrsava
```

```
main: RunnableTest
Thread-1 se izvrsava
Thread-2 se izvrsava
Thread-0 se izvrsava
```

2.8. Sinhronizacija niti

Prvi problem u višenitnom programiranju nastaje kada više različitih niti moraju da sarađuju i uzajamno komuniciraju kako bi se dobio konačan rezultat izvršavanja programa. Kada se u programu pokrenu dve ili više niti, može doći do situacije da nekoliko niti pokuša da pristupi nekom deljenom resursu. Ukoliko dve ili više niti moraju da koriste isti resurs, potrebno pažljivo programiranje kako ne bi došlo do istovremenog korišćenja resursa.

Jedan od tipičnih problema kod programa sa paralelnim izvršavanjem je stanje trke (engl. *race condition*), koji nastaje u slučaju da je jedna nit prekinuta u sredini izvršavanja neke složenije operacije, a druga nit promeni neku vrednost ili uslov od kojih zavisi ispravno izvršavanje prve niti. Sam termin stanje trke potiče od toga da, kada posmatramo višenitno okruženje gde više niti pristupa istom deljenom resursu, niti su u trci da završe izvršavanje svoje metode pre nego što budu prekinute od strane neke druge niti. Potrebno je uočiti da kod stanja trke ne postoji problem ukoliko više niti pokušava da čita iz deljenog resursa, sve dok ne pokušaju da izmene njegovu vrednost.

Problem je najlakše prikazati na konkretnom primeru, kada imamo višenitno okruženje i deljenu klasnu promenljivu. U našem primeru postoje tri niti, koje dele isti objekat klase Brojač, a samim tim i sva polja tog objekta. U našem slučaju postoji jedno polje, b, koje se prvo inkrementira, a zatim dekrementira, odnosno svaka nit bi trebalo da ga ostavi u stanju u kome ga je inicijalno i zatekla. Na primer, ukoliko je b imalo početnu vrednost 0, inkrementiranje će vrednost promeniti na 1, a dekrementiranje će vratiti vrednost ponovo na 0. Kašnjenje se simulira upotrebom sleep() metode, pošto u realnom okruženju može postojati veliki broj pokrenutih procesa, i veliki broj korisnika koji pristupaju istoj aplikaciji u istom trenutku. U takvom okruženju je jako teško predvideti kada će doći do promene konteksta između niti koje se izvršavaju, odnosno kada će neka nit biti prekinuta, a procesor dodeljen nekoj drugoj. Implementacija klase Brojač je data sledećim kodom:

```
class Brojac implements Runnable {

    private int b = 0;

    public void inkrementiraj() {
        try {
            Thread.sleep(10);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        b++;
    }

    public void dekrementiraj() {
        b--;
    }

    public int getVrednost() {
        return b;
    }

    @Override
    public void run() {
        //inkrementiranje
        this.inkrementiraj();
        System.out.println("Vrednost nakon inkrementa "
                + Thread.currentThread().getName() + " " + this.
getVrednost());
        //dekrementiranje
        this.dekrementiraj();
        System.out.println("Vrednost nakon dekrementa "
                + Thread.currentThread().getName() + " " + this.
getVrednost());
    }
}
```

U glavnom programu postoje tri niti, koje sve dele jedan isti objekat klase Brojač. Ukoliko bi se sve tri niti izvršavale jedna iza druge, bez prekida, vrednost brojača nakon završetka svake niti bi bila 0. Međutim, u metodi inkrementiraj je dodato čekanje koje simulira dugu operaciju, i za vreme kojeg nit koja se trenutno izvršava spava. Vrlo je verovatno da nit neće kompletirati celu svoju run metodu, pre nego što joj bude oduzet procesor i dodeljen nekoj drugoj niti. Glavni program je dat sledećim kodom:

```
public class ProblemTrke{
    public static void main(String[] args) {
        Brojac brojac = new Brojac();
        Thread t1 = new Thread(brojac, “Nit-1”);
        Thread t2 = new Thread(brojac, “Nit-2”);
        Thread t3 = new Thread(brojac, “Nit-3”);
        t1.start();
        t2.start();
        t3.start();
    }
}
```

Problem nastaje ukoliko više niti pristupa objektu u isto vreme, odnosno izvršavaju istu naredbu (inkrement), jer dolazi do neočekivanih rezultata izvršavanja. Moguće je da se desi da prva nit završi samo inkrement vrednosti brojača, i da joj bude oduzet procesor pre nego što stigne da sačuva novu vrednost. Tada će sledeća nit pročitati pogrešnu vrednost broja, usled čega će i njeno izračunavanje biti pogrešno. Kompletan rezultat rada celog programa je nepredvidiv čak i ukoliko se više puta izvršava na istom računaru, što se može videti iz priložena dva rezultata izvršavanja:

```
Vrednost nakon inkrementa Nit-3 2
Vrednost nakon inkrementa Nit-1 2
Vrednost nakon inkrementa Nit-2 2
Vrednost nakon dekrementa Nit-1 0
Vrednost nakon dekrementa Nit-3 1
Vrednost nakon dekrementa Nit-2 -1
```

```
Vrednost nakon inkrementa Nit-2 3
Vrednost nakon inkrementa Nit-1 3
Vrednost nakon dekrementa Nit-1 1
Vrednost nakon inkrementa Nit-3 3
Vrednost nakon dekrementa Nit-3 0
Vrednost nakon dekrementa Nit-2 2
```

U programiranju u praksi naravno nije dozvoljeno da se program ponaša nekonzistentno i nepredvidljivo, pa je neophodno naći rešenje za ovaj problem. Da bi se rešio problem trke, mora se ograničiti da samo jedna nit dobije ekskluzivno pravo da koristi deljeni resurs u jednom trenutku. Mehanizam uzajamne isključivosti u pristupu deljenom resursu u Javi se realizuje preko sinhronizovanih metoda i naredbi. To se postiže upotrebom ključne reči *synchronized*, čime se deljeni resurs u određenoj metodi ili bloku naredbi štiti tako što se obezbeđuje da se izvršavanje sinhronizovane metode ili naredbe u jednoj niti ne može prekinuti od strane druge niti.

Problem trke niti u našem primeru se može rešiti tako što se kada jedna nit uđe u svoju run metodu onemogući da bude prekinuta od strane neke druge niti. Na taj način, svaka nit će bez prekida uzeti vrednost brojača, inkrementirati je i nakon toga dekrementirati. To se postiže sinhronizovanim pristupom objektu brojača, odnosno u run metodi dodajemo sinhronizaciju na objekat this.

```
class SinhroniBrojac implements Runnable {

    private int b = 0;

    public void inkrementiraj() {
        try {
            Thread.sleep(10);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        b++;
    }

    public void dekrementiraj() {
        b--;
    }

    public int getVrednost() {
        return b;
    }

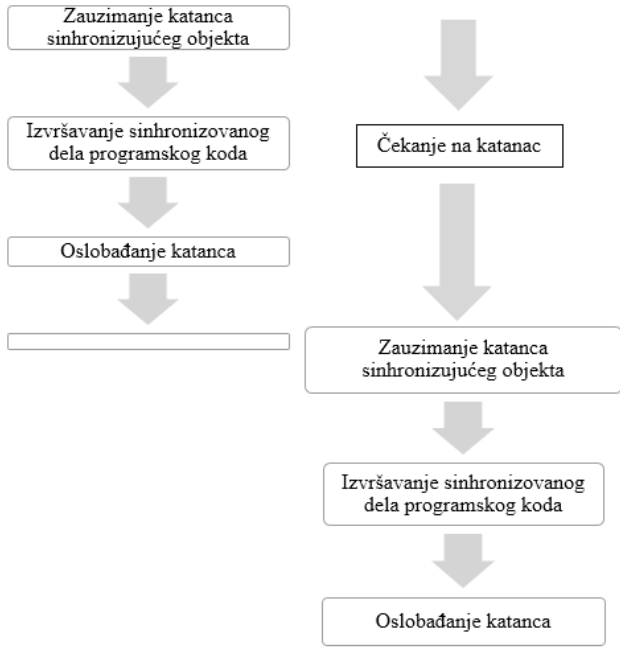
    @Override
    public void run() {
        synchronized (this) {
            //inkrementiranje
            this.inkrementiraj();
            System.out.println("Vrednost nakon inkrementa "
                  + Thread.currentThread().getName() + " " + this.
getVrednost());
            //dekrementiranje
            this.dekrementiraj();
            System.out.println("Vrednost nakon dekrementa "
                  + Thread.currentThread().getName() + " " + this.
getVrednost());
        }
    }
}
```

Ukoliko se sada ponovo pokrene izvršavanje programa, vidi se da se program u ovom slučaju ispravno izvršava.

```
Vrednost nakon inkrementa Nit-1 1
Vrednost nakon dekrementa Nit-1 0
Vrednost nakon inkrementa Nit-3 1
Vrednost nakon dekrementa Nit-3 0
Vrednost nakon inkrementa Nit-2 1
Vrednost nakon dekrementa Nit-2 0
```

U prethodnom primeru, blok naredbi unutar run metode je sinhronizovan po objektu brojača, na koga ukazuje promenljiva this. Na sličan način, i sinhronizovane metode se implicitno sinhronizuju po objektu. Objektne metode se sinhronizuju po objektu za koji se pozivaju, odnosno, ukoliko se doda modifikator synchronized nekoj objektnoj metodi, to je ekvivalentno kao da je u telu te metode na početku napisano synchronized (this). U Javi važi opšte pravilo za sinhronizaciju niti, koje kaže da se u dve različite niti ne može istovremeno izvršavati blok naredbi koji je sinhronizovan po istom objektu. Ukoliko je jedna nit sinhronizovana po nekom objektu, a druga nit pokušava da se sinhronizuje po tom istom objektu, druga nit će čekati dok prva nit ne završi rad nad sinhronizujućem objektu.

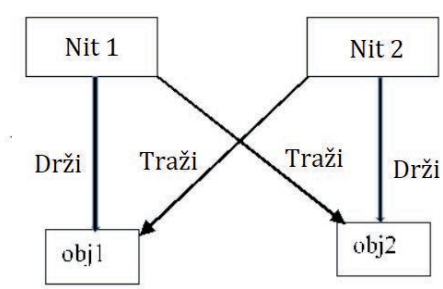
Sama realizacija sinhronizacije je bazirana na katancu (engl. *lock*). Svaki objekat u Javi sadrži i katanac, koji u bilo kom trenutku može zaključati samo jedna nit. Katanac se zaključava (zauzima) izvršavanjem sinhronizovane naredbe ili sinhronizovane metode. Ukoliko je katanac slobodan, nit ga zaključava i počinje sa izvršavanjem svojih naredbi. Nakon završetka sinhronizovane naredbe ili metode, nit oslobađa katanac. Ukoliko katanac nije slobodan, nit koja pokušava da ga zauzme će morati da čeka dok nit koja trenutno drži katanac zaključanim ne oslobodi katanac. Nit koja čeka na katanac je u stvari u stanju spavanja, i neće se probuditi sve dok katanac ne bude otključan. Postupak je ilustrovan na slici 2.2.



Slika 2.2. Mehanizam katanca za sinhronizaciju

2.8.1. Mrtva petlja

Sinhronizacija nam pomaže da rešimo problem trke niti, ali može dovesti do novog tipa problema, koji se naziva mrtva petlja (engl. *deadlock*). Mrtva petlja opisuje situaciju gde su dve ili više niti zauvek blokirane, čekajući jedna na drugu da oslobode katance objekata po kojima su sinhronizovane. Do ove situacije dolazi kada se niti sinhronizuju po više objekata. Može se desiti da dve niti zauzmu katance dva različita objekta, i obe čekaju na onu drugu nit da oslobodi katanac drugog objekta kako bi mogle da nastave rad. Kako bi se što jednostavnije objasnio problem, posmatra se situacija sa slike 2.3.



Slika 2.3. Mrtva petlja u Javi

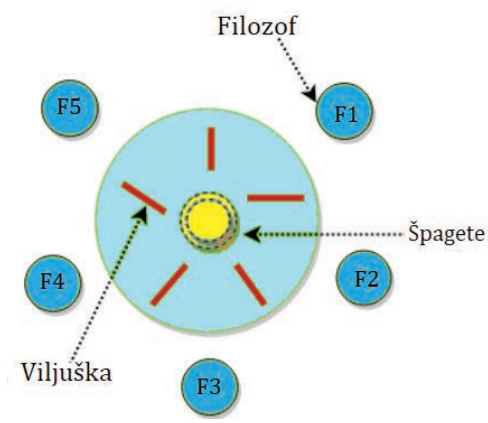
Pretpostavljamo da postoje dve niti, Nit-1 i Nit-2, i dva objekta, obj1 i obj2. Nit-1 već drži katanac nad obj1, i za dalji rad joj treba i katanac nad obj2. Sa druge strane, u isto vreme Nit-2 drži katanac nad obj2, i želi katanac nad obj1. U ovom slučaju, obe niti će čekati zauvek na onu drugu da otpusti katanac, i time je nastala mrtva petlja.

Mrtva petlja u višenitnom okruženju može nastati u sledećim slučajevima:

- Jedna sinhronizovana metoda se poziva iz druge sinhronizovane metode.
- Postoje ugnježđeni sinhronizovani blokovi.

Jedan od najpoznatijih primera koji ilustruje mrtvu petlju je problem filozofa koji večeraju. U pitanju je klasični primer iz računarstva koji demonstrira probleme sa sinhronizacijom i rešenje dizajna konkurentnih algoritama. Problem je formulisao Dijkstra 1965. godine, i u osnovi je ilustrovao računare koji se nadmeću za pristup periferijama. Mrtva petlja nastaje kada više procesa čeka na resurs koji drži neki drugi proces, a problem filozofa nudi rešenje ovog tipa grešaka.

Problem se formuliše na sledeći način. Pet filozofa naizmenično jedu i razmišljaju. Svih pet filozofa sede za okruglim stolom, na središtu stola se nalazi velika činija sa špagetama, a između svaka dva filozofa se nalazi jedna viljuška. Ukupno, postoji pet viljušaka. Svaku viljušku u jednom trenutku može koristiti samo jedan filozof. Međutim, da bi filozof mogao da jede, potrebne su mu dve viljuške – ona sa njegove leve strane, i ona sa njegove desne strane. Filozof može da uzme slobodnu viljušku, ali mu nije dozvoljeno da počne da jede sve dok nema u rukama obe viljuške – i levu i desnu. Pretpostavlja se da jedenje nije ograničeno količinom špageta – smatra se da postoji beskonačna količina špageta. Ilustracija problema je prikazana na slici 2.4.



Slika 2.4. Problem filozofa koji večeraju

Svi filozofi prate isti model ponašanja. Filozof prvo razmišlja određeno vreme, i onda ogladni. Kada je gladan, filozof prvo uzima svoju levu viljušku, a zatim posle nekog vremena uzima i svoju desnu viljušku. Ukoliko bilo koja viljuška nije slobodna

(koristi je susedni filozof), čeka da se ona oslobodi. Kada ima obe viljuške u svom posedu, počinje da jede. Nakon određenog vremena, kada završi sa jelom, filozof odlaže prvo svoju desnu viljušku, a zatim nakon nekog vremena i svoju levu viljušku. Kada je završio sa jelom, filozof nastavlja da razmišlja, sve dok ponovo ne ogladni itd. Implementacija ponašanja filozofa je data sledećom klasom Filozof. Zbog lakšeg predstavljanja akcija, sve su modelovane jednom metodom izvrsiAkciju, koja osim stringa koji označava akciju koja se trenutno izvršava, simulira i nasumično vreme trajanja te akcije, pozivom metode Thread.wait().

```
public class Filozof implements Runnable {

    private final Object levaViljuska;
    private final Object desnaViljuska;

    public Filozof (Object levaViljuska, Object desnaViljuska){
        this.levaViljuska = levaViljuska;
        this.desnaViljuska = desnaViljuska;
    }

     private void izvrsiAkciju (String akcija) throws Interrupte-
dException {
         System.out.println(Thread.currentThread().getName() + “ “
+ akcija);
        Thread.sleep((int)(Math.random() * 100));
    }

    @Override
    public void run() {
        try {
            while (true){
                izvrsiAkciju(System.nanoTime() + “: Razmislja”);
                synchronized (levaViljuska){
                     izvrsiAkciju(System.nanoTime() + “: Uzeo levu
viljusku”);
                    synchronized (desnaViljuska){
                         izvrsiAkciju(System.nanoTime() + “: Uzeo
desnu viljusku - jede”);
                        izvrsiAkciju(System.nanoTime() + “: Spusta
desnu viljusku “);
                    }
                   izvrsiAkciju(System.nanoTime() + “: Spusta levu
viljusku - ponovo razmislja”);
                }
```

```
            }
        } catch (InterruptedException e){
            Thread.currentThread().interrupt();
        }
    }

}
```

Viljuške se mogu predstaviti i posebnom klasom, ali radi pojednostavljenja, predstavljene su običnim objektima klase Object. Glavni program koji simulira problem je dat u nastavku. Prvo se inicijalizuju niz od pet filozofa, kao i niz od pet viljušaka. Svakom filozofu se dodeljuje koja mu je leva, a koja desna viljuška. Nakon toga se pokreće svih pet niti filozofa.

```
public class FilozofiZaVecerom {

    public static void main(String[] args) {
        Filozof[] filozofi = new Filozof[5];
        Object[] viljuske = new Object[5];

        for (int i = 0; i < viljuske.length; i++){
            viljuske[i] = new Object();
        }

        for (int i = 0; i < filozofi.length; i++){
            Object levaViljuska = viljuske[i];
          Object desnaViljuska = viljuske[(i+1) % viljuske.length];

            filozofi[i] = new Filozof (levaViljuska, desnaViljuska);

           Thread t = new Thread (filozofi[i], "Filozof " + (i + 1));
            t.start();
        }
    }

}
```

Nakon puštanja programa u rad, izvršavanje će neko vreme izgledati u redu, ali će, pre ili kasnije, doći do mrtve petlje. Mrtva petlja u ovom primeru nastaje u situaciji kada svih pet filozofa uzme svoju levu viljušku, i tako ostaju blokirani zauvek, pošto svaki filozof čeka svoju desnu viljušku (koju drži susedni filozof sa desne strane, kome

je to leva viljuška). Problem se može javiti odmah, kao u listingu u nastavku, ili nakon određenog proteklog vremena. Program ostaje blokiran sve do nasilnog prekida rada.

```
Filozof 2 671723372420200: Razmislja
Filozof 3 671723372718400: Razmislja
Filozof 1 671723372391600: Razmislja
Filozof 5 671723373316200: Razmislja
Filozof 4 671723373312000: Razmislja
Filozof 3 671723409974200: Uzeo levu viljusku
Filozof 2 671723417104700: Uzeo levu viljusku
Filozof 5 671723442952900: Uzeo levu viljusku
Filozof 4 671723449578700: Uzeo levu viljusku
Filozof 1 671723487866000: Uzeo levu viljusku
```

Dakle, sinhronizacija sama po sebi ne rešava sve probleme. Kao što smo videli u prethodnom primeru, može dovesti i do pojave novog problema – mrtve petlje. Kako bi se ova situacija izbegla, u situacijama kada se niti sinhronizuju po više objekata, potrebno je koristiti posebne tehnike programiranja. Najjednostavniji način rešavanja problema mrtve petlje je uređenje resursa. Ovaj način podrazumeva da se, u opštem slučaju, sinhronizujući objekti urede po nekom redosledu, i da se obezbedi da sve niti zauzimaju katance ovih objekata tim redosledom.

U slučaju pet filozofa, problem mrtve petlje se jednostavno rešava, tako što svi filozofi osim jednog uzimaju prvo levu viljušku, a nakon toga desnu. Preostali filozof se definiše na takav način da prvo uzme desnu, a zatim levu viljušku. U najgorem slučaju, može se desiti da četiri filozofa uzmu svoju levu viljušku, međutim, poslednji filozof neće ni pokušati da uzme prvo svoju levu viljušku (i time napraviti mrtvu petlju), već bi pokušao da uzme prvo viljušku sa desne strane, koja je već zauzeta, i ostao bi da čeka njeno oslobađanje, bez zauzimanja svoje leve viljuške. Jedna varijanta implementacije ovog rešenja na primeru datog koda je prikazana u nastavku.

```
public class FilozofiZaVecerom {

    public static void main(String[] args) {
        Filozof[] filozofi = new Filozof[5];
        Object[] viljuske = new Object[5];

        for (int i = 0; i < viljuske.length; i++){
            viljuske[i] = new Object();
        }
```

```
        for (int i = 0; i < filozofi.length; i++){
            Object levaViljuska = viljuske[i];
          Object desnaViljuska = viljuske[(i+1) % viljuske.length];


            if (i == filozofi.length - 1){
                filozofi[i] = new Filozof (desnaViljuska, levaVilju-
ska); // poslednji filozof prvo uzima desnu viljusku
            } else {
                filozofi[i] = new Filozof (levaViljuska, desnaVilju-
ska);
            }
           Thread t = new Thread (filozofi[i], "Filozof " + (i + 1));
            t.start();
        }
    }

}
```

Nakon unete modifikacije i pokretanja programa, vidi se da se sada program izvršava bez greške. Šta više, program se može izvršavati proizvoljno dugo, bez šanse da se javi mrtva petlja. Deo izlaza programa je prikazan u nastavku.

```
...
Filozof 3 673103174210800: Razmislja
Filozof 2 673103174273400: Uzeo desnu viljusku - jede
Filozof 4 673103228023500: Spusta levu viljusku - ponovo razmislja
Filozof 5 673103228068800: Uzeo desnu viljusku - jede
Filozof 2 673103258299500: Spusta desnu viljusku
Filozof 5 673103259484800: Spusta desnu viljusku
Filozof 2 673103307354700: Spusta levu viljusku - ponovo razmislja
Filozof 3 673103307399300: Uzeo levu viljusku
Filozof 4 673103309276200: Razmislja
Filozof 3 673103310477700: Uzeo desnu viljusku - jede
Filozof 2 673103318400400: Razmislja
Filozof 5 673103319444800: Spusta levu viljusku - ponovo razmislja
Filozof 3 673103347416400: Spusta desnu viljusku
Filozof 3 673103358581200: Spusta levu viljusku - ponovo razmislja
Filozof 4 673103358584100: Uzeo levu viljusku
Filozof 5 673103372572600: Razmislja
Filozof 1 673103372575000: Uzeo levu viljusku
Filozof 2 673103402761200: Uzeo levu viljusku
```

```
Filozof 3 673103407737300: Razmislja
Filozof 4 673103436739600: Uzeo desnu viljusku - jede
Filozof 3 673103446663900: Uzeo levu viljusku
Filozof 4 673103484185800: Spusta desnu viljusku
Filozof 4 673103543961800: Spusta levu viljusku - ponovo razmislja
...
```

2.9. Saradnja niti

Deljenje resursa nije jedini mogući način saradnje između više niti. Vrlo čest primer u programiranju je da jedna nit proizvodi rezultat koji koristi neka druga nit. Time je redosled izvršavanja instrukcija u paralelnom izvršavanju ovih niti uslovljen. Na primer, ukoliko druga nit dođe do tačke u svom izvršavanju gde joj je potreban rezultat koji proizvodi prva nit, druga nit mora da stane i sačeka, pošto prva nit još uvek nije proizvela potreban rezultat. Druga nit u tom trenutku ne može da nastavi sa radom, već mora da pređe u neaktivno stanje i da sačeka. Samim tim, mora da postoji neki način da se druga nit obavesti u trenutku kada rezultat izvršavanja prve niti bude spreman, kako bi se ta druga nit aktivirala iz blokiranog stanja i nastavila svoj rad.

U programskom jeziku Java postoje objektne metode wait() i notify(), koje su definisane u klasi Object, i samim tim nasleđene u svim ostalim klasama. Kada se u jednoj niti pozove wait() za neki objekat, nit prelazi u blokirano stanje sve dok se ne pozove notify() za taj isti objekat. Poziv metode notify() se očigledno mora desiti u nekoj drugoj niti, pošto je prva nit blokirana. Tipičan način programiranja bi bio da se u prvoj niti pozove metoda wait() u slučaju kada je potreban rezultat druge niti koji još nije spreman. Sa druge strane, kada druga nit proizvede potreban rezultat koji čeka prva nit, poziva se metoda notify() kako bi se prva nit probudila i iskoristila raspoloživ rezultat.

U računarstvu, gore opisani problem je poznat kao problem proizvođača i potrošača (engl. *producer-consumer*), i jedan je od klasičnih primera sinhronizacije u višenitnom okruženju. Problem predviđa postojanje dve niti, proizvođača i potrošača, koji dele zajednički objekat u koji se smešta, i odakle se uzima rezultat (u kompleksnijim primenama to može biti i zajednički bafer fiksne dužine). Osnovni princip rada je sledeći:

- Posao proizvođača je da generiše podatak, smesti ga u zajedničku promenljivu (ili bafer), i počne posao iz početka.
- U isto vreme, proizvođač troši podatke jedan po jedan (na primer, uzme ga iz bafera).

Problem leži u činjenici da proizvođač ne sme da pokuša da smesti podatak u zajedničku promenljivu kojoj se nalazi podatak koji potrošač još nije preuzeo (ili u bafer koji je pun), kao i da potrošač ne sme da pokuša da pročita podatak pre nego što ga proizvođač smesti u zajedničku promenljivu (odnosno da pokuša čitanje iz praznog bafera). Logika koja rešava ovaj problem se upravo zasniva na primeni wait() i notify() metoda nad zajedničkim objektom gde se smešta rezultat.

Logiku možemo ilustrovati na pojednostavljenom primeru, gde posmatramo proizvođača koji proizvodi 10 rezultata i potrošača koji konzumira ovih 10 rezultata. Prva klasa koja je potrebna je klasa Rezultat, koja predstavlja objekat koji proizvodi proizvođač, a troši potrošač. Klasa Rezultat ima dva polja. Prvo polje, sadržaj, jeste korisna informacija koju proizvodi proizvođač. Drugo polje, boolean podatak dostupan, označava da li je proizvođač stavio svež sadržaj u Rezultat (dostupan = true), odnosno da sadržaj nije validan jer ga je potrošač već uzeo (dostupan = false). U klasi Rezultat postoje i dve sinhronizovane metode, get() i put(), sa sledećim funkcionalnostima.

Metodu get() poziva potrošač kada želi da uzme proizvedeni sadržaj. Ukoliko podatak nije dostupan, to znači da proizvođač još nije postavio novi sadržaj u rezultat, odnosno da potrošač mora da sačeka. Ukoliko je podatak dostupan, potrošač uzima sadržaj, postavlja polje dostupan na vrednost false, kako bi označio da je vrednost sadržaja već uzeta i sa notifyAll() obaveštava proizvođača koji eventualno čeka da postavi novi sadržaj da sada može slobodno da postavi novi podatak.

```
public class Proizvod {
    private int sadrzaj;
    private boolean dostupan = false;

    public synchronized int get (){
        if (!dostupan){
            try {
                wait();
            } catch (InterruptedException e){
                System.out.println(e.getMessage());
            }
        }

        System.out.println("Uzima se vrednost " + sadrzaj);
        dostupan = false;
        notifyAll();
        return sadrzaj;
    }

    public synchronized void put (int value){
        if (dostupan){
            try {
                wait();
            }catch (InterruptedException e){
                System.out.println(e.getMessage());
            }
        }
```

```
        System.out.println("Stavlja se vrednost " + value);
        sadrzaj = value;
        dostupan = true;
        notifyAll();
    }
}
```

Sa druge strane, metodu put poziva proizvođač kada želi da postavi novi proizvedeni sadržaj. Ukoliko polje dostupan ima vrednost true, to znači da potrošač još uvek nije preuzeo prethodni sadržaj, odnosno da proizvođač mora da sačeka sa postavljanjem novog sadržaja. Ukoliko je vrednost polja dostupan false, to znači da je potrošač pročitao prethodni sadržaj, odnosno da proizvođač sada sme da postavi novi sadržaj. Nakon postavljanja novog sadržaja, postavlja se vrednost polja dostupan na true, i sa notifyAll() obaveštava potrošač koji eventualno čeka na postavljanje novog sadržaja da sada može preuzeti sadržaj.

Može se primetiti da je praktično sva logika implementirana u okviru klase Rezultat, tako da su u ovom primeru klase Proizvođač i Potrošač izuzetno jednostavne. Obe klase kao polje sadrže isti objekat tipa Rezultat, u koji Proizvođač smešta novi sadržaj, a odakle Potrošač uzima i konzumira taj sadržaj. Dodatno polje redniBroj samo označava redni broj proizvođača / potrošača, pošto mehanizam u potpunosti radi i u slučaju da postoji više proizvođača i više potrošača. Obe klase Proizvođač i Potrošač proširuju klasu Thread, pošto je potrebno da se paralelno izvršavaju. U klasi Proizvođač, metoda run() je implementirana tako da proizvođač postavlja 10 puta novi sadržaj u rezultat, sa nasumičnim intervalima između dva postavljanja vrednosti. U klasi Potrošač, metoda run() je implementirana tako da potrošač čita 10 vrednosti sadržaja iz rezultata.

```
public class Proizvodjac extends Thread{

    private Proizvod proizvod;
   private int redniBroj; // u slucaju da posotji vise proizvodjaca

    public Proizvodjac(Proizvod proizvod, int redniBroj){
        this.proizvod = proizvod;
        this.redniBroj = redniBroj;
    }

    public void run (){
        for (int i = 0; i < 10; i++){
            proizvod.put(i);
```

```
            try {
                sleep((int) Math.random() * 100);
            } catch (InterruptedException e){
                System.out.println(e.getMessage());
            }
        }
    }

}
```

```
public class Potrosac extends Thread {
    private Proizvod proizvod;
    private int redniBroj; //u slucaju da postoji vise potrosaca

    public Potrosac (Proizvod proizvod, int redniBroj){
        this.proizvod = proizvod;
        this.redniBroj = redniBroj;
    }

    public void run (){
        int uzetaVrednost = 0;
        for (int i = 0; i < 10; i++){
            uzetaVrednost = proizvod.get();

        }
    }

}
```

Glavni program je takođe jednostavan. Sve što treba da se uradi jeste da se kreira jedan objekat tipa Rezultat, koji je zajednički i za proizvođača i za potrošača, da se kreiraju instance proizvođača i potrošača i da se pokrenu odgovarajuće niti.

```
public class ProizvodjacPotrosac {

    public static void main(String[] args) {
        Proizvod proizvod = new Proizvod ();
        Proizvodjac p1 = new Proizvodjac(proizvod, 1);
        Potrosac c1 = new Potrosac(proizvod, 1);
```

```
        p1.start();
        c1.start();

    }

}
```

Nakon pokretanja programa, vidi se da je sinhronizacija uspešno realizovana, pošto potrošač redom uzima vrednosti koje proizvođač postavlja, kao što je prikazano u rezultatu izvršavanja:

```
Stavlja se vrednost 0
Uzima se vrednost 0
Stavlja se vrednost 1
Uzima se vrednost 1
Stavlja se vrednost 2
Uzima se vrednost 2
Stavlja se vrednost 3
Uzima se vrednost 3
Stavlja se vrednost 4
Uzima se vrednost 4
Stavlja se vrednost 5
Uzima se vrednost 5
Stavlja se vrednost 6
Uzima se vrednost 6
Stavlja se vrednost 7
Uzima se vrednost 7
Stavlja se vrednost 8
Uzima se vrednost 8
Stavlja se vrednost 9
Uzima se vrednost 9
```

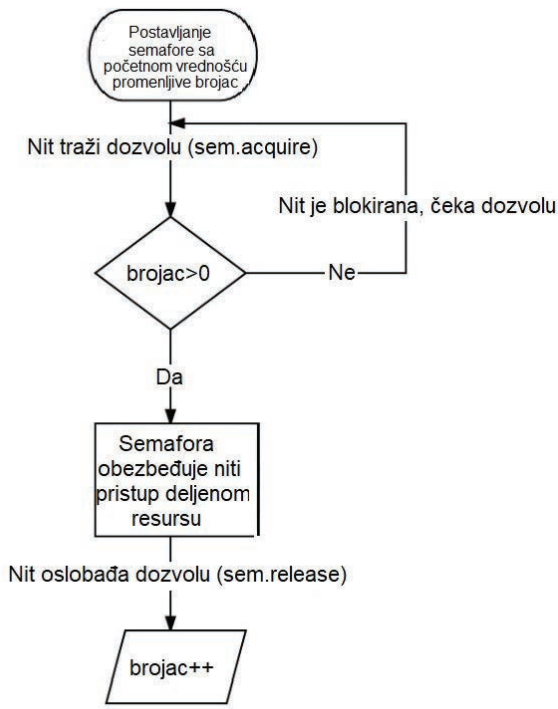
2.10. Semafori

Mehanizam semafora se može koristiti za pristup deljenom resursu uz pomoć brojača. Ukoliko je brojač veći od 0, pristup se dozvoljava. Ukoliko brojač ima vrednost 0, pristup se ne dozvoljava. Brojač služi za brojanje dozvola pristupa nekom resursu. Dakle, ukoliko neka nit želi da pristupi određenom resursu, mora dobiti dozvolu od strane semafora.

Java pruža implementaciju ovog mehanizma u klasi Semaphore, koja se nalazi u java.util.concurrent paketu. Uopšteno gledano, u Javi, ukoliko se koristi semafor, nit koja želi da pristupi resursu treba da traži dozvolu.

- Ukoliko je vrednost brojača semafora veća od 0, nit dobija dozvolu, a brojač se dekrementira.
- U suprotnom, nit je blokirana sve dok ne bude moguće dobiti dozvolu.
- Kada nit završi sa resursom i pristup joj više nije neophodan, ona oslobađa dozvolu, čime se inkrementira vrednost brojača semafora.
- Ukoliko neka nit čeka na dozvolu, ona će u tom trenutku dobiti dozvolu.

Dijagram rada semafora je dat na slici 2.5.



Slika 2.5. Dijagram rada semafora

Konstruktor klase Semaphore prihvata jednu vrednost tipa int, koja označava broj niti koje mogu paralelno pristupiti određenom resursu:

```
Semaphore semafor = new Semaphore(4);
```

Ova naredba kreira semafor sa 4 dozvole, odnosno maksimalno 4 niti može u istom trenutku pristupiti resursu. Osim konstruktora, klasa Semaphore ima još neke bitne metode. Ukoliko je semafor objekat klase Semaphore:

- semafor.availablePermits() – vraća broj trenutno raspoloživih dozvola.
- semafor.acquire() – nit se prijavljuje na semafor, ukoliko ima dostupnih dozvola dobija pristup resursu i dekrementira broj trenutno slobodnih dozvola. U suprotnom, nit čeka da se oslobodi neka dozvola.
- semafor.release() – kada niti više nije potreban pristup resursu, oslobađa svoju dozvolu, a broj trenutno slobodnih dozvola se inkrementira.

Posmatrajmo kao primer skup bankomata u banci. Ukoliko postoji četiri bankomata, to znači da maksimalno četiri korisnika može da istovremeno podiže novac, dok ostali moraju da čekaju. Realizacija ovakvog sistema niti je jednostavna korišćenjem klase Semaphore. Pošto je dozvoljeno da maksimalno četiri korisnika istovremeno pristupi, semafor se pravi na sledeći način:

```
static Semaphore bankomati = new Semaphore(4);
```

Svaki korisnik bankomata se može predstaviti kao nit, koja u svojoj run() metodi traži dozvolu za pristup, i kada dobije pristup radi broj određenih operacija na bankomatu, koji su radi jednostavnosti primera samo označeni kao operacija 1, operacija 2 i operacija 3 (u stvarnosti to bi mogli da budu unos pin koda, upit stanja, isplata novca i slično). Takođe, radi jednostavnosti, svaka od operacija traje 1 sekundu, što je implementirano sa Thread.sleep(1000). Opet, radi jednostavnosti primera, klasa KorisnikBankomata je kreirana kao unutrašnja statička klasa u okviru klase gde se nalazi i definicija semafora, kao i main metoda. Glavni program u ovom slučaju treba samo da kreira objekte određenog broja korisnika, koji su opisani klasom KorisnikBankomata, i da zatim pokrene odgovarajuće niti. Kompletan program je dat u nastavku:

```
import java.util.concurrent.Semaphore;

public class Banka {

	// max 4 osobe
	static Semaphore bankomati = new Semaphore(4);

	static class KorisnikBankomata extends Thread {

		String ime = “”;
```

```
            KorisnikBankomata(String ime) {
                  this.ime = ime;
            }

            public void run() {
                  try {
                        System.out.println(ime + " : trazi do-
zvolu za pristup...");
                        System.out.println(ime + " : trenutni
broj slobodnih bankomata: " + bankomati.availablePermits());

                        bankomati.acquire();
                        System.out.println(ime + " : dobio do-
zvolu za pristup!");
                        try {
                              for (int i = 1; i <= 3; i++) {
                                    System.out.println(ime + "
: obavlja operaciju " + i + ", trenutni broj slobodnih bankomata :
"
+ bankomati.availablePermits());
                                    // sleep 1 sekundu
                                    Thread.sleep(1000);
                              }
                        } finally {
                        // poziv release() nakon uspesnog acqu-
ire()
                              System.out.println(ime   +   "   :
oslobadja bankomat...");
                              bankomati.release();
                              System.out.println(ime + " : tre-
nutni broj slobodnih bankomata: " + bankomati.availablePermits());
                        }
                  } catch (InterruptedException e) {
                        e.printStackTrace();
                  }
            }
      }

      public static void main(String[] args) {
            System.out.println("Ukupan broj dostupnih dozvola : "
                        + bankomati.availablePermits());
```

```
            KorisnikBankomata t1 = new KorisnikBankomata(“A”);
            t1.start();
            KorisnikBankomata t2 = new KorisnikBankomata(“B”);
            t2.start();
            KorisnikBankomata t3 = new KorisnikBankomata(“C”);
            t3.start();
            KorisnikBankomata t4 = new KorisnikBankomata(“D”);
            t4.start();
            KorisnikBankomata t5 = new KorisnikBankomata(“E”);
            t5.start();
            KorisnikBankomata t6 = new KorisnikBankomata(“F”);
            t6.start();
      }
}
```

Jedan mogući rezultat izvršavanja ovog programa je dat u nastavku. Obratite pažnju da, iako je kreirano 6 korisnika, samo 4 mogu koristiti bankomate simultano. U ovom slučaju to su korisnici A, B, C i E. Korisnici D i F čekaju na oslobađanje bankomata, pre nego što mogu da nastave sa izvršavanjem svojih operacija.

```
Ukupan broj dostupnih dozvola : 4
A : trazi dozvolu za pristup...
A : trenutni broj slobodnih bankomata: 4
A : dobio dozvolu za pristup!
A : obavlja operaciju 1, trenutni broj slobodnih bankomata : 3
B : trazi dozvolu za pristup...
B : trenutni broj slobodnih bankomata: 3
B : dobio dozvolu za pristup!
B : obavlja operaciju 1, trenutni broj slobodnih bankomata : 2
C : trazi dozvolu za pristup...
C : trenutni broj slobodnih bankomata: 2
C : dobio dozvolu za pristup!
C : obavlja operaciju 1, trenutni broj slobodnih bankomata : 1
E : trazi dozvolu za pristup...
E : trenutni broj slobodnih bankomata: 1
E : dobio dozvolu za pristup!
E : obavlja operaciju 1, trenutni broj slobodnih bankomata : 0
D : trazi dozvolu za pristup...
D : trenutni broj slobodnih bankomata: 0
F : trazi dozvolu za pristup...
F : trenutni broj slobodnih bankomata: 0
```

```
A : obavlja operaciju 2, trenutni broj slobodnih bankomata : 0
B : obavlja operaciju 2, trenutni broj slobodnih bankomata : 0
C : obavlja operaciju 2, trenutni broj slobodnih bankomata : 0
E : obavlja operaciju 2, trenutni broj slobodnih bankomata : 0
A : obavlja operaciju 3, trenutni broj slobodnih bankomata : 0
B : obavlja operaciju 3, trenutni broj slobodnih bankomata : 0
C : obavlja operaciju 3, trenutni broj slobodnih bankomata : 0
E : obavlja operaciju 3, trenutni broj slobodnih bankomata : 0
A : oslobadja bankomat...
B : oslobadja bankomat...
D : dobio dozvolu za pristup!
D : obavlja operaciju 1, trenutni broj slobodnih bankomata : 1
F : dobio dozvolu za pristup!
F : obavlja operaciju 1, trenutni broj slobodnih bankomata : 0
A : trenutni broj slobodnih bankomata: 1
B : trenutni broj slobodnih bankomata: 0
E : oslobadja bankomat...
C : oslobadja bankomat...
C : trenutni broj slobodnih bankomata: 1
E : trenutni broj slobodnih bankomata: 2
D : obavlja operaciju 2, trenutni broj slobodnih bankomata : 2
F : obavlja operaciju 2, trenutni broj slobodnih bankomata : 2
F : obavlja operaciju 3, trenutni broj slobodnih bankomata : 2
D : obavlja operaciju 3, trenutni broj slobodnih bankomata : 2
D : oslobadja bankomat...
F : oslobadja bankomat...
F : trenutni broj slobodnih bankomata: 3
D : trenutni broj slobodnih bankomata: 4
```

2.11. Zadaci za vežbu

Zadatak 1: Napraviti program koji redom pronalazi proste brojeve, smešta ih u listu i štampa ih na standardnom izlazu. Ograničiti vreme izvršavanja programa, odnosno obezbediti da nakon 10 sekundi program prestane da traži nove proste brojeve.

Rešenje: Ukoliko se posmatra postavka, može se uočiti da su nam potrebne dve niti. Jedna, tzv. *worker* nit, u kojoj će biti implementirana logika za pronalaženje prostih brojeva, njihovo smeštanje u odgovarajuću strukturu podataka (niz ili u našem slučaju listu) i štampa pronađene proste brojeve. Sa druge strane, potrebna je druga nit koja će meriti 10 sekundi, i nakon isteka definisanog vremena obavestiti prvu nit da treba da prestane da traži nove proste brojeve. Dakle, potrebno je implementirati logiku tajmera. Funkciju tajmera može obavljati main nit, koja će nakon pokretanja

niti koja izračunava proste brojeve početi da meri vreme (implementirano pozivom metode sleep odgovarajući vremenski period). Nakon isteka vremena, postavlja se vrednost promenljive isteklo na true. U worker niti, na početku svake iteracije se proverava vrednost promenljive isteklo. Dokle god je vrednost false, nastavlja se sa traženjem prostih brojeva. Kada se detektuje da je vrednost isteklo true, prestaje se sa pretraživanjem. Obratite pažnju da je promenljiva isteklo deklarisana sa modifikatorom volatile. Sa ovim modifikatorom se još uvek nismo susretali. U osnovi stvari, volatile se koristi kao indikator da će nekoj promenljivoj više različitih niti pristupati i menjati vrednost. Vrednost ove promenljive se direktno čuva u glavnoj memoriji i čita iz nje, odnosno, preciznije rečeno, nikada se ne čita iz procesorske keš memorije. Pristup ovakvoj promenljivoj se uvek ponaša kao da se nalazi u sinhronizujućem bloku (iako u stvarnosti ne postoji zauzimanje katanca te promenljive). Drugim rečima, volatile je vrlo sličan sa synchronized, uz sledeće razlike:

- Promenljiva primitivnog tipa se može deklarisati kao volatile, dok se ne može koristiti za sinhronizaciju. I objekti mogu biti deklarisani sa modifikatorom volatile.
- Pristup promenljivoj koja je volatile nikada neće dovesti do blokiranja, pošto se izvršavaju samo jednostavno čitanje i upis, i ne postoji zauzimanje katanca.
- Pošto ne postoji zauzimanje katanca, volatile nije adekvatan za kompleksnije operacije čitanja-ažuriranja-upisa koje treba da se izvrše atomski.
- Promenljiva objektnog tipa sa modifikatorom volatile može imati vrednost null, za razliku od pokušaja sinhronizacije na objekat koji ima vrednost null, koji dovodi do izbacivanja izuzetka tipa NullPointerException.

```
import java.util.ArrayList;

public class ProstiBrojevi extends Thread {

    public static final int MAX_BROJ_PROSTIH = 1000000;
    public static final int DESET_SEKUNDI = 10000;
    public volatile boolean isteklo = false;

    @Override
    public void run() {
        ArrayList<Integer> prostiBrojevi = new ArrayList();
        int brojPronadjenihProstih = 0;
        for (int i = 2; brojPronadjenihProstih < MAX_BROJ_PROSTIH;
i++) {

            if (isteklo) { //provera da li je tajmer istekao
                break;
            }
```

```
            boolean prost = true;
            for (int j = 0; j < brojPronadjenihProstih; j++) {
                if (i % prostiBrojevi.get(j) == 0) {
                    prost = false;
                    break;
                }
            }
            if (prost) {
                brojPronadjenihProstih++;
                prostiBrojevi.add(i);
                System.out.println("Pronadjen prost broj: " + i);
            }
        }
    }

    public static void main(String[] args) {
        ProstiBrojevi pretrazivac = new ProstiBrojevi();
        pretrazivac.start();
        try {
            Thread.sleep(DESET_SEKUNDI);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        pretrazivac.isteklo = true;
    }
}
```

Kada se dati program pokrene, dobija se izlaz kao u nastavku:

```
run:
Pronadjen prost broj: 2
Pronadjen prost broj: 3
Pronadjen prost broj: 5
Pronadjen prost broj: 7
Pronadjen prost broj: 11
Pronadjen prost broj: 13
Pronadjen prost broj: 17
…
Pronadjen prost broj: 351811
Pronadjen prost broj: 351829
Pronadjen prost broj: 351847
BUILD SUCCESSFUL (total time: 10 seconds)
```

Zadatak 2: Napraviti sopstvenu implementaciju klase Semaphore, i prikazati na praktičnom primeru kako radi.

Rešenje: U poglavlju 2.10 prikazano je korišćenje gotove klase Semaphore iz java.util.concurrent paketa. U ovom zadatku cilj je napraviti implementaciju naše klase Semaphore koja ima istu funkcionalnost. Traženi kod dat je u nastavku:

```
public class Semaphore {
    // broj dozvola za pristup objektu
    private int s;
    // konstruktor, inicijalizacija semafora
    public Semaphore(int val) {
        s = val;
    }
     // ovaj nacin za inicijalizaciju semaphora je zastareo (eng.
depricated)
    public synchronized void initS(int i){
        s=i;
    }
     // metoda wait, mora waitS jer postoji u javi Object.wait()
metoda
    public synchronized void waitS() {
         // dokle god je vrednost semafora <=0 (dokle god nema do-
zvola)
        while (s <= 0) {
            try {
                // niti cekaju
                wait();
            } catch (InterruptedException e) {
            }
        }
         // kada nit prodje, onda smanji s za jedan, sto znaci da
je uzela dozvolu
        // smanjila je vrednost semafora
        s = s - 1;
    }
    // metoda za napustanja semafora
    public synchronized void signalS() {
        // povecava vrednost semafora za 1 (broj dozvola za 1)
        s = s + 1;
        // obavestava druge niti da se probude koje cekaju nad me-
todom waitS()
        notify();
    }
}
```

Klasu našeg semafora nazvali smo Semaphore (kao i u Javi). Prilikom implementacije semafore potrebno je definisati dve operacije wait(s) i signal(s). Pošto semafor ugrađen u programskom jeziku Java koristi metode acquire() i release(), mi smo koristili tradicionalne nazive – waitS() i signalS().

Semaforska metoda wait() smanjuje (dekrementira) vrednost semafora (s=s-1) pod uslovom da vrednost semafora s zadovoljava uslov da je s>0. Pozivajući proces se blokira dok se taj uslov ne ispuni. Da bi se ovo implementiralo u Javi, potrebno je internu semaforsku promenljivu s postaviti kao polje klase Semaphore. Obzirom da metoda wait() već postoji u Javi u osnovnoj klasi Object, kako ne bi došlo da zabune koja se metoda wait() poziva, našu metodu wait smo nazvali waitS().

Kako bi se obezbedilo atomsko izvršavanje, metoda waitS() je deklarisana kao synchronized (sinhronizovana metoda), jer je ideja da se smanjivanje i povećavanje vrednosti semaforske promenljive izvršavaju kao atomske (nedeljive) instrukcije. Sinhronizacija označava da kada jedna nit prisupi instanci klase nad kojom se sinhronizacija obavlja (sinhronizacija na objektu, ili sinhronizovana metoda), ta nit zauzima katanac, odnosno uzima ključ nad datim objektom. Uzimanje ključa omogućava datoj niti ekskluzivno pravo pristupa objektu, a onemogućava bilo kojoj drugoj niti da pristupi datoj instanci sve dok data nit koja ima ekskluzivno pravo pristupa, to pravo ne napusti, oslobodi katanac i vrati ključ koji drži nad datom instancom objekta.

Potrebno je primetiti da se ovo ograničenje odnosi samo na druge niti koje pokušavaju da pristupe datom objektu koristeći synchronized pristup. Ukoliko nit ne traži ekskluzivno pravo pristupa objektu, onda ona može da pristupi tom objektu čak i kada neko drži ekskluzivno pravo pristupa. Posledice ovoga zavise od toga da li ove niti pristupaju istim delovima. Napuštanje ekskluzivnog prava pristupa objektu se postiže na dva načina:

- Završetkom synchronized bloka i
- Blokiranjem na metodi wait() klase Object.

Kako je kod semafora potrebno omogućiti da se nit blokira sve dok uslov ne postane ispunjen, u metodi waitS() je moguće postaviti petlju sa suprotnim uslovom na kome se nit blokira sve dok uslov ne postane ispunjen.

Treba obratiti pažnju da metoda wait() klase Object može da generiše izuzetak tipa InterruptedException kada neko pozove metodu interrupt() nad datom niti (na primer kada neko uspava nit – Thread.sleep(x)), pa je potrebno uhvatiti i obraditi ovaj izuzetak. U ovom konkretnom slučaju, obrada izuzetka se svodi na ponovnu proveru uslova da li je s<=0.

Semaforska metoda signal() inkrementira (povećava za 1) vrednost semafora (s=s+1) pod uslovom da nema blokiranih procesa na tom semaforu. Ako ima blokiranih procesa,onda se jedan od njih deblokira. U skladu sa preimenovanjem prethodne metode, i ova metoda signal() je preimenovana u signalS().

Kako bi se omogućilo atomsko izvršavanje, metoda waitS() je deklarisana kao synchronized. Obzirom da klasa Object ne pruža informaciju o broju blokiranih niti, a prilikom blokiranja u metodi waitS() nije uvećana promenljiva kojom bi se brojale blokirane niti, ovde je realizovano da se uvek poziva metoda notify(). Metoda notify() jednu od blokiranih niti iz liste niti blokiranih na datom objektu prebacuje u ulazni red, gde sa drugim nitima čeka ekskluzivno pravo pristupa objektu i proveri svog uslova. Treba primetiti da ako je postojala blokirana nit koja je čekala i metodom notify() se data nit prebaci u ulazni red, to ne znači da će ta nit prva naredna dobiti ekskluzivno pravo pristupa objektu.

U Java programskom jeziku, redosled po kome se niti puštaju kroz ulazni red nije garantovan. Inicijalizacija interne semaforske promenljive se može obaviti na dva načina. Prvi način je korišćenjem posebne sinhronizovane metode initS(), a drugi način je u samom konstruktoru date klase. Prvi način je zastareo (engl. *depricated*). U slučaju inicijalizacije pomoću konstruktora, nije neophodno da konstruktor bude sinhronizvan, jer ni jedna druga nit neće da pristup instanci klase, dok se konstruktor ne izvrši.

Primer upotrebe napisane klase Semaphore dat je u nastavku. MyThread je klasa niti koja će da se izvršava, dok je MySemaphore glavna klasa sa main() metodom.

```
public class MyThread extends Thread {
    private int threadID;
    // nas semafore koji smo napravili
    Semaphore sem = null;
    // broj iteracija petlje
    private static int countToThis;
    // zajednicka promenljiva
    static  volatile int counter = 0;

    // konstruktoru prosledjujemo id niti, objekat nase klase se-
mafore i broj iteracija petlje
    public MyThread(int threadID, Semaphore sem, int countToThis) {
        this.threadID = threadID;
        this.sem = sem;
        MyThread.countToThis = countToThis;
    }

    public void run() {
        for (int i = 0; i < countToThis; i++) {
            // nit pokusava da dobije dozvolu za pristup
            // ako nema dozvola, onda ceka
            sem.waitS();
            counter++;
           System.out.println("Nit " + threadID + " je u kriticnoj
sekciji");
```

```
            // kada prodje kriticnu sekciju, oslobadja jednu dozvolu
              sem.signalS();
        }
    }
}
```

```
public class MySemaphore {

    public static void main(String[] args) {
        // broj niti
        int N = 800;
        // counter for petlje u klasi MyThread
        int countToThis = 30;
        // kreiramo objekat semafore
       // dajemo vrednost na 1, sto znaci da samo jedna nit u istom
trenutku moze da udje u kriticnu sekciju
        Semaphore sem = new Semaphore(1);

        // niz objekata myThread
        MyThread[] threads = new MyThread[N];

        for (int i = 0; i < threads.length; i++) {

            threads[i] = new MyThread(i, sem, countToThis);
            threads[i].start();
        }
        for (int i = 0; i < threads.length; i++) {
            try {
                threads[i].join();
            } catch (Exception e) {

            }
        }
         System.out.println("Vrednost counter-a je: " + MyThread.
counter);
    }
}
```

Zadatak 3: Rešiti problem čitača i pisača pomoću semafora u programskom jeziku Java.

Rešenje: Problem čitača i pisača je čest problem konkurentnog programiranja. Osnova problema je da postoje deljeni podaci, na primer baza podataka. Nad bazom podataka su moguće operacije čitanja i pisanja. Više čitača može da radi istovremeno, jer oni samo čitanju podatke. Međutim, pisač menja podatke i zato dva pisača ne smeju da rade istovremeno. Takođe, dok pisači pišu, čitači ne smeju da čitaju. Dakle, moguće je da u jednom vremenskom trenutku radi jedan pisač, ili bilo koji broj čitača.

Problem pisača i čitača ima početne uslove, koji kasnije mogu da se modifikuju. Pocetni uslovi ovog problema su:

- postoje zajedničke (deljene) promenljive (deljene promenljive mogu biti baza podataka, a mogu biti i obične promenljive).
- Čitači su bezazleni (ništa ne kvare) – mogu da rade paralelno, ne smeta im ako ih neko prekine na pola (bilo ko, bilo kada i u bilo kom broju).
- Pisači su opasni i prema čitacima i prema drugim pisačima (traže ekskluzivni pristup), ne smeju da budu prekinuti, ne trpe ni čitače, niti druge pisače (kada jedan piše, niko drugi ne sme da bude aktivan).

Najjednostavniji model problema čitača i pisača daje neograničenu prednost čitacima, tako da je nepravedan (pomeren u stranu, nije balansiran). Na primer, dok pisač piše, niko ne može da uđe, a kada uđe čitač, mogu da ulaze i drugi čitaci, ali ne mogu da ulaze pisači. Na ovaj način, ako čitača ima jako mnogo, čitači su jako favorizovani u odnosu na pisaše (jedan čitač uđe i drži otvorena vrata za druge čitače, a za to vreme pisači ne mogu da ulaze). Ovo, najjednostavnije rešenje se zasniva na dva binarna semafora. U literaturi postoji generičko ime mutex za semafor koji obezbeđuje uzajamnu isključivost (engl. *mutual exclusion*). Ovi semafori se obično zovu mutex1, mutex2, mutex3 itd., što odgovara indeksima u klasičnim programima (indekse u klasičnim programima obično nazivamo i, j, k, itd.).

Jedan semafor će se koristiti kako bi moglo da se piše po bazi – neka se zove mozePise. Ključni su prvi i poslednji čitač u jednoj partiji (dok rade zajedno). Kada čitač naiđe i nema nikoga, on je poseban, jer je prvi čitač, i kao prvi čitač, on mora da zabrani ulaz pisačima. Kada prvi čitač uđe u kritičnu sekciju, on zatvara vrata za pisače. Svaki čitač mora da proveri da li je on započeo to stanje (stanje da je zabranjen ulaz pisačima) – pre njega može da se piše, posle njega više ne može. Nakon toga, novi čitači mogu da dolaze u broju u kom hoće. Interesantana situacija je kada poslednji čitač izađe, jer nakon njega mogu da ulaze pisači. Čitači moraju da vode računa o svom broju (koliko ih ima u kritičnoj sekciji). Kritične situacije su kada prvi čitač dođe i kada poslednji čitač ode.

Čitači moraju da vode računa (evidenciju) o tome koliko ih pristupa fajlu, ili bazi podataka (o svom broju). To mora da ide preko zaštićene promenljive, kako ne bi došlo do problema x++. Čitač mora prvo da se najavi i to mora da uradi na siguran

način (wait (mutex)). Čitač takođe mora da proveri da li je on prvi čitač. Kada prvi čitač dolazi, on ne zna da li je pisač aktivan i zato prvi čitač mora da se najavi. Nakon toga čitač pita da li je on prvi čitač, i ako jeste onda reši da blokira semafor mozePise, kako ne bi mogli da ulaze pisači. Kada prvi čitač dođe, on može da „naleti“ na zatvorena vrata, jer je pisač u kritičnoj sekciji i pisač je postavio wait (mozePise).

U toj situaciji čitač mora da čeka da pisač pusti signal (mozePise). Međutim, ova situacija može da bude opasna, jer čitač čeka na signal (mozePise), a nije pustio signal (mutex), jer čeka na signal (mozePise). Međutim, u ovom slučaju situacija da čitač nije pustio signal (mutex), a čeka na signal (mozePise) nije opasna. To znači da sledeći čitači neće moći da pristupe brojuCitaca, ali to nije značajno, jer oni u svakom slucaju ne mogu da čitaju, jer im to ne dozvoljava pisač, tako da je svejedno gde čekaju u redu (da li u redu na promenjivu brojCitaca, ili u redu na promenljivu koju treba da čitaju). Prvog čitača će da pusti pisač naredbom signal (mozePise). Čim pisač pusti prvog čitača, prvi čitač će da pusti ostale čitače. Na izlasku čitač mora da registruje da je otišao (da smanji broj čitača). Ako je čitač poslednji od čitača, on nakon izlaska oslobađa i promenjivu brojCitaca i signal(mozePise) i pušta pisače da uđu.

Implementacija čitača i pisača je data u nastavku.

```
public interface ReadersWriters {

    public void startRead();
    public void endRead();
    public void startWrite();
    public void endWrite();

}

import java.util.concurrent.Semaphore;
// KLASA SEMAFOR
public class ReadersWritersSem implements ReadersWriters {

    private Semaphore rw; // stiti pisanje
    private Semaphore mutexR; // stiti promenljivu brojCitaca
    private volatile int readCount; // promenljiva brojCitaca

    public ReadersWritersSem() {
        rw = new Semaphore(1);
        mutexR = new Semaphore(1);
        readCount = 0;
    }
```

```
    public void startRead() {
        mutexR.acquireUninterruptibly();
        readCount++;
        if (readCount == 1) {
            rw.acquireUninterruptibly();
        }
        mutexR.release();
    }
    public void endRead() {
        mutexR.acquireUninterruptibly();
        readCount--;
        mutexR.release();
        if (readCount == 0) {
            rw.release();
        }
    }
    public void startWrite() {
        rw.acquireUninterruptibly();
    }
    public void endWrite() {
        rw.release();
    }
}
```

```
// KLASA READER
public class Reader extends Thread {

    private ReadersWritersSem rws;
    private int threadID;

    public Reader(ReadersWritersSem rws, int threadID) {
        this.rws = rws;
        this.threadID = threadID;
    }

    public void run(){

        int random=(int)(Math.random()*10+1);

        rws.startRead();
         System.out.println(“Reader “ + threadID + “ je poceo ci-
tanje”);
```

```
        try{
            Thread.sleep(random);
        }
        catch(Exception e){

        }
        System.out.println("Reader " + threadID + " je zavrsio sa
citanjem");
        rws.endRead();


    }
}
```

```
// KLASA WRITER
public class Writer extends Thread {

    private ReadersWritersSem rws;
    private int threadID;
    public Writer(ReadersWritersSem rws, int threadID) {
        this.rws = rws;
        this.threadID = threadID;
    }

    public void run(){
        int random=(int)(Math.random()*10+1);
        rws.startWrite();
         System.out.println("Writer " + threadID + " je poceo sa
pisanjem");

        try{
            Thread.sleep(random);
        }
        catch(Exception e){

        }
        System.out.println("Writer " + threadID + " je zavrsio sa
pisanjem");
        rws.endWrite();

    }

}
```

```
// POKRETACKA KLASA START
public class Start {

   private static final int N = 10;
    public static void main(String[] args) {
        ReadersWritersSem rws = new ReadersWritersSem();
        Reader[] readersThread = new Reader[N];
        Writer[] writersThread = new Writer[N];
        for (int i = 0; i < N; i++) {
            readersThread[i] = new Reader(rws, i);
            writersThread[i]=new Writer(rws,i);
        }

        for(int i=0;i<N;i++){
            readersThread[i].start();
            writersThread[i].start();
        }

        for(int i=0;i<N;i++){
            try{
          readersThread[i].join();
           writersThread[i].join();
            }
            catch(Exception ex){

            }
        }
    }
}
```

Problem sa ovim modelom je taj da se daje neograničena prednost čitačima, jer dokle god ima čitača, u kritičnu sekciju ne može da uđe ni jedan pisač. Dokle god je dovoljno gust protok čitača, pisači ne mogu da uđu. Kakvi bi bili mehanizmi prerade ovog modela? Trebalo bi da se namesti da pisač može da se najavi, da i on stane u red na neki način (kako sledeći čitači ne bi imali prednost nad pisačem). Ovo ne garantuje pisaču ni momentalan, niti brzi ulazak, ali garantuje da ne mogu da ulaze pre njega čitači koji su došli posle njega.

U najjednostavnijem modelu, gde se daje neograničena prednost čitačima, čitači mogu da ulaze i izlaze kada god hoće, što nije pravedno. Da bi pisač mogao da radi, svi čitači moraju da završe, a na taj način pisači možda nikada neće moći da rade. Zbog navedenog može da se primeni sledeći mehanizam: ako neki pisač hoće da uđe, mi puštamo da čitači završe, ali ne dazvoljavamo novim čitačima da ulaze. Potrebno je uvesti novu promenljivu pisacCeka koja nam govori da li neki pisač čeka da obavi pisanje.

Međutim, ni drugo rešenje nije baš najbolje. Nije problem što je promenljiva pisacCeka obična promenljiva. Drugo rešenje ne funkcioniše, jer ako prođemo jednom kroz algoritam sve radi, ali već kod drugog prolaska se javlja problem, odnosno drugo rešenje radi samo za jednog pisača. Kada prvi pisač ode, on kaže da više ne čeka i ostali pisači ostaju zaglavljeni u redu. Promenljiva pisacCeka zato ne može da bude binarna promenljiva, po principu ko čeka, a ko ne čeka, nego bi morala da bude brojačka promenljiva, kako bi se znalo koliko pisača čeka.

Potrebno je napraviti neko brojanje, da ne bi bilo pisacCeka = 1; nego pisacCeka++; i ne bi bilo pisacCeka = 0; nego pisacCeka--; Ta brojačka promenljiva mora biti ograđena semaforima. Uvodi se mutex2 koja štiti promenljivu pisacCeka. Sada je rešenje malo izbalansiranije, jer sada i pisači mogu da drže otvorenu kapiju za druge pisače. Ovakvo rešenje će da bude pravednije nego prethodno. U ovakvom rešenju imamo tzv. ortačku inercionu prednost – ko uđe, taj ortacima drži vrata otvorena da mogu da ulaze koliko god hoće. Ovaj mehanizam sa semaforima je jako dobar jer nas oslobađa razmišljanja o implementaciji. Za kontrolisanje pristupa zajedničkim promenljivama mi sada koristimo semafore i možemo da se fokusiramo na rešavanje praktičnog problema.

U nastavku je dat Java kod za ovo rešenje, gde se daje balansirana prednost i pisačima i čitačima.

```
// INTERFEJS
public interface ReadersWriters {

    public void startRead();
    public void endRead();
    public void startWrite();
    public void endWrite();

}
```

```
import java.util.concurrent.Semaphore;

// KLASA SEMAFOR
public class ReadersWritersSem implements ReadersWriters {
    private Semaphore rw; // stiti pisanje
    private Semaphore mutexR; // stiti promenljivu brojCitaca
    private Semaphore mutexW; // stiti promenljivu writersWaiting
    private volatile int readCount; // promenljiva brojCitaca
    private volatile int  writersWaiting; // promenljiva koja kaze
da li pisac ceka
    public ReadersWritersSem() {
        rw = new Semaphore(1);
```

```
        mutexR = new Semaphore(1);
        mutexW=new Semaphore(1);
        readCount = 0;
        writersWaiting=0;
    }
    public void startRead() {
        while(writersWaiting>0);
        mutexR.acquireUninterruptibly();
        readCount++;
        if (readCount == 1) {
            rw.acquireUninterruptibly();
        }
        mutexR.release();
    }
    public void endRead() {
        mutexR.acquireUninterruptibly();
        readCount--;
        mutexR.release();
        if (readCount == 0) {
            rw.release();
        }
    }
    public void startWrite() {
        mutexW.acquireUninterruptibly();
        writersWaiting++;
        mutexW.release();
        rw.acquireUninterruptibly();
    }
    public void endWrite() {
        rw.release();
        mutexW.acquireUninterruptibly();
        writersWaiting--;
        mutexW.release();
    }
}
```

```
import java.util.concurrent.Semaphore;
// KLASA READER
public class Reader extends Thread {
    private ReadersWritersSem rws;
    private int threadID;
    public Reader(ReadersWritersSem rws, int threadID) {
        this.rws = rws;
        this.threadID = threadID;
    }

    public void run(){
        int random=(int)(Math.random()*10+1);
        rws.startRead();
         System.out.println(“Reader “ + threadID + “ je poceo ci-
tanje”);
        try{
            Thread.sleep(random);
        }
        catch(Exception e){

        }
         System.out.println(“Reader “ + threadID + “ je zavrsio sa
citanjem”);
        rws.endRead();
    }
}
```

```
import java.util.concurrent.Semaphore;
// KLASA WRITER
public class Writer extends Thread {

    private ReadersWritersSem rws;
    private int threadID;

    public Writer(ReadersWritersSem rws, int threadID) {
        this.rws = rws;
        this.threadID = threadID;
    }

    public void run(){
        int random=(int)(Math.random()*10+1);
        rws.startWrite();
```

```
         System.out.println("Writer " + threadID + " je poceo sa
pisanjem");

        try{
            Thread.sleep(random);
        }
        catch(Exception e){

        }
         System.out.println("Writer " + threadID + " je zavrsio sa
pisanjem");
        rws.endWrite();
    }
}
```

```
import java.util.concurrent.Semaphore;
// KLASA START, POKRETACKA KLASA
public class Start {

   private static final int N = 10;

    public static void main(String[] args) {

        ReadersWritersSem rws = new ReadersWritersSem();

        Reader[] readersThread = new Reader[N];
        Writer[] writersThread = new Writer[N];

        for (int i = 0; i < N; i++) {

            readersThread[i] = new Reader(rws, i);
            writersThread[i]=new Writer(rws,i);
        }

        for(int i=0;i<N;i++){
            readersThread[i].start();
            writersThread[i].start();
        }

        for(int i=0;i<N;i++){
            try{
          readersThread[i].join();
```

```
            writersThread[i].join();
             }
             catch(Exception ex){

             }
        }

    }
}
```

Zadatak 4: Napisati program koji množi dve matrice (dimenzije nxn). Svako polje u rezultujućoj matrici izračunava jedna nit.

Rešenje:

```
class Matrix {
    public static int[][] A, B, C;
    public int n;

    public void stampajMatricu(int[][] M) {
            for(int i = 0; i < n; i++) {
                for(int j = 0; j < n; j++)
                        System.out.print("\t"+M[i][j]);
                System.out.println();
            }
    }

    public Matrix(int n) {
            this.n = n;
            A = new int[n][n];
            B = new int[n][n];
            C = new int[n][n];
            Random r = new Random();
            //popunjavanje matrice random brojevima do 10
            for(int i = 0; i < n; i++)
                for(int j = 0; j < n; j++) {
                        A[i][j] = r.nextInt() % 10; // %100 da bi
elementi matrice bili manji od 10
                        B[i][j] = r.nextInt() % 10;
                }
    }
```

```
    public void pomnozi() {
            ThreadMatrix[][] mt = new ThreadMatrix[n][n];

            for(int i = 0; i < n; i++)
                for(int j = 0; j < n; j++) {
                        mt[i][j] = new ThreadMatrix(3, i, j);
                        mt[i][j].start();
                }
            //ceka da zavrse sve niti
            try {
            for(int i = 0; i < n; i++)
                for(int j = 0; j < n; j++)
                        mt[i][j].join();
            } catch(InterruptedException ie) {}

            System.out.println("\nA =");
            stampajMatricu(A);
            System.out.println("\nB =");
            stampajMatricu(B);
            System.out.println("\nC =");
            stampajMatricu(C);
    }
}
```

Metoda pomnoži kreira objekte naše klase ThreadMatrix, za svaki element rezultujuće matrice, pokreće se nit, koja računa jedan element pomnoženih matrica.

```
class ThreadMatrix extends Thread {
    private int n;  // nxn matrica
    private int i, j; // red i kolona elementa koji racuna nit (ob-
jekat ove klase)
    public ThreadMatrix(int size, int ii, int jj) {
            n = size;
            i = ii;
            j = jj;
    }

    public void run() {
            Matrix.C[i][j] = 0;
            for(int x = 0; x < n; x++)
                Matrix.C[i][j] += Matrix.A[i][x] * Matrix.B[x][j];
    }
}
```

```
public class MnozenjeMatricaGlavna {
    public static void main(String[] args) {
        Matrix m = new Matrix(3);
        m.pomnozi();
    }
}
```

Zadatak 5: Garaža ima broj slobodnih mesta kao atribut. Svaki automobil predstavlja jednu nit. Simulirati rad garaže (ulazak/parkiranje/izlazak iz graže, paziti da li ima gde da se parkira). Automobili su niti koji mogu da uđu ili izađu iz garaže i ovi metodi menjaju broj slobodnih mesta. Broj slobodnih mesta ne sme biti manji od nula.

Garažu mogu istovremeno koristi više automobila, odnosno više niti pristupa zajedničkoj promenljivoj brSlobodnihMesta, pa je potrebno obezbediti ispravan rad.

Rešenje: Metode moraju biti sinhronizovane, kako bi se obezbedilo da svaka nit završi sve operacije metod. Operacije za *brSlobodnihMesta--;* izvršava se u tri koraka: pročita se vrednost promenljive brSlobodnihMesta, umanji se za jedan, upiše se izmenjena vrednost. Sve operacije moraju da se izvrše bez da druga nit preuzme rad. Ukoliko metoda ne bi bila *synhronized*, moglo bi da se desi da jedna nit pročita, smanji vrednost, pa druga nit krene sa radom, pročita (staru vrednost), prva upiše umanjenu, druga nit umanji i upiše. Rezultat je umanjenje za jedan umesto za dva.

```
class Garaza{
  private int brSlobodnihMesta;
  public Garaza(int brSlobodnihMesta) {
    //provera ispravnosti argumenta, mogu se koristiti i izuzeci
    if (brSlobodnihMesta < 0)
      brSlobodnihMesta = 0;
    else
this.brSlobodnihMesta = brSlobodnihMesta;
  }
  public synchronized void udji(String imeAuta) {
      //ukoliko nema slobodnih mesta, auto ne moze da se parkira,
pa ceka (wait()) dok se ne oslobodi mesto)
    while (brSlobodnihMesta == 0) {
      try {
        System.out.println(imeAuta+" ceka da se oslobodi mesto");
        wait();
      } catch (InterruptedException e) {}
    }
```

```
    //kad doceka, smanjuje se broj mesta
    brSlobodnihMesta--;
    System.out.println(imeAuta+" se parkirao.");
  }
  public synchronized void izadji(String imeAuta) {
    brSlobodnihMesta++;
    System.out.println(imeAuta+": je izasao");
    //kad neko izadje, oslobadja se mesto, pa se salje obavestenje
nitima koje cekaju na mesto
    notify();
  }
}
```

Pošto je svaki auto nit naslediće klasu Thread (ili implementirati Runnable interface). Metoda run() simulira jedan auto koji ulazi u garažu, malo stoji parkiran u garaži i zatim izlazi.

```
class Auto extends Thread {
  private Garaza parkingGarage;
  public Auto(String name, Garaza p) {
      super(name); //ovo je poziv konstruktora Thread, kojim se
imenuje nit
    this.parkingGarage = p;
     //cim kreiramo auto, zelimo da on zapocne sa svojim ciklusom
koji je sledeci traziMesto/udji/budi parkiran/izadji
    start();
  }
  //ciklus auta
  public void run() {
    //while (true) { //ako se stavi ovaj while, svaki kreirani auto
ce da u krug ponavlja ciklus traziMesto/udji/budi parkiran/izadji,
bez while, svaki auto samo jednom koristi garazu
      try {
         sleep((int)(Math.random() * 10000)); // simuliranje tra-
zenja mesta
      } catch (InterruptedException e) {
          System.out.println(e.getMessage());
      }

      //parkiranje - ulazak u garazu
      parkingGarage.udji(getName());
```

```
      try {
          sleep((int)(Math.random() * 20000)); //parkiran je neko
vreme
      } catch (InterruptedException e) {
          System.out.println(e.getMessage());
      }

      //izlaz iz garaze
      parkingGarage.izadji(getName());
    }
  //}
}

//Glavna klasa
public class NitiGaraza {
    public static void main(String[] args) {
        Garaza g = new Garaza(3);
        for (int i=1; i<= 10; i++) {
            Auto a = new Auto(“Auto “+i, g);
        }
    }
}
```

Primer izlaza ovog programa je:

```
Auto 5 se parkirao.
Auto 4 se parkirao.
Auto 1 se parkirao.
Auto 7 ceka da se oslobodi mesto
Auto 9 ceka da se oslobodi mesto
Auto 1: je izasao
Auto 7 se parkirao.
Auto 6 ceka da se oslobodi mesto
Auto 8 ceka da se oslobodi mesto
Auto 10 ceka da se oslobodi mesto
Auto 3 ceka da se oslobodi mesto
Auto 2 ceka da se oslobodi mesto
Auto 5: je izasao
Auto 9 se parkirao.
Auto 9: je izasao
Auto 6 se parkirao.
Auto 7: je izasao
Auto 8 se parkirao.
```

```
Auto 4: je izasao
Auto 10 se parkirao.
Auto 10: je izasao
Auto 3 se parkirao.
Auto 6: je izasao
Auto 2 se parkirao.
Auto 2: je izasao
Auto 8: je izasao
Auto 3: je izasao
```

Zadatak 6: Napisati Java program koji simulira rad auto perionice. Auto perionica ima određen broj boksova gde se mogu oprati automobili. Svaka auto koji se doveze u auto perionicu predstavlja jednu nit čiji je posao da uđe u boks, opere auto i izađe.

Klasa Perionica ima kao atribut broj slobodnih boksova. Potrebno je obezbediti da je moguće kreirati samo jedan objekat klase Perionica koristeći Singleton šablon. Metode uđiUboks() i izađiIzBoksa() su metode koje simuliraju ulazak i izlazak iz boksa (u metodama je potrebno odštampati odgovarajuće poruke: Ulazak u bok!!! ili Izlazak iz boksa!!!).

Implementirati apstraktnu klasu Vozilo koja kao atribute ima model, ime vlasnika i broj registracije. Broj registracije je string od 7 karaktera oblika SSCCCSS, gde je S slovo a C cifra. Ukoliko se pokuša postavljanje registracije na pogrešni format, izbaciti izuzetak. Potrebno je implementirati metod toString() i napraviti apstraktni metod urediVozilo(). Vozilo implementira Runnable interfejs gde metoda run() opisuje proces uređivanja jednog vozila (dolaženje u boks u perionici, uređivanje vozila, odlazak iz boksa).

Klase Auto i Kombi nasleđuju klasu Vozilo. Klase Auto i Kombi imaju kao atribut težinu, i to Vozilo je tipa Auto ukoliko je težina manja od 3.5t dok je tipa Kombi ukoliko je težina veća ili jednaka od 3.5t. Za kreiranje Vozila rešiti korišćenjem Factory design pattern-a.

U klasi Auto implementirati nadjačani metod toString(). Metod urediVozilo() simulira process uređivanja automobila po ulasku u boks. Unutar metode je potrebno odštampati na standardni izlaz poruku: "Auto + ime niti + se pere". Pranje auta traje 2000ms.

Slično klasi Auto i klasa Kombi ima nadjačani metod toString() i metod urediVozilo gde se ispisuje poruka na standardni izlaz : "Kombi + ime niti + se pere". Pranje kombija traje 3000ms.

Napisati main() metod koji simulira rad auto perionice. Na tri boksa se peru Vozila. Napraviti listu Vozila iz java.util.Collection i listi pristupati pomoću iteratora. U listu smestiti 10 vozila. Obezbediti ispravan rad niti, odnosno da kada jedno vozilo okupira boks, drugo vozilo ne može da prekida pranje prvog vozila.

Rešenje:

```
public class Perionica {
    private int brojSlobodnihBoksova;
    public static Perionica p;
    private Perionica(int brojSlobodnihBoksova) {
        this.brojSlobodnihBoksova = brojSlobodnihBoksova;
    }
   public static Perionica getInstance(int brojSlobodnihBoksova) {
        if (p == null) {
            Perionica p = new Perionica(brojSlobodnihBoksova);
            return p;
        } else {
            return p;
        }
    }
    public synchronized void udjiUBoks() {
        while (brojSlobodnihBoksova == 0) {
            try {
                wait();
            } catch (InterruptedException ex) {
                      Logger.getLogger(Perionica.class.getName()).
log(Level.SEVERE, null, ex);
            }
        }
       System.out.println(Thread.currentThread().getName()+” ulazi
u boks!!!”);
        brojSlobodnihBoksova--;
    }

public synchronized void izadjiIzBoksa() {
             System.out.println(Thread.currentThread().getName()+”
izlazi iz boksa!!!”);
        brojSlobodnihBoksova++;
        notify();
    }
}
```

```
public abstract class Vozilo implements Runnable {
    private String model;
    private String imeVlasnika;
    private String brojRegistracije;
    private Perionica p;
     public boolean proveriRegistraciju(String brojRegistracije)
throws Exception {
        boolean ispravno = true;
        if (brojRegistracije.length() != 7) {
            ispravno = false;
        } else {
            for (int i = 0; i < brojRegistracije.length(); i++) {
                if (i == 0 || i == 1 || i == 5 || i == 6) {
                     if (!Character.isAlphabetic(brojRegistracije.
charAt(i))) {
                         ispravno = false;
                     } else if (i == 2 || i == 3 || i == 4) {
                          if (!Character.isDigit(brojRegistracije.
charAt(i))) {
                             ispravno = false;
                         }
                     }
                }
            }
        }
        if (ispravno == false) {
            throw new Exception("Pogresan unos.");
        } else {
            return ispravno;
        }
    }
   public Vozilo(String model, String imeVlasnika, String brojReg-
istracije, Perionica p) throws Exception {
        this.model = model;
        this.imeVlasnika = imeVlasnika;
        if (proveriRegistraciju(brojRegistracije)) {
            this.brojRegistracije = brojRegistracije;
        }
        this.p = p;
    }
    abstract void urediVozilo();
    @Override
```

```
    public String toString() {
        return “Vozilo: “ + “model=” + model + “, imeVlasnika=” +
imeVlasnika + “, brojRegistracije=” + brojRegistracije;
    }
    @Override
    public void run() {
        p.udjiUBoks();
        urediVozilo();
        p.izadjiIzBoksa();
    }
}
```

```
public class Auto extends Vozilo {
    private int tezina;
    public Auto(String model, String imeVlasnika, String brojReg-
istracije, int tezina, Perionica p ) throws Exception {
        super(model, imeVlasnika, brojRegistracije, p);
        this.tezina=tezina;
    }
    @Override
    public synchronized void urediVozilo() {
         System.out.println(“Auto “ + Thread.currentThread().get-
Name() + “ se pere.”);
        try {
            Thread.sleep(2000);
        } catch (InterruptedException ex) {
            ex.getMessage();
        }
    }
    @Override
    public String toString() {
        return super.toString() + “ tezina: “ + tezina;
    }
}
```

```
public class Kombi extends Vozilo {
    private int tezina;
    public Kombi(String model, String imeVlasnika, String brojReg-
istracije, int tezina, Perionica p) throws Exception {
        super(model, imeVlasnika, brojRegistracije, p);
        this.tezina = tezina;
    }
```

```
    @Override
    public synchronized void urediVozilo() {
         System.out.println(“Kombi “ + Thread.currentThread().get-
Name() + “ se pere.”);
        try {
            Thread.sleep(3000);
        } catch (InterruptedException ex) {
            ex.getMessage();
        }
    }
    @Override
    public String toString() {
        return super.toString() + “ tezina: “ + tezina;
    }
}
```

```
public class VoziloFactory {
     public static Vozilo getVozilo(String model, String imeVlas-
nika, String brojRegistracije, int tezina, Perionica p) throws
Exception {
        if (tezina >= 3500)
            return new Kombi(model, imeVlasnika, brojRegistracije,
tezina, p);
        else
             return new Auto(model, imeVlasnika, brojRegistracije,
tezina, p);
    }
}
```

```
public class Glavna {
    public static void main(String[] args) throws Exception {
        Perionica p = Perionica.getInstance(3);
        List<Vozilo> vozila = new ArrayList<Vozilo>();
        String model = “NekiModel”;
        String imeVlasnika = “Marko”;
        String brojRegistracije = “AB123CD”;
        int tezina;
        for (int i = 0; i < 10; i++) {
            tezina = (int)(Math.random()*500+3250);
            vozila.add(VoziloFactory.getVozilo(model, imeVlasnika,
brojRegistracije, tezina, p));
        }
```

```
        Iterator<Vozilo> it = vozila.iterator();
        while (it.hasNext()) {
            Vozilo a=it.next();
            Thread t = new Thread(a);
            t.start();
        }
    }
}
```

Primer izlaza:

```
Thread-0 ulazi u boks!!!
Auto Thread-0 se pere.
Thread-1 ulazi u boks!!!
Kombi Thread-1 se pere.
Thread-2 ulazi u boks!!!
Kombi Thread-2 se pere.
Thread-0 izlazi iz boksa!!!
Thread-3 ulazi u boks!!!
Kombi Thread-3 se pere.
Thread-1 izlazi iz boksa!!!
Thread-4 ulazi u boks!!!
Auto Thread-4 se pere.
Thread-2 izlazi iz boksa!!!
Thread-5 ulazi u boks!!!
Auto Thread-5 se pere.
Thread-3 izlazi iz boksa!!!
Thread-6 ulazi u boks!!!
Kombi Thread-6 se pere.
Thread-4 izlazi iz boksa!!!
Thread-7 ulazi u boks!!!
Kombi Thread-7 se pere.
Thread-5 izlazi iz boksa!!!
Thread-8 ulazi u boks!!!
Auto Thread-8 se pere.
Thread-8 izlazi iz boksa!!!
Thread-9 ulazi u boks!!!
Auto Thread-9 se pere.
Thread-6 izlazi iz boksa!!!
Thread-7 izlazi iz boksa!!!
Thread-9 izlazi iz boksa!!!
```

Zadatak 7: Napisati Java program koji simulira rad šaltera u pošti. Pošta ima određen broj šaltera na kom se izvršavaju uplate i isplate. Svaka osoba koja dolazi u poštu predstavljena je kao jedna nit čiji je posao da dođe na šalter, obavi transakciju i ode sa šaltera.

Klasa Posta ima kao atribut broj slobodnih saltera. Potrebno je obezbediti da je moguće kreirati samo jedan objekat klase Posta koristeći Singleton šablon.

Implementirati apstraktnu klasu Osoba koja kao atribute ima ime, prezime i jmbg, metod toString() i apstraktni metod izvrsiTransakciju(double). Osoba implementira Runnable interfejs gde metoda run() opisuje proces jedne osobe na šalteru (dolazak na šalter, obavljanje transakcije i odlazak sa šaltera). Metode dodjiNaSalter() i zavrsiNaSalteru() se mogu implementirati ili u klasi Posta ili u klasi Osoba.

Klase Sponzor i Korisnik nasleđuju klasu Osoba. Klasa Sponzor ima kao atribut PIB (pib, tipa String) i nadjačani metod toString(). PIB bi trebalo da je string od 10 cifara, a u slučaju pogrešnog formata, izbaciti izuzetak. Metod transakcija simulira uplatu (odštampati na standardni izlaz poruku: "Sponzor + ime niti + vrsi uplatu"). Metod izvrsiTransakciju bi trebalo da traje neko vreme (koristiti metod sleep()).

Klasa Korisnik ima atribut broj računa koji je tipa String i mora biti oblika CCC-CCCCC-CC, gde je C cifra. U slučaju pogrešnog formata, izbaciti izuzetak. Metod transakcija simulira isplatu na račun korisnika tako što ispisuje poruku na standardni izlaz: "Korisnik +ime niti+podize pare".

Kreiranje Korisnika i Sponzora rešiti koristeći Factory design pattern (Korisnik ima atribut broj računa, dok Sponzor ima PIB koji su stringovi, ali različitog formata).

Napisati main metod koji simulira rad pošte. Na dva šaltera pošte se uslužuju Sponzori i Korisnici. Napraviti listu Osoba iz java.util.Collection i listi pristupati pomoću iteratora. U listu smestiti 5 osoba. Svaka Osoba predstavlja jednu nit. Obezbediti ispravan rad niti, odnosno da kad jedna nit okupira šalter, druga nit je ne može prekinuti dok ne završi posao na šalteru.

```
class Posta{
    private int brSaltera;
    //-------------singleton-----------------
    private static Posta p=null;
    private Posta(int brSaltera){
        this.brSaltera=brSaltera;
    }
    public static Posta getInstance(int brSaltera){
        if(p==null)
            p=new Posta(brSaltera);
        return p;
    }
```

```
    //--------------------------------------

    //ove metode se mogu staviti i u klasu osoba
    public synchronized void dodjiNaSalter(){
        //sve dok nema slobodnih mesta, cekaj
        while(brSaltera==0)
            try {
                wait();
            } catch (InterruptedException ex) {
                System.err.println(ex.getMessage());
            }
          System.out.println(Thread.currentThread().getName()+" je
dosao na salter!");
        brSaltera--;
    }

    public synchronized void zavrsiNaSalteru(){
        brSaltera++;
          System.out.println(Thread.currentThread().getName()+" je
otisao sa saltera!");
        //kad neko ode sa saltera, on je oslobodjen i neko ko ceka
moze da dodje
        notify();
    }

}

abstract class Osoba implements Runnable{
    private String ime;
    private String prezime;
    private String jmbg;
    private Posta posta;
    public Osoba(String ime, String prezime, String jmbg, Posta posta)
{
        this.ime = ime;
        this.prezime = prezime;
        this.jmbg = jmbg;
        this.posta = posta; //stavlja se posta da bi se obezbedila
sinhronizacija niti
    }
    abstract public void izvrsiTransakciju();
    @Override
    public String toString(){
        return ime+" "+prezime+", jmbg: "+jmbg;
    }
```

```
    //ciklus niti: dolazak na salter, transakcija i odlazak
    @Override
    public void run() {
      posta.dodjiNaSalter();
      izvrsiTransakciju();
      posta.zavrsiNaSalteru();
    }
}

class Sponzor extends Osoba{
    private String PIB;
    public Sponzor(String ime, String prezime, String jmbg, Posta
posta, String PIB) {
        super(ime, prezime, jmbg, posta);
        try {
            this.setPIB(PIB);
        } catch (Exception ex) {
            System.out.println(ex.getMessage());
        }
    }
    //pravimo set metod za PIB kako bi tu vrsili proveru formata
    public void setPIB(String PIB) throws Exception{
        if(proveriPIB(PIB))
            this.PIB = PIB;
        else
            throw new Exception("Pogresan format za PIB");
    }

   //metoda koja proverava ispravnost formata PIB koji je 10 cifara
    public boolean proveriPIB(String PIB){
        if(PIB.length()!=10)
            return false;
        for (int i = 0; i < 10; i++) {
            if(!Character.isDigit(PIB.charAt(i)))
                return false;
        }
        return true;
    }

    @Override
    public synchronized void izvrsiTransakciju() {
         //ukoliko bi ovaj metod menjao vrednost neke zajednicke
promenljiva, na primer stanje u posti, morao bi da bude synhronize,
ovako ne mora
```

```
      try {
        Thread.currentThread().sleep((int)(Math.random() * 1000));
//obavlja poslove na salteru, simulira trajanje radnje na salteru
      } catch (InterruptedException e) {
          System.out.println(e.getMessage());
      }
         System.out.println(Thread.currentThread().getName()+" iz-
vrsava uplatu!");
    }
    @Override
    public String toString(){
       return "Sponzor: "+super.toString()+"\n PIB sponzora: "+PIB;
    }
}

class Korisnik extends Osoba{
    private String brRacuna;
    public Korisnik(String ime, String prezime, String jmbg, Posta
posta, String brRacuna) {
        super(ime, prezime, jmbg, posta);
        try {
            this.setBrRacuna(brRacuna);
        } catch (Exception ex) {
            System.out.println(ex.getMessage());
        }
    }

    public void setBrRacuna(String brRacuna) throws Exception {
        if(proveriBrojRacuna(brRacuna))
            this.brRacuna = brRacuna;
        else
            throw new Exception("Pogresan format racuna!");
    }

    //oblika ccc-ccccc-cc, gde je c neka cifra
    public boolean proveriBrojRacuna(String brRacuna){
        if(brRacuna.length()!=12)
            return false;
        for (int i = 0; i < 12; i++) {
            if(i!=3 && i!=9){
                if(!Character.isDigit(brRacuna.charAt(i)))
                    return false;
            }
        }
```

```
        if(brRacuna.charAt(3)!='-' || brRacuna.charAt(9)!='-')
            return false;
        return true;
    }
    @Override
    public synchronized void izvrsiTransakciju() {
        try {
        Thread.currentThread().sleep((int)(Math.random() * 1000));
//obavlja poslove na salteru
      } catch (InterruptedException e) {
          System.out.println(e.getMessage());
      }
         System.out.println(Thread.currentThread().getName()+" po-
dize novac!");
    }
    @Override
    public String toString(){
       return "Korisnik: "+super.toString()+"\n broj racuna: "+br-
Racuna;
    }
}
class OsobaFactory{
     //na osnovu formata poslednjeg parametra moze da se zakljuci
da li je u pitanju klijent sa brojem racuna ili sponzor sa pib-om
    public Osoba getOsoba(String ime, String prezime, String jmbg,
Posta posta, String s){
        if(s.length()==10)
            return new Sponzor(ime,prezime,jmbg,posta,s);
        else if(s.length()==12)
            return new Korisnik(ime,prezime,jmbg,posta,s);
        return null;
    }
}

public static void main(String[] args) {
        Posta p=Posta.getInstance(2); //ne moze da se koristi new
Posta, posto su konstruktori private
        //posta ce biti kreirana samo ukoliko ne postoji, inace se
vraca postojeca pozivajuci metod getInstance(int brSaltera);
        OsobaFactory newOsoba=new OsobaFactory(); //pomocu Osoba-
Factory kreiramo odgovarajuce "tipove" Osobe
       List<Osoba> red=new ArrayList<>(); //lista za cuvanje Osoba
u posti
```

```
        Osoba o1=newOsoba.getOsoba("Eva","Tuba","23151232154"
,p,"1234567890");
        Osoba o2=newOsoba.getOsoba("Pera","Zikic","23151232154"
,p,"123-12345-12");
        Osoba o3=newOsoba.getOsoba("Mika","Peric","23151232154"
,p,"111-12345-11");
        Osoba o4=newOsoba.getOsoba("Zika","Mikic","23151232154"
,p,"0987654321");
        Osoba o5=newOsoba.getOsoba("Djura","Djuric","23151232154"
,p,"1231231230");

        red.add(o1);
        red.add(o2);
        red.add(o3);
        red.add(o4);
        red.add(o5);
        Iterator<Osoba> redIt=red.iterator(); //iterator za prola-
zenje kroz listu
        while(redIt.hasNext()){
            Osoba a=redIt.next();
            //kreira se i pokrece nit za svaku Osobu iz liste
            Thread t=new Thread(a);
            t.start();
        }
    }
```

Zadatak 8: Napisati Java program koji simulira rad studentske službe jednog fakulteta. Fakultet ima određen broj šalera koji su na raspolaganju studenima osnovnih, master i doktorskih studija. Svaki sudent koji dođe na šaler studentske službe predstavlja jednu nit koja okupira zaposlenog sve dok se ne reši njegov problem.

Klasa Fakultet ima kao atribut broj slobodnih šalera za rad sa studentima osnovnih studija i broj slobodnih šaltera za rad sa sudentima master i doktorskih studija. Potrebno je obezbediti da je moguće kreirati samo jedan objekat klase Fakultet koristeći Singleton šablon. Metode zatražiPomoćOsnovne(), završiOsnovne(), zatraži-PomoćNapredne(), završiNapredne (), su metode koje simuliraju dolazak i odlazak studenata sa šaltera studentske službe. U metodama je potrebno odštampati odgovarajuće poruke: *Student osnovnih/master ili doktorskih studija je zatražio pomoć!!!* i *Student osnovnih/master ili doktorskih studija je dobio pomoć od studentske službe!!!*. Obratiti pažnju da studenti osnovnih studija ne mogu da dobiju pomoć na šalterima namenjenim za studente master i doktorskih studija.

Implementirati apstraktnu klasu Student koja kao atribute ima ime, prezime i jmbg. Potrebno je implementirati metod toString() i napraviti apstraktni metod pomoc(). Klasa Student implementira Runnable interfejs gde metoda run() opisuje proces interakcije studenta i studentske službe. (dolazak na šalter, dobijanje pomoći i odlazak).

Klase StudentOsnovne i StudentNapredne nasleđuju klasu Student. Klasa StudentOsnovne ima kao atribut broj indeksa oblika 2 slova praćeno sa 6 cifara koje grade broj manji od 500000 i smer, dok klasa StudentNapredne ima kao atribut broj indeksa oblika 2 slova praćeno sa 6 cifara koje grade broj veći ili jednak 500000 i ime mentora. Kreiranje Studenta rešiti korišćenjem Factory design pattern-a.

U klasi StudentOsnovne implementirati nadjačani metod toString(). Metod pomoc() simulira interakciju studenta sa studentskom službom. Proces se sastoji od potvrđivanje identiteta studenta (štampanje podataka studenta i čekanje do 250ms da se podaci potvrde) i pomoci studentske službe koja traje maksimalno 1500ms.

Slično klasi StudentOsnovne i klasa StudentNapredne ima nadjačani metod toString() i metod pomoc(). Student master ili doktorskih studija po dolasku na šalter, predaje indeks da bi se utvrdio identitet. Na standardni izlaz se štampa broj indkesa i student master ili doktorskih studija (izabrati jedno sa verovatnoćom 0.5). Ova provera traje do 100ms. Razgovor sa sudentskom službom traje do 1000ms.

1. Napisati main() metod koji simulira rad studentske službe jednog fakulteta. Ukupno 3 šaltera je na raspolaganju studentima, od čega 1 za studente viših studijskih nivoa i 2 za studente osnovnih studija.
2. Napraviti listu Studenata iz java.util.Collection i listi pristupati pomoću iteratora.
3. U listu smestiti 5 studenata.

Obezbediti ispravan rad niti, odnosno da studenti budu usluženi na odgovarajućih šalterima i da ne može biti prekinut kad dođe na šalter.

Rešenje:

```
public class Fakultet {
    private int salteriOsnovne, salteriNapredne;
    private static Fakultet fakultet = null;
    private Fakultet(int osnovne, int napredne) {
        this.salteriOsnovne = osnovne;
        this.salteriNapredne = napredne;
    }

    public static Fakultet getInstance(int osnovne, int napredne) {
        if (fakultet == null) {
            fakultet = new Fakultet(osnovne, napredne);
        }
```

```
        return fakultet;
    }

    public synchronized void zatraziPomocOsnovne() {
        while (salteriOsnovne == 0) {
            try {
                wait();
            } catch (InterruptedException ex) {
                 System.error.println(“Greska prilikom izvrsavanja
niti”);
            }
        }
         System.out.println(“Student osnovnih studija je zatrazio
pomoc!!!”);
        salteriOsnovne--;
    }

    public synchronized void zavrsiOsnovne() {
         System.out.println(“Student osnovnih studija je dobio po-
moc!!!”);
        salteriOsnovne++;
        notify();
    }

    public synchronized void zatraziPomocNapredne() {
        while(salteriNapredne == 0) {
            try {
                wait();
            } catch (InterruptedException ex) {
              System.error.println(“Greska prilikom izvrsavanja niti”);
            }
        }

         System.out.println(“Student master/doktorskih studija za-
trazio je pomoc!!!”);
        salteriNapredne--;
    }

    public synchronized void zavrsiNapredne() {
          System.out.println(“Student master/doktorskih studija je
dobio pomoc”);
        salteriNapredne++;
        notify();
    }
}
```

```
public abstract class Student implements Runnable{
    private String ime, prezime, jmbg;
    private Fakultet faks;
   public Student(String ime, String prezime, String jmbg, Fakultet
f) {
        this.ime = ime;
        this.prezime = prezime;
        this.jmbg = jmbg;
        this.faks = f;
    }
    public abstract void pomoc();
    @Override
    public String toString() {
        return “Student “ + ime + “ prezime “ + prezime + “, JMBG:
“ + jmbg;
    }
    @Override
    public void run() {
        if(this instanceof StudentNapredne) {
            faks.zatraziPomocNapredne();
            this.pomoc();
            faks.zavrsiNapredne();
        } else {
            faks.zatraziPomocOsnovne();
            this.pomoc();
            faks.zavrsiOsnovne();
        }
    }
}
```

```
public class StudentOsnovne extends Student {

    private String brojIndeksa;
    private String smer;
    public StudentOsnovne(String ime, String prezime, String jmbg,
Fakultet f, String smer, String brojIndeksa) {
        super(ime, prezime, jmbg, f);
        this.smer = smer;
        try {
            setBrojIndeksa(brojIndeksa);
        } catch (Exception ex) {
                Logger.getLogger(StudentOsnovne.class.getName()).
log(Level.SEVERE, null, ex);
        }
    }
```

```
    public String getBrojIndeksa() {
        return brojIndeksa;
    }
    public void setBrojIndeksa(String brojIndeksa) throws Exception {
        if (brojIndeksa.length() != 8) {
              throw new Exception("Broj indeksa mora da ima tačno
osam karaktera.");
        }
        if (!Character.isLetter(brojIndeksa.charAt(0)) || !Charac-
ter.isLetter(brojIndeksa.charAt(1))) {
            throw new Exception("Prva dva karaktera indeksa moraju
da budu slova.");
        }
        for (int i = 2; i < 8; i++) {
            if (!Character.isDigit(brojIndeksa.charAt(i))) {
               throw new Exception("Poslednjih 6 karaktera indeksa
moraju da budu cifre.");
            }
        }
        this.brojIndeksa = brojIndeksa;
    }
    @Override
    public void pomoc() {
          System.out.println("Verifikacija studenta osnovih studi-
ja...");
        try {
            Thread.currentThread().sleep(250);
        } catch (InterruptedException ex) {
                Logger.getLogger(StudentOsnovne.class.getName()).
log(Level.SEVERE, null, ex);
        }
        System.out.println(this.toString() + " je verifikovan.");
        try {
              Thread.currentThread().sleep((int) (Math.random() *
1500));
        } catch (InterruptedException ex) {
                Logger.getLogger(StudentOsnovne.class.getName()).
log(Level.SEVERE, null, ex);
        }
    }
    @Override
    public String toString() {
        return super.toString() + ", osnovne studije br.Indeksa: "
+ getBrojIndeksa();
    }
}
```

```
public class StudentNapredne extends Student {
    private String brIndeksa;
    private String imeMentora;
   public StudentNapredne(String ime, String prezime, String jmbg,
Fakultet f, String smer, String brIndeksa) {
        super(ime, prezime, jmbg, f);
        this.imeMentora = imeMentora;
        try{
          setBrojIndeksa(brIndeksa);
        } catch (Exception ex) {
            System.err.println(ex.getMessage());
        }
    }
    public String getBrIndeksa() {
        return brIndeksa;
    }
     public void setBrojIndeksa(String brojIndeksa) throws Excep-
tion {
        if (brojIndeksa.length() != 8) {
              throw new Exception("Broj indeksa mora da ima tačno
osam karaktera.");
        }
        if (!Character.isLetter(brojIndeksa.charAt(0)) || !Charac-
ter.isLetter(brojIndeksa.charAt(1))) {
            throw new Exception("Prva dva karaktera indeksa moraju
da budu slova.");
        }
        for (int i = 2; i < 8; i++) {
            if (!Character.isDigit(brojIndeksa.charAt(i))) {
               throw new Exception("Poslednjih 6 karaktera indeksa
moraju da budu cifre.");
            }
        }
        this.brIndeksa = brojIndeksa;
    }
    @Override
    public void pomoc() {
         System.out.println("Verifikacija studenta naprednih studi-
ja...");
        try {
            Thread.currentThread().sleep(100);
        } catch (InterruptedException ex) {

Logger.getLogger(StudentNapredne.class.getName()).log(Level.SE-
VERE, null, ex);
```

```
        }
        System.out.println(this.toString() + " je verifikovan.");
        try {
              Thread.currentThread().sleep((int) (Math.random() *
1000));
        } catch (InterruptedException ex) {

Logger.getLogger(StudentNapredne.class.getName()).log(Level.SE-
VERE, null, ex);
        }
    }
    @Override
    public String toString() {
        return super.toString() + " napredne studije, brIndeksa: "
+ getBrIndeksa();
    }
}
```

```
public class StudentFactory {
    public static Student getInstance(String ime, String prezime,
String jmbg, String smer, String imeMentora, Fakultet t, String
indeks) {
        Integer broj = Integer.parseInt(indeks.substring(2, 7));
        if (broj < 50000) {
            return new StudentOsnovne(ime, prezime, jmbg, t, smer,
indeks);
        } else {
            return new StudentNapredne(ime, prezime, jmbg, t, ime-
Mentora, indeks);
        }
    }
}
```

```
public class Glavna {
    public static void main(String[] args) {
        Fakultet singidunum = Fakultet.getInstance(2, 1);

        List<Student> studenti = new ArrayList<>();
      Student s1 = StudentFactory.getInstance("Goran", "Milunovic",
"2761342463721","Marko Dragic","", singidunum,  "NS550000");
       Student s2 = StudentFactory.getInstance("Natasa", "Dermic",
"2761342463721", "Nenad Stokic","", singidunum, "NS5264512");
```

```
        Student s3 = StudentFactory.getInstance(“Nikola”, “Jokic”,
“2761342463721”,”Nenad Stokic”,””, singidunum, “NS5264512”);
        Student s4 = StudentFactory.getInstance(“Tamara”, “Nikic”,
“2761342463721”, “Nenad Stokic”,””, singidunum, “NS5264512”);
         Student s5 = StudentFactory.getInstance(“Ljuba”, “Simic”,
“2761342463721”, “Nenad Stokic”,””, singidunum, “NS5264512”);
        studenti.add(s1);
        studenti.add(s2);
        studenti.add(s3);
        studenti.add(s4);
        studenti.add(s5);
        Iterator<Student> iter = studenti.iterator();
        while(iter.hasNext()) {
            Student s = iter.next();
            Thread t = new Thread(s);
            t.start();
        }
    }
}
```

Zadatak 9: Napisati Java program koji simulira rad turističke agencije. Turistička agencija ima određen broj šaltera koji su na raspolaganju za rezervisanje pojedinačnih i grupnih aranžmana. Svaki turista koji dođe na šalter predstavlja jednu nit koja okupira zaposlenog sve dok se ne kompletira rezervacija.

Klasa TurističkaAgencija ima kao atribut broj slobodnih šaltera za rezervisanje pojedinačnih aranžmana i atribut koji predstavlja broj slobodnih šaltera za grupne ture. Potrebno je obezbediti da je moguće kreirati samo jedan objekat klase TurističkaAgencija koristeći Singleton šablon. Metode rezervisiPojedinacno(), razervisaoPojedinacno(), rezervisiGrupno(), rezervisaoGrupno(), su metode koje simuliraju dolazak i odlazak turista sa šaltera turističke agencije. U metodama je potrebno odštampati odgovarajuće poruke: *Turista/Predstavnik grupe je došao na red na šalter!!!* i *Turista/Predstavnik grupe je rezervisao aranžman!!!*. Obratiti pažnju da se pojedinačni aranžmani rezervišu samo na odgovarajućim šalterima (slično za grupne ture).

Implementirati apstraktnu klasu Osoba koja kao atribute ima ime, prezime i jmbg. Atribut jmbg je String koji se sastoji od 13 cifara. Ukoliko se pokuša postavljanje jmbg na pogrešni format, izbaciti izuzetak. Potrebno je implementirati metod toString() i napraviti apstraktni metod rezervacija(). Klasa Osoba implementira Runnable interfejs gde metoda run() opisuje proces rezervacije aranžmana. (dolazak na šalter, rezervisanje i odlazak).

Klase Turista i PredstavnikGrupe nasleđuju klasu Osoba. Klasa Turista ima kao atribut String koji predstavlja korisničko ime i sastoji se samo od slova, dok klasa

Predstavnik ima kao atribut String koji predstavlja šifru grupe i sastoji se samo od brojeva. Kreiranje Osobe rešiti korišćenjem Factory design pattern-a.

U klasi Turista implementirati nadjačani metod toString(). Metod rezervacija() simulira process rezervacija. Proces traje najviše 100ms, nakon rezervacije se ispisuje poruka oblika "*ImeOsobe* je zavrsio/la na salteru".

Slično klasi Turista i klasa PredstavnikGrupe ima nadjačani metod toString() i metod rezervacija() koja se sastoji od ispisivanja broja ljudi u grupi (random broj od 2 do 10) i čekanja da se rezervacija odobri što traje 180ms.

Napisati main() metod koji simulira rad turističke agencije. Ukupno 5 šaltera je na raspolaganju, od čega 3 za turiste i 2 za predstavnike grupa.

1. Napraviti listu Osoba iz java.util.Collection i listi pristupati pomoću iteratora.
2. U listu smestiti 10 osoba.
3. Obezbediti ispravan rad niti, odnosno da osobe budu uslužene na odgovarajućim šalterima i da ne mogu biti prekinute kad dođu na šalter.

Rešenje:

```
public class TuristickaAgencija {
    private int salteriPojedinac, salteriGrupe;
    private static TuristickaAgencija agencija = null;
    private TuristickaAgencija(int pojedinac, int grupe) {
        this.salteriPojedinac = pojedinac;
        this.salteriGrupe = grupe;
    }

   public static TuristickaAgencija getInstance(int pojedinac, int
grupe) {
        if (agencija == null) {
            agencija = new TuristickaAgencija(pojedinac, grupe);
        }
        return agencija;
    }
    public synchronized void rezervisiPojedinacno() {
        while (salteriPojedinac == 0) {
            try {
                wait();
            } catch (InterruptedException ex) {
                  System.err.println("Greska prilikom izvrsavanja
niti");
            }
        }
```

```
       System.out.println(“Turista je dosao na red na salter!!!”);
        salteriPojedinac--;
    }
    public synchronized void rezervisaoPojedinac() {
        System.out.println(“Turista je rezervisao aranzman!!!”);
        salteriPojedinac++;
        notify();
    }
    public synchronized void rezervisiGrupno() {
        while (salteriGrupe == 0) {
            try {
                wait();
            } catch (InterruptedException ex) {
                  System.err.println(“Greska prilikom izvrsavanja
niti”);
            }
        }
          System.out.println(“Predstavnik grupe je dosao na salt-
er!!!”);
        salteriGrupe--;
    }
    public synchronized void rezervisaoGrupno() {
        System.out.println(“Predstasvnik grupe je rezervisao aran-
zman!!!”);
        salteriGrupe++;
        notify();
    }
}
```

```
public abstract class Osoba implements Runnable{
    protected String ime, prezime, jmbg;
    protected TuristickaAgencija agencija;
     public Osoba(String ime, String prezime, String jmbg, Turis-
tickaAgencija agencija) {
        this.ime = ime;
        this.prezime = prezime;
        try {
            setJmbg(jmbg);
        } catch (Exception ex) {
            System.err.println(ex.getMessage());
        }
        this.agencija = agencija;
    }
```

```
    public void setJmbg(String jmbg) throws Exception {
        if (jmbg.length() != 13) {
            throw new Exception("JMBG mora da ima tačno 13 karak-
tera.");
        }
        for (int i = 0; i < 13; i++) {
            if (!Character.isDigit(jmbg.charAt(i))) {
                throw new Exception("Svi karakteri JMBG-a moraju
da su cifre.");
            }
        }
        this.jmbg = jmbg;
    }
    public abstract void rezervacija();
    @Override
    public String toString() {
        return "Osoba " + ime + " prezime " + prezime + ", JMBG:
" + jmbg;
    }
    @Override
    public void run() {
        if (this instanceof Turista) {
            agencija.rezervisiPojedinacno();
            this.rezervacija();
            agencija.rezervisaoPojedinac();
        } else {
            agencija.rezervisiGrupno();
            this.rezervacija();
            agencija.rezervisaoGrupno();
        }
    }
}
```

```
public class Turista extends Osoba {
    private String korIme;
    public Turista(String ime, String prezime, String jmbg, Turis-
tickaAgencija agencija, String korIme) {
        super(ime, prezime, jmbg, agencija);
        try {
            setKorIme(korIme);
        } catch (Exception ex) {
            System.err.println(ex.getMessage());
        }
    }
```

```
    public void setKorIme(String korIme) throws Exception {
        for (int i = 0; i < korIme.length(); i++) {
            if (!Character.isLetter(korIme.charAt(i))) {
                 throw new Exception(“Korisnicko ime turiste mora
sadrzati samo slova”);
            }
        }
        this.korIme = korIme;
    }
    @Override
    public void rezervacija() {
        try {
              Thread.currentThread().sleep((int) (Math.random() *
100));
        } catch (InterruptedException ex) {
            System.err.println(ex.getMessage());
        }
        System.out.println(ime+”je zavrsio/la na salteru”);
    }
}
```

```
public class PredstavnikGrupe extends Osoba{
    private String sifra;
      public PredstavnikGrupe(String ime, String prezime, String
jmbg, TuristickaAgencija agencija, String sifra) {
        super(ime, prezime, jmbg, agencija);
        try {
            setSifra(sifra);
        } catch (Exception ex) {
            System.err.println(ex.getMessage());
        }
    }
    public void setSifra(String sifra) throws Exception{
        for (int i = 0; i < sifra.length(); i++) {
            if(!Character.isDigit(sifra.charAt(i)))
                throw new Exception(“Sifra predstavnika grupe mora
sadrzati samo cifre”);
        }
        this.sifra = sifra;
    }
    public void rezervacija() {
        System.out.println(“Rezervacija za “+(int)(Math.ran-
dom()*9+2)+” osoba”);
```

```
        try {
            Thread.currentThread().sleep((int) (180));
        } catch (InterruptedException ex) {
            System.err.println(ex.getMessage());
        }
    }
}
public class OsobaFactory {
     public static Osoba getInstance(String ime, String prezime,
String jmbg, TuristickaAgencija a, String korIme) {
        if(Character.isLetter(korIme.charAt(0))) {
            return new Turista(ime, prezime, jmbg, a, korIme);
        } else {
            return new PredstavnikGrupe(ime, prezime, jmbg, a, ko-
rIme);
        }
    }
}
```

```
public class GlavnaTuristickaAgencija {
    public static void main(String[] args) {
        TuristickaAgencija singidunum = TuristickaAgencija.getIn-
stance(3,2);
        List<Osoba> osobe = new ArrayList<>();
        Osoba s1 = OsobaFactory.getInstance(“Pera”, “Peric”,
“2761342445721”, singidunum,  “ABC”);
        Osoba s2 = OsobaFactory.getInstance(“Vasa”, “Vasovic”,
“1261342463721”, singidunum, “123”);
        Osoba s3 = OsobaFactory.getInstance(“Nikola”, “Jokic”,
“2222342463721”, singidunum, “332”);
        Osoba s4 = OsobaFactory.getInstance(“Tamara”, “Nikic”,
“2765342463721”, singidunum, “CAB”);
        Osoba s5 = OsobaFactory.getInstance(“Ljuba”, “Simic”,
“2761342463725”, singidunum, “SING”);
        Osoba s6 = OsobaFactory.getInstance(“Mika”, “Mikovic”,
“2721342445721”, singidunum, “GCC”);
        Osoba s7 = OsobaFactory.getInstance(“Ana”, “Anic”,
“1261332463421”, singidunum, “7152”);
        Osoba s8 = OsobaFactory.getInstance(“Vera”, “Veric”,
“2222342463721”, singidunum, “2223”);
        Osoba s9 = OsobaFactory.getInstance(“Zuca”, “Zutic”,
“2765342333721”, singidunum, “GROUP”);
        Osoba s10 = OsobaFactory.getInstance(“Luka”, “Lukic”,
“2761342263725”, singidunum, “2341”);
```

```
        osobe.add(s1);
        osobe.add(s2);
        osobe.add(s3);
        osobe.add(s4);
        osobe.add(s5);
        osobe.add(s6);
        osobe.add(s7);
        osobe.add(s8);
        osobe.add(s9);
        osobe.add(s10);
        Iterator<Osoba> iter = osobe.iterator();
        while(iter.hasNext()) {
            Osoba o = iter.next();
            Thread t = new Thread(o);
            t.start();
        }
    }
}
```

Zadatak 10: Napisati Java program koji simulira rad graničnog prelaza. Granični prelaz ima određen broj rampi za putnika sa pasošima Evropske unije, dok ostale rampe služe za druge putnike. Svaki auto na granici predstavlja jednu nit čiji je posao da dođe na red, preda pasoše i odveze se dalje svojim putem.

Klasa Prelaz ima kao atribut broj slobodnih rampi za nosioce pasoša Evropske unije i broj slobodnih rampi za ostale pasoše. Potrebno je obezbediti da je moguće kreirati samo jedan objekat klase Prelaz koristeći Singleton šablon. Metode dodjiNaRampuEU() i produziDaljeEU() kao i odgovarajuće metode za nosioce pasoša zemalja koje nisu u EU: dodjiNaRampu() i produziDalje() su metode koje simuliraju dolazak i odlazak sa graničnog prelaza (u metodama je potrebno odštampati odgovarajuće poruke, npr. Nosioc EU pasoša došao na rampu…).

Implementirati apstraktnu klasu Putnik koja kao atribute ima ime, prezime i id. Podatak id je string od 8 cifara. Ukoliko se pokuša postavljanje id-a na pogrešni format, izbaciti izuzetak. Potrebno je implementirati metod toString() i napraviti apstraktni metod predjiGranicu(). Putnik implementira Runnable interfejs gde metoda run() opisuje proces prelaska jednog vozila preko granice.

Klase EU_putnik i Regularni_putnik nasleđuju klasu Putnik. Ove dve klase imaju kao atribut broj pasoša, i to Putnik je tipa EU_putnik ukoliko je početak broja pasoša 241 dok je tipa Regularni_putnik u suprotnom. Kreiranje Putnika rešiti korišćenjem Factory design pattern-a.

U klasi EU_putnik implementirati nadjačani metod toString(). Metod predjiGranicu() simulira proces prelaska granice. Unutar metode je potrebno odštampati na standardni izlaz poruku: "EU vozilo + ime niti + je na granici". Prelazak EU_putnika traje 2000ms.

Slično klasi EU_putnik i klasa Regularni_putnik ima nadjačani metod toString() i metod predjiGranicu() gde se ispisuje poruka na standardni izlaz : "Vozilo + ime niti + je na granici". Prelazak traje do 5000ms.

Napisati main() metod koji simulira rad graničnog prelaza. Na 2 rampe prelaze EU putnici, dok regularni putnici prelaze na 3 rampe. Napraviti listu Putnika iz java.util.Collection i listi pristupati pomoću iteratora. U listu smestiti 10 putnika. Obezbediti ispravan rad niti, odnosno da kada jedan putnik dođe na granicu, drugi putnik ne može da prekida njegov prelazak.

Rešenje:

```
public class Prelaz {
    private int rampaEU, rampaReg;
    private static Prelaz prelaz = null;
    private Prelaz(int rampaEU, int rampaReg) {
        this.rampaEU = rampaEU;
        this.rampaReg = rampaReg;
    }
    public static Prelaz getInstance(int rampaEU, int rampaReg) {
        if (prelaz == null) {
            prelaz = new Prelaz(rampaEU, rampaReg);
        }
        return prelaz;
    }
    public synchronized void dodjiNaRampuEU() {
        while (rampaEU == 0) {
            try {
                wait();
            } catch (InterruptedException ex) {
                  System.err.println("Greska prilikom izvrsavanja
niti");
            }
        }
        System.out.println("Nosilac EU pasoša došao na rampu!");
        rampaEU--;
    }
    public synchronized void produziDaljeEU() {
          System.out.println("Nosilac EU pasosa je presao grani-
cu!!!");
```

```
        rampaEU++;
        notify();
    }
    public synchronized void dodjiNaRampu() {
        while (rampaReg == 0) {
            try {
                wait();
            } catch (InterruptedException ex) {
                  System.err.println(“Greska prilikom izvrsavanja
niti”);
            }
        }
          System.out.println(“Nosilac regularnog pasosa dosao na
rampu!!!”);
        rampaReg--;
    }
    public synchronized void produziDalje() {
         System.out.println(“Nosilac regularnog pasosa je presao
granicu!!!”);
        rampaReg++;
        notify();
    }
}
```

```
public abstract class Putnik implements Runnable{
    protected String ime, prezime, id;
    protected Prelaz agencija;
     public Putnik(String ime, String prezime, String id, Prelaz
prelaz) {
        this.ime = ime;
        this.prezime = prezime;
        try {
            setId(id);
        } catch (Exception ex) {
            System.err.println(ex.getMessage());
        }
        this.agencija = prelaz;
    }
    public void setId(String id) throws Exception {
        if (id.length() != 8) {
          throw new Exception(“ID mora da ima tačno 8 karaktera.”);
        }
```

```
        this.id = id;
    }
    public abstract void predjiGranicu();
    @Override
    public String toString() {
         return “Osoba “ + ime + “ prezime “ + prezime + “, JMBG:
“ + id;
    }
    @Override
    public void run() {
        if (this instanceof EU_Putink) {
            agencija.dodjiNaRampuEU();
            this.predjiGranicu();
            agencija.produziDaljeEU();
        } else {
            agencija.dodjiNaRampu();
            this.predjiGranicu();
            agencija.produziDalje();
        }
    }
}
```

```
public class EU_Putink extends Putnik {
    private String brPasosa;
      public  EU_Putink(String  ime,  String  prezime,  String  jmbg,
Prelaz prelaz, String brPasosa) {
        super(ime, prezime, jmbg, prelaz);
        this.brPasosa=brPasosa;
    }
    @Override
    public void predjiGranicu() {
        try {
            Thread.currentThread().sleep((2000));
        } catch (InterruptedException ex) {
            System.err.println(ex.getMessage());
        }
         System.out.println(“EU vozilo “ + Thread.currentThread().
getName() + “ je na granici”);
    }
}
```

```
public class Regularni_putnik extends Putnik{
    private String brPasosa;
    public Regularni_putnik(String ime, String prezime, String id,
Prelaz p, String brPasosa) {
        super(ime, prezime, id, p);
        this.brPasosa = brPasosa;
    }
    public void predjiGranicu() {
          System.out.println("Vozilo "+Thread.currentThread().get-
Name()+" je na granici");
        try {
          Thread.currentThread().sleep((int)(Math.random()*5000));
        } catch (InterruptedException ex) {
            System.err.println(ex.getMessage());
        }
    }
}
```

```
public class PutnikFactory {
     public static Putnik getInstance(String ime, String prezime,
String id, Prelaz a, String brPasosa) {
        if(brPasosa.substring(0, 2).compareTo("241")==0) {
            return new EU_Putink(ime, prezime, id, a, brPasosa);
        } else {
             return new Regularni_putnik(ime, prezime, id, a, br-
Pasosa);
        }
    }
}
```

```
public class GlavnaGranicniPrelaz {
    public static void main(String[] args) {
        Prelaz singidunum = Prelaz.getInstance(2,3);
        List<Putnik> putnici = new ArrayList<>();
        Putnik s1 = PutnikFactory.getInstance("Pera", "Peric",
"24145678", singidunum,  "24145678");
        Putnik s2 = PutnikFactory.getInstance("Vasa", "Vasovic",
"12613424", singidunum, "12613424");
        Putnik s3 = PutnikFactory.getInstance("Nikola", "Jokic",
"24163721", singidunum, "24163721");
        Putnik s4 = PutnikFactory.getInstance("Tamara", "Nikic",
"24143721", singidunum, "24143721");
```

```
        Putnik s5 = PutnikFactory.getInstance("Ljuba", "Simic",
"27637215", singidunum, "2763725");
        Putnik s6 = PutnikFactory.getInstance("Mika", "Mikovic",
"27216571", singidunum, "27216571");
        Putnik s7 = PutnikFactory.getInstance("Ana", "Anic",
"24132442", singidunum, "24132442");
        Putnik s8 = PutnikFactory.getInstance("Vera", "Veric",
"24142463", singidunum, "24132442");
        Putnik s9 = PutnikFactory.getInstance("Zuca", "Zutic",
"27653423", singidunum, "27653423");
        Putnik s10 = PutnikFactory.getInstance("Luka", "Lukic",
"24142264", singidunum, "24142264");
        putnici.add(s1);
        putnici.add(s2);
        putnici.add(s3);
        putnici.add(s4);
        putnici.add(s5);
        putnici.add(s6);
        putnici.add(s7);
        putnici.add(s8);
        putnici.add(s9);
        putnici.add(s10);
        Iterator<Putnik> iter = putnici.iterator();
        while(iter.hasNext()) {
            Putnik o = iter.next();
            Thread t = new Thread(o);
            t.start();
        }
    }
}
```

Zadaci za samostalno rešavanje:

Zadatak 11: Napisati Java program koji simulira rad kasa u prodavnici. Svaki kupac na kasi predstavlja jednu nit čiji je posao da dođe na red, sačeka da se svi artikli otkucaju, plati i ode sa kase.

Klasa Prodavnica ima kao atribut broj slobodnih kasa i ime. Potrebno je obezbediti da je moguće kreirati samo jedan objekat klase Prodavnica koristeći Singleton šablon. Ime prodavnice ne sme da bude duže od 10 karaktera i mora da počne velikim slovom. Ukoliko se pokuša postavljanje imena na pogrešni format, izbaciti izuzetak. Metode dodjiNaKasu() i odlazakSaKase() su metode koje simuliraju dolazak i odlazak sa kase (u metodama je potrebno odštampati odgovarajuće poruke: Kupac došao na kasu i Kupac odlazi sa kase).

Implementirati apstraktnu klasu Kupac koja kao atribute ima ime, prezime i godine. Potrebno je implementirati metod toString() i napraviti apstraktni metod otkucajArtikle(). Putnik implementira Runnable interfejs gde metoda run() opisuje proces prelaska jednog vozila preko granice.

Klase Penzioner i Regularni nasleđuju klasu Kupac. Ove dve klase imaju kao atribut broj godina, i to Kupac je tipa Penzioner ukoliko je broj godina veći od 65 dok je u suprotnom tipa Regularni. Kreiranje Kupca rešiti korišćenjem Factory design šablona.

U klasi Penzioner implementirati nadjačani metod toString(). Metod otkucajArtikle() simulira proces kucanja artikla iz korpe i plaćanje. Unutar metode je potrebno odštampati na standardni izlaz poruku: "Penzioner + ime niti + je na kasi". Nakon dolaska na kasu se čeka do 1000ms da se otkucaju artikli, a zatim ispisati poruku "Plaćanje Penzionera +ime niti+ u toku". Plaćanje traje do 500ms.

Slično klasi Penzioner i klasa Regularni ima nadjačani metod toString() i metod otkucajArtikle() gde se ispisuje poruka na standardni izlaz : "Regularni kupac + ime niti + je na kasi". Nakon dolaska na kasu se čeka 300ms da se otkucaju artikli, a zatim ispisati poruku "Plaćanje Regularnog kupca +ime niti+ u toku". Plaćanje traje 500ms.

Napisati main() metod koji simulira rad kasa u prodavnici "Supermarket". Na 4 kase dolaze kupci. Napraviti listu Kupaca iz java.util.Collection i listi pristupati pomoću iteratora. U listu smestiti 12 kupaca. Obezbediti ispravan rad niti, odnosno da kada jedan kupac dođe na kasu, drugi kupac ne može da prekida njegovu transakciju.

Zadatak 12: Napisati Java program koji simulira rad telefonskih operatera neke kompanije. Kompanija ima određen broj zaposlenih operatera koji su na raspolaganju privatnim i poslovnim korisnicima. Svaki korisnik koji zatraži uslugu operatera predstavlja jednu nit koja okupira operatera sve dok se ne reši njegov problem.

Klasa Kompanija ima kao atribut broj slobodnih operatera za poslovne korisnike i broj slobodnih operatera za privatne korisnike. Potrebno je obezbediti da je moguće kreirati samo jedan objekat klase Kompanija koristeći Singleton šablon. Metode uspostaviVezuPoslovni(), prekiniVezuPoslovni(), uspostaviVezuPrivatni() i prekiniVezuPrivatni() su metode koje simuliraju uspostavljanje i prekid veze operatera. U metodama je potrebno odštampati odgovarajuće poruke: *Uspostavljanje veze, poslovni (odnosno privatni) korisnik!!!* i *Prekid veze, poslovni (odnosno privatni) korisnik!!!*. Obratiti pažnju da poslovni korisnici uspostavljaju vezu sa operaterima koji su zaduženi za rad sa njima (operator za privatne korisnike ne može da uspostavlja vezu sa poslovnim korisnikom i obrnuto).

Implementirati apstraktnu klasu Korisnik koja kao atribute ima ime i idKorisnika. Atribut idKorisnika je string od 5 karaktera oblika XXCCC, gde je XX može biti zamenjeno sa AG, BG ili NN dok je C bilo koja cifra. Ukoliko se pokuša postavljanje idKorisnika na pogrešni format, izbaciti izuzetak. Potrebno je implementirati metod toString() i napraviti apstraktni metod pozoviOperatera(). Klasa Korisnik implementira Runnable interfejs gde metoda run() opisuje proces jednog razgovora sa operaterom. (uspostavljanje veze, obavljanje razgovora i prekid veze).

Klase PrivatniKorisnik i PoslovniKorisnik nasleđuju klasu Korisnik. Klasa PrivatniKorisnik ima kao atribut jmbg oblika 13 cifara, dok klasa PoslovniKorisnik ima kao atribut pib oblika 860 praćeno sa 10 cifara (npr. 8601234567890). Kreiranje Korisnika rešiti korišćenjem Factory design pattern-a (smatrati da jmbg ne može da počne sa 860).

U klasi PrivatniKorisnik implementirati nadjačani metod toString(). Metod pozoviOperatera() simulira proces razgovora privatnog korisnika sa operaterom. Proces razgovora se sastoji od potvrđivanje identiteta korisnika (štampanje podataka korisnika i čekanje 250ms da se podaci potvrde) i razgovora koji traje maksimalno 2000ms.

Slično klasi PrivatniKorisnik i klasa PoslovniKorisnik ima nadjačani metod toString() i metod pozoviOperatera(). Razgovor poslovnog korisnika sa operaterom se sastoji od sledećih operacija: potvrđivanje identiteta i autorizacija (štampanje pib-a i imena korisnika i čekanje do 200ms da se podaci potvrde). Nakon potvrde, razgovor poslovnih korisnika sa operaterom je ograničen na 1000ms.

Napisati main() metod koji simulira rad telefonskih operatera neke kompanije. Ukupno 5 operatera je zaposleno u kompaniji od kojih 3 za poslovne i 2 za privatne korisnike.

- Napraviti listu Korisnika iz java.util.Collection i listi pristupati pomoću iteratora.
- U listu smestiti 10 korisnika.
- Obezbediti ispravan rad niti, odnosno da korisnici budu usluženi od strane odgovarajućeg operatora (poslovni korisnici od strane operatora koji radi sa poslovnim, i obrnuto za privatne korisnike) i da razgovor ne može biti prekinut od strane drugog korisnika.

NAPOMENA!!! Obratiti pažnju na vremena čekanja i trajanje razgovora, kada je dato tačno vreme, a kada najduže vreme čekanja (u drugom slučaju koristiti Random).

Primer ispravnog izlaza:

```
Uspostavljanje veze, poslovni korisnik!!!
Uspostavljanje veze, poslovni korisnik!!!
Provera identieta i autorizacija... pib = 1234567890123 korisnik Igor
Uspostavljanje veze, privatni korisnik!!!
Provera identieta i autorizacija... pib = 8601234567892 korisnik Marina
Uspostavljanje veze, privatni korisnik!!!
Provera identiteta...Kosisnik Marko, jmbg=8601234567890
Provera identiteta...Kosisnik Ana, jmbg=8601234567891
Uspostavljanje veze, poslovni korisnik!!!
Provera identieta i autorizacija... pib = 8601234567893 korisnik Zika
Poslovni korisnik ceka na operatera...
Privatni korisnik ceka na operatera...
Poslovni korisnik ceka na operatera...
Uspesna identifikacija i autorizacija...
Uspesna identifikacija i autorizacija...
U toku je razgovor poslovnog korisnika Thread-1
Uspesna identifikacija i autorizacija...
U toku je razgovor poslovnog korisnika Thread-4
U toku je razgovor poslovnog korisnika Thread-3
Privatni korisnik ceka na operatera...
Privatni korisnik ceka na operatera...
Poslovni korisnik ceka na operatera...
Privatni korisnik ceka na operatera...
U toku je razgovor privatnog korisnika Thread-2
U toku je razgovor privatnog korisnika Thread-0
Poslovni korisnik ceka na operatera...
Privatni korisnik ceka na operatera...
Privatni korisnik ceka na operatera...
Prekid veze, privatni korisnik!!!
Prekid veze, privatni korisnik!!!
Privatni korisnik ceka na operatera...
Poslovni korisnik ceka na operatera...
Prekid veze, poslovni korisnik!!!
Prekid veze, poslovni korisnik!!!
Prekid veze, poslovni korisnik!!!
Uspostavljanje veze, poslovni korisnik!!!
Provera identieta i autorizacija... pib = 8601234567895 korisnik Mirko
Uspesna identifikacija i autorizacija...
U toku je razgovor poslovnog korisnika Thread-9
Uspostavljanje veze, privatni korisnik!!!
Provera identiteta...Kosisnik Nevena, jmbg=2131647838456
Uspostavljanje veze, privatni korisnik!!!
Provera identiteta...Kosisnik Branko, jmbg=2324638761234
U toku je razgovor privatnog korisnika Thread-6
Uspostavljanje veze, poslovni korisnik!!!
Provera identieta i autorizacija... pib = 8601234567894 korisnik Pera
U toku je razgovor privatnog korisnika Thread-10
Uspesna identifikacija i autorizacija...
U toku je razgovor poslovnog korisnika Thread-5
Uspostavljanje veze, poslovni korisnik!!!
Provera identieta i autorizacija... pib = 8563856235467 korisnik Ivana
Uspesna identifikacija i autorizacija...
```

```
U toku je razgovor poslovnog korisnika Thread-7
Prekid veze, privatni korisnik!!!
Uspostavljanje veze, privatni korisnik!!!
Provera identiteta...Kosisnik Luka, jmbg=2233116644990
Prekid veze, privatni korisnik!!!
U toku je razgovor privatnog korisnika Thread-8
Prekid veze, privatni korisnik!!!
Prekid veze, poslovni korisnik!!!
Prekid veze, poslovni korisnik!!!
Prekid veze, poslovni korisnik!!!
```

Pitanja

1. Šta je multitasking?
2. Šta je programska nit u programskom jeziku Java?
3. Kako se izvršavaju niti ukoliko nema eksplicitne koordinacije između njih? Šta je problem kod takvog izvršavanja?
4. Da li Java program ima niti ukoliko se ekplicitno ne deklarišu?
5. Koje su prednosti višenitnog programiranja?
6. Da li ima smisla koristiti višenitno programiranje na jednom procesoru? Zašto?
7. Šta je predstavlja jedna nit u Java program?
8. Kako se predstavlja nit u Java programskom jeziku i koji interfejs implementira?
9. Koji interfejs mora da implementira svaka klasa zadatka koji će se izvršavati u posebnoj niti?
10. Koje metode ima interfejs Runnable?
11. Kojom metodom se pokreće nit?
12. Koja su dva osnovna načina programiranja niti?
13. Navesti neke metode klase Thread i opisati njihovu funkciju.
14. Kada se koristi metoda join() iz klase Thread?
15. Šta su anonimne klase i kako se mogu koristiti za kreiranje niti?
16. Kako se lambda izrazi mogu koristiti za kreiranje niti?
17. Šta je stanje trke (engl. *race condition*) i kada nastaje?
18. Kako se sprečava Java program da dođe u stanje trke?
19. Na čemu je bazirana realizacija sinhronizacije u Javi?
20. Šta je mrtva petlja i kada može nastati?
21. Kako se može rešiti problem mrtve petlje?
22. Kako se realizuje saradnja niti u Java programskom jeziku?
23. Čemu služe metode wait(), notify() i notifyAll() u Java programskom jeziku?
24. Šta su i čemu služe Semafore?
25. Šta je argument konstruktora klase Semaphore u Java programskom jeziku?
26. Navesti i opisati neke od značajnijih metoda klase Semaphore u Java programskom jeziku.

III
Klijent-server programiranje

Internet se, u svojoj osnovi, zasniva na povezivanju računara i njihovoj međusobnoj komunikaciji. Jedan od najboljih aspekata programskog jezika Java jeste da inkorporira model za mrežnu komunikaciju, koji je lak za upotrebu i upotrebljiv na više različitih platformi. Da bi dva programa koji se nalaze na udaljenim računarima mogla da komuniciraju preko mreže, neophodno je da ti programi formiraju po jedan soket koji predstavljaju logičke krajnje tačke konekcije između ova dva programa. Nakon konstrukcije soketa i uspostavljanja konekcije između njih, u programima se soketi dalje koriste kao najobičniji izvori i prijemnici podataka, uz standardnu primenu programskog ulaza/izlaza, odnosno ulaznog i izlaznog toka podataka preko formirane konekcije između soketa. Ukoliko jedan program želi da pošalje podatke drugom programu, on upisuje te podatke u svoj izlazni tok i oni se šalju drugom računaru. Na drugoj strani, podaci koji pristižu se usmeravaju u ulazni tok programa, odakle ih drugi program čita.

Može se primetiti da se samo pisanje i čitanje podataka preko uspostavljene konekcije ne razlikuje od najobičnijih ulazno-izlaznih operacija za datoteku koja se nalazi na disku. Jedini deo koji se razlikuje je uspostavljanje konekcije preko soketa između dva programa. Ceo proces se zasniva na kreiranju dva soketa praćenjem tačno određenog niza koraka. Jedan program konstruiše soket koji pasivno čeka dok ne stigne zahtev za uspostavljanje konekcije od strane nekog drugog soketa. Drugi program kreira aktivni soket, koji šalje zahtev za uspostavljanje konekcije pasivnom soketu. Kada pasivni soket primi zahtev, on odgovara potvrdno na zahtev za uspostavljanje konekcije i time je konekcija uspostavljena (pasivni soket može i odbiti zahtev za uspostavljanje konekcije). Kada je konekcija uspostavljena, programi koji se nalaze na logičkim krajevima uspostavljene konekcije mogu formirati ulazne i izlazne tokove preko kojih primaju i šalju podatke. Komunikacija između programa traje sve dok jedan od programa ne zatvori konekciju.

Postoje i posebni termini koji označavaju programe koji konstruišu dva osnovna tipa soketa. Ukoliko program konstruiše pasivni soket koji čeka na zahteve za uspostavljanje konekcije, on se naziva server, a pasivni soket se još naziva serverski soket. Sa druge strane, program koji želi da se poveže sa serverom se naziva klijent, a njegov aktivni soket koji šalje zahtev za uspostavljanje konekcije se naziva klijentski soket. Ovaj model mrežnog programiranja se naziva klijent-server programiranje. U

ovom modelu, server se najčešće nalazi negde u mreži i čeka zahteve za uspostavljanje konekcije od klijenata. Server obično pruža neki vid usluge koja je potrebna klijentima, a klijenti je mogu dobiti nakon uspešnog povezivanja sa serverom. U praksi je neophodno da server može da opsluži više klijenata u isto vreme, odnosno da kada se jedan klijent uspešno poveže sa serverom, serverski soket ne odbija zahteve od drugih klijenata. Kako bi se obezbedio paralelizam u radu servera, koriste se programske niti.

Na jednom računaru u istom trenutku može raditi više programa koji komuniciraju preko mreže sa drugim programima. Zbog toga nije dovoljno logičke krajeve mrežne konekcije označiti samo IP adresama računara na kojima se programi izvršavaju, već je neophodno naznačiti i broj porta. Broj porta je 16-bitni ceo broj u opsegu 0 - 65536, uz ograničenje da su brojevi od 0 do 1024 rezervisani za standardne veb servise. Prema svemu navedenom, server čeka zahteve za uspostavljanje konekcije na specifičnom portu. Klijent osim IP adrese računara na kome se izvršava serverski program mora znati i broj porta na kome server očekuje zahteve. Kao što je već navedeno, standardni veb servisi imaju dodeljene standardne brojeve portova, kao što je prikazano u tabeli 3.1. Sa tabele se može uočiti da standardni veb serveri koji koriste HTTP očekuju zahteve na portu 80.

Tabela 3.1. Primeri poznatih portova

Broj porta	Opis
20	FTP – Data
21	FTP – Control
22	SSH Remote Login Protocol
23	Telnet
25	Simple Mail Transfer Protocol (SMTP)
80	HTTP
110	POP3
118	SQL Services
143	Interim Mail Access Protocol (IMAP)
156	SQL Server
389	Lightweight Directory Access Protocol (LDAP)
443	HTTPS

3.1. Klijent-server

Kao što smo već spominjali, podaci se preko Interneta šalju u paketima. Svaki paket sadrži zaglavlje (engl. *header*) i tovar (engl. *payload*). U zaglavlju se nalazi adresa i port destinacije paketa, adresa i port izvora odakle je paket poslat i dodatne informacije koje služe za osiguravanje pouzdanog prenosa. Sami podaci koji se prenose u paketu se nalaze u tovaru. Pošto su paketi ograničene dužine, često je neophodno da se podaci podele na više paketa prilikom slanja, i da se ponovo sastave na destinaciji. Moguće je da se tokom transporta neki paketi izgube ili oštete, pa je neophodno njihovo ponovno slanje. Takođe, pošto paketi putuju kroz mrežu nezavisno jedni od drugih, moguće je da na destinaciju stignu u pogrešnom redosledu, pa je neophodno da se preurede u odgovarajući redosled.

Praćenje svih ovih zadataka – podele podataka u pakete, generisanja zaglavlja, parsiranja zaglavlja dolazećih paketa, praćenja koji paketi su stigli na destinaciju, a koji nisu stigli ili su stigli oštećeni itd. – bi zahtevalo mnogo posla i mnogo dodatnog koda. Na sreću, programer ne mora da ovo radi samostalno. Soketi omogućavaju programeru da tretira mrežnu konekciju kao još jedan tok podatka u koji može upisivati podatke i iz koga može čitati podatke. Soketi zaklanjaju detalje mreže na nižim nivoima, poput detekcije grešaka, veličine paketa, retransmisije izgubljenih paketa itd.

Programer posmatra soket kao prostu konekciju između dva programa. Soket može da izvrši sedam osnovnih operacija:

- Povezivanje na udaljenu mašinu
- Slanje podataka
- Prijem podataka
- Zatvaranje konekcije
- Vezivanje za port
- Osluškivanje dolazećih zahteva
- Prihvatanje (ili odbijanje) zahteva za konekcijama udaljenih mašina na odgovarajući port

U programskom jeziku Java, u paketu java.net se nalazi klasa Socket, koju koriste i klijenti i serveri. Ova klasa ima metode koje poklapaju sa prve četiri osnovne operacije. Poslednje tri operacije su neophodne samo za server, koji čeka na zahteve klijenata. Oni su implementirani u klasi ServerSocket, koji se takođe nalazi u paketu java.net.

Objekat tipa ServerSocket ne učestvuje neposredno u komunikaciji između klijenata i servera, već služi samo da čeka na zahteve za uspostavljanje konekcije od strane klijenata. Sa druge strane, objekat klase Socket predstavlja jednu krajnju tačku uspostavljene konekcije. Može biti klijentski soket, koji šalje serveru zahtev za uspostavljanje konekcije, a takođe može biti i soket konstruisan na strani servera radi

uspostavljanja same konekcije sa klijentom, odnosno obezbeđuje drugu logičku tačku konekcije. Da rezimiramo, na strani servera objekat tipa ServerSocket čeka na zahteve za uspostavljanje konekcije od strane klijenata i konstruiše objekat tipa Socket radi uspostavljanja veze i omogućavanja dalje komunikacije.

Kada se piše programski kod za server, nije unapred poznato koji klijenti će kontaktirati server, a čak i da je ta informacija poznata, ne znamo kada će klijent kontaktirati server. Drugim rečima, server se ponaša kao telefonski operater koji sedi pored telefona i čeka dolazeće pozive. On ne zna ko će i kada zvati, već samo zna da kada telefon pozvoni, treba da se javi i da priča sa osobom na drugoj strani veze. Kao što je već napomenuto, za javljanje na telefon, odnosno prihvatanje dolazećih konekcija, Java pruža posebnu klasu ServerSocket koja predstavlja serverske sokete. U svojoj srži, zadatak serverskog soketa jeste da sedi pored telefona i čeka dolazeće pozive, odnosno, tehnički govoreći, serverski soket se izvršava na serveru i osluškuje dolazeće TCP konekcije. Svaki serverski soket osluškuje na određenom portu na serverskoj mašini. Kada klijent sa udaljenog računara pokuša da se poveže na taj port, server se budi, uspostavlja konekciju između klijenta i servera i vraća objekat tipa Socket koji predstavlja logički kraj konekcije na strani servera. Kada se konekcija uspostavi, server koristi običan objekat tipa Socket za slanje podataka klijentu. Podaci uvek putuju preko običnog soketa, odnosno objekta tipa Socket.

Klasa ServerSocket sadrži sve što je neophodno za serversku komponentu, između ostalog konstuktore za kreiranje novih ServerSocket objektata, metode koje osluškuju konekcije na zadatom portu, metode koje služe za konfiguraciju različitih podešavanja i standardne uslužne metode poput toString() metode.

Osnovni životni ciklus serverskog programa prati tačno određen skup radnji. Na početku, neophodno je kreirati novi objekat tipa ServerSocket na odgovarajućem portu upotrebom konstruktora. Broj porta se navodi kao argument u pozivu konstruktora, na primer:

```
ServerSocket serverSoket = new ServerSocket (port);
```

Specificirani port u argumentu konstruktora mora biti broj tipa int, u intervalu 0-65536. Kao što je ranije napomenuto, prvih 1024 portova su zauzeti standardnim servisima, tako da u praksi taj broj treba biti veći od 1024. Ukoliko dođe do problema prilikom kreiranja serverskog soketa i povezivanja sa zadatim portom, izbacuje se izuzetak tipa IOException. U praksi to obično znači da je navedeni port već u upotrebi.

Odmah nakon kreiranja, serverski soket počinje da osluškuje i čeka zahteve za konekcijom na zadatom portu. Kako bi se odgovorilo na zahtev, koristi se metoda accept() iz klase ServerSocket. Metodom accept() se prihvata zahtev, uspostavlja konekcija sa klijentom, a kao rezultat se vraća objekt tipa Socket preko koga ide sva dalja komunikacija sa klijentom:

```
Socket soket = server.accept();
```

Prilikom poziva metode accept(), izvršavanje se blokira sve dok ne stigne zahtev za uspostavljanje konekcije od strane nekog klijenta, ili ne dođe do neke greške pri čemu se izbacuje izuzetak tipa IOException. Ovde treba biti oprezan, pošto se blokira i metoda u kojoj je pozvana metoda accept(), odnosno cela ta programska nit (druge niti istog programa se i dalje normalno izvršavaju). Ukoliko je potrebno, moguće je pozivati metodu accept() više puta za isti serverski soket, kako bi se prihvatili zahtevi od više klijenata. Serverski soket će nastaviti da osluškuje zahteve za konekcijom sve do zatvaranja pozivom metode close(), ili dok ne dođe do neke greške u programu.

U zavisnosti od tipa servera, nakon uspostavljanja konekcije na čija se oba kraja nalaze soketi tipa Socket, formira se ulazni tok, izlazni tok ili oba, koji služe za komunikaciju sa klijentom. Ulazni tok se formira pozivom metode getInputStream(), a izlazni tok se formira pozivom metode getOutputStream().

Nakon formiranja konekcije, server i klijent komuniciraju sve do zatvaranja konekcije. Zatvaranje konekcije može inicirati server, klijent ili oba. Nakon toga se server vraća u stanje osluškivanja i čekanja zahteva za novom konekcijom.

Osnovna struktura programa sa serverske strane, koji osluškuje zahteve na portu 2000, data je u nastavku.

```
try {
     ServerSocket serverSoket = new ServerSocket(2000);
        while (true) {
            //za komunikaciju sa klijentom koristi se Socket
            Socket soket = serverSoket.accept();
            //za ulazni tok otvara se InputStream
            InputStream ulaz = soket.getInputStream();
            //za izlazni tok otvara se OutputStream
            OutputStream izlaz = soket.getOutputStream();
            //Dalje se za komunikaciju sa klijentom koriste ulaz i
izlaz
              . . .
            //Na kraju je potrebno zatvoriti soket
            soket.close();
       }
} catch (IOException e){
       //u catch grani se hvata izuzetak i vrsi oporavak od greske
}
```

Na strani klijenta, Java program koristi sokete na jednostavan način. Prvo, neophodno je kreirati novi soket pozivom konstruktora klase Socket. Konstruktor prihvata dva argumenta. Prvi argument je IP adresa ili domensko ime ciljanog računara na kome se izvršava serverska aplikacija sa kojom klijent želi da se poveže. Drugi argument je broj porta na kome server osluškuje zahteve za novim konekcijama. Konstruktor klase Socket ima sledeći oblik:

```
public Socket (String IP, int port) throws IOException
```

Nakon kreiranja, soket pokušava da se konektuje na udaljeni računar. Kada se uspostavi konekcija, na sličan način kao i na serverskoj strani, formira se ulazni tok, izlazni tok ili oba, koji služe za komunikaciju sa serverom. Ulazni tok se formira pozivom metode getInputStream(), a izlazni tok se formira pozivom metode getOutputStream(). Na kraju komunikacije, klijent može inicirati zatvaranje konekcije.

Osnovna struktura programa sa klijentske strane, koji se povezuje sa serverom na adresi 192.168.1.10 i portu 2000 je data u nastavku.

```
try {
     Socket soket = new Socket(„192.168.1.10“, 2000);
     //za ulazni tok otvara se InputStream
     InputStream ulaz = soket.getInputStream();
     //za izlazni tok otvara se OutputStream
     OutputStream izlaz = soket.getOutputStream();
     //Dalje se za komunikaciju sa serverom koriste ulaz i izlaz
              . . .
     //Na kraju je potrebno zatvoriti soket
     soket.close();
} catch (IOException e){
       //u catch grani se hvata izuzetak i vrsi oporavak od greske
}
```

Treba napomenuti da je za ispravan rad svakog programa koji koristi sokete sa serverske i klijentske strane, neophodno prvo pokrenuti serverski program, a tek nakon toga klijentski.

3.2. Jednostavni klijent-server programi

Posmatrajmo kao prvi primer veoma jednostavni klijent-server program, gde server prihvata zahtev od klijenta, a zatim vraća povezanom klijentu neki podatak. U

ovom slučaju, server vraća trenutno serversko vreme klijentu. U ovom slučaju, postoji samo prenos podataka od servera ka klijentu, tako da sa serverske strane postoji samo OutputStream.

```
import java.io.IOException;
import java.io.PrintWriter;
import java.net.ServerSocket;
import java.net.Socket;
import java.util.Date;

public class DatumServer {

    public static void main(String[] args) throws IOException {
        ServerSocket serverskiSoket = new ServerSocket(1818);
        try {
            while (true) {
                Socket soket = serverskiSoket.accept();
                try {
                    PrintWriter out =
                          new PrintWriter(soket.getOutputStream(),
true);
                    out.println(new Date().toString());
                } finally {
                    soket.close();
                }
            }
        }
        finally {
            serverskiSoket.close();
        }
    }
}
```

Server prvo otvara serverski soket i očekuje konekcije na portu 1818. Kada pristigne zahtev od klijenta, server prihvata konekciju i šalje trenutni datum klijentu. Zbog lakše manipulacije izlaznim tokom podataka, OutputStream je omotan objektom klase PrintWriter, koji na dalje pruža isti interfejs kao standardni izlaz sistema (System.out). Potrebno je još jednom naglasiti da se serverski program mora pokrenuti pre pokretanja klijentskog programa, pošto se klijent ne može povezati sa serverom koji ne radi.

Klijentski program u ovom slučaju pruža rudimentarni grafički korisnički interfejs, gde se prvo očekuje da korisnik unese IP adresu serverskog računara, i nakon

toga pokušava povezivanje sa portom 1818 na unetoj adresi. Pošto se u našem slučaju i serverski i klijentski programi izvršavaju na istom računaru, kao IP adresa se može uneti localhost. Ukoliko se ova dva programa izvršavaju na različitim računarima, u ovom koraku korisnik treba da unese IP adresu servera.

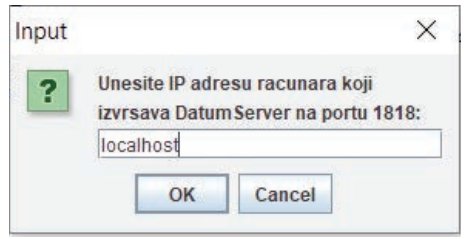
Pošto u ovom slučaju samo server šalje podatke klijentu, nakon povezivanja klijent otvara samo svoj InputStream. Radi lakšeg rada sa ulaznim tokom podatka, InputStream je omotan objektom tipa InputStreamReader, i zatim objektom tipa BufferedReader. Klijent čita jednu liniju iz svog ulaznog toka (podatak sa datumom), i prikazuje to u prostom grafičkom korisničkom interfejsu. Kod klijentskog programa je dat u nastavku:

```
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.net.Socket;
import javax.swing.JOptionPane;

public class DatumKlijent {

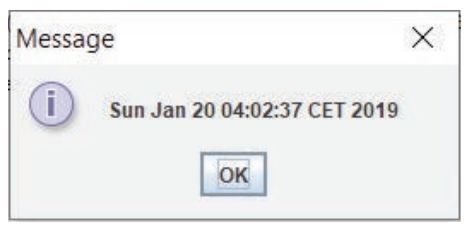
    public static void main(String[] args) throws IOException {
        String adresaServera = JOptionPane.showInputDialog(
            “Unesite IP adresu racunara koji\n” +
            “izvrsava DatumServer na portu 1818:”);
        Socket s = new Socket(adresaServera, 1818);
        BufferedReader input =
              new BufferedReader(new InputStreamReader(s.getInput-
Stream()));
        String answer = input.readLine();
        JOptionPane.showMessageDialog(null, answer);
        s.close();
        System.exit(0);
    }
}
```

Kada se klijentski program pokrene, prikazuje mu se jednostavan dijalog gde je potrebno uneti IP adresu servera. Kao što smo već napomenuli, pošto se oba programa izvršavaju na istom računaru, dovoljno je uneti localhost, kao što je prikazano na slici 3.1.



Slika 3.1. Unos IP adrese u DatumKlijent programu

Nakon unosa odgovarajuće adrese i klika na dugme OK, serverski program prihvata zahtev za konekcijom, i šalje trenutni datum preko svog izlaznog toka podataka. Klijentski program prihvata taj podatak u svom ulaznom toku, čita jednu liniju iz ulaznog toka i prikazuje je u jednostavnom prozoru, kao što je prikazano na slici 3.2.



Slika 3.2. Prikaz podataka pristiglih sa DatumServer programa

U ovom prvom primeru, implementirana je samo jednosmerna komunikacija, odnosno serverski program nakon povezivanja klijentskog programa šalje određene podatke klijentu. U opštem slučaju, postoji dvosmerna komunikacija između serverskog i klijentskog programa. Osnovna paradigma ovakvog modela jeste da klijent šalje određene podatke serveru, server prihvata te podatke, koristi ih za neku obradu i izračunavanje, i na kraju dobijeni rezultat server šalje klijentu.

Takođe, može se uočiti da server sve zahteve prihvata u jednoj niti. Dok obrađuje zahtev jednog povezanog klijenta, server ne može prihvatiti novi zahtev. Tek nakon zatvaranja trenutne konekcije, server može prihvatiti novi zahtev. Dvosmerna komunikacija i višenitni server su opisani u narednom poglavlju.

3.3. Višenitno mrežno programiranje

U prethodnom poglavlju posmatrali smo veoma jednostavan klijent-server program, gde je postojala samo jednosmerna komunikacija. Sada se posmatra nešto složeniji program, koji ilustruje dvosmernu komunikaciju. Nakon povezivanja, klijentski program šalje poruke serverskom programu, a na svaku poruku server šalje odgovor klijentu.

Implementacija serverskih programa vrlo često podrazumeva da se više zadataka može obavljati paralelno, odnosno da više klijenata istovremeno uspostavi konekciju sa istim serverom, kako bi dobili neke rezultate od njega. U primeru iz prethodnog poglavlja to nije bilo moguće, pošto je serverski program zahteve prihvatao i obrađivao u jednoj niti. Svaki sledeći zahtev koji bi stigao od nekog drugog klijenta bi morao da čeka završetak obrade klijenta koji se trenutno opslužuje. U prethodnom primeru to možda i ne bi bio preveliki problem, pošto server nakon uspostavljanja veze sa klijentom šalje samo jednu poruku i odmah zatvara konekciju. Međutim, u slučaju kompleksnijeg programa, kada klijent i server mogu razmenjivati veliki broj poruka međusobno, to znači da se sledeći zahtev ne može prihvatiti sve dok se trenutna komunikacija ne završi. U ovom slučaju je neophodno koristiti više niti u serverskom programu za paralelnu obradu više klijenata.

Neka se sada posmatra jedan takav sofisticiraniji server, koji može da opsluži više klijenata u isto vreme. Kada se neki klijent poveže, server kreira novu nit za komunikaciju sa klijentom. Klijent šalje string serveru, a server odgovara slanjem istog stringa gde su sva mala slova pretvorena u velika. Proces se ponavlja sve dok klijent ne pošalje tačku (“.”).

Server prihvata zahteve na portu 1818, i svakom povezanom klijentu dodeljuje redni broj. Pored toga, kreira novu nit u kojoj se dalje odvija komunikacija sa klijentom, i vraća se čekanju na nove zahteve. Na ovaj način, server može da opsluži više klijenata u paraleli. Svaka nit je predstavljena klasom Kapitalizacija koja je izvedena iz klase Thread. U okviru run () metode se kreiraju i ulazni tok i izlazni tok, pošto je u ovom slučaju komunikacija dvosmerna. Kao i ranije, zbog lakšeg i komformnijeg rada, ulazni tok je omotan objektom klase BufferedReader, a izlazni tok objektom klase PrintWriter. Server na početku šalje klijentu pozdravnu poruku u dve linije teksta (samo jednom, prilikom povezivanja klijenta), a nakon toga svaki string koji stigne od klijenta obrađuje (kapitalizuje, odnosno pretvara sva mala slova u velika) i vraća klijentu rezultat (kapitalizovani string). Ove operacije se ponavljaju sve dok od klijenta ne stigne karakter “.”. Kod serverskog programa je dat u nastavku:

```
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.io.PrintWriter;
import java.net.ServerSocket;
import java.net.Socket;

public class KapitalizacijaServer {

    public static void main(String[] args) throws IOException {
        System.out.println(“Server je pokrenut”);
        int brojKlijenta = 1;
        ServerSocket server = new ServerSocket(1818);
```

```
        try {
            while (true) {
              new Kapitalizacija(server.accept(), brojKlijenta++).
start();
            }
        } finally {
            server.close();
        }
    }

    private static class Kapitalizacija extends Thread {

        private Socket soket;
        private int brojKlijenta;

        public Kapitalizacija(Socket soket, int brojKlijenta) {
            this.soket = soket;
            this.brojKlijenta = brojKlijenta;
        }

        public void run() {
            try {
                        BufferedReader ulaz = new BufferedReader(new
InputStreamReader(soket.getInputStream()));
                 PrintWriter izlaz = new PrintWriter(soket.getOut-
putStream(), true);

                // saljemo pozdravnu poruku klijentu
                izlaz.println("Dobar dan, vi ste klijent broj " +
brojKlijenta + ".");
                 izlaz.println("Unesite liniju teksta koju zelite
da kapitalizujete.");

                while (true) {
                    String inputString = ulaz.readLine();
                      if (inputString == null || inputString.equ-
als(".")) {
                        break;
                    }
                    izlaz.println(inputString.toUpperCase());
                }

            } catch (IOException e) {
                 System.out.println("Doslo je do greske u radu sa
klijentom");
```

```
            } finally {
                try {
                    soket.close();
                } catch (IOException e) {
                System.out.println("Nije moguce zatvoriti soket!!!");
                }
            }

            System.out.println("Klijent sa brojem " + brojKlijenta
+ " je opsluzen i konekcija je zatvorena");

        }
    }
}
```

Klijentski program je realizovan sa rudimentarnim grafičkim korisničkim interfejsom, gde je potrebno prvo uneti IP adresu servera na koji se klijent konektuje, a nakon toga omogućava unos teksta koji se šalje serveru na kapitalizaciju. Kapitalizovani tekst koji se primi od servera se takođe prikazuje u okviru korisničkog interfejsa. Pošto je u pitanju dvosmerna komunikacija, slično kao i u implementaciji serverskog programa koristimo i ulazni i izlazni tok podataka, koji su omotani odgovarajućim objektima tipa BufferedReader i PrintWriter. Kod za klijentski program je dat u nastavku:

```
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.io.PrintWriter;
import java.net.Socket;
import javax.swing.JFrame;
import javax.swing.JOptionPane;
import javax.swing.JScrollPane;
import javax.swing.JTextArea;
import javax.swing.JTextField;

public class KapitalizacijaKlijent {

    private BufferedReader in;
    private PrintWriter out;
   private JFrame frame = new JFrame("Klijent za kapitalizaciju");
    private JTextField dataField = new JTextField(40);
    private JTextArea messageArea = new JTextArea(8, 60);
```

```
    public KapitalizacijaKlijent() {
        //iniciramo gui
        messageArea.setEditable(false);
        frame.getContentPane().add(dataField, “North”);
          frame.getContentPane().add(new JScrollPane(messageArea),
“Center”);
        //dodajemo listener
        dataField.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                out.println(dataField.getText());
                String odgovor;
                try {
                    odgovor = in.readLine();
                    if (odgovor == null || odgovor.equals(“”)) {
                        System.exit(0);
                    }
                } catch (IOException ex) {
                    odgovor = “greska”;
                }
                messageArea.append(odgovor + “\n”);
                dataField.selectAll();
            }
        });
    }

    public void poveziSeSaServerom () throws IOException {

        String adresaServera = JOptionPane.showInputDialog (
          frame, “Unesite IP adresu servera”, JOptionPane.QUESTI-
ON_MESSAGE);

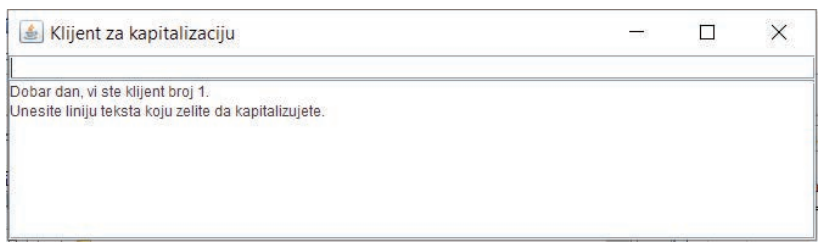
        Socket soket = new Socket (adresaServera, 1818);
         in = new BufferedReader(new InputStreamReader(soket.getIn-
putStream()));
        out = new PrintWriter (soket.getOutputStream(), true);

        //2 linije teksta dobijamo kao pozdravnu poruku od servera
        for (int i = 0; i < 2; i++){
            messageArea.append(in.readLine() + “\n”);
        }
    }
```

```
    public static void main(String[] args) throws IOException {
        KapitalizacijaKlijent klijent = new KapitalizacijaKlijent
();
       klijent.frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLO-
SE);
        klijent.frame.pack();
        klijent.frame.setVisible(true);
        klijent.poveziSeSaServerom();
    }

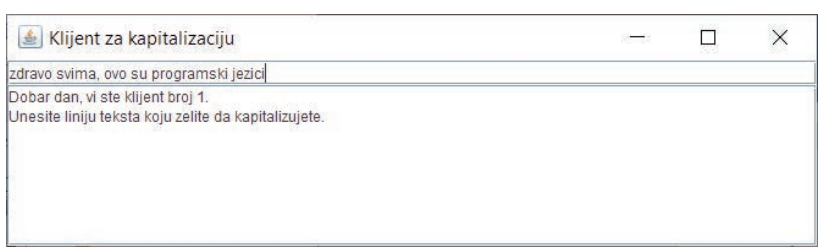
}
```

Nakon pokretanja izvršavanja klijentskog programa i povezivanja sa serverom, na klijentu se prikazuje pozdravna poruka kao na slici 3.3.



Slika 3.3. Pozdravna poruka servera

Korisnik nakon toga može uneti tekst koji je potrebno kapitalizovati, kao na slici 3.4. Pritiskom na taster enter, string se šalje serveru. Server kapitalizuje string i vraća odgovor klijentu. Odgovor servera, koji je kapitalizovani string, se na klijentskoj strani prikazuje kao na slici 3.5.



Slika 3.4. Unos teksta na klijentskoj strani



Slika 3.5. Prikaz kapitalizovanog stringa koji je stigao od server

3.4. Zadaci za vežbu

Zadatak 1: Napisati Java programe koji simuliraju rad servera i klijenata. Server ima listu korisnika, klijent mora da je na listi i da se predstavi korisničkim imenom i šifrom da bi mogao da pretraži listu korisnika. Svaki korisnik koji zatraži uslugu servera predstavlja jednu nit.

Klasa **Korisnik** ima atribute korisnickoIme, sifra, zarada i godine. Implementirati konstruktor, get metode i metod toString().

Napisati funkcionalni interfejs koji će služiti za proveru korisnika.

Java program **Server**. Implementirati Server koji čeka zahtev od klijenta. Svaki klijent bi trebalo da predstavlja jednu nit. Kada se uspostavi konekcija, server šalje pozdravnu poruku i zahtev za predstavljanje:

```
Uspesna konekcija, klijent broj brojKlijenta. Unesite korisnicko
ime i sifru:
```

Nakon toga, server čita od korisnika poslate podatke i proverava da li u listi svojih korisnika ima poslatu kombinaciju korisničkog imena i šifre. Ukoliko ne postoji, šalje se poruka i prekida se konekcija. Ukoliko korisnik postoji u listi, šalje se odgovarajuća poruka.

Nakon uspešne konekcije i identifikacije, server čeka parametre za pretragu liste. Prvo se učitava granica za zaradu, pa granica za godine. Nakon unetih parametara pomoću lambda izraza se kreira odgovarajući test. Za unete vrednosti manje od 0, potrebno je pronaći korisnike koji imaju manju vrednost odgovarajućeg atributa od unete apsolutne vrednosti (na primer, ako se za zaradu unesemo -1000, traže se korisnici koji imaju zaradu manju od 1000), za vrednosti veće od nule traže se korisnici sa većom ili jednakom vrednošću od zadate (na primer, ako se unese 1000 za zaradu, traže se korisnici koji imaju zaradu veću od 1000) i konačno ako se unese 0 za pretragu po nekom atributu, znači da ne želimo da filtritamo po njemu.

Nakon kreiranja odgovarajućeg testa, iz liste se izdvajaju korisnici koji zadovoljavaju parametre pretrage i šalju se korisniku (jedan po jedan).

Kreirati 10 korisnika na strani servera i upisati ih u listu.

Java program **Klijent**. Implementirati klijentsku stranu prethodno opisane komunikacije. Potrebno je odštampati na standardni izlaz sve poruke primljene od servera.

NAPOMENA: U rešenju postoje dodatna ispisivanja na standardnom izlazu servera koja pomažu proveri.

Funkcionalni interfejs:

```
public interface TestKorisnici {
    boolean test(Korisnici k);
}
```

Klasa Korisnik:

```
public class Korisnik {
    //moze da se koristi map
    //private Map<String, String> user;
    private String username;
    private String pass;
    private double zarada;
    private int godine;
     public Korisnik(String username, String pass, double zarada,
int godine) {
        this.username = username;
        this.pass = pass;
        this.zarada = zarada;
        this.godine=godine;
    }
    public String getUsername() {
        return username;
    }
    public String getPass() {
        return pass;
    }
    public double getZarada() {
        return zarada;
    }
    public int getGodine() {
        return godine;
    }
    @Override
    public String toString() {
        return “Korisnici “ + “username=” + username + “, zarada=”
+ zarada+”, godina=”+godine;
    }
}
```

Server:

```
public class Server {
    static List<Korisnik> korisnici;
    public static void main(String[] args) throws IOException {
        korisnici = new ArrayList<>();
        korisnici.add(new Korisnik(“Pera”, “1”, 200, 30));
        korisnici.add(new Korisnik(“Zika”, “2”, 400, 45));
        korisnici.add(new Korisnik(“Mika”, “3”, 600, 25));
        korisnici.add(new Korisnik(“Ana”, “4”, 800, 55));
        korisnici.add(new Korisnik(“Sanja”, “5”, 1000, 38));
        korisnici.add(new Korisnik(“Vanja”, “6”, 1200, 23));
        korisnici.add(new Korisnik(“Tanja”, “7”, 1400, 65));
        System.out.println(“Server je pokrenut”);
        int brojKlijenta = 1;
        ServerSocket server = new ServerSocket(9001);
        try {
            while (true) {
                  new Konekcija(server.accept(), brojKlijenta++).
start();
            }
        } finally {
            server.close();
        }
    }
    private static class Konekcija extends Thread {
        private Socket soket;
        private int brojKlijenta;
        public Konekcija(Socket soket, int brojKlijenta) {
            this.soket = soket;
            this.brojKlijenta = brojKlijenta;
        }
        public void run() {
            try {
              Scanner input = new Scanner(soket.getInputStream());
                PrintWriter output = new PrintWriter(soket.getOut-
putStream(), true);
                // posaljemo pozdravnu poruku klijentu
                 output.println(“Uspesna konekcija, klijent broj “
+ brojKlijenta + “.”);
               output.println(“Unesite korisnicko ime i sifru: “);
                String user = input.nextLine(); //cita username
                String pass = input.nextLine(); //cita pass
                System.out.println(user + “ “ + pass);
```

```
                //provera da li korisnik postoji
                boolean postoji = false;
                for (Korisnik k : korisnici) {
                      if (k.getUsername().compareTo(user) == 0 &&
k.getPass().compareTo(pass) == 0) {
                        output.println("Prepoznat korisnik");
                        postoji = true;
                        break;
                    }
                }
                if (!postoji) {
                       output.println("Nepoznat korisnik, zatvaram
konekciju");
                    throw new Exception("Nepoznat korisnik");
                }
                double zarada;
                int godine;
                //unosi se uslov pretrage: broj<0 znaci trazi se <
zarada/godine, >0 > i ako se unese 0, nema pretrage po tom uslovu
                TestKorisnici tester = k -> (true);
                zarada = input.nextDouble();
                godine = input.nextInt();
                System.out.println("zarada: " + zarada + ", godine
" + godine);
                if (zarada < 0) {
                    if (godine < 0) {
                         tester = k -> (k.getZarada() < zarada &&
k.getGodine() < -godine);
                    } else if (godine > 0) {
                         tester = k -> (k.getZarada() < zarada &&
k.getGodine() > godine);
                    } else {
                        tester = k -> (k.getZarada() < zarada);
                    }
                } else if (zarada > 0) {
                    if (godine < 0) {
                         tester = k -> (k.getZarada() > zarada &&
k.getGodine() < -godine);
                    } else if (godine > 0) {
                         tester = k -> (k.getZarada() > zarada &&
k.getGodine() > godine);
                    } else {
                        tester = k -> (k.getZarada() > zarada);
                    }
```

```
                } else if (godine < 0) {
                    tester = k -> (k.getGodine() < -godine);
                } else if (godine > 0) {
                    tester = k -> (k.getGodine() > godine);
                }
                System.out.println("gotovo tester");
                for (Korisnik k : korisnici) {
                    if (tester.test(k)) {
                        output.println(k.toString());
                        System.out.println(k.toString());
                    }
                }
                System.out.println("gotovo stampanje, ");
            } catch (IOException e) {
                 System.out.println("Doslo je do greske u radu sa
klijentom");
            } catch (Exception ex) {
                System.out.println(ex.getMessage());
                try {
                    soket.close();
                } catch (IOException e) {
                     System.out.println("Nije moguce zatvoriti so-
ket!!!");
                }
            } finally {
                try {
                    soket.close();
                } catch (IOException e) {
                     System.out.println("Nije moguce zatvoriti so-
ket!!!");
                }
            }
            System.out.println("Klijent sa brojem " + brojKlijenta
+ " je opsluzen i konekcija je zatvorena");
        }
    }
}
```

Klijent:

```
public class Klijent {
    public static void main(String[] args) throws IOException {
        BufferedReader in;
        PrintWriter out;
        Scanner sc = new Scanner(System.in);
        System.out.println("Unesite ip adresu servera");
        String serverAddress = sc.nextLine();
        Socket socket = new Socket(serverAddress, 9001);
        in = new BufferedReader(new InputStreamReader(socket.getIn-
putStream()));
        out = new PrintWriter(socket.getOutputStream(), true);
        //2 linije teksta: pozdrav i poruka unesite korisnicko ime
i sifru
        for (int i = 0; i < 2; i++) {
            System.out.println(in.readLine());
        }
        //unosi se korisnicko ime i sifra
        String odgovor = sc.nextLine();
        out.println(odgovor);
        odgovor = sc.nextLine();
        out.println(odgovor);
        //potvrda korisnika
        System.out.println(odgovor = in.readLine());
        //unosenje parameara pretrage
          if (!odgovor.equals("Nepoznat korisnik, zatvaram konek-
ciju")) {
            System.out.println("Unesite granicu za zaradu");
            out.println(sc.nextDouble()); //zarada
            System.out.println("Unesite granicu za godine");
            out.println(sc.nextInt()); //godine
          //citanje korisnika koji odgovaraju kriterijumu pretrage
            while ((odgovor = in.readLine()) != null) {
                System.out.println(odgovor);
            }
        }
    }
}
```

Primer izlaza:

Server:

```
Server je pokrenut
Pera 1
zarada: 1000.0, godine 37
gotovo tester
Korisnici username=Tanja, zarada=1400.0, godina=65
gotovo stampanje,
Klijent sa brojem 1 je opsluzen i konekcija je zatvorena
```

Klijent:

```
Unesite ip adresu servera
localhost
Uspesna konekcija, klijent broj 1.
Unesite korisnicko ime i sifru:
Pera
1
Prepoznat korisnik
Unesite granicu za zaradu
1000
Unesite granicu za godine
37
Korisnici username=Tanja, zarada=1400.0, godina=65
```

Zadatak 2: Napisati Java programe koji simuliraju rad servera i klijenata. Server ima listu korisnika, klijent mora da je na listi i da se predstavi korisničkim imenom i šifrom da bi mogao da pretraži listu korisnika. Ukoliko korisnik nije na listi, trebalo bi ga dodati. Svaki korisnik koji zatraži uslugu servera predstavlja jednu nit.

Klasa Korisnik ima atribute korisnickoIme i sifra (umesto klase, mogu se koristiti mape). Kreirati listu korisnika na strani server1.

Klasa Student ima aribute ime, prezime, smer, prosek, godina upisa. Implementirati konstruktor

Napisati funkcionalni interfejs koji će služiti za proveru studenata.

Java program Server. Implementirati Server koji čeka zahtev od klijenta. Svaki klijent bi trebalo da predstavlja jednu nit. Kada se uspostavi konekcija, server šalje pozdravnu poruku i zahtev za predstavljanje:

`Uspesna konekcija, klijent broj` *`brojKlijenta`*`.`

`Unesite korisnicko ime i sifru:`

Nakon toga, server čita od korisnika poslate podatke i proverava da li u listi svojih korisnika ima poslatu kombinaciju korisničkog imena i šifre. Ukoliko ne postoji, dodaje se u listu, šalje se poruka i prekida se konekcija. Ukoliko korisnik postoji u listi, šalje se odgovarajuća poruka.

Nakon uspešne konekcije i identifikacije, server čeka parametre za pretragu liste. Prvo se učitava smer, pa prosek, pa granica za godinu upisa. Nakon unetih parametara pomoću lambda izraza se kreira odgovarajući test. Za unete vrednosti manje od 0, potrebno je pronaći studente koji imaju manju vrednost odgovarajućeg atributa od unete apsolutne vrednosti (na primer, ako se za presek unese -7.80, traže se studenti koji imaju prosek manji od 7.80), za vrednosti veće od nule traže se korisnici sa većom ili jednakom vrednošću od proseka (na primer, ako se unese 9.0 za prosek, traže se studenti koji imaju prosek veći ili jednak od 9.00) i konačno ako se unese 0 za pretragu po nekom atributu, znači da ne želimo da filtritamo po njemu. Za smer se unosi 's', 'i' ili 'x'. Za 's' se traže studenti sa smera softversko inženjerstvo, za 'i' se traže studente sa informatike, 'x' znači da se ne filtrita po ovom parametru. Ukoliko dođe do neispravnog unosa parametra, ispisati poruku i prekinuti konekciju. Nakon kreiranja odgovarajućeg testa, iz liste se izdvajaju sudenti koji zadovoljavaju parametre pretrage i šalju se klijentu (jedan po jedan). Kreirati 10 studenata na strani servera i upisati ih u listu.

Java program Klijent. Implementirati klijentsku stranu prethodno opisane komunikacije. Potrebno je odštampati na standardni izlaz sve poruke primljene od servera.

Rešenje:

Funkcionalni interfejs:

```
public interface StudentTester {
    public boolean test(Student s);
}
```

Klasa Korisnik:

```
public class Korisnik {
    private String korisnickoIme;
    private String sifra;
    public Korisnik(String korisnickoIme, String sifra) {
        this.korisnickoIme = korisnickoIme;
        this.sifra = sifra;
    }
    public String getKorisnickoIme() {
        return korisnickoIme;
    }
```

```
    public void setKorisnickoIme(String korisnickoIme) {
        this.korisnickoIme = korisnickoIme;
    }
    public String getSifra() {
        return sifra;
    }
    public void setSifra(String sifra) {
        this.sifra = sifra;
    }
    @Override
    public String toString() {
        return “Korisnik{“ + “korisnickoIme=” + korisnickoIme + “,
sifra=” + sifra + ‘}’;
    }
}
```

Server:

```
public class Server {
    public static List<Korisnik> listaKorisnika;
    public static List<Student> listaStudenata;
    public static void main(String[] args) throws IOException {
        listaKorisnika = new ArrayList<>();
        listaKorisnika.add(new Korisnik(“Pera”, “1”));
        listaKorisnika.add(new Korisnik(“Mika”, “2”));
        listaKorisnika.add(new Korisnik(“Zika”, “3”));
        listaKorisnika.add(new Korisnik(“Laza”, “5”));
        listaKorisnika.add(new Korisnik(“Ana”, “6”));
        listaKorisnika.add(new Korisnik(“Mara”, “4”));
        listaKorisnika.add(new Korisnik(“Sanja”, “7”));
        listaKorisnika.add(new Korisnik(“Vanja”, “8”));
        listaKorisnika.add(new Korisnik(“Zuca”, “9”));
        listaKorisnika.add(new Korisnik(“Luca”, “10”));
        listaStudenata = new ArrayList<>();
        listaStudenata.add(new Student(“Pera”, “Peric”, “Softver-
sko inzenjerstvo”, 8.53, 2));
        listaStudenata.add(new Student(“Mika”, “Mikic”, “Softver-
sko inzenjerstvo”, 6.7, 3));
        listaStudenata.add(new Student(“Zika”, “Zikic”, “Informa-
tika”, 9.55, 2));
        listaStudenata.add(new Student(“Laza”, “Lazic”, “Informa-
tika”, 10.0, 2));
        listaStudenata.add(new Student(“Zuca”, “Zutic”, “Softver-
sko inzenjerstvo”, 6.53, 2));
```

```
        listaStudenata.add(new Student(“Vera”, “Veric”, “Softver-
sko inzenjerstvo”, 7.7, 1));
        listaStudenata.add(new Student(“Mara”, “Maric”, “Informa-
tika”, 10.0, 2));
        listaStudenata.add(new Student(“Lana”, “Lazic”, “Informa-
tika”, 8.12, 3));
        listaStudenata.add(new Student(“Luka”, “Lukic”, “Softver-
sko inzenjerstvo”, 7.45, 2));
        listaStudenata.add(new Student(“Mika”, “Mikovic”, “Soft-
versko inzenjerstvo”, 8.9, 3));
        System.out.println(“Server je pokrenut...”);
        ServerSocket server = new ServerSocket(9001);
        int brojKlijenta = 1;
        try {
            while (true) {
                  new Konekcija(server.accept(), brojKlijenta++).
start();
            }
        } finally {
            server.close();
        }
    }
    public static class Konekcija extends Thread {

        private Socket soket;
        private int brojKlijenta;

        public Konekcija(Socket soket, int brojKlijenta) {
            this.soket = soket;
            this.brojKlijenta = brojKlijenta;
        }
        @Override
        public void run() {
            try {
              Scanner input = new Scanner(soket.getInputStream());
                PrintWriter output = new PrintWriter(soket.getOut-
putStream(), true);
                 output.println(“Uspesna konkekcija, klijent broj”
+ brojKlijenta);
                  System.out.println(brojKlijenta + “: se uspesno
konektovao”);

                 System.out.println(brojKlijenta + “: Unosi ime i
sifru”);
```

```
                output.println(“Unesite ime i sifru”);
                String ime = input.nextLine();
                String sifra = input.nextLine();
                boolean postoji = false;
                for (Korisnik k : listaKorisnika) {
                        if (k.getKorisnickoIme().equals(ime) &&
k.getSifra().equals(sifra)) {
                    output.println(“Uspesno logovanje”);
                       System.out.println(brojKlijenta + “ se
uspesno ulogovao”);
                    postoji = true;
                    break;
                }
                }
                if (!postoji) {
                    output.println(“Losi kredincijali”);
                      System.out.println(brojKlijenta + “ se nije
ulogovao”);
                    listaKorisnika.add(new Korisnik(ime, sifra));
                    soket.close();
                } else {
                    StudentTester studentTester = (s) -> {
                        return true;
                    };
                       System.out.println(brojKlijenta + “: Unosi
prosek i godinu”);
                    output.println(“Unesi smer, proesk i godinu”);

                    String smer = input.nextLine();
                    double prosek = input.nextDouble();
                    int godina = input.nextInt();

                     System.out.println(brojKlijenta + “: smer=” +
smer + “, prosek=” + prosek + “, godina=” + godina);

                    if (smer.equals(“s”)) {
                        if (prosek < 0) {
                            if (godina < 0) {
                                studentTester = (s) -> {
                             return s.getSmer().equals(“Softversko
inzenjerstvo”)
                                                    && s.getProsek() <
-prosek && s.getGodinaUpisa() < -godina;
                                };
```

```
                            } else if (godina > 0) {
                                studentTester = (s) -> {
                             return s.getSmer().equals("Softversko
inzenjerstvo")
                                                    && s.getProsek() <
-prosek && s.getGodinaUpisa() >= godina;
                                };
                            } else {
                                studentTester = (s) -> {
                             return s.getSmer().equals("Softversko
inzenjerstvo")
                                                    && s.getProsek() <
-prosek;
                                };
                            }
                        } else if (prosek > 0) {
                            if (godina < 0) {
                                studentTester = (s) -> {
                             return s.getSmer().equals("Softversko
inzenjerstvo")
                                                 && s.getProsek() > prosek
&& s.getGodinaUpisa() < -godina;
                                };
                            } else if (godina > 0) {
                                studentTester = (s) -> {
                             return s.getSmer().equals("Softversko
inzenjerstvo")
                                                 && s.getProsek() > prosek
&& s.getGodinaUpisa() >= godina;
                                };
                            } else {
                                studentTester = (s) -> {
                             return s.getSmer().equals("Softversko
inzenjerstvo")
                                                && s.getProsek() > prosek;
                                };
                            }
                        } else {
                            studentTester = (s) -> {
                             return s.getSmer().equals("Softversko
inzenjerstvo");
                            };
                        }
```

```
                    } else if (smer.equals("i")) {
                        if (prosek < 0) {
                            if (godina < 0) {
                                studentTester = (s) -> {
                                                return s.getSmer().
equals("Informatika")
                                                 && s.getProsek() <
-prosek && s.getGodinaUpisa() < -godina;
                                };
                            } else if (godina > 0) {
                                studentTester = (s) -> {
                                                return s.getSmer().
equals("Informatika")
                                                 && s.getProsek() <
-prosek && s.getGodinaUpisa() >= godina;
                                };
                            } else {
                                studentTester = (s) -> {
                                                return s.getSmer().
equals("Informatika")
                                                 && s.getProsek() <
-prosek;
                                };
                            }
                        } else if (prosek > 0) {
                            if (godina < 0) {
                                studentTester = (s) -> {
                                                return s.getSmer().
equals("Informatika")
                                            && s.getProsek() > prosek
&& s.getGodinaUpisa() < -godina;
                                };
                            } else if (godina > 0) {
                                studentTester = (s) -> {
                                                return s.getSmer().
equals("Informatika")
                                             && s.getProsek() > prosek
&& s.getGodinaUpisa() >= godina;
                                };
                            } else {
                                studentTester = (s) -> {
                                                return s.getSmer().
equals("Informatika")
                                           && s.getProsek() > prosek;
```

```
                                };
                            }
                        } else {
                            studentTester = (s) -> {
                         return s.getSmer().equals("Informatika");
                            };
                        }
                    } else if (smer.equals("x")) {
                        if (prosek < 0) {
                            if (godina < 0) {
                                studentTester = (s) -> {
                                    return s.getProsek() < -prosek
&& s.getGodinaUpisa() < -godina;
                                };
                            } else if (godina > 0) {
                                studentTester = (s) -> {
                                    return s.getProsek() < -prosek
&& s.getGodinaUpisa() >= godina;
                                };
                            } else {
                                studentTester = (s) -> {
                                   return s.getProsek() < -prosek;
                                };
                            }
                        } else if (prosek > 0) {
                            if (godina < 0) {
                                studentTester = (s) -> {
                                     return s.getProsek() > prosek
&& s.getGodinaUpisa() < -godina;
                                };
                            } else if (godina > 0) {
                                studentTester = (s) -> {
                                     return s.getProsek() > prosek
&& s.getGodinaUpisa() >= godina;
                                };
                            } else {
                                studentTester = (s) -> {
                                    return s.getProsek() > prosek;
                                };
                            }
                        }
                    }
                System.out.println(brojKlijenta + "Ispis upita");
```

```
                    output.println("Ispis upita");
                    for (Student s : listaStudenata) {
                        if (studentTester.test(s) == true) {
                            output.println(s);
                            System.out.println(s);
                        }
                    }
                    System.out.println(brojKlijenta + "Kraj ispisa
upita");
                    output.println("Kraj ispisa upita");
                }
            } catch (IOException e) {
              System.out.println("Doslo je do greske na serveru");
            } finally {
                try {
                    soket.close();
                } catch (IOException ex) {
                    Logger.getLogger(Server.class
                          .getName()).log(Level.SEVERE, null, ex);
                }
            }
        }
    }
}
```

Klijent:

```
public class Klijent {
    public static void main(String[] args) throws IOException {
        System.out.println("Kljient je startovan...");
        Scanner sc = new Scanner(System.in);
        System.out.println("Unesite adresu servera");
        String adresaServera = sc.nextLine();
        try {
            Socket soket = new Socket(adresaServera, 9001);
                    BufferedReader input = new BufferedReader(new
InputStreamReader(soket.getInputStream()));
             PrintWriter output = new PrintWriter(soket.getOutput-
Stream(), true);
           //cita liniju: output.println("Uspesna konkekcija, kli-
jent broj" + brojKlijenta);
            System.out.println(input.readLine());
```

```
            //cita liniju: output.println("Unesite ime i sifru");
            System.out.println(input.readLine());
            String ime = sc.nextLine();
            String sifra = sc.nextLine();
            output.println(ime);
            output.println(sifra);
            //cita liniju: output.println("Uspesno logovanje");
            String odgovor = input.readLine();
            System.out.println(odgovor);
            if (odgovor.equals("Losi kredincijali")) {
                System.exit(0);
            }
              //cita liniju: output.println("Unesi smer, proesk i
godinu");
            System.out.println(input.readLine());
            String smer = sc.nextLine();
            double prosek = sc.nextDouble();
            int godina = sc.nextInt();
            output.println(smer);
            output.println(prosek);
            output.println(godina);
            //cita liniju: output.println("Ispis upita");
            System.out.println(input.readLine());
            while ((odgovor = input.readLine()) != null) {
                System.out.println(odgovor);
            }
            //cita liniju: output.println("Kraj ispisa upita");
            System.out.println(input.readLine());
        } catch (IOException e) {
            System.out.println("Greska u konekciji");
        }
    }
}
```

Primer izlaza:

Server:

```
Server je pokrenut...
1: se uspesno konektovao
1: Unosi ime i sifru
1 se uspesno ulogovao
1: Unosi prosek i godinu
1: smer=s, prosek=0.0, godina=2
1Ispis upita
```

```
Student{ime=Pera, prezime=Peric, smer=Softversko inzenjerstvo,
prosek=8.53, godinaUpisa=2}
Student{ime=Mika, prezime=Mikic, smer=Softversko inzenjerstvo,
prosek=6.7, godinaUpisa=3}
Student{ime=Zuca, prezime=Zutic, smer=Softversko inzenjerstvo,
prosek=6.53, godinaUpisa=2}
Student{ime=Vera, prezime=Veric, smer=Softversko inzenjerstvo,
prosek=7.7, godinaUpisa=1}
Student{ime=Luka, prezime=Lukic, smer=Softversko inzenjerstvo,
prosek=7.45, godinaUpisa=2}
Student{ime=Mika, prezime=Mikovic, smer=Softversko inzenjerstvo,
prosek=8.9, godinaUpisa=3}
1Kraj ispisa upita
2: se uspesno konektovao
2: Unosi ime i sifru
2 se uspesno ulogovao
2: Unosi prosek i godinu
2: smer=x, prosek=8.5, godina=0
2Ispis upita
Student{ime=Pera, prezime=Peric, smer=Softversko inzenjerstvo,
prosek=8.53, godinaUpisa=2}
Student{ime=Zika, prezime=Zikic, smer=Informatika, prosek=9.55,
godinaUpisa=2}
Student{ime=Laza, prezime=Lazic, smer=Informatika, prosek=10.0,
godinaUpisa=2}
Student{ime=Mara, prezime=Maric, smer=Informatika, prosek=10.0,
godinaUpisa=2}
Student{ime=Mika, prezime=Mikovic, smer=Softversko inzenjerstvo,
prosek=8.9, godinaUpisa=3}
2Kraj ispisa upita
```

Klijent 1:

```
Kljient je startovan...
Unesite adresu servera
localhost
Uspesna konkekcija, klijent broj1
Unesite ime i sifru
Mika
2
Uspesno logovanje
Unesi smer, proesk i godinu
s
0
2
```

```
Ispis upita
Student{ime=Pera, prezime=Peric, smer=Softversko inzenjerstvo,
prosek=8.53, godinaUpisa=2}
Student{ime=Mika, prezime=Mikic, smer=Softversko inzenjerstvo,
prosek=6.7, godinaUpisa=3}
Student{ime=Zuca, prezime=Zutic, smer=Softversko inzenjerstvo,
prosek=6.53, godinaUpisa=2}
Student{ime=Vera, prezime=Veric, smer=Softversko inzenjerstvo,
prosek=7.7, godinaUpisa=1}
Student{ime=Luka, prezime=Lukic, smer=Softversko inzenjerstvo,
prosek=7.45, godinaUpisa=2}
Student{ime=Mika, prezime=Mikovic, smer=Softversko inzenjerstvo,
prosek=8.9, godinaUpisa=3}
Kraj ispisa upita
```

Klijent 2:

```
Kljient je startovan...
Unesite adresu servera
localhost
Uspesna konkekcija, klijent broj2
Unesite ime i sifru
Vanja
8
Uspesno logovanje
Unesi smer, proesk i godinu
x
8.5
0
Ispis upita
Student{ime=Pera, prezime=Peric, smer=Softversko inzenjerstvo,
prosek=8.53, godinaUpisa=2}
Student{ime=Zika, prezime=Zikic, smer=Informatika, prosek=9.55,
godinaUpisa=2}
Student{ime=Laza, prezime=Lazic, smer=Informatika, prosek=10.0,
godinaUpisa=2}
Student{ime=Mara, prezime=Maric, smer=Informatika, prosek=10.0,
godinaUpisa=2}
Student{ime=Mika, prezime=Mikovic, smer=Softversko inzenjerstvo,
prosek=8.9, godinaUpisa=3}
Kraj ispisa upita
```

Zadatak 3: Napisati Java programe koji simuliraju rad servera i klijenata. Server ima listu turističkih aranžmana koje klijent može da pretražuje. Serveru može da pristupi samo jedan klijent i nakon usluživanja tog klijenta, server se zatvara.

Klasa TurističkiAranžman ima celobrojni aribut broj dana trajanja aranžmana, realan broj koji predstavlja cenu i tekstualni atribute za ime grada i ime turističke agencije koja ima taj aranžman u ponudi. Implementirati konstruktor i dodatne metode po potrebi.

Napisati funkcionalni interfejs koji će služiti za proveru aranžmana.

Java program Server. Implementirati Server koji ima listu od 10 turističkih aranžmana i čeka zahtev od klijenta. Kada se uspostavi konekcija, server šalje pozdravnu poruku klijentu oblika:

Uspostavljena konekcija sa serverom.

Nakon uspešne konekcije server čeka parametre za pretragu liste aranžmana. Klijent šalje serveru jedan podatak, tekst oblika minimalan broj dana aranžmana, maksimalan broj dana aranžmana, minimalna cena, maksimalna cena, deo imena mesta aranžmana. Svi podaci su razdvojeni jednim razmakom (npr. "5 10 1000 2000 ba" treba da vrati sve aranžmane koji traju između 5 i 10 dana (uključujući 5 i 10), koštaju između 1000 i 2000 evra (uključujući i granične cene) i ime mesta sadrži u sebi **ba** (npr Du**ba**i, **Ba**rselona, …)).

Nakon učitanih parametara, kreirati odgovarajući test koristeći lambda izraz. Pomoću kreiranog testa iz liste se izdvajaju aranžmani koji zadovoljavaju parametre pretrage i šalju se klijentu (jedan po jedan).

Ukoliko dođe do neispravnog unosa parametra, ispisati poruku i prekinuti konekciju.

Java program Klijent. Implementirati klijentsku stranu prethodno opisane komunikacije. Potrebno je odštampati na standardni izlaz sve poruke primljene od servera.

Rešenje:

Funkcionalni interfejs

```
public interface AranzmanTester {
    boolean test(TuristickiAranzman t);
}
```

```
Klasa TuristickiAranzman
public class TuristickiAranzman {
    private int brDanaAranzmana;
    private double cenaAranzmana;
    private String grad, imeAgencije;
```

```
   public TuristickiAranzman(int brDanaAranzmana, double cenaAran-
zmana, String grad, String imeAgencije) {
        this.brDanaAranzmana = brDanaAranzmana;
        this.cenaAranzmana = cenaAranzmana;
        this.grad = grad;
        this.imeAgencije = imeAgencije;
    }
    public int getBrDanaAranzmana() {
        return brDanaAranzmana;
    }
    public double getCenaAranzmana() {
        return cenaAranzmana;
    }
    public String getGrad() {
        return grad;
    }
    public String getImeAgencije() {
        return imeAgencije;
    }
    @Override
    public String toString() {
         return “Broj dana “ + getBrDanaAranzmana() + “ cena: “ +
getCenaAranzmana() + “ grad: “ + getGrad() + “ tur.agencija: “ +
getImeAgencije();
    }
}
```

Server:

```
public class Server {
    private static List<TuristickiAranzman> aranzmani;
    public static void main(String[] args) throws IOException {
        aranzmani = new ArrayList<TuristickiAranzman>();
        aranzmani.add(new TuristickiAranzman(5, 1250, “Madrid”,
“Laza-turs”));
        aranzmani.add(new TuristickiAranzman(7, 1750, “Moskva”,
“Pera-turs”));
        aranzmani.add(new TuristickiAranzman(9, 2235, “Milano”,
“Laza-turs”));
        aranzmani.add(new TuristickiAranzman(11, 2455, “Barselo-
na”, “Laza-turs”));
        aranzmani.add(new TuristickiAranzman(13, 2200,
“Budimpešta”, “Pera-turs”));
        aranzmani.add(new TuristickiAranzman(10, 2000, “Dubai”,
“Laza-turs”));
```

```
            aranzmani.add(new TuristickiAranzman(7, 1300, “Porto”,
“Pera-turs”));
        aranzmani.add(new TuristickiAranzman(9, 2235, “Rim”, “Ana-
turs”));
          aranzmani.add(new TuristickiAranzman(11, 1150, “Berlin”,
“Ana-turs”));
            aranzmani.add(new TuristickiAranzman(3, 500, “Malaga”,
“Ana-turs”));
        System.out.println(“Pokrećem server...”);
        ServerSocket server = new ServerSocket(9090);
        Socket soket = server.accept();
        Scanner in = new Scanner(soket.getInputStream());
        PrintWriter out = new PrintWriter(soket.getOutputStream(),
true);
        out.println(“Uspostavljena konekcija sa serverom...”);
        //String upit = in.nextLine();
        int minDana = in.nextInt();
        int maxDana = in.nextInt();
        double minCena = in.nextDouble();
        double maxCena = in.nextDouble();
        String deoImena = in.next();
        AranzmanTester tester;
            tester = k -> (k.getBrDanaAranzmana() >= minDana &&
k.getBrDanaAranzmana() <= maxDana
          && k.getCenaAranzmana() >= minDana && k.getCenaAranzmana()
<= maxCena
                    && k.getGrad().toLowerCase().contains(deoImena.
toLowerCase()));

        for (TuristickiAranzman t : aranzmani) {
            if (tester.test(t)) {
                out.println(t.toString());
                System.out.println(t.toString());
            }
        }
        out.println(“Kraj!”);
        server.close();
        in.close();
        out.close();
    }
}
```

Klijent:

```
public class Klijent {
    public static void main(String[] args) throws UnknownHostExcep-
tion, IOException {
        InetAddress ia = InetAddress.getByName(“localhost”);
        Socket soket = new Socket(ia, 9090);
        Scanner inputUser = new Scanner(System.in);
                BufferedReader incoming = new BufferedReader(new
InputStreamReader(soket.getInputStream()));
          PrintWriter outgoing = new PrintWriter(soket.getOutput-
Stream(), true);
           //cita liniju: out.println(“Uspostavljena konekcija sa
serverom...”);
        String fromServer = incoming.readLine();
        System.out.println(fromServer);

         System.out.println(“Unesite sledeće podatke:\n-Minimalan
broj dana aranžmana;”
                          + “\n-maksimalan broj dana aranžmana;\n-
minimalna cena;\n-maksimalna cena;\n-deo imena mesta aranžmana”);
        outgoing.println(inputUser.nextLine());

        while((fromServer = incoming.readLine()) != null){
            System.out.println(fromServer);
        }
        incoming.close();
        outgoing.close();
    }
}
```

Primer izlaza:

Server:

```
Pokrećem server...
Broj dana 11 cena: 2455.0 grad: Barselona tur.agencija: Laza-turs
Broj dana 10 cena: 2000.0 grad: Dubai tur.agencija: Laza-turs
```

Klijent:

```
Uspostavljena konekcija sa serverom...
Unesite sledeće podatke:
-Minimalan broj dana aranžmana;
-maksimalan broj dana aranžmana;
```

```
-minimalna cena;
-maksimalna cena;
-deo imena mesta aranžmana
10 15 2200 5000 ba
Broj dana 11 cena: 2455.0 grad: Barselona tur.agencija: Laza-turs
Broj dana 10 cena: 2000.0 grad: Dubai tur.agencija: Laza-turs
Kraj!
```

Zadatak 4: Napraviti Klijent/Server program za komunikaciju u realnom vremenu više korisnika (eng. chat room).

Rešenje: Ovaj zadatak će se rešavati postepenim proširivanjem osnovnih klasa. U prvom koraku će biti napravljen jednostavan server koji prihvata samo jednog klijenta i ispisuje sve što klijent pošalje na standardni izlaz. Konekcija se zatvara kada klijent pošalje poruku ".zdravo".

Server:

```
public class ChatServer {
    private Socket soket = null;
    private ServerSocket server = null;
    private DataInputStream citajPoruku = null;
    public ChatServer(int port) {
        try {
            server = new ServerSocket(port);
            System.out.println("Server je pokrenut...");
            System.out.println("Server ceka klijenta ...");
            soket = server.accept();
            System.out.println("Konektovan klijen...");
     citajPoruku = new DataInputStream(new BufferedInputStream(soket.
getInputStream()));
            boolean zavrsio = false;
            while (!zavrsio) {
                try {
                    String poruka = citajPoruku.readUTF();
                    System.out.println(poruka);
                    zavrsio = poruka.equals(".zdravo");
                } catch (IOException ioe) {
                    zavrsio = true;
                  }
               }
               soket.close();
               citajPoruku.close();
```

```
        } catch (IOException ioe) {
            System.out.println(ioe);
        }
    }
    public static void main(String args[]) {
        ChatServer server = new ChatServer(9001);
    }
}
```

Klijent:

```
public class ChatKlijent {
    private Socket soket = null;
    private DataInputStream upis = null;
    private DataOutputStream slanje = null;
     public ChatKlijent(String imeServera, int serverPort) throws
IOException {
        System.out.println("Uspostavljanje konekcije...");
        try {
            soket = new Socket(imeServera, serverPort);
        } catch (IOException ex) {
            System.err.println("Greska pri konekciji!");
        }
        System.out.println("Konektovan na soket: " + soket);
        upis = new DataInputStream(System.in);
        slanje = new DataOutputStream(soket.getOutputStream());
        String poruka = "";
        while (!poruka.equals(".zdravo")) {
            try {
                poruka = upis.readLine();
                slanje.writeUTF(poruka);
                slanje.flush();
            } catch (IOException ioe) {
             System.out.println("Greska prilikom slanja poruke!");
            }
        }
        upis.close();
        slanje.close();
        soket.close();
    }
    public static void main(String args[]) throws IOException {
        ChatKlijent client = new ChatKlijent("localhost", 9001);
    }
}
```

Primer izlaza:

Server	**Klijent**
Server je pokrenut... Server ceka klijenta ... Konektovan klijen... pera .zdravo	Uspostavljanje konekcije... Konektovan na soket: Socket[localhost,port=9001,localport=50657] pera .zdravo

Predstavljeno rešenje će biti nadograđeno tako da server može da prihvati novog klijenta kada završi sa jednim. Klijentska strana ostaje ista.

Server:

```
public class ChatServer implements Runnable{
    private Socket soket = null;
    private ServerSocket server = null;
    private DataInputStream citajPoruku = null;
    private Thread nitKlijent = null;
    public ChatServer(int port) {
        try {
            server = new ServerSocket(port);
            System.out.println("Server je pokrenut...");
            System.out.println("Server ceka klijenta ...");
            soket = server.accept();
            System.out.println("Konektovan klijent...");
            nitKlijent = new Thread(this);
            nitKlijent.start();
        } catch (IOException ioe) {
            System.out.println(ioe);
        }
    }
    @Override
    public void run()
   {  while (nitKlijent != null)
      {   try
         {  System.out.println("Server ceka klijenta ...");
            soket = server.accept();
            System.out.println("Konektovan klijent ....");
     citajPoruku = new DataInputStream(new BufferedInputStream(soket.
getInputStream()));
            boolean zavrsio = false;
            while (!zavrsio)
            {  try
               {  String poruka = citajPoruku.readUTF();
```

```
                System.out.println(poruka);
                zavrsio = poruka.equals(".zdravo");
             }
             catch(IOException ioe)
             {  zavrsio = true;  }
          }
          soket.close();
          citajPoruku.close();
       }
       catch(IOException ie)
       {  System.out.println("Acceptance Error: " + ie);  }
    }
  }
   public static void main(String args[]) {
       ChatServer server = new ChatServer(9001);
   }
}
```

Primer izlaza:

Server	**Klijent 1:**
Server je pokrenut...	Uspostavljanje konekcije...
Server ceka klijenta ...	Konektovan na soket: Socket[addr=localhost,port=9001,localport=50688]
Konektovan klijent...	**Poruka klijenta 1**
Poruka klijenta 1	**.zdravo**
.zdravo	
Server ceka klijenta ...	**Klijent 2:**
Konektovan klijent	Uspostavljanje konekcije...
Poruka klijenta 2	Konektovan na soket: Socket[addr=localhost,port=9001,localport=50688]
.zdravo	**Poruka klijenta 2**
Server ceka klijenta ...	**.zdravo**

Sledeći korak će biti nadogradnja servera tako da može da opslužuje više klijenata istovremeno. Klijentska strana i dalje ostaje ista. Kod server je nepotrebna promenljiva tipa Socket i fajl za čitanje pošto će se ona prebaciti u novu klasu. Dodaje se promenljiva tipa nove klase koja će predstavljati nit Servera kako bi server mogao da opslužuje više klijenata:

```
private ChatServerNit klijent = null;
```

Nova klasa ChatServerNit.

```
public class ChatServerNit extends Thread {
    private Socket soket = null;
    private ChatServer server = null;
    private int ID = -1;
    private DataInputStream citajPoruku = null;
    public ChatServerNit(ChatServer server, Socket soket) {
        this.server = server;
        this.soket = soket;
        ID = soket.getPort();
    }
    public void run() {
        System.out.println("Server nit " + ID + " radi.");
        while (true) {
            try {
                System.out.println(citajPoruku.readUTF());
            } catch (IOException ioe) {
            }
        }
    }
    public void otvori() throws IOException {
     citajPoruku = new DataInputStream(new BufferedInputStream(soket.
getInputStream()));
    }
    public void zatvori() throws IOException {
        soket.close();
        citajPoruku.close();
    }
}
```

Izmenjen server:

```
public class ChatServer implements Runnable{
    private ServerSocket server = null;
    private Thread nitKlijent = null;
    private ChatServerNit klijent = null;
    public ChatServer(int port) {
        try {
            server = new ServerSocket(port);
            System.out.println("Server je pokrenut...");
            nitKlijent = new Thread(this);
            nitKlijent.start();
```

```
        } catch (IOException ioe) {
            System.out.println(ioe);
        }
    }
    @Override
    public void run()
   {  while (nitKlijent != null)
      {   try
         {  System.out.println("Server ceka klijenta ...");
            Socket soket = server.accept();
            klijent = new ChatServerNit(this, soket);
            System.out.println("Konektovan klijent ....");
            klijent.otvori();
            klijent.start();
         }
         catch(IOException ie)
         {  System.out.println("Acceptance Error: " + ie);  }
      }
   }
    public static void main(String args[]) {
        ChatServer server = new ChatServer(9001);
    }
}
```

Poslednji korak je prosleđivanje poruka svim klijentima. U prethodnom rešenju samo je server dobijao sve poruke. Da bi to postigli, pravi se klasa ChatKlijetNit koja će slično kao i u slučaju servera biti zadužena za prosleđivanje poruka svim klijentima.

Server:

```
public class ChatServer implements Runnable {
    private ServerSocket server = null;
    private Thread nitKlijent = null;
    private ChatServerNit klijenti[] = new ChatServerNit[20];
    private int brKlijenata = 0;
    public ChatServer(int port) {
        try {
            server = new ServerSocket(port);
            System.out.println("Server je pokrenut...");
            nitKlijent = new Thread(this);
            nitKlijent.start();
        } catch (IOException ioe) {
            System.out.println(ioe);
        }
    }
```

```
    @Override
    public void run() {
        while (nitKlijent != null) {
            try {
                System.out.println("Server ceka klijenta ...");
                Socket soket = server.accept();
                if (brKlijenata < klijenti.length) {
                 System.out.println("Prihvacen klijent: " + soket);
                   klijenti[brKlijenata] = new ChatServerNit(this,
soket);
                    try {
                        klijenti[brKlijenata].otvori();
                        klijenti[brKlijenata].start();
                        brKlijenata++;
                    } catch (IOException ioe) {
                        System.out.println("Greska prilokom pokre-
tanja niti: " + ioe);
                    }
                } else {
                    System.out.println("Odbijen klijent: dostiguto
maksimalnih " + klijenti.length + " klijenata.");
                }
            } catch (IOException ie) {
                System.out.println("Greska pri povezivanju sa kli-
jentom: " + ie);
            }
        }
    }
    private int pronadjiKlijenta(int ID)
   {  for (int i = 0; i < brKlijenata; i++)
         if (klijenti[i].getID() == ID)
            return i;
      return -1;
   }
    public synchronized void handle(int ID, String poruka) {
        if (poruka.equals(".zdravo")) {
            klijenti[pronadjiKlijenta(ID)].posalji(".zdravo");
            remove(ID);
        } else {
            for (int i = 0; i < brKlijenata; i++) {
                klijenti[i].posalji(ID + ": " + poruka);
            }
        }
    }
```

```
    public synchronized void remove(int ID) {
        int pos = pronadjiKlijenta(ID);
        if (pos >= 0) {
            ChatServerNit zaZatvaranje = klijenti[pos];
             System.out.println("Removing client thread " + ID + "
at " + pos);
            if (pos < brKlijenata - 1) {
                for (int i = pos + 1; i < brKlijenata; i++) {
                    klijenti[i - 1] = klijenti[i];
                }
            }
            brKlijenata--;
            try {
                zaZatvaranje.zatvori();
            } catch (IOException ioe) {
               System.out.println("Greska pri zatvaranju klijenta:
" + ioe);
            }
            zaZatvaranje.stop();
        }
    }
    public static void main(String args[]) {
        ChatServer server = new ChatServer(9001);
    }
}
```

ChatServerNit:

```
public class ChatServerNit extends Thread {
    private Socket soket = null;
    private ChatServer server = null;
    private int ID = -1;
    private DataInputStream citajPoruku = null;
    private DataOutputStream prosledi Poruku = null;
    public ChatServerNit(ChatServer server, Socket soket) {
        this.server = server;
        this.soket = soket;
        ID = soket.getPort();
    }
    public void posalji(String poruka){
        try {
            prosledi Poruku.writeUTF(poruka);
            prosledi Poruku.flush();
```

```
        } catch (IOException ex) {
            System.out.println("Greska prilikom slanja poruke kli-
jenta " + ID);
            server.remove(ID);
            stop();
        }
    }
    public int getID(){
        return ID;
    }
    public void run() {
        System.out.println("Server nit " + ID + " radi.");
        while (true) {
            try {
                server.handle(ID, citajPoruku.readUTF());
            } catch (IOException ioe) {
                System.out.println("Greska pri citanju poruke kli-
jenta " + ID);
                server.remove(ID);
                stop();
            }
        }
    }
    public void otvori() throws IOException {
     citajPoruku = new DataInputStream(new BufferedInputStream(soket.
getInputStream()));
                    prosledi Poruku = new DataOutputStream(new
BufferedOutputStream(soket.getOutputStream()));
    }
    public void zatvori() throws IOException {
        soket.close();
        citajPoruku.close();
        prosledi Poruku.close();
    }
}
```

Klijent:

```
public class ChatKlijent implements Runnable{
    private Socket soket = null;
    private DataInputStream upis = null;
    private DataOutputStream slanje = null;
    private Thread nitKlijent = null;
    private ChatKlijentNit klijent = null;
```

```
     public ChatKlijent(String imeServera, int serverPort) throws
IOException {
        System.out.println("Uspostavljanje konekcije...");
        try {
            soket = new Socket(imeServera, serverPort);
        } catch (IOException ex) {
            System.err.println("Greska pri konekciji!");
        }
        System.out.println("Konektovan na soket: " + soket);
        start();
    }
    public void run() {
        while (nitKlijent != null) {
            try {
                slanje.writeUTF(upis.readLine());
                slanje.flush();
            } catch (IOException ioe) {
                System.out.println("Sending error: " + ioe.getMes-
sage());
                stop();
            }
        }
    }
    public void rukujPorukom(String poruka) {
        if (poruka.equals(".zdravo")) {
            System.out.println("Dovidjenja...");
            stop();
        } else {
            System.out.println(poruka);
        }
    }
    public void start() throws IOException {
        upis = new DataInputStream(System.in);
        slanje = new DataOutputStream(soket.getOutputStream());
        if (nitKlijent == null) {
            klijent = new ChatKlijentNit(this, soket);
            nitKlijent = new Thread(this);
            nitKlijent.start();
        }
    }
    public void stop() {
        try {
            nitKlijent.stop();
            nitKlijent = null;
```

```
            upis.close();
            slanje.close();
            soket.close();
            klijent.zatvori();
            klijent.stop();
        } catch (IOException ex) {
          Logger.getLogger(ChatKlijent.class.getName()).log(Level.
SEVERE, null, ex);
        }
    }
    public static void main(String args[]) throws IOException {
        ChatKlijent client = new ChatKlijent("localhost", 9001);
    }
}
```

ChatKlijentNit:

```
public class ChatKlijentNit extends Thread {
    private Socket soket = null;
    private ChatKlijent klijent = null;
    private DataInputStream citajPoruku = null;
    public ChatKlijentNit(ChatKlijent klijent, Socket soket) {
        this.klijent = klijent;
        this.soket = soket;
        otvori();
        start();
    }
    public void otvori() {
        try {
                citajPoruku = new DataInputStream(soket.getInput-
Stream());
        } catch (IOException ioe) {
             System.out.println("Greska prilikom citanja poruke: "
+ ioe);
            klijent.stop();
        }
    }
    public void zatvori() {
        try {
            citajPoruku.close();
        } catch (IOException ioe) {
             System.out.println("Greska prilikom zatvaranja fajla:
" + ioe);
        }
    }
```

```
    public void run() {
        while (true) {
            try {
                klijent.rukujPorukom(citajPoruku.readUTF());
            } catch (IOException ioe) {
               System.out.println("Greskar: " + ioe.getMessage());
                klijent.stop();
            }
        }
    }
}
```

Zadaci za samostalno rešavanje:

Zadatak 5: Napisati Java programe koji simuliraju rad servera i klijenata. Server ima listu kurseva koje klijent može da pretražuje. Serveru može da pristupi samo jedan klijent i nakon usluživanja tog klijenta, server se zatvara.

Klasa **Kursevi** ima realan broj koji predstavlja cenu, ceo broj koji predstavlja maksimalnu veličinu grupe, nivo (enum: pocetni, srednji, napredni) i tekstualni atribut za ime kursa u jednoj školi. Implementirati konstruktor i dodatne metode po potrebi.

Napisati funkcionalni interfejs koji će služiti za pretragu kurseva.

Java program Server. Implementirati Server koji ima listu od 8 kurseva i čeka zahtev od klijenta. Kada se uspostavi konekcija, server šalje pozdravnu poruku klijentu oblika:

```
Uspostavljena konekcija sa serverom.
```

Nakon uspešne konekcije server čeka parametre za pretragu liste kurseva. Klijent šalje serveru jedan podatak, tekst oblika maksimalna cena, maksimalna veličina grupe, nivo (ceo broj I to: 1 za pocetni, 2 za srednji ili 3 za napredni nivo), deo imena kursa (za ukucano *ka* potrebno je pronaći i kurs BazePodata*ka* i Matemati*ka*). Svi podaci su razdvojeni jednim razmakom.

Nakon učitanih parametara, kreirati odgovarajući test koristeći lambda izraz. Pomoću kreiranog testa iz liste se izdvajaju kursevi koji zadovoljavaju parametre pretrage i šalju se klijentu (jedan po jedan).

Ukoliko dođe do neispravnog unosa parametra, ispisati poruku i prekinuti konekciju.

Java program Klijent. Implementirati klijentsku stranu prethodno opisane komunikacije. Potrebno je odštampati na standardni izlaz sve poruke primljene od servera.

Zadatak 6: Napisati Java programe koji simuliraju rad servera i klijenata. Server ima spisak artikla u prodavnici koje klijent može da pretražuje. Serveru može da pristupi samo jedan klijent i nakon usluživanja tog klijenta, server se zatvara.

Klasa **Artikal** ima realan broj koji predstavlja cenu, ceo broj koji predstavlja raspoloživu količinu, i tekstualni atribut za naziv artikla. Implementirati konstruktor i dodatne metode po potrebi.

Napisati funkcionalni interfejs koji će služiti za pretragu artikla.

Java program Server. Implementirati Server koji ima listu od 10 artikla i čeka zahtev od klijenta. Kada se uspostavi konekcija, server šalje pozdravnu poruku klijentu oblika:

Uspostavljena konekcija sa serverom.

Nakon uspešne konekcije server čeka parametre za pretragu liste artikla. Klijent šalje serveru jedan podatak, tekst oblika maksimalna cena, maksimalna cena, željena količina i deo imena kursa (za ukucano *ka* potrebno je pronaći npr. Pavla*ka*, *Ka*jmak, Ma*ka*rone, …). Svi podaci su razdvojeni jednim razmakom.

Nakon učitanih parametara, kreirati odgovarajući test koristeći lambda izraz. Pomoću kreiranog testa iz liste se izdvajaju artikli koji zadovoljavaju parametre pretrage i šalju se klijentu (jedan po jedan).

Ukoliko dođe do neispravnog unosa parametra, ispisati poruku i prekinuti konekciju.

Java program Klijent. Implementirati klijentsku stranu prethodno opisane komunikacije. Potrebno je odštampati na standardni izlaz sve poruke primljene od servera.

Pitanja

1. Šta su soketi?
2. Kako se uspostavlja komunikacija pomoću soketa u Java programskom jeziku?
3. Kada se zatvara konekcija između dva programa?
4. Koji soketi se koriste za uspostavljanje klijent-server komunikacije?
5. Koje su osnovne operacije koje soket može da izvrši? Koje od njih su neophodne samo za server soket?
6. Koja klasa u Java programskom jeziku se koristi za server soket?
7. Šta je argument konstruktora klase ServerSocket u Java programskom jeziku?
8. Koja metoda služi za prihvatanje zahteva i uspostavljanje konekcije sa klijentom?
9. Koji je povratni tip metod accept() klase ServerSocket?
10. Koja klasa u Java programskom jeziku se koristi za klijentski soket?
11. Šta su argumenti konstruktora klase Socket?

IV
Uvod u Spring okruženje

Spring okruženje je kreirano sa veoma jasnim ciljem – da se pojednostavi razvoj Java EE aplikacija. Pružena je alternativa znatno težim Java *enterprise* tehnologijama, poput EJB (engl. *Enterprise JavaBeans*). Spring je ponudio znatno lakši programerski model u poređenju sa EJB. Spring se zasniva na POJO (engl. *Plain Old Java Object*), zajedno sa konceptima injekcije zavisnosti (engl. *Dependency Injection – DI*) i aspektno orijentisanog programiranja (AOP), koji su značajno doprineli uspehu Spring okruženja. Vremenom, EJB i J2EE (engl. *Java 2 Enterprise Edition*) su takođe počeli da nude koncepte POJO, DI i AOP, inspirisani uspehom Spring-a. Međutim, Spring i dalje ide korak ispred njih, sa inovacijama poput razvoja mobilnih aplikacija, integracije različitih API-ja (engl. *Application Programming Interface – API*), NoSQL baza i novijih tehnologija poput *cloud computing* i *big data*.

Spring je open-source platforma, koja je dostupna na adresi: https://spring.io/projects/spring-framework. Spring nije predviđen isključivo za razvoj serverske strane, već svaka Java aplikacija može imati koristi u smislu jednostavnosti, lakoće testiranja i održavanja, kao i labave sprege koju Spring nudi. Labava sprega (engl. *loose coupling*) je termin koji se pre svega odnosi na pristup povezivanju komponenti nekog sistema na takav način da te komponente minimalno zavise jedna od druge. Spring koristi reč zrno (engl. *bean*) kada se govori o komponentama aplikacije, međutim, komponente mogu biti i bilo koji tip POJO.

Spring može da uradi puno stvari – to je sistem za organizaciju Java koda, upotrebom XML fajlova ili anotacija. Međutim, skoro sve mogućnosti koje Spring pruža se zasnivaju na nekoliko fundamentalnih ideja, koje imaju za cilj da pojednostave Java programiranje. Kako Spring olakšava razvoj Java aplikacija? Spring koristi četiri osnovne strategije kako bi smanjio kompleksnost Java programiranja:

- Lako i minimalno invazivno programiranje sa POJO.
- Labava sprega kroz injekciju zavisnosti i upotrebu interfejsa.
- Deklarativno programiranje kroz aspekte.
- Eliminisanje viška koda koji se ponavlja (engl. *boilerplate code*) upotrebom aspekata i šablona.

Skoro sve funkcionalnosti Spring okruženja se mogu povezati sa jednom ili više od ove četiri strategije. U nastavku ovog poglavlja ćemo se osvrnuti na svaku od ovih strategija.

4.1. POJO

Skoro svaki Java programer se susreo sa nekim okruženjem koje ga ograničava u razvoju aplikacija time što forsira proširenje jedne od klasa tog okruženja, ili implementaciju nekog od interfejsa ugrađenih u okruženje. Programiranje u nekom od takvih okruženja se često naziva invazivno programiranje, i može se naći u velikom broju programerskih okvira poput EJB, ranijih verzija Struts, WebWork i brojnih drugih. Spring u najvećoj mogućoj meri ne meša svoj API u kod aplikacije. Umesto toga, klase u Spring aplikaciji vrlo često nemaju nikakvu indikaciju da ih Spring koristi. U najčešćem slučaju, klase su samo anotirane sa nekom od Spring anotacija – osim toga, u pitanju su najobičniji čisti Java objekti - POJO. Kako bismo ilustrovali problem, posmatrajmo najprostiju *Hello world* klasu u Spring okviru, koja ima samo jednu metodu sayHello(), koja vraća string "Hello World".

```
package primer.spring;
public class HelloWorld {
       public String sayHello() {
              return "Hello World";
       }
}
```

Kao što se može videti, ovo je najprostija obična Java klasa – POJO. Ništa specijalno ne nagoveštava da je u pitanju Spring komponenta. Zbog toga se za Spring kaže da koristi neinvazivni model programiranja, odnosno ova klasa bi mogla da funkioniše kako u Spring aplikaciji, tako i u bilo kojoj drugoj Java aplikaciji (koja nije Spring). Iako POJO sam po sebi ima vrlo jednostavnu i nama uobičajenu formu, Spring ga čini mnogo moćnijim na nekoliko načina. Jedan način je da Spring omogućava povezivanje objekata pomoću injekcije zavisnosti.

4.2. Injektovanje zavisnosti (DI)

Sam izraz injektovanje zavisnosti (DI) može zvučati kompleksno i zastrašujuće, i stvoriti utisak da je u pitanju neka veoma komplikovana tehnika programiranja. Na sreću, DI uopšte nije komplikovan koliko zvuči. Sama primena DI značajno pojednostavljuje kod, i čini ga razumljivijim i lakšim za testiranje. Šta je u stvari i kako radi DI?

Svaka netrivijalna aplikacija u Javi se sastoji iz većeg broja klasa koje sarađuju zajedno kako bi se obavila funkcija programa. U slučaju tradicionalnog programiranja, svaki objekat bi bio zadužen da dobavi reference na objekte sa kojima sarađuje

(odnosno zavisnosti). To dovodi do tesne sprege između objekata, kao i koda koji je izuzetno težak za testiranje. Ilustrovaćemo problem na primeru klase Vitez prikazane u nastavku.

```
public interface Vitez {

  void zapocniAvanturu();

}
```

```
public class VitezKojiSpasavaDamu implements Vitez {

  private AvanturaSpasavanjaDame avantura;

  public VitezKojiSpasavaDamu() {
    this.avantura = new AvanturaSpasavanjaDame(); //tesna sprega
  }

  public void zapocniAvanturu() {
    avantura.zapocniAvanturu();
  }

}
```

Avanture su opisane interfejsom Avantura, i klasama različitih avantura koje implementiraju ovaj interfejs, na sledeći način.

```
public interface Avantura {

  void zapocniAvanturu();

}
public class AvanturaSpasavanjaDame implements Avantura {

  public void zapocniAvanturu() {
    System.out.println("Zapocinje avantura spasavanja dame.");
  }

}
```

U priloženom primeru, VitezKojiSpasavaDamu kreira svoju sopstvenu avanturu tipa AvanturaSpasavanjaDame u svom konstruktoru. Ovakav način programiranja čini da je VitezKojiSpasavaDamu u tesnoj sprezi sa avanturom spašavanja dame (vitez zavisi od avanture) i samim tim sužava manevarski prostor viteza. Ukoliko je potrebno spašavanje dame u nevolji, ovaj vitez je tu, ali ukoliko je potrebna bilo koja druga herojska avantura, poput borbe sa zmajem, ovaj vitez neće moći da je prihvati.

Sprega objekata u programu je mač sa dve oštrice. Sa jedne strane, program sa tesnom spregom i visokim nivoom zavisnosti je težak za testiranje, razumevanje i ponovnu upotrebu. Sa druge strane, određen nivo sprege je neophodan, jer program bez ikakve sprege i zavisnosti između klasa ne bi mogao da uradi ništa. Kako bi moglo išta da se uradi u programu, klase moraju na neki način znati za postojanje drugih klasa. Sprega je dakle neophodna, ali se mora držati pod kontrolom i pažljivo pratiti.

U slučaju primene DI, objektima se dodeljuju zavisnosti za vreme njihovog kreiranja, a te zavisnosti dodeljuje neka treća strana koja koordiniše svim objektima u sistemu. Od objekata se više ne očekuje da kreiraju ili dobave reference objekata od kojih zavise. Kako bismo ilustrovali ovaj pojam, posmatrajmo klasu HrabriVitez, koji nije samo hrabar već je i dovoljno fleksibilan da može da učestvuje u bilo kojoj avanturi na koju naiđe. Za razliku od klase VitezKojiSpasavaDamu, HrabriVitez ne kreira sam svoju avanturu, već je dobija u vreme kreiranja kao argument konstruktora. Ovaj tip injekcije zavisnosti je još poznat pod nazivom injekcija u konstruktoru (engl. *constructor injection*).

```
public class HrabriVitez implements Vitez {

  private Avantura avantura;

  public HrabriVitez(Avantura avantura) {
    this.avantura = avantura;
  }

  public void zapocniAvanturu() {
    avantura.zapocniAvanturu();
  }

}
```

Dodatno možemo primetiti da je avantura koja se dodeljuje tipa interfejsa Avantura. To je interfejs koji implementiraju sve avanture, koje se mogu proslediti svugde gde se očekuje tip Avantura. To znači da se HrabriVitez može uključiti u avanturu spašavanja dame, borbe sa zmajem, ili bilo kojom drugom avanturom koja implementira interfejs Avantura. HrabriVitez dakle nije u tesnoj sprezi ni sa jednom specifičnom

implementacijom interfejsa Avantura. Zapravo, što se tiče viteza, njemu je svejedno koja će mu se avantura dodeliti, sve dok ta avantura implementira interfejs Avantura. Ovo je ujedno i glavna prednost DI – labava sprega putem interfejsa. Ukoliko neki objekat čuva svoje zavisnosti u obliku njihovih interfejsa (a ne kroz neku konkretnu implementaciju interfejsa), zavisnost se može zameniti sa nekom drugom (koja takođe implementira dati interfejs), a da objekat koji zavisi ne primeti razliku.

Sada kada je klasa HrabriVitez napisana na takav način da se vitezu može proslediti bilo koja avantura, kako možemo specificirati koju tačno avanturu želimo da mu damo? Pretpostavimo sada da želimo hrabrom vitezu da dodelimo avanturu borbe sa zmajem. U tom slučaju, potrebna nam je klasa AvanturaBorbeSaZmajem koja će proširiti interfejs Avantura.

```
import java.io.PrintStream;

public class AvanturaBorbeSaZmajem implements Avantura {

  private PrintStream izlaz;

  public AvanturaBorbeSaZmajem(PrintStream izlaz) {
    this.izlaz = izlaz;
  }

  public void zapocniAvanturu() {
    izlaz.println("Zapocinje avantura borbe sa zmajem!");
  }

}
```

Klasa AvanturaBorbeSaZmajem će služiti kao avantura koju ćemo injektovati hrabrom vitezu. To je moguće, pošto AvanturaBorbeSaZmajem implementira interfejs Avantura, što je u skladu sa konstruktorom klase HrabriVitez koji kao tip argumenta očekuje objekat klase koja implementira interfejs Avantura. Može se još primetiti da je u klasi AvanturaBorbeSaZmajem običan System.out.println() zamenjen objektom tipa PrintStream koji se injektuje kroz konstruktor. Sada ostaje pitanje – kako možemo da postavimo objekat klase AvanturaBorbeSaZmajem hrabrom vitezu, odnosno kako možemo da postavimo PrintStream objektu klase AvanturaBorbeSaZmajem?

Postupak kreiranja veza i zavisnosti između komponenti aplikacije se često naziva ožičavanje ili povezivanje, odnosno engl. *wiring*. U Spring okruženju postoji više načina za povezivanje komponenti, ali najčešći pristup je putem XML konfiguracionog fajla. Priimer jednog ovakvog fajla, vitez.xml, služi za povezivanje objekata klasa HrabriVitez, AvanturaBorbeSaZmajem i PrintStream zajedno.

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans.
xsd">

  <bean id="vitez" class="pj.vitezovi.HrabriVitez">
    <constructor-arg ref="avantura" />
  </bean>

  <bean id="avantura" class="pj.vitezovi.AvanturaBorbeSaZmajem">
    <constructor-arg value="#{T(System).out}" />
  </bean>

</beans>
```

U ovom konfiguracionom fajlu su HrabriVitez i AvanturaBorbeSaZmajem deklarisani kao zrna u Spring okruženju. U slučaju zrna HrabriVitez, u konstruktoru se kao argument prosleđuje referenca na zrno sa imenom avantura. Sa druge strane, deklaracija zrna avantura koristi SpEL (engl. *Spring Expression Language*) za prosleđivanje System.out konstruktoru klase AvanturaBorbeSaZmajem. SpEL se odnosi na jezik koji podržava upite i manipulaciju grafom objekata za vreme izvršavanja programa.

Kao što smo ranije napomenuli, postoji više načina za ožičavanje zrna. XML konfiguracija je standardni i najčešći način konfiguracije zrna. Ukoliko međutim takav način konfiguracije nekome ne odgovara postoji još nekoliko načina koje Spring nudi. Na primer, sledeći Java kod u potpunosti odgovara prethodno navedenom XML fajlu. U ovom slučaju, koriste se anotacije, o kojima će biti reči kasnije.

```
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;

import pj.vitezovi.HrabriVitez;
import pj.vitezovi.Vitez;
import pj.vitezovi.Avantura;
import pj.vitezovi.AvanturaBorbeSaZmajem;

@Configuration
public class VitezoviKonfiguracija {
```

```
  @Bean
  public Vitez vitez() {
    return new HrabriVitez(avantura());
  }

  @Bean
  public Avantura avantura() {
    return new AvanturaBorbeSaZmajem(System.out);
  }

}
```

Potpuno je svejedno da li se koristi XML ili Java konfiguracija, prednosti DI su u oba slučaja iste. Iako HrabriVitez zavisi od avanture, on nije programiran sa bilo kakvim predznanjem kakav će tip avanture dobiti, kao ni odakle će ta avantura doći. Na isti način, AvanturaBorbeSaZmajem ne zna koji će tip PrintStream dobiti, kao ni odakle će on doći. Jedini koji ima sve ove informacije je Spring, koji kroz svoju konfiguraciju (XML ili Java) povezati sve delove zajedno. Na taj način moguće je po potrebi izmeniti zavisnosti, bez ikakve intervencije u kodu samih klasa. Prethodno opisani primer služi samo da pokaže pristup koji Spring koristi u ožičavanju zrna – nije potrebno udubljivati se u detalje sad, pošto će konfiguracija biti detaljno opisana u kasnijim poglavljima, gde će biti pokazano i kako se može omogućiti samom Spring okruženju da automatski otkrije zrna i poveže ih međusobno (što je zapravo i najčešće korišćeni način konfiguracije).

Sada, nakon deklaracije zrna i definisanja odnosa između njih, potrebno je učitati XML konfiguracioni fajl i pokrenuti aplikaciju. U Spring aplikaciji, koristi se kontekst aplikacije (engl. *application context*) za učitavanje definicija zrna i njihovo međusobno povezivanje. Springov kontekst aplikacije je u potpunosti odgovoran za kreiranje i ožičavanje objekata koji čine aplikaciju. Postoji nekoliko različitih implementacija aplikacionog konteksta, koji se razlikuju samo u načinu kako se učitava konfiguracija. Pošto je u našem primeru konfiguracija zrna definisana u XML fajlu vitez.xml, odgovarajući oblik konteksta aplikacije bi bio ClassPathXmlApplicationContext. Ovaj kontekst učitava jedan ili više XML fajlova koji se nalaze na putanji aplikacije (engl. *classpath*). Sledeći kod pokazuje kako bi izgledao main() metod aplikacije koji koristi ovaj kontekst za učitavanje fajla vitez.xml i dohvatanje reference na objekat klase Vitez.

```
import org.springframework.context.support.
                    ClassPathXmlApplicationContext;
public class VitezMain {

  public static void main(String[] args) throws Exception {
    ClassPathXmlApplicationContext context =
        new ClassPathXmlApplicationContext(
            "pj/vitezovi/springconfig/vitez.xml");
    Vitez vitez = context.getBean(Vitez.class);
    vitez.zapocniAvanturu();
    context.close();
  }
}
```

U ovom slučaju, unutar main() metode se kreira Springov aplikacioni kontekst baziran na vitez.xml fajlu. Ovaj kontekst se nakon toga koristi kao fabrika za dobijanje zrna čiji ID je vitez. Kada je referenca na Viteza dobijena na ovaj način, poziva se metoda zapocniAvanturu() koja će pokrenuti avanturu koja je u XML fajlu dodeljena vitezu. Potrebno je obratiti pažnju da čak ni VitezMain klasa nema pojma o tipu avanture koji je dodeljen vitezu. Šta više, ova klasa čak nema pojma ni sa kojom implementacijom interfejsa Vitez radi (u ovom slučaju u pitanju je naravno HrabriVitez). Jedini koji zna tačne implementacije klasa je vitez.xml. Ovo je naravno samo uvod u DI – kroz dalja poglavlja ćemo detaljnije videti kako se sve koristi injektovanje zavisnosti.

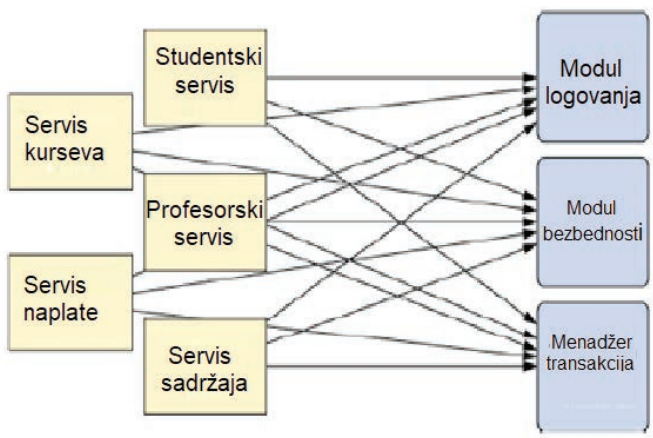
4.3. Primena aspekata

Aspekti su, pored DI, drugi glavni stub na kojima se zasniva Spring. Aspektno orijentisano programiranje (AOP) se koristi za izdvajanje funkionalnosti koju je potrebno koristiti kroz celu aplikaciju. AOP se najčešće definiše kao tehnika koja promoviše razdvajanje briga u softverskom sistemu. O kojim brigama je ovde reč? Sistem se najčešće sastoji od više komponenti, a svaka komponenta je zadužena za određen deo funkcionalnosti celokupnog programa. Problem se javlja u tome da sve ove komponente često moraju da brinu o dodatnim obavezama koje obično nemaju nikakve veze sa njihovom glavnom funkcionalnosti. To su najčešće neki sistemski servisi poput logovanja, menadžmenta transakcija ili sigurnosti, koji vrlo često nađu mesto u komponentama koje imaju potpuno drugu funkcionalnost.

Ovi servisi se obično protežu kroz više komponenti u sistemu, čime se kod duplira. To obično znači da ukoliko želimo da promenimo način rada nekog od ovih servisa, potrebno je izmeniti kod u više komponenti – na primer, ukoliko se promeni način logovanja, u svim klasama u kojima se radi logovanje bi bilo potrebno izmeniti

odgovarajuće delove koda. To je samo jedan problem. Drugi problem je da je programski kod klasa praktično zagađen kodom koji nije u liniji sa osnovnom funkcijom tih klasa. Prosto gledano, metoda koja dodaje jedan unos u npr. listu kontakata treba da brine samo o tome kako da doda ime, adresu i broj telefona u odgovarajuću listu kontakata – ne treba da brine o tome da li je logovana, sigurna ili da li se kompletan upis u bazu radi u okviru transakcije ili ne.

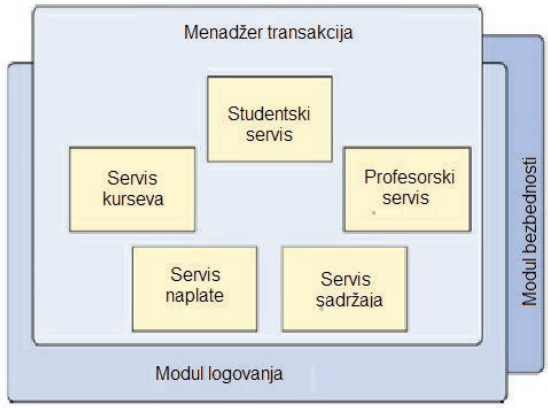
Na slici 4.1 ilustrovana je kompleksnost ovog problema. Objekti sa leve strane, koji su bitni za glavnu funkcionalnost programa i biznis logiku softvera su previše intimno povezani sa sistemskim servisima sa desne strane. Ne samo da svaki od objekata sa leve strane zna da je logovan, osiguran i ubačen u transakcioni kontekst, već, što je još gore, svaki od tih objekata je odgovoran da obavlja ove servise sam za sebe. Sami pozivi ovih servisa su raštrkani po modulima kojima logovanje ili sigurnost nisu glavna briga. Ukoliko bi bilo potrebno promeniti na primer logovanje, u svakoj od klasa sa leve strane bi bilo neophodno uneti izmene u kodu.



Slika 4.1. Pozivi sistemskih servisa u modulima gde im to nije osnovna funkcionalnost

AOP omogućava modularizaciju ovakvih servisa, i njihovo deklarativno primenjivanje u aplikaciji nad tačno određenim komponentama na koje treba da imaju uticaj. Kao rezultat, komponente više nemaju dodatni kod koji se bavi na primer logovanjem, već se mogu fokusirati na svoju glavnu funkconalnost. Zapravo, komponente su primenom aspekata potpuno neupućene u to koji sistemski servisi utiču na njih. Ukratko, aspekti osiguravaju da POJO ostanu čisti i jednostavni.

Aspekti se mogu zamisliti kao pokrivači koji pokrivaju više komponenti u aplikaciji, kao što je prikazano na slici 4.2. Aplikacija se sastoji od modula u kojima je implementirana funkcionalnost programa. Ti moduli se zatim prekrivaju nivoima dodatnih servisa, koji se deklarativno primenjuju u modulima na takav način da jezgro aplikacije čak ne zna ni da postoje.



Slika 4.2. AOP i primena servisa na komponente na koje trebaju da imaju uticaj

Koncept aspekata je moćan, ali pošto se većina mlađih programera nije susrela sa ovakvim načinom programiranja do sada, najlakše je koncept prikazati na konkretnom primeru. Možemo ponovo da posmatramo primer vitezova iz prethodnog poglavlja, i demonstriramo kako se aspekti primenjuju u Spring okruženju.

Pretpostavimo da je u aplikaciji vitez potrebno dodati logovanje. Kako bismo ostali u duhu vremena vitezova, poznato je da su dela vitezova opisivali pesnici i pevači poznati pod imenom bardovi. Pretpostavimo sada da želimo da logujemo našeg hrabrog viteza, odnosno njegov odlazak u avanturu i njegov povratak iz avanture, upotrebom servisa logovanja – barda. Potrebno je implementirati dodatnu klasu bard koja će se koristiti za logovanje.

```
import java.io.PrintStream;

public class Bard {

  private PrintStream izlaz;

  public Bard(PrintStream izlaz) {
    this.izlaz = izlaz;
  }

  public void pevajPreAvanture() {
    izlaz.println(“La la la, nas vitez je veoma hrabar!”);
  }
```

```
  public void pevajPosleAvanture() {
    izlaz.println("La la la, hrabri vitez je bio u avanturi!");
  }

}
```

Klasa Bard je veoma prosta klasa, sa samo dve metode. Metoda pevajPreAvanture() je predviđena da se poziva pre nego što vitez započne avanturu, a metoda pevajPosleAvanture() bi trebalo da se poziva nakon završetka avanture. U oba slučaja, bard treba da opeva vitezova dela kroz PrintStream koji se injektuje kroz konstruktor klase Bard. Ovo je zapravo klasičan primer logovanja – samo predstavljen u takvom kontekstu gde će biti potpuno jasno zašto tradicionalan način programiranja servisa logovanja nije pogodan (što možda nije toliko očigledno kada se gleda klasa koja radi klasično logovanje).

Na prvi pogled, ubacivanje opevavanja dela viteza u kod bi trebalo da bude krajnje jednostavno – potrebno je samo da injektujemo barda u viteza. Ukoliko se urade odgovarajuće izmene klase HrabriVitez kako bi mogao da koristi usluge servisa – barda, dolazi se do sledećeg koda.

```
public class HrabriVitez implements Vitez {

  private Avantura avantura;
  private Bard bard;

  public HrabriVitez(Avantura avantura, Bard bard) {
    this.avantura = avantura;
    this.bard = bard;
  }

  public void zapocniAvanturu() {
    bard.pevajPreAvanture();
    avantura.zapocniAvanturu();
    bard.pevajPosleAvanture();
  }

}
```

To bi trebalo da bude to, potrebno je još samo pronaći konfiguracioni XML i deklarisati zrno tipa Bard, i nakon toga ga injektovati kroz konstruktor zrna HrabriVitez, zar ne? Ali, ovde je potrebno zastati i razmisliti. Nešto se ovde ne uklapa baš najbolje. Da li je zaista vitezova briga da vodi računa o svom bardu? Zar nije logično

da bi bard trebalo da radi svoj posao bez da ga neko pita da to radi? Ili još gore, da vodi računa o tome kako i kada to radi? Jer, upravo to je bardov posao, da peva o delima hrabrog viteza. Gde je onda svrha da vitez mora stalno da podseća barda da radi svoj posao?

Ne samo to, pošto je neophodno da vitez bude svestan postojanja barda, primorani smo da injektujemo barda kroz konstruktor klase HrabriVitez. Ovo ne samo da komplikuje i zagađuje kod klase HrabriVitez, već donosi čitav niz pitanja. Šta raditi u slučaju da je potrebno da postoji vitez koji nema svog barda. Odnosno, prevedeno na programski jezik Java, šta se dešava ukoliko je Bard null? Da li je potrebno ubaciti logiku za proveru null vrednosti kako bi se pokrio taj slučaj (pošto u suprotnom dolazi do izbacivanja izuzetka tipa NullPointerException)? Klasa HrabriVitez postaje sve složenija i složenija, naročito u slučaju da se mora obraditi i scenario sa null vrednosti.

Upotrebom AOP, moguće je deklarisati da bard treba da peva o vitezovim hrabrim delima na takav način da se vitez oslobodi direktnog petljanja sa metodama klase Bard. Kako bi se Bard pretvorio u aspekt, potrebno je samo deklarisati ga kao aspekt u jednom od konfiguracionih fajlova. Na primer, u XML fajlu vitez.xml se može dodati deo koji deklariše da je Bard aspekt na sledeći način.

```
<?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:aop="http://www.springframework.org/schema/aop"
  xsi:schemaLocation="http://www.springframework.org/schema/aop
      http://www.springframework.org/schema/aop/spring-aop.xsd
            http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans.
xsd">

 <bean id="vitez" class="pj.vitezovi.HrabriVitez">
    <constructor-arg ref="avantura" />
  </bean>

  <bean id="avantura" class="pj.vitezovi.AvanturaBorbeSaZmajem">
    <constructor-arg value="#{T(System).out}" />
  </bean>

  <bean id="bard" class="pj.vitezovi.Bard">
    <constructor-arg value="#{T(System).out}" />
  </bean>
```

```
  <aop:config>
    <aop:aspect ref="bard">
      <aop:pointcut id="pocetakAvanture"
          expression="execution(* *.zapocniAvanturu(..))"/>

      <aop:before pointcut-ref="pocetakAvanture"
          method="pevajPreAvanture"/>

      <aop:after pointcut-ref="pocetakAvanture"
          method="pevajPosleAvanture"/>
    </aop:aspect>
  </aop:config>

</beans>
```

U ovom slučaju se koristi Springov aop imenski prostor, kako bi se zrno tipa Bard definisalo kao aspekt. Prvo se deklariše Bard kao zrno. Nakon toga, to zrno se referencira iz <aop:aspect> elementa. Dalje, upotrebom <aop:before> elementa deklariše se da će, pre izvršavanja metode zapocniAvanturu() biti izvršena bardova metoda pevajPreAvanture(). To se u terminologiji AOP naziva savet pre (engl. *before advice*). Na sličan način, upotrebom <aop:after> elementa deklariše se da će, nakon izvršavanja metode zapocniAvanturu() biti izvršena bardova metoda pevajPosleAvanture(). To se u terminologiji AOP naziva savet posle (engl. *after advice*).

U oba slučaja, koristi se pointcut-ref atribut koji referencira tačku preseka (engl. *pointcut*) pod nazivom pocetakAvanture. Ova tačka je definisana u elementu <pointcut> kako bi se pomoću odgovarajućeg izraza označilo na kom tačno mestu u kodu savet aspekta treba da se primeni – u ovom slučaju, to je poziv metode zapocniAvanturu(). Izraz koji definiše mesto u kodu gde se savet primenjuje koristi AspectJ sintaksu. Ponovo, ne treba brinuti previše ukoliko ovde nisu svi detalji jasni – o aspektima će biti reči u narednim poglavljima. Za sada, dovoljno je znati da smo na ovaj način tražili od Spring okruženja da pozove bardove metode pevajPreAvanture() i pevajPosleAvanture pre i posle poziva vitezove metode zapocniAvanturu(). Sa malo dodatnog koda u XML konfiguraciji, Bard je pretvoren u Spring aspekt.

Ovde je potrebno još naglasiti dve stvari. Prvo, klasa Bard je i dalje POJO – ništa u njemu ne naglašava da se koristi kao aspekt. Bard je postao aspekt kada je definisan kao takav u kontekstu Springa. Drugo, i još važnije, bard se može primeniti na hrabrog viteza bez potrebe da HrabriVitez eksplicitno pozove bardove metode. Još bolje, klasa HrabriVitez je u potpunosti nesvesna postojanja objekta klase Bard.

4.4. Eliminisanje viška koda upotrebom šablona

U programiranju se često dešava da je potrebno pisati isti kod više puta. Ovakav kod se naziva na engl. *boilerplate* kod. Dakle, to je kod koji se iznova ponavlja kako bi se obavili česti i prosti zadaci. Na žalost, u programskom jeziku Java ovaj problem je veoma čest. Najosnovniji primer ovakvog koda sa kojim se većina programera najčešće susreće jeste rad sa JDBC (engl. *Java Database Connectivity*), gde je potrebno dohvatiti neke podatke iz baze podataka. Svako ko je radio sa JDBC je najverovatnije morao da napiše nešto slično sledećem kodu.

```
public Radnik getRadnikById(long id){
            Connection konekcija = null;
            PreparedStatement ps = null;
            ResultSet rs = null;
            try {
                  konekcija = dataSource.getConnection();
                  ps  =  konekcija.prepareStatement("select  id,
ime, prezime, plata from radnik where id=?");
                  ps.setLong(1,  id);
                  rs = ps.executeQuery();
                  Radnik radnik  = null;
                  if(rs.next()){
                        radnik = new Radnik();
                        radnik.setId(rs.getLong("id"));
                        radnik.setIme(rs.getString("ime"));
                        r a d n i k . s e t P r e z i m e ( r s .
getString("prezime"));
                        radnik.setPlata(rs.getDouble("plata"));
                  }
                  return radnik;
            } catch (SQLException e){
                  //rukovanje SQL izuzetkom
            } finally {
                  //ciscenje nereda
                  if (rs != null){
                        try {
                              rs.close();
                        } catch (SQLException e) {}

                  }
```

```
                if (ps != null){
                        try {
                                ps.close();
                        }catch (SQLException e) {}
                }
                if (konekcija != null){
                        try {
                                konekcija.close();
                        } catch (SQLException e) {}
                }
        }
        return null;
    }
```

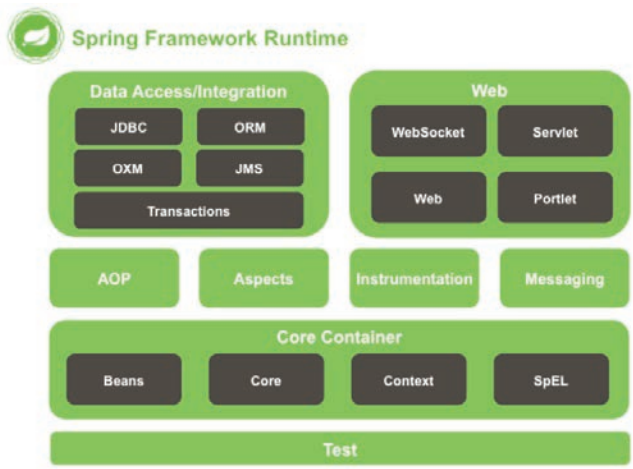
Navedeni kod služi da dovuče iz baze ime, prezime radnika, kao i platu. Ono što je ovde problem jeste činjenica da samo mali deo koda zaista služi da se specificira upit za radnika, dok je ostatak hrpa "ceremonijalnih" JDBC naredbi – prvo je potrebno kreirati konekciju, zatim upit, i na kraju izvršiti upit. Što je još gore, na kraju rada sa JDBC, obavezno se mora uhvatiti SQLException. SQLException je proveravani izuzetak, što znači da se obavezno mora hvatati, iako u tom slučaju zaista ne možemo mnogo toga da uradimo kao opravak. Tu naravno nije kraj, jer se nakon svega rečenog mora počistiti nered, odnosno zatvoriti otvorene objekte tipa Connection, ResultSet i Statement. Svako od ovih zatvaranja može takođe izbaciti SQLException, koji se naravno mora opet uhvatiti. Ono što bi trebalo uočiti ovde je da je zapravo većina napisanog koda identičan kod koji će se pisati za bilo koju JDBC operaciju. Manje od trećine navedenog koda se zaista bavi upitom za radnika, dok je sve preostalo višak koda koji se bavi JDBC.

JDBC nije jedini koji zahteva gomilu viška koda. Postoji čitav niz aktivnosti koje zahtevaju sličan ceremonijalni kod. Spring stoga teži da eliminiše ovakav kod enkapsulacijom u šablone. Na primer, u slučaju JDBC, Spring nudi JdbcTemplate, kao i nešto sepcijalizovaniju verziju SimpleJdbcTemplate, koji izvršavaju operacije nad bazom podataka bez nepotrebnih ceremonijalnih naredbi koje zahteva tradicionalni JDBC. O ovim šablonima će biti reči u kasnijim poglavljima.

4.5. Spring arhitektura okruženja

U prethodnim delovima ovog poglavlja videli smo da je Spring fokusiran na pojednostavljenje JEE razvoja aplikacija kroz DI, AOP i redukciju viška nepotrebnog koda. Čak i kad bi samo to bila kompletna funkcionalnost Springa, bilo bi vredno koristiti ga. Naravno, Spring nudi još brojne mogućnosti, koje će biti predstavljene u nastavku knjige.

Spring nudi veliki broj mogućnosti za projekte koji se razvijaju na Spring jezgru, šireći svoj uticaj na veb servise, REST, mobilne aplikacije, NoSQL itd. Kompletan Spring se sastoji od funkcionalnosti koje su podeljene u 20 osnovnih modula. Ovih 20 modula se može grupisati u šest kategorija funkcionalnosti, kao što je prikazano na slici 4.3. Gledani kao celina, ovi moduli pružaju sve što je neophodno za razvoj aplikacija. Međutim, nije obavezno da se aplikacija u potpunosti bazira na Springu. Moguće je odabrati samo module koji odgovaraju našoj aplikaciji, i kombinovati sa drugim opcijama ukoliko je potrebno. Sam Spring nudi nekoliko tačaka integracije sa drugim programerskim okvirima i bibliotekama. Pošto to prevazilazi okvire ovog udžbenika, mi ćemo se fokusirati samo na upotrebu čistog Spring okruženja, bez kombinacije sa drugim okvirima. U nastavku ovog poglavlja biće ukratko opisani svi pojedinačni moduli Spring okruženja, kako bi čitalac stekao kompletnu sliku o osnovnim mogućnostima okruženja i kako bi znao gde se pojedinačne opcije nalaze.



Slika 4.3. Spring moduli
(izvor: https://docs.spring.io/spring/docs/5.0.0.RC3/spring-framework-reference/overview.html)

Core container – centralni deo Spring okvira je kontejner koji određuje kako se zrna kreiraju, konfigurišu i zadužen je za upravljanje njima. U ovom modulu se nalazi fabrika zrna, koja je zadužena za pružanje podrške za DI. Pored fabrike zrna, postoji nekoliko implementacija konteksta aplikacije, koji omogućavaju različite načine konfiguracije Spring aplikacije. Svi ostali moduli Springa su nadograđeni na ovo jezgro.

AOP modul – Spring pruža bogatu podršku za aspektno orijentisano programiranje kroz svoj AOP modul. Ovaj model predstavlja osnovu za razvoj aspekata u Spring aplikaciji.

Data Access / Integration – Rad sa JDBC često rezultuje gomilom viška koda, poput dohvatanja konekcije, pripreme upita, izvršavanja upita, obrade rezultata i zatvaranja

konekcije. Springov JDBC modul sa podrškom za DAO (engl. *Data Access Object*) uklanja sav ovaj nepotreban kod, tako da metode za rad sa bazom ostaju jednostavne i čiste, a ujedno i sprečava potencijalne probleme poput zaboravljanja zatvaranj konekcije ka bazi i slično. Dodatno, Spring sa ovim modulom dodaje sloj razumljivih izuzetaka iznad izuzetaka koje izbacuju različite baze podataka, čime se uprošćava rukovanje izuzecima. Pored čistog JDBC, Spring daje podršku i za ORM (engl. *Object Relational Mapping*). Ovaj modul se nadograđuje na DAO podršku, čime se daje mogućnost lakog kreiranja DAO objekata za više različitih ORM platformi. Spring ne pokušava da implementira i nametne svoje ORM rešenje, već nudi veze sa više standardnih ORM platformi, poput Hibernate, Java Persistence API, Java Data Objects i slično. Menadžment transakcija u okviru Springa podržava sve ORM platforme, kao i JDBC. Transakcije se oslanjaju na AOP modul za pružanje servisa transakcija. Ovaj modul uključuje i Springovu nadogradnju JMS (engl. *Java Message Service*) za asinhronu integraciju sa drugim aplikacijama putem poruka. Tu se nalazi i OXM (engl. *Object to XML Mapping*).

Web modul – Spring donosi veoma bogati MVC (engl. *Model View Controller*) okvir, sa podrškom za labavu spregu koju Spring promoviše u okviru veb sloja aplikacije. Sama MVC paradigma je opšte prihvaćen pristup razvoju veb aplikacija koje koriste korisnički interfejs razdvojen od logike aplikacije. Pošto sama Java ima više drugih MVC okvira, Spring može da se lako integriše i sa njima, a među podržanim su Apache Struts, JSF, WebWork i slično. Pored podrške za veb aplikacije, ovaj modul pruža i podršku za RMI (engl. *Remote Method Invocation*), kao i druge opcije na daljinu (engl. remote). Spring takođe pruža odličnu podršku za REST API.

Instrumentation – ovaj modul daje podršku za dodavanje agenata Java virtuelnoj mašini. Preciznije, pruža podršku za agenta za Tomcat, koji transformiše fajlove sa klasama u trenutku učitavanja. Ovaj modul se koristi za vrlo uzak skup korisničkih scenarija, i neće biti obrađen u ovoj knjizi.

Testing – pošto je veoma važno da programeri provere svoj kod testovima, Spring pruža modul koji je posvećen testiranju Spring aplikacija. U njemu se nalazi kolekcija *mock* objekata za pisanje jediničnih testova koji rade sa JNDI, servletima i slično. Ovaj modul ima i podršku za integraciono testiranje, kroz učitavanje različitih kolekcija zrna u kontekst aplikacije.

Osim navedenih modula, Spring u širem spektru podržava i dodatne sfere razvoja Java aplikacija, od kojih ćemo navesti samo najbitnije stavke:

- *Spring Web Flow* – nadogradnja Spring MVC okvira za pravljenje veb aplikacija koje se zasnivaju na toku, u smislu vođenja korisnika ka cilju (npr. *wizard*).
- *Spring Security* – deklarativni mehanizam sigurnosti implementiran pomoću Spring AOP.
- *Spring Data* – Podrška za NoSQL baze poput MongoDB, Neo4j i slično.
- *Spring Mobile* – ekstenzija Spring MVC okvira za razvoj moblinih veb aplikacija.

- *Spring Android* – pojednostavljenje razvoja nativnih Android aplikacija uz pomoć mogućnosti Spring okruženja.
- *Spring Boot* – pojednostavljenje samog Springa. Podrška za automatsko konfigurisanje koje eliminiše većinu konfigurisanja samog Springa.

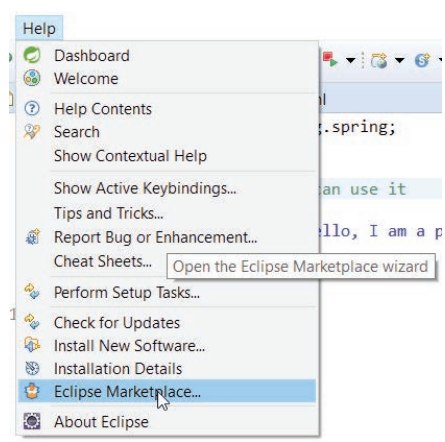
4.6. Osnovno konfigurisanje Spring projekta

U okviru ovog udžbenika za razvoj Java aplikacija koristiće se okruženje Eclipse IDE (engl. *Integrated Development Environment*). Potrebno je da Eclipse verzija bude *Enterprise Edition* (JEE). Okruženje se može skinuti sa adrese:

https://www.eclipse.org/downloads/packages/

Moguće je naravno koristiti i druga okruženja za razvoj Java aplikacija, poput NetBeans ili IntelliJ. Eclipse okruženje je odabrano iz prostog razloga što većina profesionalnih Java programera koristi ovo okruženje, a i većina kompanija koje se bave programiranjem u Javi traže razvoj upravo na ovoj platformi.

U okviru ovog poglavlja biće pokazan tradicionalni način konfigurisanja Spring aplikacije, uz upotrebu Maven. U narednim poglavljima biće pokazano i konfigurisanje aplikacije pomoću Spring Boot automatskog konfigurisanja. Na samom početku, neophodno je dodati Spring Framework i Maven u okviru Eclipse okruženja, ukoliko već nisu prisutni. Konkretno, Maven bi trebalo da bude automatski uključen u sve novije verzije Eclipse okruženja. Ukoliko Maven nije automatski uključen, potrebno je instalirati ga. Najlakši način za dodavanje novih biblioteka je kroz Eclipse Marketplace, koji se u najnovijim verzijama Eclipse okruženja nalazi pod padajućim menijem Help, kao na slici 4.4.



Slika 4.4. Eclipse Marketplace za dodavanje biblioteka

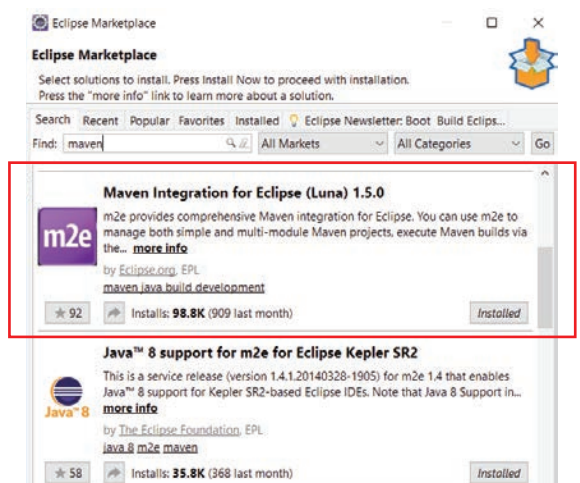
Sve biblioteke se dodaju tako što se unese kompletno ime ili deo imena u polje za pretragu. Nakon unosa Maven u polje za pretragu, od izlistanih rezultata pretrage potrebno je odabrati *Maven Integration for Eclipse*, kao na slici 4.5.

Šta je zapravo Maven i zbog čega se intenzivno koristi u razvoju Java aplikacija? Maven je alat koji se koristi za *build* i rukovođenje bilo kojim Java projektom. Maven ima nekoliko glavnih funkcionalnosti:

- Olakšava proces build-a aplikacije.
- Pruža kvalitetne informacije o projektu.
- Pruža smernice za razvoj uz primenu najboljih praksi.
- Dozvoljava lako dodavanje novih biblioteka.

Maven omogućava build projekta pomoću POM (engl. *Project Object Model*) i skupa *plug-in* dodataka koji su dele svi projekti koji koriste Maven, čime se pravi uniformisani build sistem. Maven takođe pruža puno korisnih informacija koje se delom uzimaju iz POM fajla, a delom su generisane iz izvornog koda projekta.

Maven takođe teži da na lak način usmeri projekat u smeru primene najnovijih najboljih praksi u programiranju. Na primer, specifikacija, izvršavanje i izveštaj jediničnih testova su normalan ciklus builda uz pomoć Maven-a.



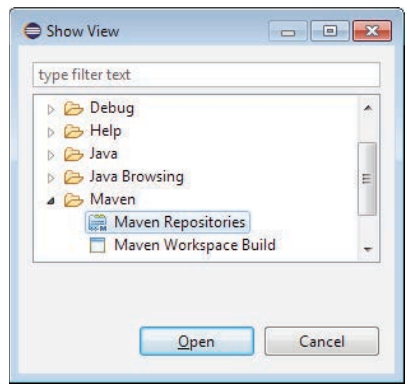
Slika 4.5. Instalacija Maven preko Eclipse Marketplace

Neke od najboljih praksi (naročito po pitanju jediničnih testova za projekat) koje Maven unosi su:

- Razdvajanje izvornog koda jediničnih testova u odvojenu, paralelnu granu.
- Upotreba odgovarajućih konvencija za imenovanje, kako bi se olakšalo lociranje i izvršavanje testova.

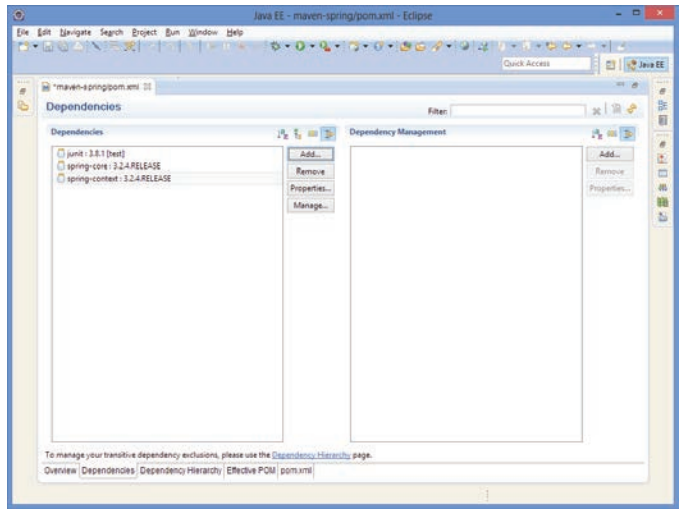
- Posebno okruženje u kome se izvršavaju testovi.
- Sprečavanje builda ukoliko svi jedinični testovi ne prođu uspešno.
- Upravljanje listom zavisnosti projekta – lako dodavanje novih zavisnosti.

Nakon dodavanja Maven Integration podrške, najčešće je potrebno izgraditi indekse po prvi put. Potrebno je u glavnom meniju odabrati redom opcije Window -> Show View -> Other, a zatim u Show View prozoru odabrati Maven -> Maven Repositories, kao što je prikazano na slici 4.6.



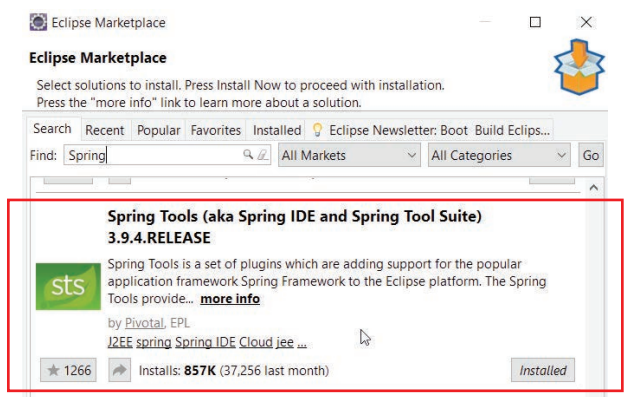
Slika 4.6. Maven Repositories

U sledećem prozoru koji se pojavi potrebno je kliknuti desnim dugmetom miša na Global Repositories i odabrati Go Into. Nakon toga, desni klik na "central (http://repo.maven.apache.org/maven2)" i odabrati Rebuild Index. Ovaj korak može da potraje dosta dugo prilikom prvog pokretanja indeksiranja. Kada se indeksiranje završi, nove zavisnosti (eksterne biblioteke) se dodaju veoma prosto, desnim klikom na projekat, a zatim odabirom opcije Maven -> Add Dependency. Moguće je i u okviru POM fajla odabrati Dependencies tab, i zatim prosto dodati reference ka spoljašnjim bibliotekama, kao što je prikazano na slici 4.7.



Slika 4.7. Dodavanje referenci kroz pom.xml

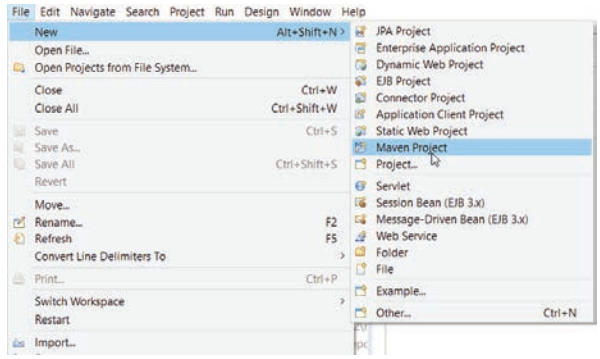
Spring se takođe mora dodati u Eclipse okruženje, na isti način kao i Maven, kroz Marketplace, kao na slici 4.8.



Slika 4.8. Dodavanje Spring Tools kroz Marketplace

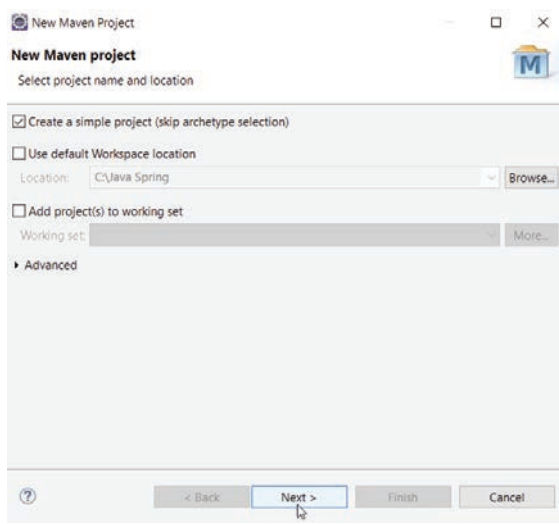
4.7. Prvi Spring projekat

U nastavku ovog poglavlja prikazaćemo kreiranje prvog Spring projekta upotrebom Maven-a. Koraci primenjeni ovde se podrazumevaju u narednim poglavljima. Nakon pokretanja Eclipse okruženja za razvoj Java aplikacija, potrebno je iz glavnog menija odabrati File -> New -> Maven project, kao na slici 4.9.



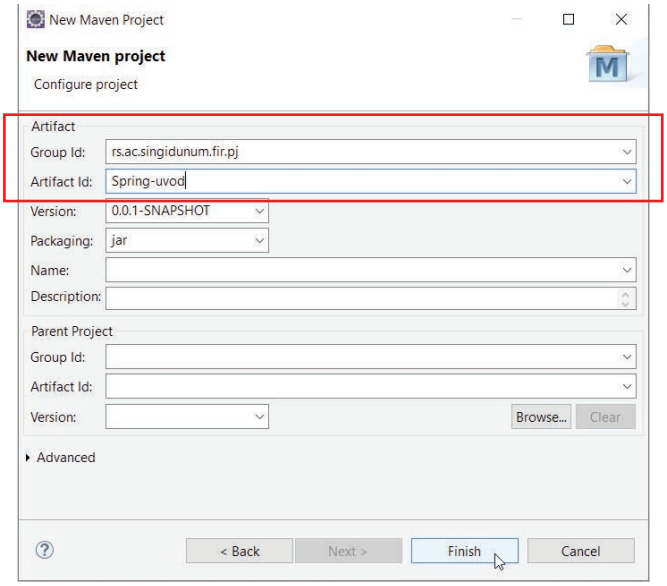
Slika 4.9. Kreiranje Maven projekta

U sledećem prozoru, potrebno je odabrati opciju *Create a simple project (skip archetype selection)*, i specificirati lokaciju na kojoj će se čuvati radno okruženje. Svaki Spring projekat bi trebalo da ima jedinstveni *Workplace location* folder.



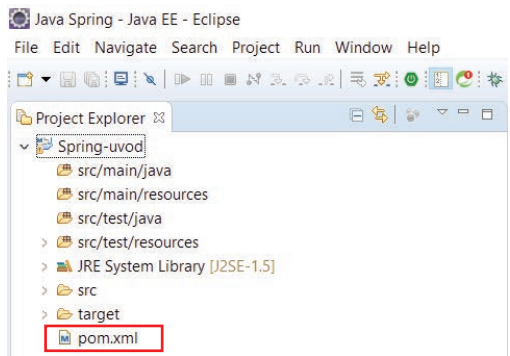
Slika 4.10. Osnovna podešavanja Maven projekta

U sledećem meniju, potrebno je popuniti polja *Group Id* i *Artefact Id. Group Id* obično odgovara imenu paketa u kome se nalazi izvorni kod, dok *Artefact Id* prestavlja ime našeg programa. Ime paketa obično prati obrnuti Internet domen kompanije koja ga razvija, tako da bi u našem slučaju paket mogao da glasi: rs.ac.singidunum.fir.pj. Sva ostala polja se mogu preskočiti, kao na slici 4.11.



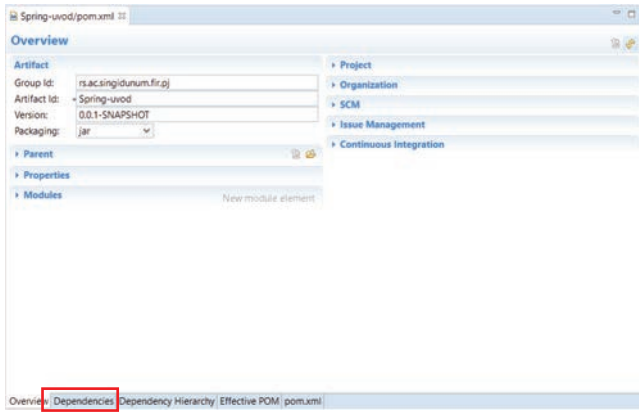
Slika 4.11. Podešavanje Group Id i Artefact Id

Nakon ovog osnovnog podešavanja i klika na dugme Finish, projekat bi trebalo da postane vidljiv u okviru Project Explorer taba u Eclipse okruženju, kao na slici 4.12.



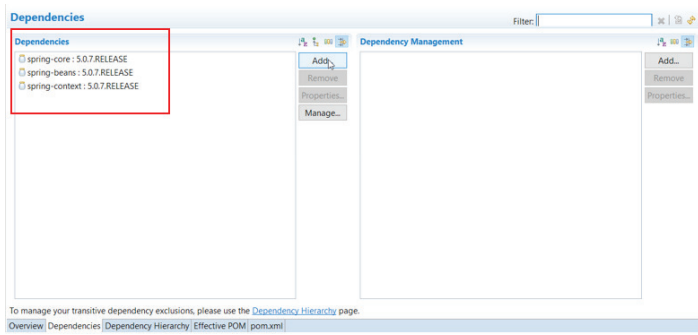
Slika 4.12. Prikaz projekta nakon kreiranja u Project Explorer tabu

Na slici 4.12. se vidi jedan XML fajl koji je od izuzetnog značaja – pom.xml fajl. Kroz ovaj fajl mogu se dodati reference (zavisnosti) ka eksternim bibliotekama. Ukoliko se dvostruko klikne na ovaj fajl, dobija se pregledni prikaz prikazan na slici 4.13. Referencama se upravlja klikom na tab Dependencies.



Slika 4.13. Pregledni prikaz pom.xml fajla

Za osnovni Spring projekat, potrebno je dodati sledeće reference: Spring Core, Spring Beans i Spring Context. Reference se dodaju kroz tab Dependencies u okviru preglednog prikaza pom.xml fajla, kao što je prikazano na slici 4.14.



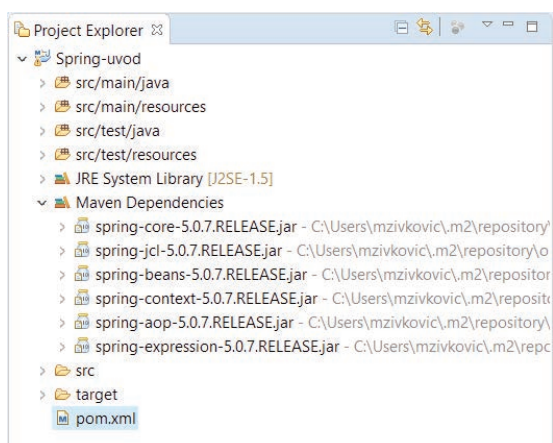
Slika 4.14. Dodavanje zavisnosti u pom.xml

Nakon dodavanja ovih osnovnih referenci, one su vidljive i ukoliko se pogleda direktno kod pom.xml fajla:

```
<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>rs.ac.singidunum.fir.pj</groupId>
  <artifactId>Spring-uvod</artifactId>
  <version>0.0.1-SNAPSHOT</version>
```

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

Potrebno je obratiti pažnju da se reference na eksterne biblioteke mogu dodati i direktno pisanjem u pom.xml fajl, bez upotrebe grafičkog editora. Nakon dodavanja, reference se mogu pronaći i u Project Explorer tabu, pod stavkom Maven Dependencies, kao što je prikazano na slici 4.14.



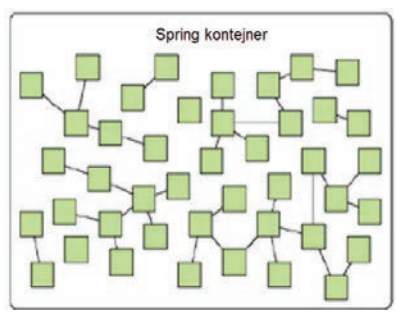
Slika 4.15. Prikaz dodatih referenci u tabu Project Explorer

Pitanja

1. Na čemu se zasniva Spring?
2. Šta je labava sprega (engl. *loose coupling*)?
3. Koje strategije koristi Spring kako bi se smanjila kompeksnost Java programiranja?
4. Zašto se kaže da Spring koristi neinvazivni model programiranje?
5. Šta je injektovanje zavisnosti i čemu služi?
6. Na koji način injektovanje zavisnosti smanjuje spregu između objekata?
7. Šta je ožičavanje u AOP terminologiji?
8. Koji je najčešći način ožičavanja u Spring okruženju? Koji još načni postoje?
9. Šta se koristi za učitavanje definicija zrna i njihovo međusobno povezivanje?
10. Šta je aspektno orjentisano programiranje?
11. Šta su aspekti (u AOP terminologiji)?
12. Na koji način Spring nastoji da se eliminiše ponavljanje koda za jednostavne i česte zadatke?
13. Kojih 6 kategorija funkcionalnosti nudi Spring?
14. Da li je obavezno da se aplikacija u potpunosti bazira na Springu?
15. Šta je centralni deo Spring okvira?
16. Čemu služi AOP modul u Springu?
17. Čemu služi Data Access / Integration modul u Springu?
18. Šta je Web modul u Springu?
19. Čemu služi Instrumentation modul u Springu?
20. Čemu služi Testing modul u Springu?
21. Navesti neke dodatne sfere razvoja Java aplikacija koje Spring podržava.
22. Šta je Maven i zbog čega se intenzivno koristi u razvoju Java aplikacija?
23. Koje reference je potrebno dodati u Maven projekat za osnovni Spring projekat?

V
Konfiguracija zrna u Spring okruženju

U aplikaciji koja se bazira na Spring okviru, objekti aplikacije žive unutar Spring kontejnera. Kontejner je zadužen za kreiranje objekata, njihovo međusobno ožičavanje, njihovu konfiguraciju, odnosno upravljanje njihovim kompletnim životnim ciklusom od kolevke pa do groba (u Javi od new() sve do finalize()), kao što je prikazano na slici 5.1.



Slika 5.1. Spring kontejner koji kreira i ožičava objekte

U nastavku ovog poglavlja ćemo opisati kako se Spring konfiguriše na takav način da zna koje objekte treba da kreira i kako da ih poveže. Na početku, bitno je poznavati osnovne osobine kontejnera u kome se objekti nalaze. Na taj način će biti jasnije kako se upravlja objektima u Spring aplikaciji.

Kontejner predstavlja jezgro Spring okruženja. Spring koristi injekciju zavisnosti (DI) za upravljanje komponentama koje čine jednu aplikaciju. To uključuje i kreiranje veza između komponenti koje treba da sarađuju zajedno. Osim DI, ovaj pojam je poznat i pod drugim nazivom IoC (engl. *Inversion of* Control). Na taj način kod objekata je čistiji i lakši za razumevanje, objekti se lakše ponovo upotrebljavaju, i lakši su za jedinično testiranje. Kontejneri u Springu se zbog primene inverzije kontrole često označavaju terminom IoC kontejneri. U Springu ne postoji samo jedan kontejner. Postoji nekoliko implementacija kontejnera, koji se uopšteno gledano mogu podeliti u dva osnovna tipa. Prvi tip su tzv. fabrike zrna, najjednostavniji kontejneri koji pružaju osnovnu podršku za DI i definisani su interfejsom org.springframework.beans.factory.BeanFactory. Drugi tip su konteksti aplikacije, koji nadograđuju fabrike zrna, a

izvedeni su iz osnovnog interfejsa org.springframework.context.ApplicationContext. Konteksti aplikacije su znatno kompleksniji od prostih fabrika zrna, i nude mnogobrojne dodatne funkcionalnosti, poput podrške za objavljivanje događaja aplikacije koje mogu uhvatiti svi prijavljeni osluškivači događaja (engl, *event listeners*). U praksi, proste fabrike zrna su često previše niskog nivoa za većinu aplikacija, pa se favorizuje upotreba konteksta aplikacije, na koje ćemo se i mi fokusirati u nastavku knjige.

Spring nudi nekoliko različitih oblika konteksta aplikacije:

- AnnotationConfigApplicationContext – učitava kontekst aplikacije iz jedne ili više Java konfiguracionih klasa.
- AnnotationConfigWebApplicationContext – učitava kontekst veb aplikacije iz jedne ili više Java konfiguracionih klasa.
- ClassPathXmlApplicationContext – učitava kontekst iz jednog ili više XML fajlova koji se nalaze na klasnoj putanji projekta.
- FileSystemXmlApplicationContext – učitava kontekst iz jednog ili više XML fajlova koji se nalaze u sistemu fajlova.
- XmlWebApplicationContext – učitava kontekst iz jednog ili više XML fajlova koje sadrži veb aplikacija.

Na početku, najbitnije je opisati učitavanje konteksta aplikacije iz sistema fajlova ili iz klasne putanje. Ukoliko se koristi učitavanje konteksta iz sistema fajlova, potrebno je proslediti putanju na kojoj se nalazi odgovarajući XML fajl:

```
ApplicationContext context = new
FileSystemXmlApplicationContext(“c:/vitez.xml”);
```

Na sličan način, ukoliko se koristi kontekst aplikacije, učitavanje konteksta se radi na sledeći način:

```
ApplicationContext context = new
ClassPathXmlApplicationContext(“vitez.xml”);
```

Razlika između ova dva osnovna učitavanja konteksta je u tome da se u prvom slučaju traži XML fajl na specifičnoj lokaciji u sistemu fajlova, dok se u drugom slučaju XML fajl može nalaziti bilo gde u klasnoj putanji projekta (uključujući i JAR fajlove). Kada je kontekst učitan, referenca na bilo koje zrno se može dohvatiti iz kontejnera pozivom metode getBean() nad objektom konteksta.

```
ClassPathXmlApplicationContext context =
        new ClassPathXmlApplicationContext(
            “vitez.xml”);
Vitez vitez = context.getBean(Vitez.class);
```

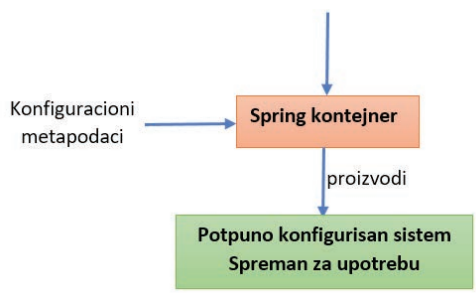
5.1. Zrna i njihov životni ciklus

Sada kada je uveden pojam kontejnera, može se posmatrati i životni ciklus zrna u okviru kontejnera u kome se zrna nalaze. Šta su zapravo zrna u okviru Spring okruženja? To su svi objekti koji čine aplikaciju i kojima upravlja Spring IoC kontejner. Zrno (engl. *bean*) je dakle objekat koga instancira i kojim upravlja kontejner. Zrna se kreiraju uz pomoć metapodataka koji se prosleđuju kontejneru, na primer u obliku XML fajla i tagova <bean> <bean/>.

Pošto se kontejneri u Springu zasnivaju na principu inverzije kontrole (IoC odnosno DI), objekti deklarišu svoje zavisnosti, odnosno druge objekte koji su im potrebni za normalan rad isključivo na jedan od sledećih načina:

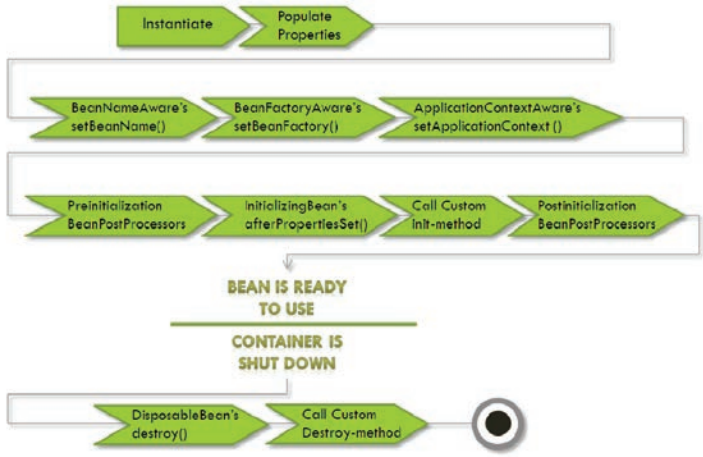
- Kroz argumente konstruktora.
- Kroz argumente *factory* metode
- Kao *property* kojem se instanca objekta setuje nakon konstrukcije.

Nakon toga kontejner injektuje ove zavisnosti prilikom kreiranja zrna. Na slici 5.2 prikazan je osnovni proces kreiranja zrna u Spring kontejneru.



Slika 5.2. Proces kreiranja zrna

Životni ciklus zrna je jednostavan u tradicionalnim Java aplikacijama. Koristi se ključna reč new za instanciranje objekta – zrna, i ono je spremno za upotrebu. Kada se više zrno ne koristi, odnosno kada se ukine poslednja referenca na taj objekat na hipu (engl. *heap* – mesto u memoriji gde se smeštaju objekti), zrno postaje označeno za sakupljanje đubreta (engl. *Garbage collection*) i nakon nekog vremena se uklanja. Nasuprot tome, životni ciklus zrna u Spring kontejneru je dosta kompleksniji. Poptuno razumevanje životnog ciklusa zrna u kontejneru otvara dodatne mogućnosti koje Spring nudi za kustomizaciju načina na koji se zrno kreira. Na slici 5.3 prikazan je tipičan životni ciklus zrna koje je učitano u kontekst aplikacije Springa.



Slika 5.3. Tipičan životni ciklus zrna u Spring okruženju
(slika preuzeta sa https://docs.spring.io)

Sa slike 5.3 se vidi da fabrika zrna izvršava podešavanje zrna u nekoliko koraka pre nego što ono postane spremno za upotrebu. Koraci su sledeći:

- Spring instancira zrno.
- Spring injektuje vrednosti i reference ka drugim zrnima u polja zrna koje se kreira.
- Ukoliko zrno implementira BeanNameAware interfejs, Spring prosleđuje ID zrna setBeanName() metodi.
- Ukoliko zrno implementira BeanFacroryAware interfejs, Spring poziva setBeanFactory() metodu, pri čemu prosleđuje kao parametar samu fabriku zrna.
- Ukoliko zrno implementira ApplicationContextAware interfejs, Spring poziva setApplicationContext() metodu, pri čemu prosleđuje kao parametar referencu na okružujući kontekst aplikacije.
- Ukoliko zrno implementira BeanPostProcesor interfejs, Spring poziva postProcessBeforeInitialization() metodu.
- Ukoliko zrno implementira InitializingBean interfejs, Spring poziva njegovu metodu afterPropertiesSet(). Slično, ukoliko je zrno deklarisano sa init metodom, specificirana metoda za inicijalizaciju se poziva.
- Ukoliko zrno implementira BeanPostProcessor interfejs, Spring poziva njegovu metodu postProcessAfterInitialization().
- U ovom trenutku, zrno je spremno za upotrebu u aplikaciji, i ostaje u kontekstu aplikacije sve dok se kontekst aplikacije ne uništi.
- Ukoliko zrno implementira DisposableBean interfejs, Spring poziva njegovu destroy() metodu. Slično, ukoliko je zrno deklarisano sa destroy metodom, specificirana metoda se poziva.

5.2. Deklaracija zrna

Svaka netrivijalna aplikacija se sastoji od više objekata koji moraju da rade zajedno kako bi se postigao neki zajednički cilj, odnosno izvršila funkcija aplikacije. Svi ovi objekti moraju biti svesni jedni drugih, uz neophodnu međusobnu komunikaciju kako bi se izvršio posao. Ukoliko se radi o online aplikaciji za kupovinu, na primer, komponenta za upravljanje narudžbinama bi morala da sarađuje sa komponentom za upravljanje proizvodima i komponentom za autorizaciju kreditne kartice. Sve ove komponente zajedno bi morale da rade sa komponentom za pristup podacima, kako bi se čitali i upisivali podaci u bazu podataka.

Tradicionalni pristup kreiranju veza između objekata aplikacije (kroz konstrukciju na primer) dovodi do komplikovanog koda koji se teško ponovo upotrebljava i još teže testira. U najblažem obliku, od ovih objekta se zahteva da rade više nego što bi trebalo. U najgorem slučaju, objekti su u tesnoj sprezi, što dovodi do gorepomenutih problema sa ponovnom upotrebom i testiranjem. U okviru Spring okruženja, od objekata se ne očekuje da traže ili kreiraju druge objekte koji su im potrebni kako bi obavljali svoju funkciju. Umesto toga, daju im se reference na objekte sa kojima sarađuju unutar kontejnera. Komponenti za upravljanje porudžbinama, na primer, je potrebna autorizacija kreditne kartice – ali ona nema potrebu da kreira objekat za autorizaciju. Dovoljno je da se pojavi praznih ruku, i biće joj dodeljen odgovarajući objekat klase za autorizaciju od strane kontejnera.

Kreiranje ovakvih veza između objekata aplikacije je srž injekcije zavisnosti, i često se naziva ožičavanje (engl. *wiring*). Injekcija zavisnosti (DI) je najelementarnija stvar koju Spring radi, i ova tehnika se upotrebljava svaki put kada se razvijaju aplikacije bazirane na Spring okviru. O ovom postupku će detaljno biti reči u sledećem poglavlju. Sada je neophodno objasniti kako se kreiraju zrna u Springu. Kao što smo već ranije napomenuli, Spring je baziran na kontejneru – ukoliko se ne konfiguriše ispravno, to je samo prazan kontejner koji ne služi srvsi. Neophodno je na neki način reći Springu koja zrna kontejner treba da sadrži, i kako da ih ožiči međusobno. Postoji nekoliko načina za konfiguraciju Springa, ali za sada ćemo se fokusirati na tradicionalan način upotrebom XML fajlova.

Kada se deklarišu zrna u XML fajlu, koreni element konfiguracionog fajla je <beans> element iz Springove šeme zrna. Tipični konfiguracioni XML fajl je u sledećem obliku:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://www.springframework.org/schema/
beans http://www.springframework.org/schema/beans/spring-beans.xsd">
```

```
    <!-- Deklaracija zrna ide ovde -->


</beans>
```

Unutar <beans> elementa se smešta kompletna Spring konfiguracija, računajući i deklaracije pojedinačnih zrna unutar <bean> tagova. Imenski prostor zrna nije jedini – postoji još dodatnih imenskih prostora, među kojima su aop (elementi za deklarisanje aspekata), mvc (podrška za Springov MVC šablon) itd. Element <bean> je najosnovnija konfiguraciona jedinica u Springu. Ona prosto kaže Spring okruženju da kreira objekat za nas.

5.3. Osnovna konfiguracija zrna

U svakom Java projektu postoje određene klase i objekti sa kojima se radi. Spring Java projekti nisu izuzetak. Na primer, jednostavna klasa Osoba u Spring okruženju bi bila definisana na sledeći način (podrazumeva se da je već napravljen Maven projekat kao što je opisano u poglavlju 4.7):

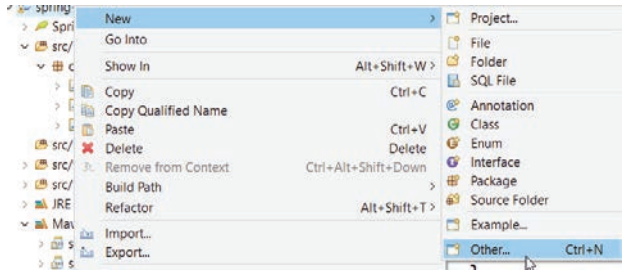
```
package rs.ac.singidunum.fir.pj;

public class Osoba {

    public void govori () {
          System.out.println("Zdravo, ja sam osoba");
    }

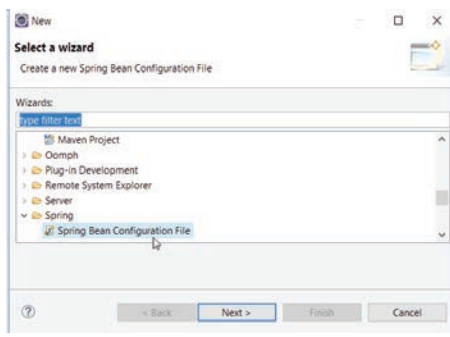
}
```

Kako bi se objekti ove klase mogli koristiti u okviru konteksta aplikacije, neophodno je u projekat dodati konfiguracioni fajl zrna (engl. *Spring Bean Configuration File*). Ovaj fajl se dodaje tako što se izvrši desni klik na projekat, zatim odabere New -> Other, kao na slici 5.4.

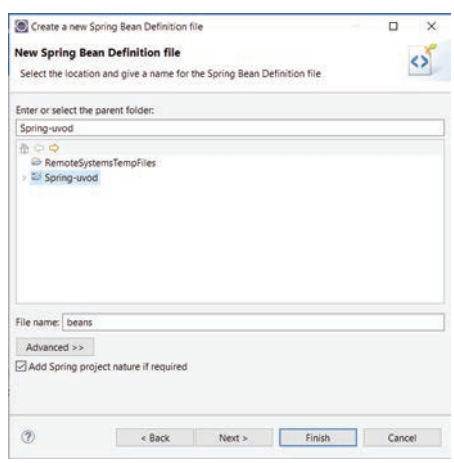


Slika 5.4. Dodavanje konfiguracionog fajla zrna, korak 1

U novom ekranu potrebno je odabrati *Spring Bean Configuration file*, koji se nalazi u okviru Spring foldera, kao na slici 5.5. U sledećem koraku, potrebno je dati ime konfiguracionom fajlu (npr. ime *beans*, bez ekstenzije .xml, koja se dodaje automatski), i odabrati folder u kome će se ovaj fajl generisati (na primer može se odabrati root projekta), kao što je prikazano na slici 5.6.

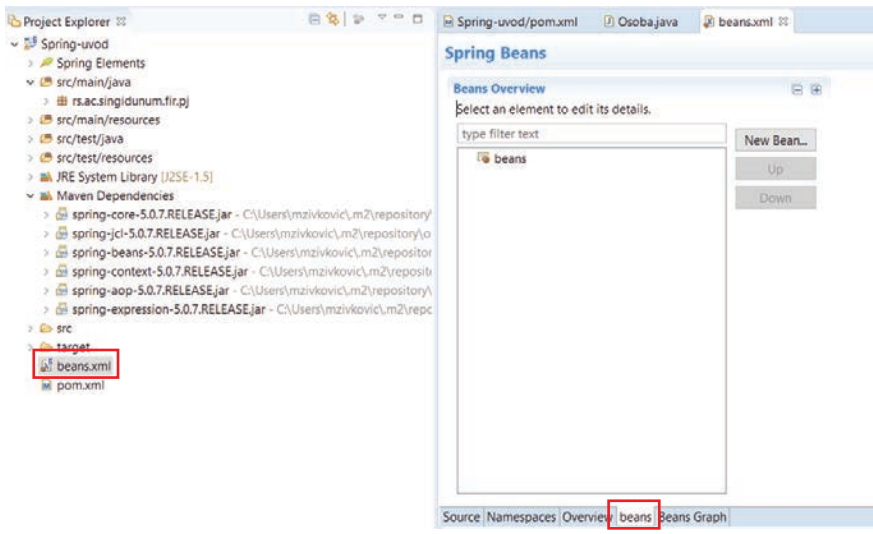


Slika 5.5. Dodavanje konfiguracionog fajla zrna, korak 2



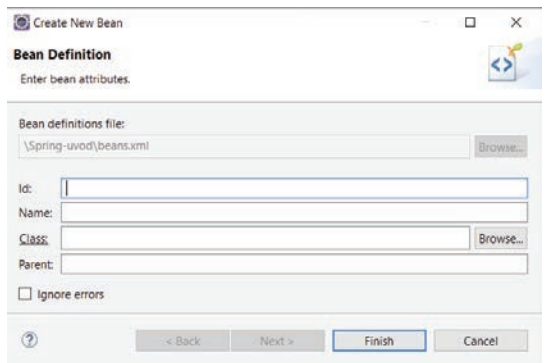
Slika 5.6. Dodavanje konfiguracionog fajla zrna, korak 3

Nakon potvrde klikom na dugme Finish, beans.xml fajl će biti dodat u odgovarajući folder – u ovom primeru u root folder projekta. Kada se otvori beans.xml i odabere beans tab na dnu ekrana, dobija se korisnički grafički editor kao na slici 5.7.



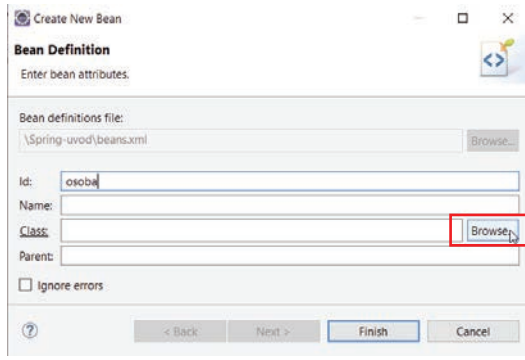
Slika 5.7. Beans.xml editor

Novo zrno se dodaje tako što se odabere opcija New Bean, koja otvara meni prikazan na slici 5.8. U ovom meniju potrebno je definisati Id zrna, i povezati za sa klasom Osoba.

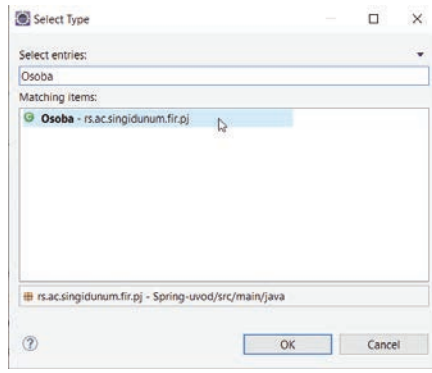


Slika 5.8. Meni za dodavanje novog zrna

U polje Id se unosi željena vrednost, u ovom primeru *osoba*, a povezivanje sa klasom osoba se vrši klikom na dugme Browse, kao na slici 5.9.

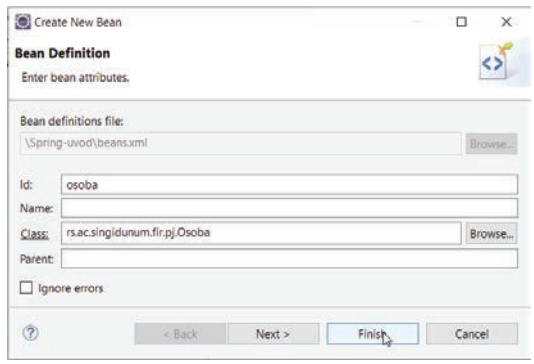


Slika 5.9. Postavljanje id zrna i povezivanje sa klasom



Slika 5.10. Odabir klase sa kojom se povezuje zrno sa Id osoba

Na slici 5.10 je prikazan odabir klase Osoba iz paketa rs.ac.singidunum.fir.pj za povezivanje sa konkretnim zrnom koje ima Id *osoba*. Nakon potvrde odabira klase, vraća se prethodni ekran sa popunjenim poljem Class kao na slici 5.11. Nakon toga je samo potrebno potvrditi kreiranje zrna. Nakon toga, zrno sa Id osoba postaje vidljivo i u grafičkom editoru, kao i u beans.xml fajlu.



Slika 5.11. Potvrda kreiranja zrna osoba

Nakon dodavanja zrna osoba, možemo pogledati sadržaj fajla beans.xml, tako što se odabere tab Source.

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://www.springframework.org/schema/
beans    http://www.springframework.org/schema/beans/spring-beans.
xsd">

       <bean   id="osoba"   class="rs.ac.singidunum.fir.pj.Osoba"></
bean>
</beans>
```

Naravno, sve ovo je bilo moguće i ručno pisati u beans.xml izvornom kodu. Međutim, zbog jednostavnosti upotrebe editora, najčešće se kompletna konfiguracija vrši na taj način, naročito u situacijama postojanja više zrna i velikog broja zavisnosti između njih, kada xml fajl postaje kompleksan.

Kako se koristi zrno koje je upravo napravljeno? Standardan način u Javi je kreiranje objekta pozivom operatora new. U Springu, koristi se kontejner koji je zadužen za instanciranje zrna. Potrebno je naglasiti da je zrno sa Id *osoba*, koje je tipa Osoba, instancirano već pri učitavanju konteksta iz XML fajla. Pozivom metode getBean() se zapravo samo dobija referenca na to zrno (koje je već kreirano i postoji u kontejneru). Metoda getBean() očekuje parametar tipa String, koji označava Id zrna za koje se želi dobiti referenca. Kod koji dohvata zrno sa Id *osoba* je dat u nastavku:

```
package rs.ac.singidunum.fir.pj;

import org.springframework.context.ApplicationContext;
import   org.springframework.context.support.FileSystemXmlApplica-
tionContext;

public class App {
       public static void main (String[] args){
              //standardni Java nacin
              /*Osoba osoba = new Osoba();
              osoba.govori();*/
              //Spring nacin
              //ApplicationContext - kontejner, klasa koja instan-
cira zrna za nas
              ApplicationContext context = new FileSystemXmlApplic
ationContext("C:\\Spring\\Uvod\\beans.xml");
```

```
            Osoba osoba = (Osoba) context.getBean("osoba");
            osoba.govori();
            //na kraju je potrebno zatvoriti kontekst aplikacije
            ((FileSystemXmlApplicationContext)context).close();
    }
}
```

Ovaj kod zahteva malo pojašnjenja. Prvo, neophodno je kreirati kontekst aplikacije, u ovom slučaju se koristi FileSystemXmlApplicationContext, kome se kao argument daje putanja ka xml fajlu. U ovom trenutku, kontekst je učitan, i sva zrna koja su definisana u beans.xml fajlu su instancirana i spremna za upotrebu. Dakle, u kontejneru već postoji zrno sa Id osoba, koje je klasnog tipa Osoba. Referenca na zrno sa Id *osoba* se dobija pozivom metode context.getBean("osoba"). Nakon toga, zrno se koristi na uobičajen način, kao i u standardnom Java programiranju. Na samom kraju, potrebno je zatvoriti kontekst aplikacije. Izvršavanje ovog programa daje izlaz u sledećem obliku:

```
Jan 29, 2019 11:33:53 PM org.springframework.context.support.Ab-
stractApplicationContext prepareRefresh
INFO: Refreshing org.springframework.context.support.FileSystemXml
ApplicationContext@6659c656: startup date [Tue Jan 29 23:33:53 CET
2019]; root of context hierarchy
Jan 29, 2019 11:33:53 PM org.springframework.beans.factory.xml.
XmlBeanDefinitionReader loadBeanDefinitions
INFO: Loading XML bean definitions from file [C:\Spring\Uvod\beans.
xml]
Zdravo, ja sam osoba
Jan 29, 2019 11:33:53 PM org.springframework.context.support.Ab-
stractApplicationContext doClose
INFO: Closing org.springframework.context.support.FileSystemXmlAp
plicationContext@6659c656: startup date [Tue Jan 29 23:33:53 CET
2019]; root of context hierarchy
```

Ono što se može uočiti je da Spring sam dodaje veliki broj dodatnih informacija u konzoli. U slučaju kompleksnijih programa, izlaz može biti preopširan, pa je ponekada malo teže doći do informacija i detalja o eventualnim greškama.

Klasa Osoba u ovom primeru je izuzetno jednostavna. Postoji samo jedna metoda. Klase koje se programiraju u praksi su znatno složenije. Dodajmo zato sada dva polja, jedno koje označava matični broj osobe, i drugo koje predstavlja ime osobe. Dodatno, potreban je konstruktor koji prihvata vrednosti ova dva podatka i smešta ih u odgovarajuća polja. Proširena klasa Osoba data je u nastavku:

```
package rs.ac.singidunum.fir.pj;

public class Osoba {

	private long jmbg;
	private String ime;

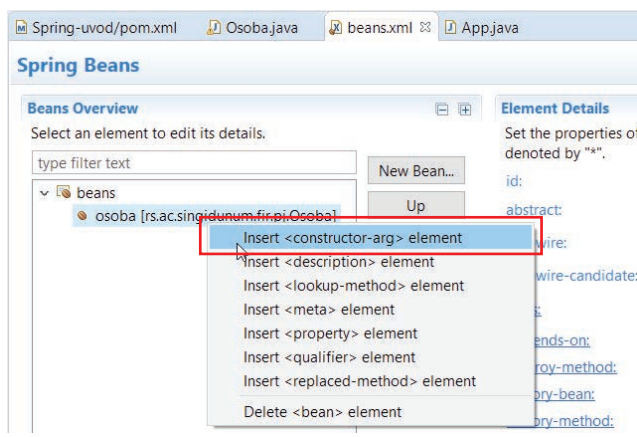
	public Osoba () { }

	public Osoba(long jmbg, String ime) {
		this.jmbg = jmbg;
		this.ime = ime;
	}

	public void govori () {
		System.out.println("Zdravo, ja sam " + this.ime);
	}

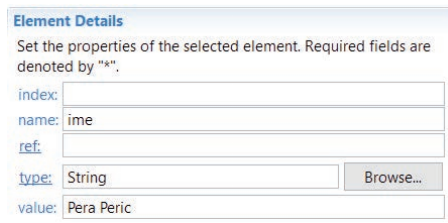
}
```

U ovom slučaju, ukoliko želimo da objekat klase Osoba bude ispravno inicijalizovan, potrebno je koristiti injekciju kroz konstruktor kako bismo prosledili vrednosti matičnog broja osobe i imena osobe odgovarajućim poljima. Da bi smo to uradili, potrebno je otići na beans.xml, odabrati beans tab, i kliknuti desnim dugmetom miša na zrno osoba. Iz liste ponuđenih opcija, bira se opcija Insert constructor-arg element, kao na slici 5.12.

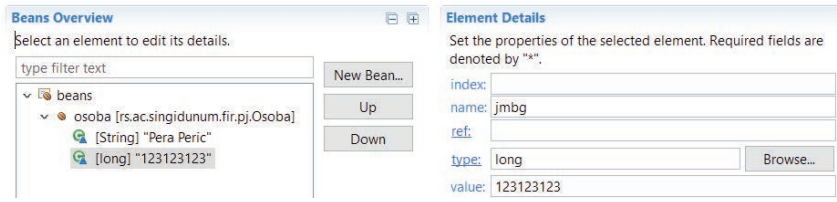


Slika 5.12. Dodavanje argumenta u konstruktor

U sledećem meniju možemo dodati sledeće parametre za argument koji se odnosi na ime osobe, kao na slici 5.13. Na isti način, možemo dodati još jedan argument, matični broj osobe, kao na slici 5.14. Polje *name* u ovom meniju označava ime formalnog parametra konstruktora u koje se injektuje odgovarajuća vrednost čiji je tip definisan u polju *type*, a konkretna vrednost u polju *value*. Na slici 5.14 se sa leve strane takođe vidi i pregled definisanih argumenata konstruktora.



Slika 5.13. Dodavanje parametra tipa String u konstruktor



Slika 5.14. Dodavanje parametra tipa long u konstruktor

Ukoliko se sada pogleda sadržaj beans.xml fajla, može se uočiti da su dodati novi elementi u okviru zrna osoba, koji predstavljaju ožičavanje parametara konstruktora klase Osoba:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://www.springframework.org/schema/
beans    http://www.springframework.org/schema/beans/spring-beans.
xsd">

       <bean id="osoba" class="rs.ac.singidunum.fir.pj.Osoba">
              <constructor-arg name="ime" type="String" value="Pera
              Peric"></constructor-arg>
              <constructor-arg name="jmbg" type="long"
              value="123123123" ></constructor-arg>
       </bean>
</beans>
```

Naravno, i ovo je sve bilo moguće ručno uneti u beans.xml. Argument konstruktora se dodaje upotrebom elementa sa tagom <constructor-arg>, sa atributima *name* gde se definiše ime formalnog parametra konstruktora koji treba da prihvati vrednost atributa *value*, čiji je tip definisan u atributu *type*. Sada, kada se dohvata referenca na zrno osoba iz kontejnera, ono će biti inicijalizovano sa imenom i matičnim brojem definisanim u beans.xml fajlu.

5.4. Injekcija zavisnosti - DI

U prethodnom primeru smo injektovali dve primitivne vrednosti kroz konstruktor. Iako je String objekat, možemo ga u ovom kontekstu posmatrati kao primitivnu vrednost. Do prave injekcije zavisnosti dolazi u slučaju kada se u zrno injektuje neko drugo zrno (objekat neke druge klase). Posmatrajmo sada takav slučaj, injekciju drugog objekta kao polje (engl. izraz *property*) u klasu Osoba. Logično, potrebno je da postoji i neka druga klasa, neka to bude klasa Adresa.

```
package rs.ac.singidunum.fir.pj;

public class Adresa {

	private String ulica;
	private String postanskiBroj;

	public Adresa () { }

	public Adresa(String ulica, String postanskiBroj) {
		super();
		this.ulica = ulica;
		this.postanskiBroj = postanskiBroj;
	}

	@Override
	public String toString() {
		return "Adresa [ulica=" + ulica + ", postanskiBroj="
+ postanskiBroj + "]";
	}

}
```

Uvedimo i neke izmene u klasi Osoba. Dodaćemo dva polja, jedno je objekat tipa Adresa, a drugo uzrast tipa integer. Mogućnost za postavljanje vrednosti ova dva polja je data kroz odgovarajuće set metode. Izmenjena klasa Osoba je data u nastavku:

```
package rs.ac.singidunum.fir.pj;

public class Osoba {

	private long jmbg;
	private String ime;

	private Adresa adresa;
	private int uzrast;

	public Osoba () { }

	public Osoba(long jmbg, String ime) {
		this.jmbg = jmbg;
		this.ime = ime;
	}

	public void setAdresa(Adresa adresa) {
		this.adresa = adresa;
	}

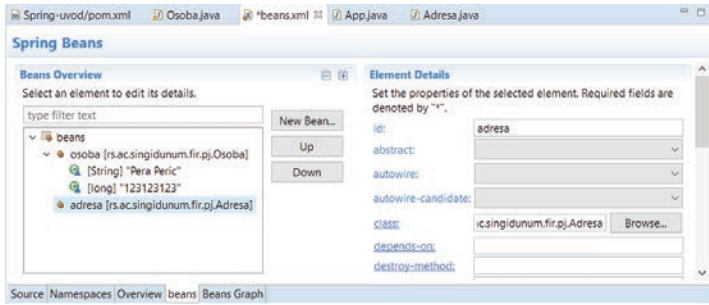
	public void setUzrast(int uzrast) {
		this.uzrast = uzrast;
	}

	public void govori () {
		System.out.println("Zdravo, ja sam " + this.ime + " i
imam " + this.uzrast + " godina!");
		System.out.println("Moja adresa je: " + this.adresa);
	}

}
```

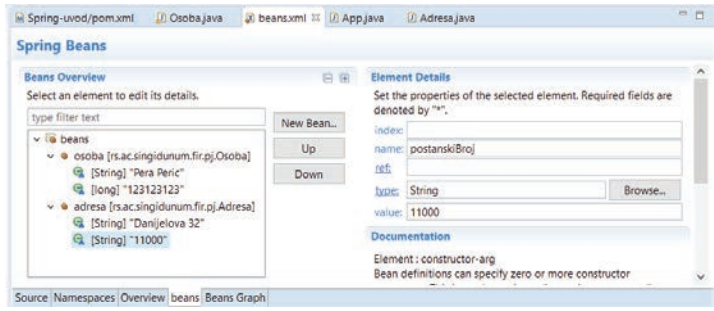
Za klasu Osoba se sada kaže da zavisi od objekta klase Adresa, odnosno postoji zavisnost (engl. *dependency*). Potrebno je injektovati objekat klase Adresa u objekat klase Osoba. U klasi osoba su dodate set metode, kako bismo prikazali drugi oblik injekcije zavisnosti. Već smo videli kako izgleda injekcija kroz argument konstruktora. Drugi način je injekcija polja (engl. *Property Injection*) kroz set metode.

Kako bismo uradili injekciju adrese, potrebno je da postoji zrno tipa Adresa. Ovo zrno se dodaje na isti način kao što je prethodno bilo dodato zrno sa Id *osoba*. Potrebno je otići u beans.xml i odabrati opciju New Bean, i kreirati zrno sa Id *adresa*, koje je tipa Adresa, kao što je prikazano na slici 5.15.



Slika 5.15. Dodavanje zrna klase Adresa

Kao i u slučaju zrna osoba, potrebno je inicijalizovati zrno adresa. Pošto se inicijalizacija objekta klase Adresa radi kroz konstruktor, neophodno je injektovati ulicu i poštanski broj kroz argumente konstruktora klase adresa. Kao i u slučaju zrna osoba, dodaju se dva <constructor-arg> elementa (desni klik na zrno adresa, i odabir opcije insert <constructor-arg> element, kao što je prikazano na slici 5.16.



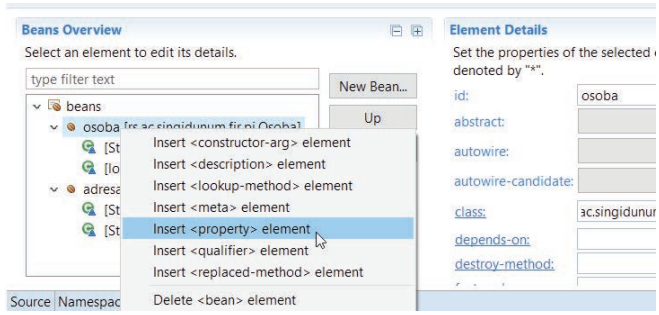
Slika 5.16. Injekcija parametara klase Adresa kroz konstruktor

Sada se u beans.xml može videti i novo zrno sa Id *adresa*, kao što je prikazano u nastavku:

```
<bean id="adresa" class="rs.ac.singidunum.fir.pj.Adresa">
            <constructor-arg name="ulica" type="String"
                  value="Danijelova 32">
            </constructor-arg>
            <constructor-arg name="postanskiBroj" type="String"
                  value="11000">
            </constructor-arg>
</bean>
```

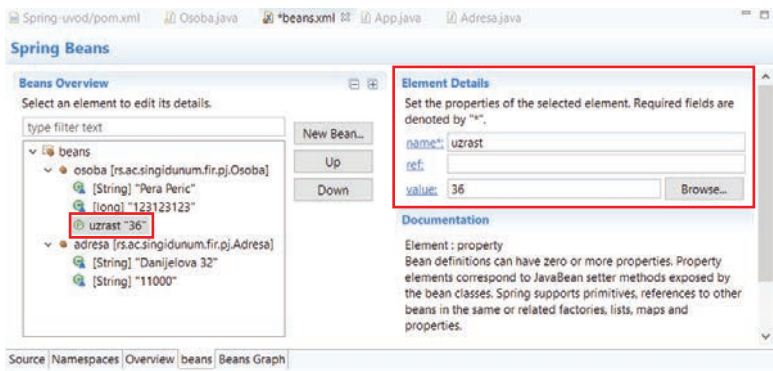
Ovo zrno se sada može iskoristiti za injekciju u zrno osoba. Unutar beans.xml editora, desnim klikom dobijaju se opcije za insertovanje. U prethodnom slučaju smo

koristili Insert <constructor-arg> element, pošto je injekcija išla kroz argumente konstruktora. U slučaju injekcije uzrasta i adrese u zrno osoba, nije moguće koristiti taj pristup, zato što ne postoji konstruktor koji prihvata te parametre. Klasa Osoba je implementirana na takav način da očekuje postavljanje polja uzrast i adresa kroz set metode. Zbog toga, i za uzrast i za adresu se uzima opcija Insert <property> element, kao na slici 5.17.



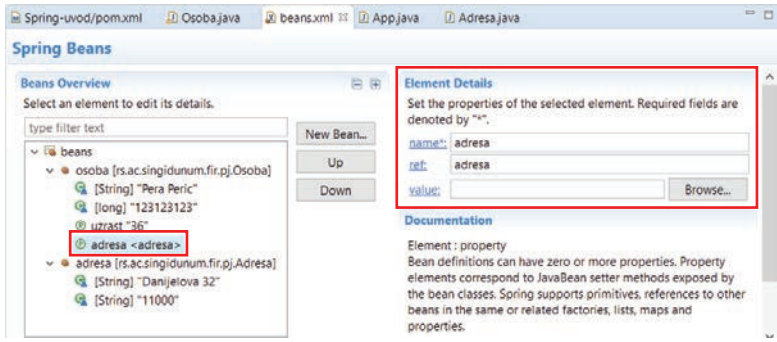
Slika 5.17. Injekcija polja u klasu Osoba

Potrebno je dodati dva polja, uzrast (tipa integer) i adresu (objekat klase Adresa). Uzrast se može specificirati kao vrednost, pošto je primitivnog tipa, kao na slici 5.18.



Slika 5.18. Injekcija primitivne vrednosti kroz set metodu

U slučaju injekcije adrese, potrebno je postaviti referencu na odgovarajuće zrno, tako što se u polju *ref* postavlja Id zrna koje je potrebno injektovati, kao što je prikazano na slici 5.19.



Slika 5.19. Injekcija objekta kroz set metodu

Sadržaj beans.xml fajla sada izgleda ovako:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://www.springframework.org/schema/
beans    http://www.springframework.org/schema/beans/spring-beans.
xsd">

       <bean id="osoba" class="rs.ac.singidunum.fir.pj.Osoba">
              <constructor-arg name="ime" type="String" value="Pera
Peric"></constructor-arg>
              <constructor-arg    name="jmbg"    type="long"    val-
ue="123123123" ></constructor-arg>
              <property name="uzrast" value="36"></property>
              <property name="adresa" ref="adresa"></property>
       </bean>
       <bean id="adresa" class="rs.ac.singidunum.fir.pj.Adresa">
              <constructor-arg name="ulica" type="String"
                     value="Danijelova 32">
              </constructor-arg>
              <constructor-arg name="postanskiBroj" type="String"
                     value="11000">
              </constructor-arg>
       </bean>
</beans>
```

Ukoliko se izvrši program sa ovim modifikacijama, dobija se sledeći izlaz:

```
Jan 30, 2019 1:19:00 AM org.springframework.context.support.Ab-
stractApplicationContext prepareRefresh
INFO: Refreshing org.springframework.context.support.FileSystemXml
ApplicationContext@6659c656: startup date [Wed Jan 30 01:19:00 CET
2019]; root of context hierarchy
Jan 30, 2019 1:19:00 AM org.springframework.beans.factory.xml.Xml-
BeanDefinitionReader loadBeanDefinitions
INFO: Loading XML bean definitions from file [C:\Spring\Uvod\beans.
xml]
Zdravo, ja sam Pera Peric i imam 36 godina!
Moja adresa je: Adresa [ulica=Danijelova 32, postanskiBroj=11000]
Jan 30, 2019 1:19:01 AM org.springframework.context.support.Ab-
stractApplicationContext doClose
INFO: Closing org.springframework.context.support.FileSystemXmlAp
plicationContext@6659c656: startup date [Wed Jan 30 01:19:00 CET
2019]; root of context hierarchy
```

5.5. Metode init i destroy

Svakom zrnu se mogu dodati metode koje će biti izvršene odmah nakon kreiranja zrna (init) i neposredno pre uništenja zrna (destroy). U okviru klase Osoba, mogu se dodati dve metode koje će definisati init i destroy procedure. U datom primeru su metode nazvane onCreate() i onDestroy(), po analogiji na konvenciju koja se tipično koristi za ovakve metode (npr. u razvoju Android aplikacija). Metode su mogle biti imenovane i drugačije, potpuno proizvoljno. U ovom primeru, metode onCreate() i onDestroy() samo ispisuju odgovarajuće poruke na standardni izlaz. Ukoliko je potrebno, u ovim metodama se mogu raditi dodatni postupci inicijalizacije ili uništavanja objekta.

```
package rs.ac.singidunum.fir.pj;

public class Osoba {

	private long jmbg;
	private String ime;

	private Adresa adresa;
	private int uzrast;

	public Osoba () { }
```

```
	public Osoba(long jmbg, String ime) {
		this.jmbg = jmbg;
		this.ime = ime;
	}

	public void setAdresa(Adresa adresa) {
		this.adresa = adresa;
	}

	public void setUzrast(int uzrast) {
		this.uzrast = uzrast;
	}

	public void govori () {
		System.out.println("Zdravo, ja sam " + this.ime + " i
imam " + this.uzrast + " godina!");
		System.out.println("Moja adresa je: " + this.adresa);
	}

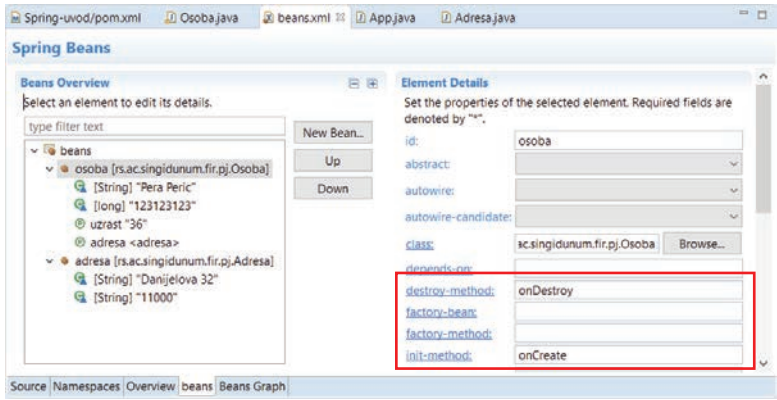
	public void onCreate () {
		System.out.println("Kreirana osoba: " + this);
	}

	public void onDestroy () {
		System.out.println("Uklonjena osoba: " + this);
	}

@Override
	public String toString() {
		return "Osoba [jmbg=" + jmbg + ", ime=" + ime + ",
adresa=" + adresa + ", uzrast=" + uzrast + "]";
	}

}
```

Sada se, unutar beans.xml fajla, može odabrati zrno osoba, i dodati init-method i destroy-method. Ove metode se dodaju po imenima metoda u definiciji klase, dakle, u slučaju init-method, potrebno je postaviti metodu onCreate, dok je u slučaju destroy-method potrebno postaviti onDestroy kao metodu koja se izvršava. Da su imena ovih metoda bila drugačija u definiciji klase, koristili bismo ta imena iz definicije klase na ovom mestu. Postavljanje ovih metoda je prikazano na slici 5.20.



Slika 5.20. Dodavanje init i destroy metoda

U okviru izvornog koda beans.xml fajla, mogu se uočiti ove dve nove dodate metode:

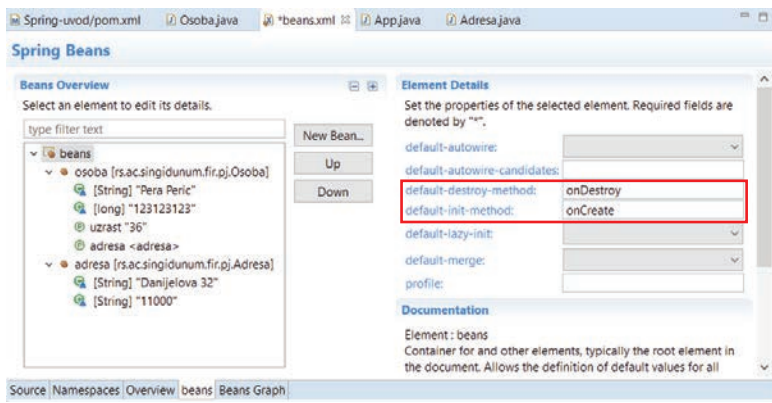
```
<bean id="osoba" class="rs.ac.singidunum.fir.pj.Osoba"
            destroy-method="onDestroy" init-method="onCreate">
            <constructor-arg name="ime" type="String" value="Pera
Peric"></constructor-arg>
            <constructor-arg    name="jmbg"    type="long"    val-
ue="123123123" ></constructor-arg>
            <property name="uzrast" value="36"></property>
            <property name="adresa" ref="adresa"></property>
</bean>
```

Ukoliko se program izvrši u ovom obliku, dobija se izlaz prikazan u nastavku, gde se vidi da je odmah nakon kreiranja zrna, a pre njegove upotrebe, pozvana init metoda, kao i da je neposredno pre uništenja zrna pozvana metoda destroy.

```
Jan 30, 2019 1:56:02 AM org.springframework.context.support.Ab-
stractApplicationContext prepareRefresh
INFO: Refreshing org.springframework.context.support.FileSystemXml
ApplicationContext@6659c656: startup date [Wed Jan 30 01:56:02 CET
2019]; root of context hierarchy
Jan 30, 2019 1:56:02 AM org.springframework.beans.factory.xml.Xml-
BeanDefinitionReader loadBeanDefinitions
INFO: Loading XML bean definitions from file [C:\Spring\Uvod\beans.
xml]
Kreirana osoba: Osoba [jmbg=123123123, ime=Pera Peric, adresa=Adresa
[ulica=Danijelova 32, postanskiBroj=11000], uzrast=36]
Zdravo, ja sam Pera Peric i imam 36 godina!
Moja adresa je: Adresa [ulica=Danijelova 32, postanskiBroj=11000]
```

```
Jan 30, 2019 1:56:03 AM org.springframework.context.support.Ab-
stractApplicationContext doClose
INFO: Closing org.springframework.context.support.FileSystemXmlAp
plicationContext@6659c656: startup date [Wed Jan 30 01:56:02 CET
2019]; root of context hierarchy
Uklonjena osoba: Osoba [jmbg=123123123, ime=Pera Peric, adresa=Adresa
[ulica=Danijelova 32, postanskiBroj=11000], uzrast=36]
```

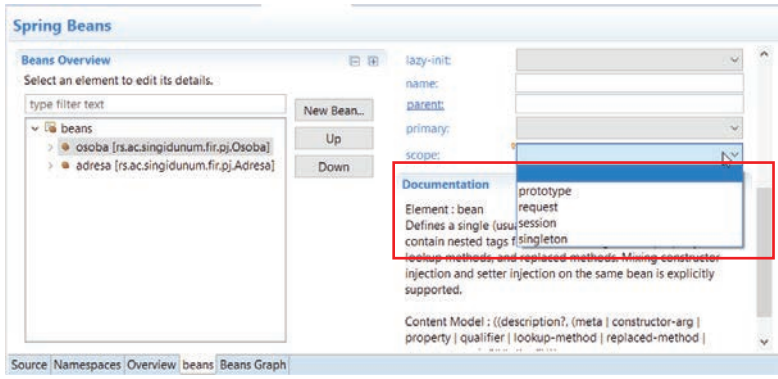
Ukoliko je potrebno, mogu se definisati init i destroy metode na nivou elementa bean, kao na slici 5.21. U ovom slučaju, to znači da se za svako zrno pozivaju metode onCreate() prilikom kreiranja, i onDestroy() prilikom uništavanja zrna. Ukoliko u nekoj klasi ove metode nisu definisane, za takvo zrno se ne izvršava ništa.



Slika 5.21. Dodavanje podrazumevanih init i destroy metoda

5.6. Opseg zrna

U Spring okruženju sva zrna su podrazumevano singleton (postoji tačno jedna instanca tog zrna). Kada kontejner vrati zrno, za potrebe ožičavanja ili kao rezultat poziva getBean() metode kontejnera, to je uvek ista instanca zrna. Ponekad je međutim potrebno da se dobije jedinstvena instanca zrna svaki put kada se traži zrno od kontejnera. U Springu postoji opcija definisanja opsega zrna (engl. *scope*). U editoru postoji opcija za specificiranje opsega, kao na slici 5.22.



Slika 5.22. Opseg zrna

Ukoliko je potrebno da se dobije novo zrno svaki put kad se pozove metoda getBean(), opseg je potrebno postaviti na vrednost *prototype*. Postoje ukupno četiri moguće vrednosti za opseg zrna, koje su prikazane u tabeli 5.1 sa kratkim opisima.

Tabela 5.1. Mogući opsezi zrna u Springu

Opseg	Način rada
singleton	Opseg zrna je jedna instanca po kontejneru (podrazumevano)
prototype	Dozvoljava više instanciranja zrna
request	Opseg važenja zrna je jedan HTTP zahtev. Ima smisla koristiti samo u veb kontekstu (npr. sa Spring MVC)
session	Opseg važenja je jedna HTTP sesija. Ima smisla koristiti samo u veb kontekstu (npr. sa Spring MVC)

5.7. Injektovanje kolekcija

Do sada je prikazano kako se Spring koristi za injekciju kako polja primitivnih tipova (upotrebom atributa *value*), tako i polja klasnih tipova, pomoću referenci na druga zrna (upotrebom atributa *ref*). Međutim, injekcija primitivnih vrednosti i referenci na druga zrna je korisna samo u slučaju da zrno u koje se vrši injekcija ima polja koja su singularna. Kako se radi ožičavanje u slučaju da zrno ima polje koje je kolekcija vrednosti?

Spring nudi četiri tipa konfiguracionih elemenata koji odgovaraju kolekcijama, i koji se mogu koristiti za injekciju u polja drugih zrna. Ovi tipovi elemenata, zajedno sa kratkim opisom, su prikazani u tabeli 5.2.

Tabela 5.2. Elementi za ožičavanje kolekcija

Element	Koristi se za
<list>	Ožičavanje liste vrednosti, duplikati su dozvoljeni
<set>	Ožičavanje skupa vrednosti, duplikati nisu dozvoljeni
<map>	Ožičavanje kolekcije u obliku parova ime-vrednost, gde ime i vrednost mogu biti bilo kog tipa
<props>	Ožičavanje kolekcije u obliku parova ime-vrednost, gde su i ime i vrednost tipa String

Elementi <list> i <set> su korisni za konfigurisanje polja neke klase koja su ili nizovi ili bilo koja implementacija interfejsa java.util.Collection. Stvarna implementacija kolekcije koja se koristi za definisanje polja nema previše korelacije sa izborom <list> ili <set> elementa. Oba tipa elementa se mogu praktično nesmetano koristiti sa bilo kojim tipom java.util.Collection.

Što se tiče <map> i <props> elemenata, oni odgovaraju kolekcijama koje su tipa java.util.Map i java.util.Properties, respektivno. Ova dva tipa su korisna u slučaju da je potrebno injektovanje kolekcije koja je u obliku parova ključ-vrednost. Razlika između ova dva elementa je da kada se koristi <props>, i ključ i vrednost moraju biti tipa String. U slučaju <map> elementa, dozvoljeni su ključevi i vrednosti proizvoljnih tipova.

Posmatrajmo primer kada je potrebno injektovati listu objekata tipa String. U pom.xml fajlu se nalaze iste zavisnosti ka Spring modulima i na isti način se dodaje beans.xml kao i u prethodnim poglavljima. Sadržaj pom.xml fajla je:

```
<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>rs.ac.singidunum.fir.pj</groupId>
  <artifactId>Spring-uvod</artifactId>
  <version>0.0.1-SNAPSHOT</version>
  <dependencies>
            <dependency>
            <groupId>org.springframework</groupId>
            <artifactId>spring-core</artifactId>
            <version>5.0.7.RELEASE</version>
      </dependency>
      <dependency>
            <groupId>org.springframework</groupId>
            <artifactId>spring-beans</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>
</project>
```

Klasa u koju je potrebno injektovati listu objekata tipa String je klasa KorpaSaVocem, čiji je kod dat u nastavku. Polje koje prihvata listu je polje vocke. Klasa ima konstruktor koji inicijalizuje sva polja, kao i toString() metodu koja ispisuje sadržaj korpe sa voćem, tako što se svaka pojedinačna voćka ispisuje u novom redu.

```
package rs.ac.singidunum.fir.pj;

import java.util.List;

public class KorpaSaVocem {

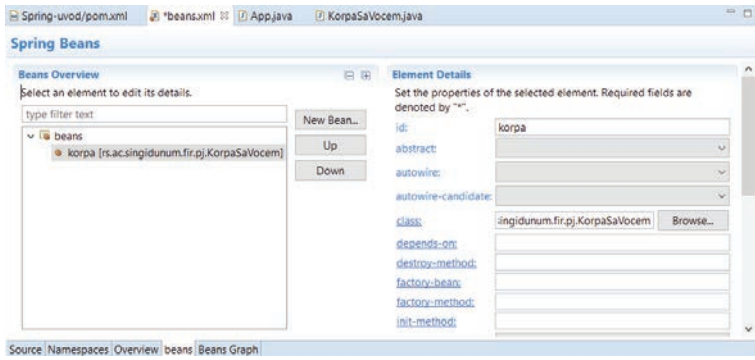
        private String imeKorpe;
        private List<String> vocke;

        public KorpaSaVocem(String imeKorpe, List<String> vocke) {
                this.imeKorpe = imeKorpe;
                this.vocke = vocke;
        }

        public String toString(){
                StringBuilder sb = new StringBuilder();
                sb.append(imeKorpe);
                sb.append(“ sadrzi:\n”);
                for (String fruit: vocke){
                        sb.append(fruit);
                        sb.append(“\n”);
                }
                return sb.toString();
        }

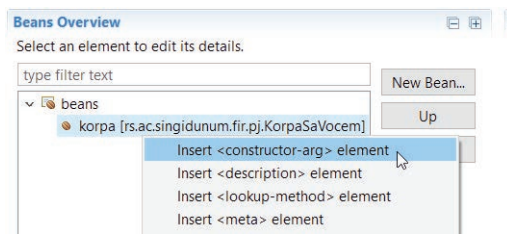
}
```

Nakon implementacije klase KorpaSaVocem, potrebno je kreirati zrno *korpa*, koje se povezuje sa klasom KorpaSaVocem, na uobičajen način pokazan u prethodnim poglavljima, kao na slici 5.23.



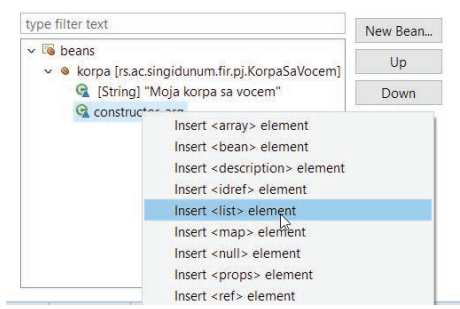
Slika 5.23. Konfigurisanje zrna za korpu sa voćem

Pošto se ime korpe i lista stringova prosleđuju kroz konstruktor, koristi se insert constructor-arg element, kao na slici 5.24. Ime korpe se injektuje kao prosta vrednost tipa String, kao što je već opisano u ranijim poglavljima.



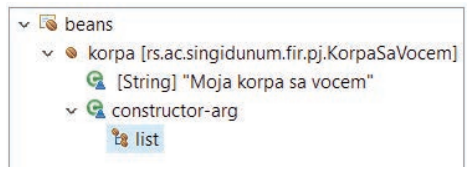
Slika 5.24. Injektovanje liste stringova kroz konstruktor, korak 1

Kada je ubačen element koji označava argument konstruktora, i kada se on prikaže u okviru zrna *korpa*, potrebno je kliknuti desnim dugmetom miša na njega kako bi se dobile dodatne opcije, od kojih je potrebno u ovom slučaju izabrati listu, kao na slici 5.25.

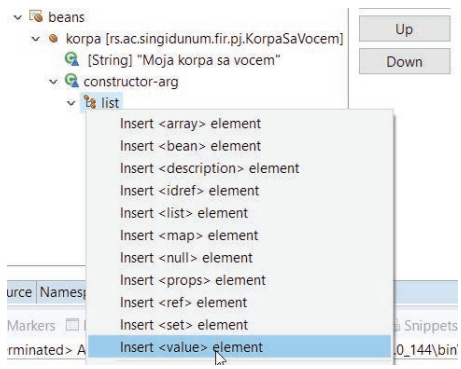


Slika 5.25. Injektovanje liste stringova kroz konstruktor, korak 2

Nakon ubacivanja elementa <list>, on će biti prikazan u okviru zrna *korpa*, kao na slici 5.26. Pojedinačni elementi se dodaju desnim klikom na listu, i odabirom elementa <value>, kao na slici 5.27.



Slika 5.26. Injektovanje liste stringova kroz konstruktor, korak 3



Slika 5.27. Injektovanje liste stringova kroz konstruktor, korak 4

Nakon ubacivanja <value> elementa, sadržaj beans.xml fajla je:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://www.springframework.org/schema/
beans http://www.springframework.org/schema/beans/spring-beans.xsd">

       <bean id="korpa" class="rs.ac.singidunum.fir.pj.KorpaSaVocem">
              <constructor-arg name="imeKorpe" type="String"
                     value="Moja korpa sa vocem">
              </constructor-arg>
              <constructor-arg>
                     <list>
                            <value></value>
                     </list>
              </constructor-arg>
       </bean>
</beans>
```

Obično je lakše koristiti editor za kreiranje beans.xml fajla, međutim u slučaju ubacivanja elemenata liste, brže i lakše je unositi konkretne elemente direktno u beans.xml fajlu. Prosto, kopiranjem je moguće napraviti dodatne elemente, u koje je posle toga lako dodati njihove vrednosti. Nakon ručne izmene fajla, beans.xml ima sledeći sadržaj:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://www.springframework.org/schema/
beans    http://www.springframework.org/schema/beans/spring-beans.
xsd">

       <bean         id="korpa"         class="rs.ac.singidunum.fir.
pj.KorpaSaVocem">
              <constructor-arg name="imeKorpe" type="String"
                     value="Moja korpa sa vocem">
              </constructor-arg>
              <constructor-arg>
                     <list>
                            <value>jabuka</value>
                            <value>ananas</value>
                            <value>banana</value>
                            <value>kruska</value>
                            <value>banana</value>
                            <value>pomorandza</value>
                            <value>banana</value>
                            <value>ananas</value>
                     </list>
              </constructor-arg>
       </bean>
</beans>
```

Lista dozvoljava unošenje više istih vrednosti – u ovom slučaju ananas je unet dva puta, dok je banana uneta tri puta. Sada se može napisati i kratak program koji će demonstrirati rad ovog primera. Klasa App.java ima sledeći sadržaj:

```
package rs.ac.singidunum.fir.pj;

import org.springframework.context.ApplicationContext;
import org.springframework.context.support.FileSystemXmlApplica-
tionContext;
```

```
public class App {
	public static void main (String[] args){

		ApplicationContext context = new FileSystemXmlApplic
ationContext("beans.xml");
		KorpaSaVocem korpa = (KorpaSaVocem) context.
getBean("korpa");
		System.out.println(korpa);
		((FileSystemXmlApplicationContext)context).close();

	}
}
```

Nakon izvršavanja datog programa, dobija se izlaz kao u nastavku. Može se primetiti da su elementi liste koji se ponavljaju i dalje tu, pošto je u slučaju <list> dozvoljeno ponavljanje elemenata.

```
Jan 31, 2019 1:38:38 AM org.springframework.context.support.Ab-
stractApplicationContext prepareRefresh
INFO: Refreshing org.springframework.context.support.FileSystemXml
ApplicationContext@6659c656: startup date [Thu Jan 31 01:38:38 CET
2019]; root of context hierarchy
Jan 31, 2019 1:38:38 AM org.springframework.beans.factory.xml.Xml-
BeanDefinitionReader loadBeanDefinitions
INFO: Loading XML bean definitions from file [C:\Spring\Uvod\beans.
xml]
Moja korpa sa vocem sadrzi:
jabuka
ananas
banana
kruska
banana
pomorandza
banana
ananas

Jan 31, 2019 1:38:38 AM org.springframework.context.support.Ab-
stractApplicationContext doClose
INFO: Closing org.springframework.context.support.FileSystemXmlAp
plicationContext@6659c656: startup date [Thu Jan 31 01:38:38 CET
2019]; root of context hierarchy
```

Neka se sada posmatra slučaj da se unutar beans.xml fajla promeni element <list> u <set>, odnosno da se koristi skup umesto liste za injekciju vrednosti. Nakon unetih izmena, beans.xml fajl ima sledeći sadržaj:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://www.springframework.org/schema/
beans http://www.springframework.org/schema/beans/spring-beans.xsd">

       <bean id="korpa" class="rs.ac.singidunum.fir.pj.KorpaSaVocem">
              <constructor-arg name="imeKorpe" type="String"
                     value="Moja korpa sa vocem">
              </constructor-arg>
              <constructor-arg>
                     <set>
                            <value>jabuka</value>
                            <value>ananas</value>
                            <value>banana</value>
                            <value>kruska</value>
                            <value>banana</value>
                            <value>pomorandza</value>
                            <value>banana</value>
                            <value>ananas</value>
                     </set>
              </constructor-arg>
       </bean>
</beans>
```

Ukoliko se nakon samo ove jedne izmene program pokrene ponovo, u izlazu se može primetiti da, u ovom slučaju, u listu koja se nalazi u polju voćke zrna *korpa* nisu unete višestruke vrednosti, zbog toga što <set> ne dozvoljava ponavljanja.

```
Jan 31, 2019 3:00:30 AM org.springframework.context.support.Ab-
stractApplicationContext prepareRefresh
INFO: Refreshing org.springframework.context.support.FileSystemXml
ApplicationContext@6659c656: startup date [Thu Jan 31 03:00:30 CET
2019]; root of context hierarchy
Jan 31, 2019 3:00:30 AM org.springframework.beans.factory.xml.Xml-
BeanDefinitionReader loadBeanDefinitions
INFO: Loading XML bean definitions from file [C:\Spring\Uvod\beans.xml]
```

```
Moja korpa sa vocem sadrzi:
jabuka
ananas
banana
kruska
pomorandza

Jan 31, 2019 3:00:30 AM org.springframework.context.support.Ab-
stractApplicationContext doClose
INFO: Closing org.springframework.context.support.FileSystemXmlAp
plicationContext@6659c656: startup date [Thu Jan 31 03:00:30 CET
2019]; root of context hierarchy
```

Potrebno je još jedanput napomenuti da ovo podešavanje između elemenata list i set u beans.xml konfiguracionom fajlu ne utiče na odabir kolekcije u klasi koja će čuvati ove vrednosti. Na primer, i list i set podešavanje unutar beans.xml se mogu čuvati u objektu tipa List unutar Java klase.

5.8. Injektovanje liste zrna

U prethodnom poglavlju pokazano je kako se injektuju kolekcije vrednosti primitivnih tipova i stringova. Šta se dešava u slučaju da je potrebno injektovati kolekciju drugih zrna? Šta bi se desilo da pojedinačne voćke u primeru iz prethodnog poglavlja nisu samo prosti stringovi, već objekti neke klase, odnosno zrna u terminologiji Springa? Posmatrajmo sada takvu implementaciju, gde su pojedinačne voćke opisane klasom Vocka.

```
package rs.ac.singidunum.fir.pj;

public class Vocka {

	private String vrsta;

	public Vocka (String vrsta){
		this.vrsta = vrsta;
	}

	public String toString() {
		return this.vrsta;
	}

}
```

Sada se na standardan način može kreirati nekoliko zrna voća unutar beans.xml, gde se svako zrno inicijalizuje pomoću injekcije kroz konstruktor.

```
<bean id=”jabuka” class=”rs.ac.singidunum.fir.pj.Vocka”>
       <constructor-arg name=”vrsta” type=”String” value=”Jabuka”></
       constructor-arg>
</bean>
<bean id=”banana” class=”rs.ac.singidunum.fir.pj.Vocka”>
       <constructor-arg name=”vrsta” type=”String” value=”Banana”></
       constructor-arg>
</bean>
<bean id=”ananas” class=”rs.ac.singidunum.fir.pj.Vocka”>
       <constructor-arg name=”vrsta” type=”String” value=”Ananas”></
       constructor-arg>
</bean>
```

Naravno, u ovom slučaju neophodno je prilagoditi klasu KorpaSaVocem, pošto polje koje sadrži voćke više nije lista stringova, već lista objekata tipa Vocka.

```
package rs.ac.singidunum.fir.pj;

import java.util.List;

public class KorpaSaVocem {

       private String imeKorpe;
       private List<Vocka> vocke;

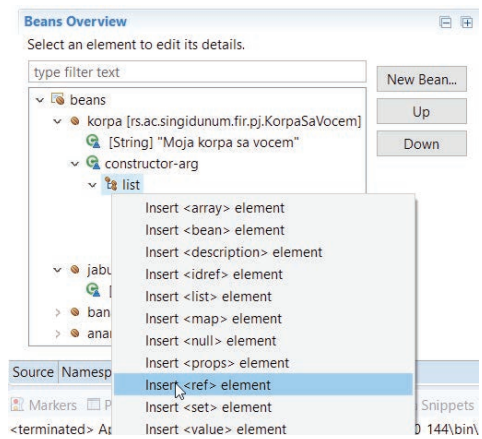
       public KorpaSaVocem(String imeKorpe, List<Vocka> vocke) {
              this.imeKorpe = imeKorpe;
              this.vocke = vocke;
       }

       public String toString(){
              StringBuilder sb = new StringBuilder();
              sb.append(imeKorpe);
              sb.append(“ sadrzi:\n”);
              for (Vocka fruit: vocke){
                     sb.append(fruit);
                     sb.append(“\n”);
              }
```

```
            return sb.toString();
        }

}
```

Na sličan način kao u prethodnom poglavlju, potrebno je injektovati listu voćki, sa tom razlikom što pojedinačne voćke više nisu stringovi već zrna tipa Vocka. Ukoliko se koristi editor za konfiguraciju zrna, unutar liste bi sada bilo potrebno dodavati ref elemente, kao što je prikazano na slici 5.28.



Slika 5.28. Injektovanje liste zrna

Za svaki ref element bi trebalo postaviti referencu na jedno zrno. Međutim, kao i u prethodnom poglavlju, ovde je takođe lakše raditi direktno u beans.xml. XML kod za unošenje referenci na zrna je dovoljno prost da je značajno brže raditi unutar samog fajla nego se snalaziti po editoru. Zrna koja se stavljaju u list element se jednostavno referišu pomoću njihovog Id. Unutar list elementa se dodaju reference na sledeći način.

```
   <?xml version="1.0" encoding="UTF-8"?>
   <beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://www.springframework.org/schema/
beans http://www.springframework.org/schema/beans/spring-beans.xsd">

       <bean id="korpa" class="rs.ac.singidunum.fir.pj.KorpaSaVocem">
```

```
            <constructor-arg name="imeKorpe" type="String"
                  value="Moja korpa sa vocem">
            </constructor-arg>
            <constructor-arg>
                  <list>
                        <ref bean="jabuka"/>
                        <ref bean="banana"/>
                        <ref bean="ananas"/>
                  </list>
            </constructor-arg>
      </bean>
      <bean id="jabuka" class="rs.ac.singidunum.fir.pj.Vocka">
            <constructor-arg      name="vrsta"      type="String"
value="Jabuka"></constructor-arg>
      </bean>
      <bean id="banana" class="rs.ac.singidunum.fir.pj.Vocka">
            <constructor-arg      name="vrsta"      type="String"
value="Banana"></constructor-arg>
      </bean>
      <bean id="ananas" class="rs.ac.singidunum.fir.pj.Vocka">
            <constructor-arg      name="vrsta"      type="String"
value="Ananas"></constructor-arg>
      </bean>
</beans>
```

Bez ikakve naknadne promene glavnog programa, izvršavanje daje sledeći rezultat.

```
Jan 31, 2019 3:45:27 AM org.springframework.context.support.Ab-
stractApplicationContext prepareRefresh
INFO: Refreshing org.springframework.context.support.FileSystemXml
ApplicationContext@6659c656: startup date [Thu Jan 31 03:45:27 CET
2019]; root of context hierarchy
Jan 31, 2019 3:45:27 AM org.springframework.beans.factory.xml.Xml-
BeanDefinitionReader loadBeanDefinitions
INFO: Loading XML bean definitions from file [C:\Spring\Uvod\beans.
xml]
Moja korpa sa vocem sadrzi:
Jabuka
Banana
Ananas
Jan 31, 2019 3:45:27 AM org.springframework.context.support.Ab-
stractApplicationContext doClose
```

```
INFO: Closing org.springframework.context.support.FileSystemXmlAp
plicationContext@6659c656: startup date [Thu Jan 31 03:45:27 CET
2019]; root of context hierarchy
```

5.9. Zadaci za vežbu

Zadatak 1: Napisati Spring projekat upotrebom Maven-a koji sadrži klasu Instrument koji predstavlja jedan muzički instrument. Polja klase su naziv tipa String i polje tipa enum TipInstrumenta koji ima vrednosti: UDARALJKE, DUVACKI i ZICANI. Klasa sadrži konstruktor koji postavlja vrednosti polja na prosleđene i metod toString(). Kreirati jedan objekat (zrno i koristiti injekciju kroz konstruktor kako bi prosledili vrednosti polja klase) i ispisati na standardni izlaz.

Rešenje: pom.xml i beans.xml fajlovi su generisani korišćenjem editora, dok su klasa Instrument i klasa App za testiranje pisane ručno.

pom.xml:

```
<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://ma-
ven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>rs.ac.singidunum.fir.pj</groupId>
  <artifactId>Primer1_Instrumenti</artifactId>
  <version>0.0.1-SNAPSHOT</version>
  <dependencies>
       <dependency>
              <groupId>org.springframework</groupId>
              <artifactId>spring-beans</artifactId>
              <version>5.1.4.RELEASE</version>
       </dependency>
       <dependency>
              <groupId>org.springframework</groupId>
              <artifactId>spring-context</artifactId>
              <version>5.1.4.RELEASE</version>
       </dependency>
       <dependency>
              <groupId>org.springframework</groupId>
              <artifactId>spring-core</artifactId>
              <version>5.1.4.RELEASE</version>
       </dependency>
  </dependencies>
</project>
```

Klasa Instrument:

```
package ra.ac.singidunum.fir.pj;

public class Instrument {
	private String naziv;
	enum TipInstrumenta{UDARALJKE,DUVACKI,ZICANI};
	private TipInstrumenta tip;

	public Instrument(String naziv, TipInstrumenta tip) {
		this.naziv = naziv;
		this.tip = tip;
	}

	public String toString() {
		return "Naziv: " + naziv + ", tip: " + tip;
	}
}
```

beans.xml:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
	xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
	xsi:schemaLocation="http://www.springframework.org/schema/
beans http://www.springframework.org/schema/beans/spring-beans.
xsd">


	<bean id="instrument" class="ra.ac.singidunum.fir.
pj.Instrument">
		<constructor-arg name="naziv" type="String"
value="violina"></constructor-arg>
		<constructor-arg name="tip" type="TipInstrumenta"
			value="ZICANI">
		</constructor-arg>
	</bean>
</beans>
```

Glavna klasa za testiranje:

```
package ra.ac.singidunum.fir.pj;

import org.springframework.context.ApplicationContext;
import
org.springframework.context.support.ClassPathXmlApplicationContext;
```

```
import
org.springframework.context.support.FileSystemXmlApplicationContext;

public class App {
	public static void main(String[] args) {
		ApplicationContext context = new
FileSystemXmlApplicationContext("C:\\Users\\John\\eclipse-work-
space\\Primer1_Instrumenti\\beans.xml");
		Instrument instrument = (Instrument) context.
getBean("instrument");
		System.out.println(instrument.toString());
		//zatvaranje kontekst aplikacije
		((FileSystemXmlApplicationContext)context).close();
	}
}
```

Ispravan izlaz:

```
Naziv: violina, tip: ZICANI
```

Zadatak 2: Proširiti prethodni zadatak klasom Muzičar koja ima polje instrument koji svira tipa Instrument i polje ime tipa String, konstruktor i metod toString() koji će ispisati i podatke instrumenta. Konfigurisati 3 objekta (kao zrna sa odgovarajućom konfiguracijom) klase Instrument i 4 objekta (ponovo zrna sa odgovarajućom konfiguracijom, referiše se adresa za polje instrument) klase Muzičar.

Rešenje:

pom.xml je identičan kao u prethodnom zadatku (samo su generisane zavisnosti).

Klasa Muzicar:

```
package ra.ac.singidunum.fir.pj;

public class Muzicar {
	private Instrument instrument;
	private String ime;

	public Muzicar(String ime) {
		this.ime = ime;
	}

	public void setInstrument(Instrument i) {
		this.instrument = i;
	}
```

```
	public String toString() {
		return “Muzicar “ + ime + “ svira “ + instrument.to-
String();
	}
}
```

beans.xml: U ovom zadatku je kreirano po jedno zrno klase Instrument i klase Muzicar preko editora, a preostali elementi su dodati direktno u beams.xml fajlu jednostavnim kopiranjem i izmenama polja.

```
<?xml version=”1.0” encoding=”UTF-8”?>
<beans xmlns=”http://www.springframework.org/schema/beans”
	xmlns:xsi=”http://www.w3.org/2001/XMLSchema-instance”
	xsi:schemaLocation=”http://www.springframework.org/schema/
beans http://www.springframework.org/schema/beans/spring-beans.
xsd”>

	<bean id=”instrument1” class=”ra.ac.singidunum.fir.
pj.Instrument”>
		<constructor-arg name=”naziv” type=”String”
value=”violina”></constructor-arg>
		<constructor-arg name=”tip” type=”TipInstrumenta”
			value=”ZICANI”>
		</constructor-arg>
	</bean>
	<bean id=”instrument2” class=”ra.ac.singidunum.fir.
pj.Instrument”>
		<constructor-arg name=”naziv” type=”String”
value=”tuba”></constructor-arg>
		<constructor-arg name=”tip” type=”TipInstrumenta”
			value=”DUVACKI”>
		</constructor-arg>
	</bean>
	<bean id=”instrument3” class=”ra.ac.singidunum.fir.
pj.Instrument”>
		<constructor-arg name=”naziv” type=”String”
value=”bubanj”></constructor-arg>
		<constructor-arg name=”tip” type=”TipInstrumenta”
			value=”UDARALJKE”>
		</constructor-arg>
	</bean>
```

```
	<bean id=”muzicar1” class=”ra.ac.singidunum.fir.pj.Muzicar”>
		<constructor-arg name=”ime” type=”String”
value=”Pera Peric”></constructor-arg>
		<property name=”instrument” ref=”instrument1”>
		</property>
	</bean>
	<bean id=”muzicar2” class=”ra.ac.singidunum.fir.pj.Muzicar”>
		<constructor-arg name=”ime” type=”String” value=”Ana
Anic”></constructor-arg>
		<property name=”instrument” ref=”instrument2”>
		</property>
	</bean>
	<bean id=”muzicar3” class=”ra.ac.singidunum.fir.pj.Muzicar”>
		<constructor-arg name=”ime” type=”String”
value=”Mika Mikovic”></constructor-arg>
		<property name=”instrument” ref=”instrument3”>
		</property>
	</bean>
	<bean id=”muzicar4” class=”ra.ac.singidunum.fir.pj.Muzicar”>
		<constructor-arg name=”ime” type=”String”
value=”Zika Zikic”></constructor-arg>
		<property name=”instrument” ref=”instrument1”>
		</property>
	</bean>
</beans>
```

Glavna klasa za testiranje:

```
package ra.ac.singidunum.fir.pj;
import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplica-
tionContext;
import org.springframework.context.support.FileSystemXmlApplica-
tionContext;
public class App {
	public static void main(String[] args) {
		ApplicationContext context = new FileSystemXmlAppli
cationContext(“C:\\Users\\John\\eclipse-workspace\\Primer2_Muzi-
cari\\beans.xml”);

		Muzicar muzicar1 = (Muzicar) context.getBean(“muzicar1”);
		Muzicar muzicar2 = (Muzicar) context.getBean(“muzicar2”);
		Muzicar muzicar3 = (Muzicar) context.getBean(“muzicar3”);
		Muzicar muzicar4 = (Muzicar) context.getBean(“muzicar4”);
```

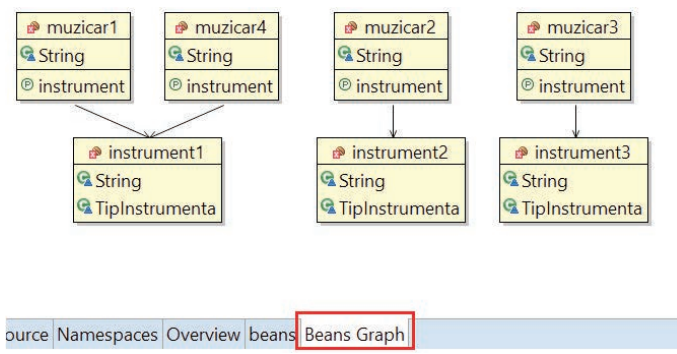
```
            System.out.println(muzicar1.toString());
            System.out.println(muzicar2.toString());
            System.out.println(muzicar3.toString());
            System.out.println(muzicar4.toString());
            //zatvaranje kontekst aplikacije
            ((FileSystemXmlApplicationContext)context).close();

      }
}
```

Ispravan izlaz:

```
Muzicar Pera Peric svira Naziv: violina, tip: ZICANI
Muzicar Ana Anic svira Naziv: tuba, tip: DUVACKI
Muzicar Mika Mikovic svira Naziv: bubanj, tip: UDARALJKE
Muzicar Zika Zikic svira Naziv: violina, tip: ZICANI
```

Interesantno je pogledati Beans Graph tab:



Slika 5.29. Beans Graph tab u editoru

Zadatak 3: Napisati Spring projekat upotrebom Maven-a koji sadrži klasu Teretana koja ima listu članova – listu sa podacima tipa String. Klasa sadrži konstruktor koji postavlja vrednosti polja na prosleđene i metod toString(). Kreirati jedan objekat (zrno i koristiti injekciju kroz konstruktor kako bi prosledili vrednosti polja klase) i ispisati na standardni izlaz.

Rešenje: Kao i u prethodnom zadatku, pom.xml fajl nije menjan. U ovom zadatku se kao argument konstruktora navodi lista. Klasa Teretana je definisana na sledeći način:

```
package rs.ac.singidunum.fir.pj;

import java.util.List;

public class Teretana {
	private String imeTeretane;
	private List<String> clanovi;

	public Teretana(String imeTeretane, List<String> clanovi) {
		this.imeTeretane = imeTeretane;
		this.clanovi = clanovi;
	}
	public String toString(){
		StringBuilder sb = new StringBuilder();
		sb.append("Clanovi teretane ");
		sb.append(imeTeretane + " su:\n");
		for (String clan: clanovi){
			sb.append(clan);
			sb.append("\n");
		}
		return sb.toString();
	}
}
```

Za konfiguraciju zrna *teretana* je ponovo korišćen editor kako bi se definisao prvi argument konstruktora (ime teretane), a elementi liste su dodavani direktno u beans.xml fajlu:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
	xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
	xsi:schemaLocation="http://www.springframework.org/schema/
beans http://www.springframework.org/schema/beans/spring-beans.xsd">

	<bean id="teretana" class="rs.ac.singidunum.fir.pj.Teretana">
		<constructor-arg name="imeTeretane" type="String"
value="Snagator Teretana">
		</constructor-arg>
		<constructor-arg>
			<list>
				<value>Zika Peric</value>
				<value>Mira Mikic</value>
				<value>Jova Lazic</value>
				<value>Zile Zivic</value>
				<value>Maja Maric</value>
				<value>Zika Peric</value>
				<value>Zile Zivic</value>
```

```
                    </list>
                </constructor-arg>
        </bean>
    </beans>
```

Glavna klasa:

```
package rs.ac.singidunum.fir.pj;

import org.springframework.context.ApplicationContext;
import   org.springframework.context.support.FileSystemXmlApplica-
tionContext;

public class Glavna {
        public static void main(String[] args) {
                ApplicationContext context = new FileSystemXmlApplic
ationContext(“beans.xml”);

                Teretana    teretana    =    (Teretana)    context.
getBean(“teretana”);
                System.out.println(teretana);

                ((FileSystemXmlApplicationContext)context).close();
        }
}
```

Izlaz ovog koda je:

```
Clanovi teretane Snagator Teretana su:
Zika Peric
Mira Mikic
Jova Lazic
Zile Zivic
Maja Maric
Zika Peric
Zile Zivic
```

Zadatak 4: Napisati Spring projekat upotrebom Maven-a koji sadrži klasu Proizvod koja ima polja naziv tipa String i barkod tipa long, konstruktor, metodu toString() i metodu za promenu barkoda. Kreirati 3 objekta klase Proizvod – zrna sa odgovarajućom konfiguracijom. Klasa Prodavnica sadrži lisut proivoda i naziv prodavnice. Klasa sadrži konstruktor koji postavlja vrednosti polja na prosleđene i metod toString(). Kreirati jedan objekat (zrno i koristiti injekciju kroz konstruktor kako bi prosledili vrednosti polja klase) klase Prodavnica i ispisati ga na standardni izlaz.

Rešenje: Ponovo, pom.xml fajl nije menjan. Ovaj zadatak ćemo raditi postepeno. Prvo je potrebno kreirati klasu Proizvod i konfigurisati tri zrna.

```
package rs.ac.singidunum.fir.pj;

public class Proizvod {
	private String naziv;
	private long barkod;

	public Proizvod(String naziv, long barkod) {
		this.naziv = naziv;
		this.barkod = barkod;
	}

	public String toString() {
		return naziv + “, barkod: “ + barkod;
	}

	public void setBarkod(long barkod) {
		this.barkod = barkod;
	}
}
```

Fajl beans.xml koji konfiguriše tri zrna klase Proizvod:

```
<?xml version=”1.0” encoding=”UTF-8”?>
<beans xmlns=”http://www.springframework.org/schema/beans”
	xmlns:xsi=”http://www.w3.org/2001/XMLSchema-instance”
	xsi:schemaLocation=”http://www.springframework.org/schema/
beans http://www.springframework.org/schema/beans/spring-beans.xsd”>

	<bean id=”prozvod1” class=”rs.ac.singidunum.fir.pj.Proizvod”>
		<constructor-arg	name=”naziv”	type=”String”
value=”Sok”></constructor-arg>
		<constructor-arg	name=”barkod”	type=”long”	val-
ue=”11223345”></constructor-arg>
	</bean>
	<bean id=”prozvod2” class=”rs.ac.singidunum.fir.pj.Proizvod”>
		<constructor-arg	name=”naziv”	type=”String”
value=”Mleko”></constructor-arg>
		<constructor-arg	name=”barkod”	type=”long”	val-
ue=”123456”></constructor-arg>
	</bean>
```

```
      <bean id="prozvod3" class="rs.ac.singidunum.fir.pj.Proizvod">
            <constructor-arg name="naziv" type="String"
value="Hleb"></constructor-arg>
            <constructor-arg name="barkod" type="long" val-
ue="111111"></constructor-arg>
      </bean>
   </beans>
```

Sada pišemo klasu Prodavnica koja ima kao polje listu proizvoda i naziv prodavnice:

```
import java.util.List;

public class Prodavnica {
      private String naziv;
      private List<Proizvod> proizvodi;

      public Prodavnica(String naziv, List<Proizvod> proizvodi) {
            this.naziv = naziv;
            this.proizvodi = proizvodi;
      }

      public List<Proizvod> getProizvodi() {
            return proizvodi;
      }

      public String toString() {
            StringBuilder sb = new StringBuilder();
            sb.append("Spisak proizvoda u prodavnici ");
            sb.append(naziv + " je:\n");
            for (Proizvod proizvod : proizvodi) {
                  sb.append(proizvod);
                  sb.append("\n");
            }
            return sb.toString();
      }
}
```

Sledeći zadatak je da se konfiguriše zrno za jednu Prodavnicu što se postiže dodavanjem iz editora a rezultat u beans.xml fajlu su dodati sledeći redovi:

```
<bean id="prodavnica" class="rs.ac.singidunum.fir.pj.Prodavnica">
            <constructor-arg name="naziv" type="String"
                  value="Nasa mala prodavnica">
```

```
            </constructor-arg>
            <constructor-arg>
                  <list>
                  </list>
            </constructor-arg>
```

Naša lista proizvoda je trenutno prazna. U prethodnom zadatku su dodavane vrednosti za prmitivni tip unutar value tagova. U ovom primeru želimo da injektujemo listu proizvoda. To se može postići povezivanjem preko Id zrna. U beans.xml naša lista sada izgleda ovako:

```
<constructor-arg>
      <list>
            <ref bean="proizvod1" />
            <ref bean="proizvod2" />
            <ref bean="proizvod3" />
      </list>
</constructor-arg>
```

Konačni beans.xml fajl:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xsi:schemaLocation="http://www.springframework.org/schema/
beans http://www.springframework.org/schema/beans/spring-beans.
xsd">

      <bean id="proizvod1" class="rs.ac.singidunum.fir.
pj.Proizvod">
            <constructor-arg name="naziv" type="String"
value="Sok"></constructor-arg>
            <constructor-arg name="barkod" type="long" val-
ue="11223345"></constructor-arg>
      </bean>
      <bean id="proizvod2" class="rs.ac.singidunum.fir.
pj.Proizvod">
            <constructor-arg name="naziv" type="String"
value="Mleko"></constructor-arg>
            <constructor-arg name="barkod" type="long" val-
ue="123456"></constructor-arg>
      </bean>
```

```
	<bean id="proizvod3" class="rs.ac.singidunum.fir.
pj.Proizvod">
		<constructor-arg name="naziv" type="String"
value="Hleb"></constructor-arg>
		<constructor-arg name="barkod" type="long" val-
ue="111111"></constructor-arg>
	</bean>
	<bean id="prodavnica" class="rs.ac.singidunum.fir.
pj.Prodavnica">
		<constructor-arg name="naziv" type="String"
			value="Nasa mala prodavnica">
		</constructor-arg>
		<constructor-arg>
			<list>
				<ref bean="proizvod1" />
				<ref bean="proizvod2" />
				<ref bean="proizvod3" />

			</list>
		</constructor-arg>
	</bean>
</beans>
```

Na kraju glavna klasa:

```
package rs.ac.singidunum.fir.pj;

import org.springframework.context.ApplicationContext;
import   org.springframework.context.support.FileSystemXmlApplica-
tionContext;

public class Glavna {
	public static void main(String[] args) {
		ApplicationContext context = new FileSystemXmlApplic
ationContext("beans.xml");
		Prodavnica   prodavnica   =   (Prodavnica)   context.
getBean("prodavnica");
		System.out.println(prodavnica);

		//menjamo bar kod hlebu
		prodavnica.getProizvodi().get(2).setBarkod(234567);
		System.out.println(prodavnica);

		((FileSystemXmlApplicationContext)context).close();
	}
}
```

Izlaz ovog programa:

```
Spisak proizvoda u prodavnici Nasa mala prodavnica je:
Sok, barkod: 11223345
Mleko, barkod: 123456
Hleb, barkod: 111111

Spisak proizvoda u prodavnici Nasa mala prodavnica je:
Sok, barkod: 11223345
Mleko, barkod: 123456
Hleb, barkod: 234567
```

Zadatak 5: Napisati Spring projekat upotrebom Maven-a koji sadrži klasu Kurs koja ima polja nazivFakulteta tipa String i parove <šifra predmeta, naziv>, konstruktor, metodu toString(). Kreirati nekoliko objekata (zrno i koristiti injekciju kroz konstruktor kako bi prosledili vrednosti polja klase) klase Kurs i ispisati ih na standardni izlaz.

Rešenje: Ponovo, pom.xml fajl nije menjan. Za parove <šifra predmeta, naziv> ćemo koristiti java.util.Map

Klasa Kurs:

```
package rs.ac.singidunum.fir.pj;
import java.util.*;
import java.util.Map.Entry;

public class Kurs {
       private String fakultet;
       private Map<String,String> idKursa;
       public Kurs(String fakultet, Map<String, String> idKursa) {
              this.fakultet = fakultet;
              this.idKursa = idKursa;
       }

       public String toString() {
              StringBuilder sb = new StringBuilder();
              sb.append(“Kursevi na fakultetu “ + fakultet + “: \n”);
         Iterator<Entry<String, String>> iter = idKursa.entry-
Set().iterator();
          while (iter.hasNext()) {
              Entry<String, String> entry = iter.next();
              sb.append(“Kurs: “ + entry.getValue());
              sb.append(“, sifra kursa: “ +entry.getKey());
```

```
            sb.append("\n");
        }
        return sb.toString();
      }
}
```

Fajl beans.xml:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://www.springframework.org/schema/
beans http://www.springframework.org/schema/beans/spring-beans.
xsd">

       <bean id="fir" class="rs.ac.singidunum.fir.pj.Kurs">
             <constructor-arg name="fakultet" type="String"
value="FIR"></constructor-arg>

             <constructor-arg>
                   <map>
                         <entry key="1" value="Osnove programi-
ranja"></entry>
                         <entry key="2" value="Objektno orjenti-
sano programiranje"></entry>
                         <entry key="3" value="Programski jezi-
ci"></entry>
                         <entry key="4" value="Razvoj mobilnih
aplikacija"></entry>
                         <entry key="5" value="Cloud programi-
ranje"></entry>
                         <entry key="6" value="Baze podataka"></
entry>
                   </map>
             </constructor-arg>
       </bean>
</beans>
```

Glavna klasa:

```
package rs.ac.singidunum.fir.pj;

import org.springframework.context.ApplicationContext;
import org.springframework.context.support.FileSystemXmlApplica-
tionContext;
```

```
public class Glavna {
	public static void main(String[] args) {
		ApplicationContext context = new FileSystemXmlApplic
ationContext("beans.xml");
		Kurs kurs = (Kurs) context.getBean("fir");
		System.out.println(kurs);

		((FileSystemXmlApplicationContext)context).close();
	}
}
```

Izlaz:

```
Kursevi na fakultetu FIR:
Kurs: Osnove programiranja, sifra kursa: 1
Kurs: Objektno orjentisano programiranje, sifra kursa: 2
Kurs: Programski jezici, sifra kursa: 3
Kurs: Razvoj mobilnih aplikacija, sifra kursa: 4
Kurs: Cloud programiranje, sifra kursa: 5
Kurs: Baze podataka, sifra kursa: 6
```

Zadatak 6: Napisati Spring projekat upotrebom Maven-a koji sadrži klasu Profesor koja ima polja ime, jmbg i listu predmeta koje predaje (lista stringova). Potrebno je ispisati poruku „Profesor *imeProfesora* je počeo da predaje" svaki put kad se kreira objekat (koristiti metodu onCreate()) i poruku „Profesor *imeProfesora* vise ne predaje" neposredno pre uništavanja zrna (onDestroy() metod).

Rešenje:

Klasa Profesor:

```
package ac.rs.singidunum.fir.pj;

import java.util.List;

public class Profesor {
	private String ime, jmbg;
	private List<String> predmeti;
	public Profesor(String ime, String jmbg, List<String> pred-
meti) {
		this.ime = ime;
		this.jmbg = jmbg;
		this.predmeti = predmeti;
	}
```

```
	public String toString() {
		StringBuilder sb = new StringBuilder();
		sb.append(“Profesor “ + ime + “ (jmbg: “ + jmbg + “)
predaje sledece predmete:\n”);
		for (String predmet : predmeti) {
			sb.append(predmet);
			sb.append(“\n”);
		}
		return sb.toString();
	}

	public void onCreate () {
		System.out.println(“Profesor “ + this.ime + “ je po-
ceo da predaje.”);
	}

	public void onDestroy () {
		System.out.println(“Profesor “ + this.ime + “ vise ne
predaje.”);
	}

}
```

Fajl beans.xml:

```
<?xml version=”1.0” encoding=”UTF-8”?>
<beans xmlns=”http://www.springframework.org/schema/beans”
	xmlns:xsi=”http://www.w3.org/2001/XMLSchema-instance”
	xsi:schemaLocation=”http://www.springframework.org/schema/
beans http://www.springframework.org/schema/beans/spring-beans.xsd”
	default-destroy-method=”onDestroy” default-init-
method=”onCreate”>

	<bean id=”pera” class=”ac.rs.singidunum.fir.pj.Profesor”>
		<constructor-arg name=”ime” type=”String”
value=”Pera Peric”></constructor-arg>
		<constructor-arg name=”jmbg” type=”String” val-
ue=”2601901231231”></constructor-arg>
		<constructor-arg>
			<list>
				<value>Programski jezici</value>
				<value>OOP</value>
				<value>Osnove programiranja</value>
			</list>
```

```
            </constructor-arg>
      </bean>

      <bean id=”mara” class=”ac.rs.singidunum.fir.pj.Profesor”>
            <constructor-arg name=”ime” type=”String” value=”Mara
Lazic”></constructor-arg>
            <constructor-arg   name=”jmbg”   type=”String”   val-
ue=”2301901231232”></constructor-arg>
            <constructor-arg>
                  <list>
                        <value>Engleski jezik</value>
                        <value>Ruski jezik</value>
                  </list>
            </constructor-arg>
      </bean>
</beans>
```

Glavna klasa:

```
package ac.rs.singidunum.fir.pj;

import org.springframework.context.ApplicationContext;
import   org.springframework.context.support.FileSystemXmlApplica-
tionContext;

public class Glavna {
      public static void main(String[] args) {
            ApplicationContext context = new FileSystemXmlApplic
ationContext(“beans.xml”);
            Profesor prof = (Profesor) context.getBean(“pera”);
            System.out.println(prof);

            Profesor profM = (Profesor) context.getBean(“mara”);
            System.out.println(profM);
            ((FileSystemXmlApplicationContext)context).close();
      }
}
```

Izlaz:

```
Profesor Pera Peric je poceo da predaje.
Profesor Mara Lazic je poceo da predaje.
Profesor Pera Peric (jmbg: 2601901231231) predaje sledece predmete:
Programski jezici
OOP
Osnove programiranja

Profesor Mara Lazic (jmbg: 2301901231232) predaje sledece predmete:
Engleski jezik
Ruski jezik

Profesor Mara Lazic vise ne predaje.
Profesor Pera Peric vise ne predaje.
```

Zadaci za samostalnu vežbu:

Proširiti drugi zadatak klasom Orkestar koja će sadržati listu Muzičara. Testirati rad kreiranjem zrna orkestar i konfigurisati listu muzičara.

Napsati klasu Profesor koja ima polja ime i jmbg. Koristiti klasu Kurs iz petog primera i napraviti klasu Zaposleni koja sadrži spisak svih zaposlenih na univerzitetu. Potrebno je da se profesori ne ponavljaju. Koristiti Spring i Maven i sve objekte konfigurisati kao zrna. Implementirati injekciju objekata po potrebi.

Napisati klasu Firma koja će imati polje naziv, pib i listu osoba gde je svaka osoba zrno klase Osoba opisana u ovom poglavlju. Koristiti Klasu Osoba koja ima polje tipa Adresa.

Pitanja

1. Kako se još naziva i za šta je zadužen Sring kontejner?
2. Koja su osnovna dva tipa Spring kontejnera?
3. Koje oblike konteksta aplikacije nudi Spring?
4. Koja je razlika između učitavanja aplikacije korišćenjem `FileSystemXmlApplicationContext i ClassPathXmlApplicationContext`?
5. Kako objekti mogu da deklarišu svoje zavisnosti?
6. Koji je koreni element konfiguracionog XML fajla za deklaraciju zrna?
7. Koji tag se koriste za kreiranje zrna?
8. Kada se koristi injekcija kroz konstruktor?
9. Kada se koristi injekcija polja?
10. Čemu služe metode init i destroy?
11. Koji opsezi zrna postoje u Springu?
12. Koje konfiguracione elemente koji odgovaraju kolekcijama u Javi nudi Spring?
13. Koja je razlika između <list> i <set> elemenata za konfiguracija polja?
14. Da li se <list> može koristiti za konfiguraciju nizova?
15. Koja je razlika između <map> i <prps> elemenata za konfiguracija polja?
16. Koji tag se koristi za upisivanje vrednosti u listu u XML konfiguracionom fajlu?
17. Da li se mogu injektovati liste zrna ili je moguće koristiti samo liste primitivnih tipova?
18. Kako se injektuju liste zrna u Spring-u?

VI
Spring - Automatsko ožičavanje zrna

Deklarisanje zrna upotrebom <bean> elementa, kao i njihova konfiguracija pomoću injekcije odgovarajućih primitivnih vrednosti ili referenci na druga zrna upotrebom <constructor-arg> ili <property> elemenata, sasvim je prihvatljiva u slučaju male aplikacije sa malim ukupnim brojem zrna. Međutim, kako aplikacija raste, tako raste i količina XML konfiguracije koju je potrebno napisati. Kako bi se taj problem rešio, Spring nudi opciju za automatsko ožičavanje (engl. *autowiring*), čime se u velikoj meri redukuje, ili čak u potpunosti eliminiše potreba za upotrebom <constructor-arg> i <property> elemenata – Springu se dozvoljava da automatski odluči kako da ožiči zrna.

Druga opcija koju Spring nudi – automatsko otkrivanje zrna (engl. *autodiscovery*) ide još jedan korak dalje, tako što se Springu omogućava da samostalno odluči koje klase treba da se konfigurišu kao Spring zrna, čime se dodatno umanjuje i potreba za <bean> elementima. Kada se koriste zajedno, automatsko ožičavanje i automatsko otkrivanje zrna mogu drastično smanjiti količinu koda u XML konfiguracionom fajlu. Najčešće na kraju bude potrebno samo nekoliko linija XML, nezavisno od veličine projekta i broja zrna u kontekstu aplikacije.

Koja je logika u pozadini automatskog ožičavanja? Ako bismo rekli na primer rečenicu:

- Mesec je svetao noćas.

Verovatno niko ne bi odgovorio pitanjem – koji mesec? To je zato što se svi nalazimo na planeti Zemlji, i potpuno je očigledno da se pojam mesec odnosi na Zemljin satelit Mesec. Da se hipotetički nalazimo na Jupiteru i kažemo istu rečenicu, u tom slučaju bi pitanje na koji mesec se misli bilo opravdano, pošto Jupiter ima više od 60 prirodnih satelita. Na Zemlji, dvosmislenost ne postoji. Ista logika se koristi i u ožičavanju zrna. Kada je potrebno ožičiti zrna, ponekad je poptuno očigledno koje zrno treba ožičiti, odnosno injektovati u dato polje drugog zrna. Na primer, ukoliko u kontekstu aplikacije postoji samo jedno zrno tipa javax.sql.DataSource, svakom zrnu kome treba DataSource treba upravo to jedno zrno DataSource. Spring iskorišćava situaciju da vrlo često postoji veliki broj očiglednih i nedvomislenih ožičavanja, i nudi opciju automatskog ožičavanja.

Spring nudi nekoliko načina automatskog ožičavanja:

- *byName* – po imenu, odnosno Spring pokušava da upari sva polja automatski ožičenog zrna sa zrnima koja imaju isto ime (odnosno Id) kao i ime polja. Polja za koje nije pronađeno zrno koje može da se upari ostaju neožičena.
- *byType* – po tipu, odnosno Spring pokušava da upari sva polja automatski ožičenog zrna sa zrnima čiji se tip može dodeliti polju. Isto kao i u prethodnom slučaju, polja za koje nije pronađeno zrno koje može da se upari ostaju neožičena.
- *constructor* – po konstruktoru, odnosno Spring pokušava da upari konstruktor automatski ožičenog zrna sa zrnima čiji se tipovi mogu dodeliti argumentima konstruktora.

U nastavku ovog poglavlja biće pokriveni svi načini automatskog ožičavanja zrna, kao i automatskog otkrivanja zrna.

6.1. Automatsko ožičavanje po imenu

U Springu, ime se daje praktično svemu. Svako zrno ima polja kojima su data imena, kao što i druga zrna koja treba da se injektuju u ta polja imaju svoja imena. Spring koristi situaciju koja se često javlja, a to je da se ime polja obično poklapa sa imenom zrna koje je potrebno injektovati (ožičiti) u to polje. To poklapanje po imenu služi Springu kao nagoveštaj koje zrno je potrebno automatski ožičiti u polje.

Postupak je najlakše pokazati kroz primer. Posmatra se program koji ima funkciju logera. Loger ima nekoliko različitih implementacija logovanja, koje su realizovane kao zasebne klase, koje sve implementiraju jedan dati interfejs za logovanje. Projekat je podešen na način kao što je prikazano u poglavlju 4.6. Potrebne su iste zavisnosti unutar pom.xml fajla, čiji je sadržaj prikazan 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://ma-
ven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>rs.ac.singidunum.fir.pj</groupId>
  <artifactId>Spring-uvod</artifactId>
  <version>0.0.1-SNAPSHOT</version>
  <dependencies>
             <dependency>
             <groupId>org.springframework</groupId>
             <artifactId>spring-core</artifactId>
             <version>5.0.7.RELEASE</version>
```

```
		</dependency>
		<dependency>
			<groupId>org.springframework</groupId>
			<artifactId>spring-beans</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>
</project>
```

Neka u projektu postoje jedan interfejs i dve klase. Interfejs LogPisac definiše metodu koju sve implementirajuće klase imaju. To je metoda pisiLiniju(), koja upisuje jednu liniju u odgovarajući log. Klasa FajlPisac implementira ovaj interfejs i piše jednu liniju teksta u fajl. Sa druge strane, klasa KonzolniPisac ispisuje jednu liniju teksta na standardnom izlazu. Ovo je standardna struktura klasa kada je potrebno imati mogućnost logovanja na nekoliko različitih načina. Izvorni kod interfejsa i ove dve klase dat je u nastavku.

```
package rs.ac.singidunum.fir.pj;

public interface LogPisac {

	void pisiLiniju (String linija);

}
```

```
package rs.ac.singidunum.fir.pj;

public class FajlPisac implements LogPisac {

	public void pisiLiniju(String linija) {
		//dummy implementacija
		//ovde bi trebalo da ide upis u fajl
		System.out.println("Upis u fajl: " + linija);
	}

}
```

```
package rs.ac.singidunum.fir.pj;

public class KonzolniPisac implements LogPisac {

	public void pisiLiniju (String linija) {
		System.out.println("Ispis na konzolu: " + linija);
	}

}
```

Pored ove dve klase, postoji i klasa Loger, koja ima dva polja, jedno tipa FajlPisac, a drugo tipa KonzolniPisac. Oba ova polja se setuju kroz set metode. Ispis u fajl se poziva metodom pisiUFajl(), dok se ispis na konzolu poziva metodom pisiNaKonzolu(). Kod klase Loger je dat u nastavku.

```
package rs.ac.singidunum.fir.pj;

//dummy implementacija logera
//loger moze da upisuje i u fajl i na konzolu
public class Loger {

	private KonzolniPisac konzolniPisac;
	private FajlPisac fajlPisac;

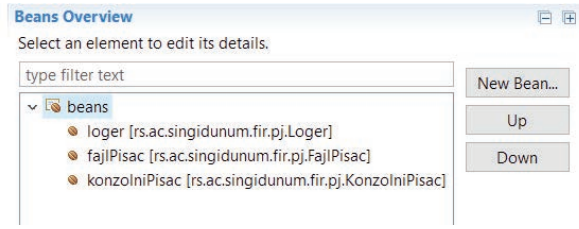
	public void setKonzolniPisac(KonzolniPisac konzolniPisac) {
		this.konzolniPisac = konzolniPisac;
	}
	public void setFajlPisac(FajlPisac fajlPisac) {
		this.fajlPisac = fajlPisac;
	}

	public void pisiUFajl (String text){
		fajlPisac.pisiLiniju(text);
	}

	public void pisiNaKonzolu (String text){
		konzolniPisac.pisiLiniju(text);
	}

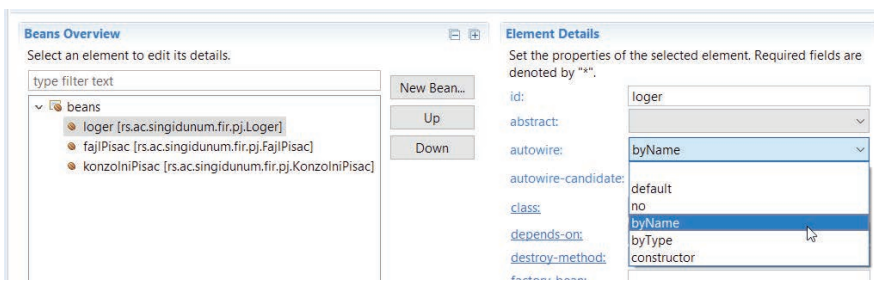

}
```

Sada je potrebno kreirati tri zrna, loger, konzolniPisac i fajlPisac unutar beans.xml, kao što je prikazano na slici 6.1.



Slika 6.1. Zrna u programu Loger

Sada se može iskoristiti povoljna situacija da polja unutar klase Loger, konzolniPisac i fajlPisac, imaju identična imena kao i zrna koja je potrebno injektovati na te pozicije, uz potrebu i da set metode budu nazvane po konvenciji. U tom slučaju, u konfiguraciji zrna Loger moguće je odabrati *autowire byName*, kao što je prikazano na slici 6.2.



Slika 6.2. Podešavanje automatskog ožičavanja po imenu u editoru

Na ovaj način, podesili smo da će Spring pokušati samostalno, automatski da odredi prema imenu zrna i imenima polja zrna Loger da odredi kako da izvrši ožičavanje. Ovde se samo mora napomenuti da u ovom slučaju, kako bi ožičavanje po imenu radilo ispravno, Id zrna i ime polja u koje bi to zrno trebalo biti injektovano moraju biti identični, računajući i mala i velika slova. Takođe, set metode moraju biti imenovane po konvenciji, odnosno setKonzolniPisac() i setFajlPisac(). Nakon podešavanja automatskog ožičavanja po imenu, beans.xml će imati sledeći sadržaj (uz napomenu da je naravno ovo bilo moguće pisati direktno u beans.xml, bez upotrebe editora).

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://www.springframework.org/schema/
beans http://www.springframework.org/schema/beans/spring-beans.xsd">
```

```
        <bean id="loger" class="rs.ac.singidunum.fir.pj.Loger"
                autowire="byName">
        </bean>
        <bean id="fajlPisac"
class="rs.ac.singidunum.fir.pj.FajlPisac"></bean>
        <bean id="konzolniPisac"
                class="rs.ac.singidunum.fir.pj.KonzolniPisac">
        </bean>
</beans>
```

Sada možemo napisati jednostavnu klasu App koja će demonstrirati rad programa.

```
package rs.ac.singidunum.fir.pj;

import org.springframework.context.ApplicationContext;
import org.springframework.context.support.FileSystemXmlApplica-
tionContext;

public class App {
        public static void main (String[] args){

                ApplicationContext context = new FileSystemXmlApplic
ationContext("beans.xml");
                Loger loger = (Loger) context.getBean("loger");
                loger.pisiUFajl("Zdravo");
                loger.pisiNaKonzolu("Zdravo");
                ((FileSystemXmlApplicationContext)context).close();

        }
}
```

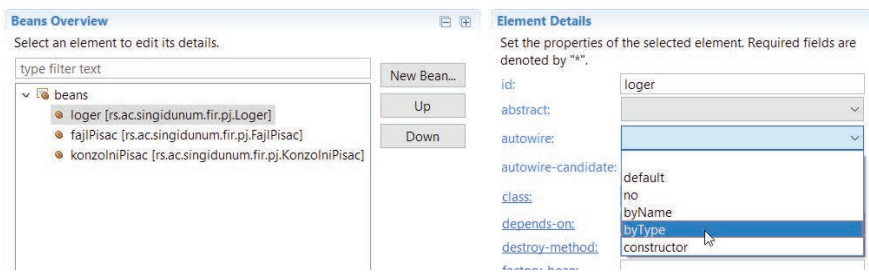
Nakon pokretanja izvršavanja, program daje sledeći izlaz.

```
Jan 31, 2019 10:02:56 PM org.springframework.context.support.Ab-
stractApplicationContext prepareRefresh
INFO: Refreshing org.springframework.context.support.FileSystemXml
ApplicationContext@6659c656: startup date [Thu Jan 31 22:02:56 CET
2019]; root of context hierarchy
Jan 31, 2019 10:02:56 PM org.springframework.beans.factory.xml.
XmlBeanDefinitionReader loadBeanDefinitions
INFO: Loading XML bean definitions from file [C:\Spring\Uvod\beans.xml]
```

```
Upis u fajl: Zdravo
Ispis na konzolu: Zdravo
   Jan 31, 2019 10:02:56 PM org.springframework.context.support.
AbstractApplicationContext doClose
   INFO: Closing org.springframework.context.support.FileSystemXml
ApplicationContext@6659c656: startup date [Thu Jan 31 22:02:56 CET
2019]; root of context hierarchy
```

6.2. Automatsko ožičavanje po tipu

Automatsko ožičavanje po tipu radi na sličan način kao i po imenu, uz razliku da se umesto provere imena polja, u ovom slučaju proverava tip polja. Kada pokušava ožičavanje po tipu, Spring traži zrna čiji se tipovi mogu automatski dodeliti poljima. Posmatrajmo ponovo primer Logera. U ovom slučaju se bira *autowire byType*, kao na slici 6.3.



Slika 6.3. Podešavanje automatskog ožičavanja po tipu u editoru

Sadržaj beans.xml u ovom slučaju je skoro identičan kao u slučaju automatskog ožičavanja po imenu. Razlikuje se zapravo samo vrednost atributa *autowire* unutar loger zrna.

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.xsd">
       <bean id="loger" class="rs.ac.singidunum.fir.pj.Loger"
             autowire="byType">
       </bean>
```

```
      <bean id=”fajlPisac” class=”rs.ac.singidunum.fir.
pj.FajlPisac”></bean>
      <bean id=”konzolniPisac”
            class=”rs.ac.singidunum.fir.pj.KonzolniPisac”>
      </bean>
  </beans>
```

Prilikom učitavanja konteksta, Spring će pokušati da u polje fajlPisac zrna loger injektuje zrno čiji je tip FajlPisac. Pošto postoji zrno tog tipa (zrno sa Id *konzolniPisac*), Spring će ga odabrati i ožičiti u polje fajlPisac zrna loger. Nije potrebno menjati nijedan drugi deo programa, rezultat izvršavanja će biti isti.

```
Jan 31, 2019 10:18:28 PM org.springframework.context.support.Ab-
stractApplicationContext prepareRefresh
INFO: Refreshing org.springframework.context.support.FileSystemXml
ApplicationContext@6659c656: startup date [Thu Jan 31 22:18:28 CET
2019]; root of context hierarchy
Jan 31, 2019 10:18:28 PM org.springframework.beans.factory.xml.
XmlBeanDefinitionReader loadBeanDefinitions
INFO: Loading XML bean definitions from file [C:\Spring\Uvod\beans.
xml]
Upis u fajl: Zdravo
Ispis na konzolu: Zdravo
Jan 31, 2019 10:18:28 PM org.springframework.context.support.Ab-
stractApplicationContext doClose
INFO: Closing org.springframework.context.support.FileSystemXmlAp
plicationContext@6659c656: startup date [Thu Jan 31 22:18:28 CET
2019]; root of context hierarchy
```

Šta se dešava u slučaju da Spring pronađe više od jednog zrna kada pokušava da uradi automatsko ožičavanje po tipu, odnosno da postoji više zrna koja se mogu dodeliti u neko polje automatski ožičenog zrna? U tom slučaju, Spring neće nagađati koje zrno treba da se ožiči i jednostavno će izbaciti izuzetak. Spring ne dozvoljava dvosmislenost ni u jednom obliku. Dakle, dozvoljeno je imati samo jedno zrno koje se može dodeliti u neko polje automatski ožičenog zrna, kada se radi ožičavanje po tipu. Kako bismo ilustrovali ovaj problem, posmatrajmo nešto drugačiju implementaciju klase Loger.

```
package rs.ac.singidunum.fir.pj;

//dummy implementacija logera
//loger moze da upisuje i u fajl i na konzolu
public class Loger {

	private LogPisac konzolniPisac;
	private LogPisac fajlPisac;

	public void setKonzolniPisac(LogPisac konzolniPisac) {
		this.konzolniPisac = konzolniPisac;
	}
	public void setFajlPisac(LogPisac fajlPisac) {
		this.fajlPisac = fajlPisac;
	}

	public void pisiUFajl (String text){
		fajlPisac.pisiLiniju(text);
	}

	public void pisiNaKonzolu (String text){
		konzolniPisac.pisiLiniju(text);
	}


}
```

Sada postoje dve klase koje se mogu injektovati u bilo koje od dva polja klase Loger. Svako polje je tipa LogPisac, što znači da se u njega može postaviti bilo objekat tipa FajlPisac, bilo KonzolniPisac. Spring neće moći da odredi koje od dva zrna treba da ide u koje polje, i izbaciće izuzetak. Ukoliko se program pokrene, rezultat izvršavanja je dat u nastavku.

```
Jan 31, 2019 10:38:29 PM org.springframework.context.support.Ab-
stractApplicationContext prepareRefresh
INFO: Refreshing org.springframework.context.support.FileSystemX
mlApplicationContext@6659c656: startup date [Thu Jan 31 22:38:29
CET 2019]; root of context hierarchy
Jan 31, 2019 10:38:29 PM org.springframework.beans.factory.xml.
XmlBeanDefinitionReader loadBeanDefinitions
INFO: Loading XML bean definitions from file [C:\Spring\Uvod\beans.xml]
Jan 31, 2019 10:38:30 PM org.springframework.context.support.Ab-
stractApplicationContext refresh
```

```
WARNING: Exception encountered during context initialization -
cancelling refresh attempt: org.springframework.beans.factory.
UnsatisfiedDependencyException: Error creating bean with name
'loger' defined in file [C:\Spring\Uvod\beans.xml]: Unsatisfied
dependency expressed through bean property 'fajlPisac'; nested
exception is org.springframework.beans.factory.NoUniqueBeanDefini-
tionException: No qualifying bean of type 'rs.ac.singidunum.fir.
pj.LogPisac' available: expected single matching bean but found
2: fajlPisac,konzolniPisac
Exception in thread "main" org.springframework.beans.factory.
UnsatisfiedDependencyException: Error creating bean with name
'loger' defined in file [C:\Spring\Uvod\beans.xml]: Unsatisfied
dependency expressed through bean property 'fajlPisac'; nested
exception is org.springframework.beans.factory.NoUniqueBeanDefini-
tionException: No qualifying bean of type 'rs.ac.singidunum.fir.
pj.LogPisac' available: expected single matching bean but found
2: fajlPisac,konzolniPisac
...
```

Izbačeni izuzetak jasno pokazuje da je Spring pronašao dva poklapanja po tipu, iako je dozvoljeno samo jedno (`NoUniqueBeanDefinitionException: No qualifying bean of type 'rs.ac.singidunum.fir.pj.LogPisac' available: expected single matching bean but found 2: fajlPisac,konzolniPisac`). U ovakvom slučaju bi se moglo koristiti ožičavanje po imenu, koje bi rešilo ovaj problem (svaki Id zrna je jedinstven pa ne može doći do dvosmislenosti). Drugi način za rešavanje ovog problema bilo bi podešavanje primarnog kandidata za automatsko ožičavanje po tipu. Unutar <bean> elementa se može postaviti atribut *primary*, koji sa vrednostima true ili false označava da li je zrno primarni kandidat za ožičavanje. Ukoliko samo jedan kandidat za automatsko ožičavanje ima *primary* atribut podešen na true, on će biti odabran isperd drugih kandidata. Samo treba biti oprezan, za *primary* atribut važi da je podrazumevana vrednost true. To znači da je podrazumevano da su svi kandidati za ožičavanje primarni (odnosno da se nijedan ne preferira u odnosu na druge). Kako bi se koristio ovaj atribut da bi se neko zrno proglasilo za primarni izbor, potrebno je u sva ostala zrna postaviti vrednost false kako bi se označilo da nisu primarni izbor.

6.3. Automatsko ožičavanje po konstruktoru

Ukoliko je zrno konfigurisano na takav način da se koristi injekcija kroz konstruktor, umesto <constructor-arg> elementa može se dozvoliti Springu da automatski odabere argumente konstruktora iz skupa zrna koji se nalaze u kontekstu. Automatsko ožičavanje po konstruktoru se prvo pokušava po tipu, i samim tim ima ista ograničenja kao i ožičavanje po tipu, odnosno može doći do dvosmislenosti. Spring

neće pokušati da pogađa koje zrno da ožiči u konstruktor ukoliko pronađe više zrna koja odgovaraju argumentima konstuktora. Ukoliko dođe to takve situacije, Spring će pokušati da uradi automatsko ožičavanje po imenu. Posmatrajmo ponovo isti primer Logera kao u prethodnim poglavljima, sa malom izmenom klase Loger – u ovom slučaju dodat je konstruktor, pošto ožičavanje treba da bude implementirano po konstruktoru.

```
package rs.ac.singidunum.fir.pj;

//dummy implementacija logera
//loger moze da upisuje i u fajl i na konzolu
public class Loger {

	private LogPisac konzolniPisac;
	private LogPisac fajlPisac;

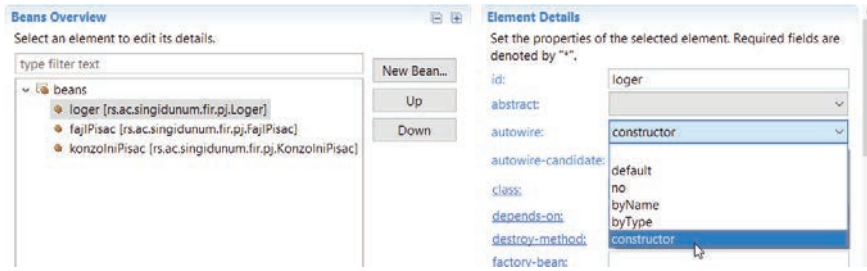
	public Loger(LogPisac konzolniPisac, LogPisac fajlPisac) {
		this.konzolniPisac = konzolniPisac;
		this.fajlPisac = fajlPisac;
	}
	public void setKonzolniPisac(LogPisac konzolniPisac) {
		this.konzolniPisac = konzolniPisac;
	}
	public void setFajlPisac(LogPisac fajlPisac) {
		this.fajlPisac = fajlPisac;
	}

	public void pisiUFajl (String text){
		fajlPisac.pisiLiniju(text);
	}

	public void pisiNaKonzolu (String text){
		konzolniPisac.pisiLiniju(text);
	}


}
```

U ovom slučaju, u editoru se bira *autowire constructor*, kao na slici 6.4.



Slika 6.4. Podešavanje automatskog ožičavanja po konstruktoru u editoru

Sadržaj beans.xml fajla je skoro isti kao i u slučaju automatskog ožičavanja po tipu ili po imenu, uz razliku da je vrednost atributa *autowire* sada *constructor*.

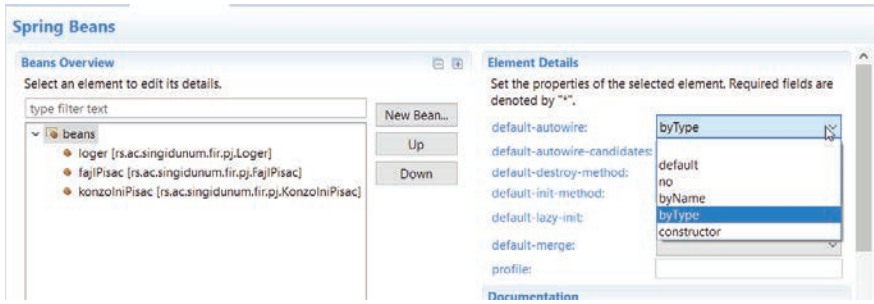
```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://www.springframework.org/schema/
beans    http://www.springframework.org/schema/beans/spring-beans.
xsd">

       <bean id="loger" class="rs.ac.singidunum.fir.pj.Loger"
             autowire="constructor">
       </bean>
       <bean      id="fajlPisac"      class="rs.ac.singidunum.fir.
pj.FajlPisac"></bean>
       <bean id="konzolniPisac"
             class="rs.ac.singidunum.fir.pj.KonzolniPisac">
       </bean>
</beans>
```

Automatsko ožičavanje po konstruktoru je slično ožičavanju po tipu, sa tim što se primenjuje na argumente konstruktora. Spring kontejner prvo traži zrna čiji je atribut *autowire* postavljen na vrednost *constructor* u XML konfiguracionom fajlu. Nakon toga, pokušava da upari i ožiči argumente konstruktora sa tačno jednim zrnom koje se nalazi u kontejneru. Prvo se pokušava uparivanje po tipu, pa ako ono ne uspe, pokušava se uparivanje po imenu. Naravno, i u ovom slučaju je moglo da se direktno piše u XML fajlu umesto korišćenja editora.

6.4. Podrazumevano automatsko ožičavanje

U praksi se vrlo često dešava da je potrebno podesiti isti atribut za automatsko ožičavanje u svim zrnima u kontekstu aplikacije. U slučaju da postoji veliki broj zrna, ručno podešavanje nije praktično. Stvari se mogu pojednostaviti tako što se može tražiti od Springa da primeni isti stil automatskog ožičavanja za sva zrna koja kreira. Sve što treba uraditi u tom slučaju, jeste da se podesi *default-autowire* atribut na korenom <beans> elementu. Ukoliko se koristi editor, podešavanje se radi na korenom bean elementu na način prikazan na slici 6.5.



Slika 6.5. Postavljanje podrazumevanog ožičavanja u editoru

XML fajl u tom slučaju izgleda ovako:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://www.springframework.org/schema/
beans http://www.springframework.org/schema/beans/spring-beans.xsd"
       default-autowire="byType">

<!-- zrna idu ovde -->

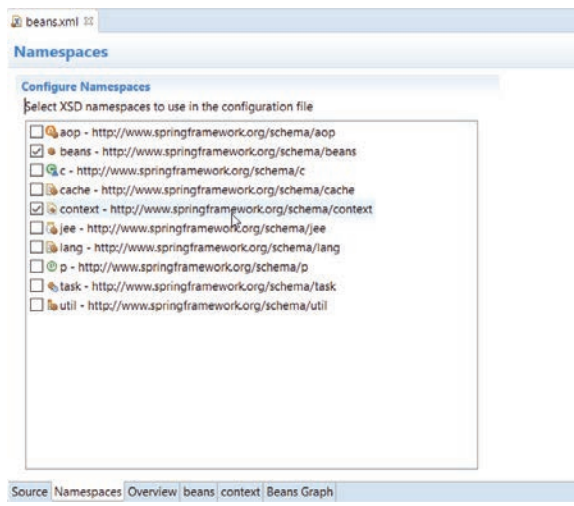
</beans>
```

Podrazumevano podešavanje *default-autowire* atributa je postavljeno na vrednost *none*, odnosno nijedno zrno neće biti automatski ožičeno ukoliko ono samo nije konfigurisano za automatsko ožičavanje putem *autowire* atributa. Moguće je nadjačati podrazumevano podešavanje na nivou zrna upotrebom *autowire* atributa.

6.5. Automatsko ožičavanje pomoću anotacija

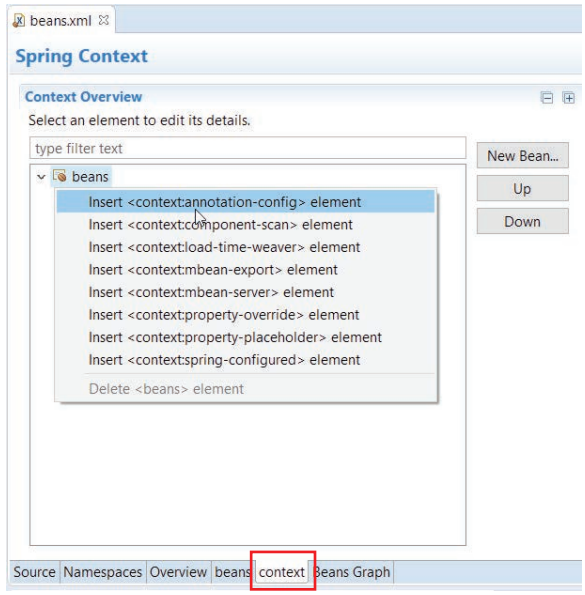
Spring od svoje verzije 2.5 pruža najinteresantniji način za automatsko ožičavanje zrna – upotrebom anotacija. Programski jezik Java je uveo podršku za anotacije od verzije 5.0. Anotacije su oblik metapodataka, odnosno pružaju dodatne informacije o programu, iako same po sebi nisu deo programa. Predstavljaju dodatne informacije za kompajler ili dodatno procesiranje za vreme izvršavanja. Pre anotacija, ponašanje Spring okruženja se kontrolisalo kroz XML konfiguraciju. Danas, uz pomoć anotacija se može podešavati veliki broj opcija Spring okruženja.

Automatsko ožičavanje pomoću anotacija nije mnogo drugačije od konfiguracije *autowire* atributa u XML fajlu, ali omogućava preciznije podešavanje. Anotacije nisu podrazumevane u Spring konfiguraciji, pa ih je neophodno uključiti pomoću <context:annotation-config> elementa. Ukoliko je se koristi editor, potrebno je prvo uključiti *context* imenski prostor, u tabu *Namespaces*, kao na slici 6.7.



Slika 6.6. Uključivanje context imenskog prostora pomoću editora

Nakon uključivanja ovog imenskog prostora, novi tab će se pojaviti na dnu editora, sa labelom *context*, kao na slici 6.7. Na ovaj način se mogu dodati i drugi imenski prostori sa slike 6.6, i svaki od njih će dodati odgovarajući tab na dnu editora. Ukoliko se odabere tab *context*, dobijaju se nove opcije koje se mogu uključiti u konfiguraciju. Pošto je potrebno dodati podršku za anotacije, u okviru *context* taba se desnim klikom na beans element otvara meni sa dostupnim opcijama, od kojih je potrebno odabrati Insert <context:annotation-config> element, kao na slici 6.7. Upotrebu anotacija ćemo demonstrirati na istom primeru Logera kao u prethodnim poglavljima.



Slika 6.7. Dodavanje podrške za anotacije pomoću editora

Nakon dodavanja ovog elementa, može se primetiti izmena u beans.xml izvornom kodu.

```
<?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:context="http://www.springframework.org/schema/context"
       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-4.3.xsd">

       <bean id="loger" class="rs.ac.singidunum.fir.pj.Loger">
       </bean>
       <bean id="fajlPisac" class="rs.ac.singidunum.fir.
pj.FajlPisac"></bean>
       <bean id="konzolniPisac"
             class="rs.ac.singidunum.fir.pj.KonzolniPisac">
       </bean>
       <context:annotation-config></context:annotation-config>
</beans>
```

Nakon dodavanja ovog elementa u beans.xml fajl, moguće je koristiti anotacije za automatsko ožičavanje unutar Java klasa, umesto konfigurisanja unutar beans.xml. Moguće je koristiti @Autowired anotaciju nad seter metodama (odnosno bilo kojom metodom koja postavlja vrednost), nad konstruktorom, čak i direktno nad poljima klase. @Autowired anotacija radi po principu automatskog ožičavanja po tipu, pa je za njen ispravan rad potrebno da ne postoji nikakva dvosmislenost, odnosno mora postojati jedan i samo jedan kandidat za svako polje koje se automatski ožičava. U nastavku ćemo pogledati sve načine na koje možemo anotirati automatsko ožičavanje klase loger.

Prvi način na koji se može anotirati klasa je preko set metoda. Nad svakom set metodom se može postaviti anotacija @Autowired, što je prikazano sledećom modifikacijom klase Loger. Potrebno je obratiti pažnju i na import paketa anotacija iz Spring paketa.

```
package rs.ac.singidunum.fir.pj;

import org.springframework.beans.factory.annotation.Autowired;

//dummy implementacija logera
//loger moze da upisuje i u fajl i na konzolu
public class Loger {

	private KonzolniPisac konzolniPisac;
	private FajlPisac fajlPisac;

	@Autowired
	public void setKonzolniPisac(KonzolniPisac konzolniPisac) {
		this.konzolniPisac = konzolniPisac;
	}
	@Autowired
	public void setFajlPisac(FajlPisac fajlPisac) {
		this.fajlPisac = fajlPisac;
	}

	public void pisiUFajl (String text){
		fajlPisac.pisiLiniju(text);
	}

	public void pisiNaKonzolu (String text){
		konzolniPisac.pisiLiniju(text);
	}


}
```

Drugi oblik anotiranja jeste anotacija konstruktora, koja je prikazana u sledećoj varijaciji implementacije klase Loger.

```
package rs.ac.singidunum.fir.pj;

import org.springframework.beans.factory.annotation.Autowired;

//dummy implementacija logera
//loger moze da upisuje i u fajl i na konzolu
public class Loger {

	private KonzolniPisac konzolniPisac;
	private FajlPisac fajlPisac;

	@Autowired
	public Loger(KonzolniPisac konzolniPisac, FajlPisac fajlPi-
sac) {
		this.konzolniPisac = konzolniPisac;
		this.fajlPisac = fajlPisac;
	}

	public void setKonzolniPisac(KonzolniPisac konzolniPisac) {
		this.konzolniPisac = konzolniPisac;
	}

	public void setFajlPisac(FajlPisac fajlPisac) {
		this.fajlPisac = fajlPisac;
	}

	public void pisiUFajl (String text){
		fajlPisac.pisiLiniju(text);
	}

	public void pisiNaKonzolu (String text){
		konzolniPisac.pisiLiniju(text);
	}


}
```

Treći način je direktna anotacija polja, kao što je prikazano u sledećoj varijaciji implementacije klase Loger. U tom slučaju čak nisu potrebne ni set metode.

```
package rs.ac.singidunum.fir.pj;

import org.springframework.beans.factory.annotation.Autowired;

//dummy implementacija logera
//loger moze da upisuje i u fajl i na konzolu
public class Loger {

	@Autowired
	private KonzolniPisac konzolniPisac;
	@Autowired
	private FajlPisac fajlPisac;

	public void pisiUFajl (String text){
		fajlPisac.pisiLiniju(text);
	}

	public void pisiNaKonzolu (String text){
		konzolniPisac.pisiLiniju(text);
	}


}
```

Jedino ograničenje kod @Autowired anotacije je da mora postojati tačno jedno zrno koje se može ožičiti. Ukoliko nema odgovarajućih zrna, ili ukoliko postoji više od jednog zrna koja bi mogla biti injektovana u jedno polje, @Autowired može naleteti na probleme. Na primer, ukoliko nema zrna koje bi moglo da se poveže, ožičavanje puca sa izuzetkom NoSuchBeanDefinitionException. To je zapravo sigurniji način – bolje da Spring aplikacija odmah pukne sa izuzetkom prilikom lošeg ožičavanja, nego da se kasnije naleti na NullPointerException. Međutim, ponekad je ožičavanje nekog polja potpuno opciono, odnosno da je null vrednost nekog polja potpuno prihvatljiva. Kako bi se izbeglo izbacivanje izuzetka prilikom ožičavanja u tom slučaju, može se dodatno naglasiti da je ožičavanje nekog polja opciono sa:

```
@Autowired(required=false)
private KonzolniPisac konzolniPisac;
```

U ovakvom obliku bilo bi dozvoljeno da polje konzolniPisac ostane null, u slučaju da se ne pronađe odgovarajuće zrno za ožičavanje, i ne bi došlo do izbacivanja izuzetka. Atribut required se može koristiti na bilo kom mestu gde se može staviti i anotacija @Autowired.

6.6. Kvalifikatori

Ukoliko postoji više od jednog zrna koja bi mogla da se ožiče na isto polje, Spring neće pogađati koje od zrna treba da ožiči već će kao u prethodnom slučaju izbaciti izuzetak tipa NoSuchBeanDefinitionException. Kako bi se sugerisalo Springu koje zrno se želi (od više njih koji bi mogli da se ožiče na isto polje), može se dodatno koristiti @Qualifier anotacija.

Na primer, u klasi KonzolniPisac se može dodati kvalifikator u sledećem obliku.

```
package rs.ac.singidunum.fir.pj;

import org.springframework.beans.factory.annotation.Qualifier;

@Qualifier(“toconsole”)
public class KonzolniPisac implements LogPisac {

	public void pisiLiniju (String linija) {
		System.out.println(“Ispis na konzolu: “ + linija);
	}

}
```

Kvalifikator anotacijom se obeležava cela klasa, tako što se anotacija stavlja iznad definicije klase. Na isti način se u klasi FajlPisac može dodati sledeći kvalifikator:

```
package rs.ac.singidunum.fir.pj;

import org.springframework.beans.factory.annotation.Qualifier;

@Qualifier(“tofile”)
public class FajlPisac implements LogPisac {

	public void pisiLiniju(String linija) {
		//dummy implementacija
		//ovde bi trebalo da ide upis u fajl
		System.out.println(“Upis u fajl: “ + linija);
	}

}
```

Primer upotrebe se može videti u modifikovanom obliku klase Loger.

```
package rs.ac.singidunum.fir.pj;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.beans.factory.annotation.Qualifier;

//dummy implementacija logera
//loger moze da upisuje i u fajl i na konzolu
public class Loger {

	@Autowired
	@Qualifier("toconsole")
	private LogPisac konzolniPisac;

	@Autowired
	@Qualifier("tofile")
	private LogPisac fajlPisac;

	public void pisiUFajl (String text){
		fajlPisac.pisiLiniju(text);
	}

	public void pisiNaKonzolu (String text){
		konzolniPisac.pisiLiniju(text);
	}


}
```

U ovakvom obliku klase Loger postojala bi dvosmislenost. Oba polja klase Loger su tipa LogPisac, odnosno u svako od njih se može smestiti i FajlPisac i KonsolniPisac. Sa kvalifikatorima se Springu sugeriše da u prvo polje smesti onaj LogPisac koji ima kvalifikator "toconsole", a da u drugo polje smesti onaj LogPisac koji ima kvalifikator "tofile". Čak i da postoji još nekoliko zrna koja su tipa neke implementacije interfejsa LogPisac, ne bi došlo do problema u ožičavanju pošto kvalifikatorima sugerišemo koja zrna treba da se ožiče.

6.7. Inject anotacija

Potrebno je samo još dodatno napomenuti da je uvođenjem podrške za injekciju zavisnosti u programski jezik Java, uveden i zajednički DI model za sve Java okvire. Specifikacija ovog modela je poznata u Java zajednici pod imenom JSR-330, ili prosto

rečeno Inject. Pošto je i Spring zapravo Java okvir, od verzije Spring 3, pružena je podrška za Inject model. Sa ovim modelom uvedena je i @Inject anotacija, koja u potpunosti može da zameni @Autowired anotaciju koja je specifična isključivo za Spring. Kao i @Autowired, i anotacija @Inject se može koristiti za automatsko ožičavanje polja, metoda i konstruktora. Razlika je u tome da @Inject ne sadrži atribut required, odnosno, ukoliko se koristi @Inject anotacija, očekuje se da kompletno ožičavanje bude uspešno (nije dozvoljeno imati null vrednost), inače se izbacuje izuzetak.

Ukoliko se koristi @Inject anotacija, potrebno je dodati još jednu zavisnost u pom.xml:

```
<dependency>
	<groupId>javax.inject</groupId>
	<artifactId>javax.inject</artifactId>
	<version>1</version>
</dependency>
```

U tom slučaju, klasa Loger bi mogla da izgleda ovako:

```
package rs.ac.singidunum.fir.pj;

import javax.inject.Inject;

//dummy implementacija logera
//loger moze da upisuje i u fajl i na konzolu
public class Loger {

	@Inject
	private LogPisac konzolniPisac;

	@Inject
	private LogPisac fajlPisac;

	public void pisiUFajl (String text){
		fajlPisac.pisiLiniju(text);
	}

	public void pisiNaKonzolu (String text){
		konzolniPisac.pisiLiniju(text);
	}

}
```

Prilikom izvršavanja programa, na izlazu se može i videti da je učitan JSR-330:

```
Feb 01, 2019 3:39:30 AM org.springframework.context.support.Ab-
stractApplicationContext prepareRefresh
INFO: Refreshing org.springframework.context.support.FileSystemX
mlApplicationContext@6659c656: startup date [Fri Feb 01 03:39:30
CET 2019]; root of context hierarchy
Feb 01, 2019 3:39:30 AM org.springframework.beans.factory.xml.
XmlBeanDefinitionReader loadBeanDefinitions
INFO: Loading XML bean definitions from file [C:\Spring\Uvod\beans.
xml]
Feb 01, 2019 3:39:30 AM org.springframework.beans.factory.annota-
tion.AutowiredAnnotationBeanPostProcessor <init>
INFO: JSR-330 'javax.inject.Inject' annotation found and support-
ed for autowiring
Upis u fajl: Zdravo
Ispis na konzolu: Zdravo
Feb 01, 2019 3:39:30 AM org.springframework.context.support.Ab-
stractApplicationContext doClose
INFO: Closing org.springframework.context.support.FileSystemXmlAp
plicationContext@6659c656: startup date [Fri Feb 01 03:39:30 CET
2019]; root of context hierarchy
```

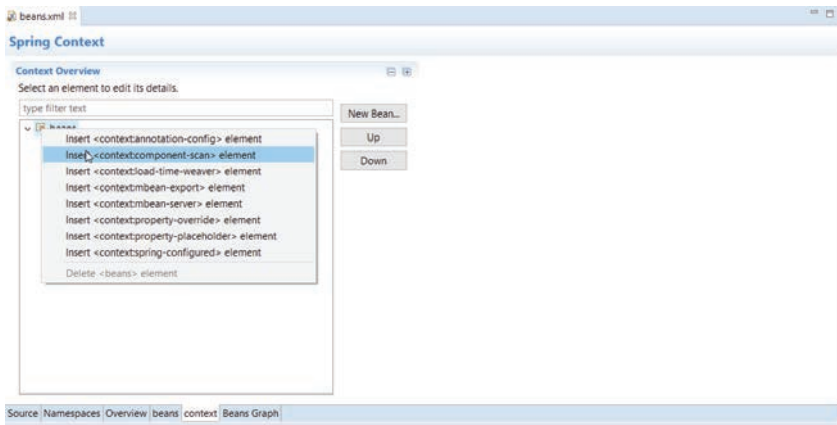
Na kraju se može još dodatno napomenuti da @Inject ima dodatnu anotaciju @Named, koja funkcioniše vrlo slično kao i Springova @Qualifier anotacija, sa tom razlikom što @Qualifier sužava selekciju, a @Named bira zrno sa tačno određenim Id.

6.8. Automatsko otkrivanje zrna

Dodavanjem <context:annotation-config> elementa u konfiguraciju, Springu je rečeno da će se u kodu pisati određeni skup anotacija u definicijama zrna. Te anotacije će voditi sam proces ožičavanja zrna. Na taj način se eliminiše većina upotreba elemenata <property> i <constructor-arg> u XML fajlu, ali je i dalje neophodno da se zrna eksplicitno deklarišu upotrebom <bean> elementa. Šta ako bi postojao način da se eliminišu i <bean> elementi iz konfiguracije?

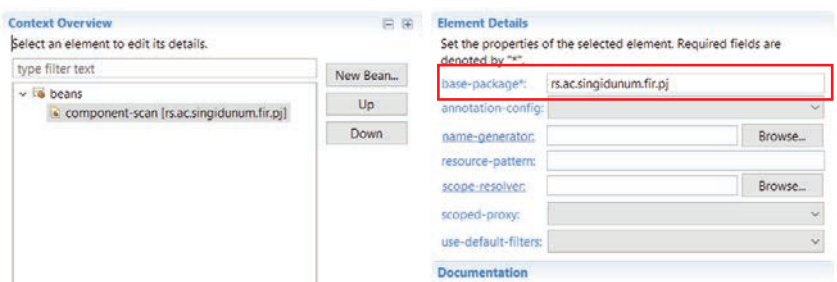
Element <context:component-scan> radi sve što radi <context:annotation-config> element, uz jednu dodatnu funkcionalnost. On konfiguriše Spring da automatski otkrije zrna i deklariše ih umesto nas. To kao posledicu ima da se sva (ili skoro sva) zrna u Spring aplikaciji mogu deklarisati i ožičiti bez upotrebe elementa <bean> u XML fajlu. Dakle, da bi se Spring konfigurisao za automatsko otkrivanje zrna, umesto elementa <context:annotation-config> potrebno je koristiti element <context:component-scan>.

Ukoliko je potrebno dodati <context:component-scan> element, to se uz pomoć editora može uraditi odabirom taba *context*, i dodavanjem odgovarajućeg elementa kao na slici 6.8.



Slika 6.8. Dodavanje podrške za automatsko skeniranje zrna pomoću editora

Automatsko otkrivanje zrna radi tako što se skenira zadati paket, kao i svi njegovi potpaketi, pri čemu se traže klase koje je potrebno automatski registrovati kao zrna u Spring kontejneru. Atribut *base-package* služi za zadavanje paketa od kojeg počinje pretraga. Nakon dodavanja elementa u editoru, u njegovim podešavanjima je potrebno postaviti željeni paket u opisani atribut *base-package*, kao na slici 6.9.



Slika 6.9. Postavljanje početnog paketa za automatsko skeniranje

Nakon uvedenih izmena, sadržaj beans.xml fajla je dat u nastavku. Potrebno je obratiti pažnju da su sve definicije zrna uklonjene iz XML fajla, i da u njemu stoji samo element koji definiše skeniranje komponenti.

```
<?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:context="http://www.springframework.org/schema/context"
       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-4.3.xsd">

       <context:component-scan
              base-package="rs.ac.singidunum.fir.pj">
       </context:component-scan>
</beans>
```

Nakon konfigurisanja automatskog otkrivanja zrna, zadavanja početnog paketa od kojeg kreće pretraga i uklanjanja definicija zrna iz XML fajla, postavlja se logično pitanje – kako <context:component-scan> uopšte zna koje klase da registruje kao zrna? Automatsko otkrivanje zrna radi tako što se traže klase koje su anotirane sa jednom od nekoliko stereotipnih anotacija:

- @Component – anotacija opšte namene koja nagoveštava da je klasa koja je označena na ovaj način Spring komponenta.
- @Controller – anotacija koja specificira da klasa definiše Spring MVC kontroler.
- @Repository – anotacija koja označava da je klasa repozitorijum podataka.
- @Service – anotacija koja označava da klasa definiše servis.

To znači da je sada potrebno markirati sve klase koje treba da budu automatski otkrivene u našem programu anotacijom • @Component.

```
package rs.ac.singidunum.fir.pj;

import org.springframework.stereotype.Component;

@Component
public class FajlPisac implements LogPisac {

       public void pisiLiniju(String linija) {
              //dummy implementacija
              //ovde bi trebalo da ide upis u fajl
              System.out.println("Upis u fajl: " + linija);
       }

}
```

```
package rs.ac.singidunum.fir.pj;

import org.springframework.stereotype.Component;

@Component
public class KonzolniPisac implements LogPisac {

	public void pisiLiniju (String linija) {
		System.out.println("Ispis na konzolu: " + linija);
	}
}
package rs.ac.singidunum.fir.pj;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Component;

//dummy implementacija logera
//loger moze da upisuje i u fajl i na konzolu
@Component
public class Loger {

	@Autowired
	private LogPisac konzolniPisac;

	@Autowired
	private LogPisac fajlPisac;

	public void pisiUFajl (String text){
		fajlPisac.pisiLiniju(text);
	}

	public void pisiNaKonzolu (String text){
		konzolniPisac.pisiLiniju(text);
	}

}
```

Na ovaj način je količina koda u XML konfiguracionom fajlu značajno redukovana. U našem slučaju, iz XML fajla uklonjene su sve definicije zrna i sva pravila za automatsko ožičavanje, a ostao je samo jedan element koji definiše automatsko otkrivanje zrna i paket u kome treba skenirati zrna. Sve ostalo se rešava direktno u kodu Java klasa upotrebom anotacija. Ovo je i tipičan primer konfiguracije Spring projekta.

Još je na kraju potrebno napomenuti da se svim zrnima u kontekstu aplikacije u Spring okruženju dodeljuje Id. To važi i za zrna koja su automatski otkrivena jer su njihove klase označene anotacijom @Component. To možda nije bilo očigledno u prethodnom primeru, pošto se nigde eksplicitno ne dodeljuje. Spring zrnu dodeljuje Id koji se dobija iz imena klase. Preciznije, zrno dobija Id tako što se napiše ime klase sa malim početnim slovom. Dakle, u prethodnom primeru, zrno klase Loger ima Id *loger*, zrno klase KonzolniPisac ima Id *konzolniPisac*, itd. Ukoliko je potrebno dodeliti neko drugo ime zrnu, na primer ukoliko želimo da zrnu klase Loger dodelimo Id *mojSjajniLoger*, klasa Loger se anotira na sledeći način: @Component("mojSjajniLoger").

6.9. Zadaci za vežbu

Zadatak 1: Napisati Spring projekat upotrebom Maven-a koji sadrži klasu Auto koja sadrži polja marka i tip, klasu Osoba koja sadrži polja ime, prezime i jmbg i klasu SaobracajnaDozvola koji ima polja auto tipa Auto i polje vlasnik tipa Osoba. Klase sadrže konstruktore, set i toString() metode. Kreirati po jedan objekat svake klase (zrna i koristiti injekciju kroz konstruktor kako bi prosledili vrednosti polja klasa). Izvršti automatsko ožičavanje:

- Po imenu
- Po tipu
- Po konstruktoru

U glavnoj klasi odštampati podatke jedne saobraćajne dozvole.

Rešenje: Prvo će biti kreirane sve potrebne klase sa odgovarajućim metodama. Klase se kao i do sada standardno implementiraju u Java programskom jeziku kao POJO. Potrebno je naglasiti da u slučaju automatskom ožičavanja po imenu i po tipu klasa SaobracajnaDozvola ne sme da ima konstruktor inače će biti izbačen org.springframework.beans.BeanInstantiationException.

Implementacija klasa:

```
package rs.ac.singidunum.fir.pj;

public class Osoba {
	private String ime, prezime;
	private long jmbg;
	public Osoba(String ime, String prezime, long jmbg) {
		this.ime = ime;
		this.prezime = prezime;
		this.jmbg = jmbg;
	}
	public void setIme(String ime) {
		this.ime = ime;
	}
```

```
	public void setPrezime(String prezime) {
		this.prezime = prezime;
	}
	public void setJmbg(long jmbg) {
		this.jmbg = jmbg;
	}
	public String toString() {
		return ime + " " + prezime + ", jmbg: " +jmbg;
	}

}
```

Klasa Auto:

```
package rs.ac.singidunum.fir.pj;

public class Auto {
	private String marka, tip;

	public Auto(String marka, String tip) {
		this.marka = marka;
		this.tip = tip;
	}

	public void setMarka(String marka) {
		this.marka = marka;
	}

	public void setTip(String tip) {
		this.tip = tip;
	}

	public String toString() {
		return "Auto [marka=" + marka + ", tip=" + tip + "]";
	}

}
```

Klasa SaobracajnaDozvola:

```
package rs.ac.singidunum.fir.pj;

public class SaobracajnaDozvola {
	private Osoba vlasnik;
	private Auto auto;
```

```
	public void setVlasnik(Osoba vlasnik) {
		this.vlasnik = vlasnik;
	}
	public void setAuto(Auto auto) {
		this.auto = auto;
	}

	@Override
	public String toString() {
		return "SaobracajnaDozvola: Vlasnik=" + vlasnik + ",
auto: " + auto;
	}
}
```

Pomoću editora su napravljena zrna za objekte klasa Auto, Osoba i SaobracajnaDozvola. Polja Auta i Osobe su postavljene kao u prethodnom poglavlju, preko argumenata konstruktora, dok su polja klase SaobracajnaDozvola automatski ožičene po imenu.

beans.xml fajl

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
	xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
	xsi:schemaLocation="http://www.springframework.org/schema/
beans http://www.springframework.org/schema/beans/spring-beans.xsd">


	<bean id="auto" class="rs.ac.singidunum.fir.pj.Auto">
		<constructor-arg type="String" value="Zastava"
name="marka"></constructor-arg>
		<constructor-arg type="String" value="128"
name="tip"></constructor-arg>
	</bean>
	<bean id="vlasnik" class="rs.ac.singidunum.fir.pj.Osoba">
		<constructor-arg name="ime" type="String"
value="Pera"></constructor-arg>
		<constructor-arg name="prezime" type="String"
value="Peric"></constructor-arg>
		<constructor-arg name="jmbg" type="Long" val-
ue="2121214142"></constructor-arg>
	</bean>
	<bean id="saobracajnaDozvola"
		class="rs.ac.singidunum.fir.pj.SaobracajnaDozvola"
autowire="byName">
	</bean>
</beans>
```

Glavna klasa:

```
package rs.ac.singidunum.fir.pj;
import org.springframework.context.ApplicationContext;
import   org.springframework.context.support.FileSystemXmlApplica-
tionContext;

public class Glavna {
      public static void main(String[] args) {
            ApplicationContext context = new FileSystemXmlApplic
ationContext(“beans.xml”);
            SaobracajnaDozvola  dozvola  =  (SaobracajnaDozvola)
context.getBean(“saobracajnaDozvola”);
            System.out.println(dozvola.toString());
            ((FileSystemXmlApplicationContext) context).close();
      }
}
```

Izlaz:

```
SaobracajnaDozvola: Vlasnik=Pera Peric, jmbg: 2121214142, auto:
Auto [marka=Zastava, tip=128]
```

Za automatsko ožičavanje po tipu potrebno je samo promeniti beans.xml fajl. Pri konfigurisanju zrna saobracajnaDozvola potrebno je za vrednost atributa autowire staviti byType umesto byName.

```
<bean id=”saobracajnaDozvola”
      class=”rs.ac.singidunum.fir.pj.SaobracajnaDozvola”
autowire=”byType”>
</bean>
```

Izlaz sve ostale klase ostaju identične.

Za automatsko ožičavanje po konstruktoru u klasi SaobracajnaDozvola ćemo dodati konstruktor tako da naša klasa sada izgleda ovako:

```
package rs.ac.singidunum.fir.pj;

public class SaobracajnaDozvola {
	private Osoba vlasnik;
	private Auto auto;
	public SaobracajnaDozvola(Osoba vlasnik, Auto auto) {
		this.vlasnik = vlasnik;
		this.auto = auto;
	}
	public void setVlasnik(Osoba vlasnik) {
		this.vlasnik = vlasnik;
	}
	public void setAuto(Auto auto) {
		this.auto = auto;
	}

	@Override
	public String toString() {
		return "SaobracajnaDozvola: Vlasnik=" + vlasnik + ",
auto: " + auto;
	}

}
```

Pored ove izmene kao i u prethodnom slučaju potrebno je promeniti vrednost atributa autowire pri konfigurisanju zrna saobracajnaDozvola tako da ima vrednost *constructor*.

```
<bean id="saobracajnaDozvola"
	class="rs.ac.singidunum.fir.pj.SaobracajnaDozvola"
	autowire="constructor">
</bean>
```

Zadatak 2: Izmeniti prehodni zadatak tako da se automatsko ožičavanje vrši pomoću anotacije.

Rešenje: Prvo je potrebno ukloniti automatsku anotaciju za zrno saobracajnaDozvola. Zatim je potrebno uključiti <context:annotation-config> element na način opisan u ovom poglavlju. Treći korak je dodavanje podrške za anotacije što je takođe opisano ranije u ovom poglavlju. Kada je ovo sve urađeno, možemo izmeniti klasu SaobracajnaDozvola dodavanjem anotacija. Anotacije mogu biti dodate ili za set metode ili za konstruktor ili direknom anotacijom polja, ali samo jedan od načina se

može koristiti (ne mogu sva polja i metode da imaju @Autowired anotaciju). Klasa SaobracajnaDozvola sa anotacijom set metoda i zakomentarisanim mogućim anotacija je sledeća:

```
package rs.ac.singidunum.fir.pj;

import org.springframework.beans.factory.annotation.Autowired;

public class SaobracajnaDozvola {
//@Autowired
	private Osoba vlasnik;
	//@Autowired
private Auto auto;
	/*@Autowired
	public SaobracajnaDozvola(Osoba vlasnik, Auto auto) {
		this.vlasnik = vlasnik;
		this.auto = auto;
	}*/
	@Autowired
	public void setVlasnik(Osoba vlasnik) {
		this.vlasnik = vlasnik;
	}
	@Autowired
	public void setAuto(Auto auto) {
		this.auto = auto;
	}

	@Override
	public String toString() {
		return "SaobracajnaDozvola: Vlasnik=" + vlasnik + ",
auto: " + auto;
	}
}
```

Ostale klase ostaju iste. Pokretanjem glavne klase dobija se identičan ispis kao u prethodnom zadatku.

Zadatak 3: Napisati Spring projekat upotrebom Maven-a koji sadrži funkcionalni interfejs Posao koji ima metod uradiPosao(). Klase Menadzer i Radnik implementiraju interefejs Posao tako što menadzer ispisuje poruku „Plan rada je napravljen" dok radnik ispisuje poruku „Dodeljen posao je izvrsen". Klasa Firma ima dva polje tipa Posao, menadzer i radnik, i metode naparaviPlan() i sprovediPlan(). Prva metoda

poziva metodu uradiPosao polja menadzer, a druga metoda poziv uradiPosao polja radnik. Kreirati po jedan objekat svake klase (zrna) i izvršiti automatsko ožičavanje.

Rešenje: U ovom zadatku, pošto u klasi Firma postoji dva polja istog tipa ne može se koristiti automatsko ožičavanje po tipu, ukoliko se implementira konstruktor, može se koristiti ožičavanje po konstruktoru pošto su imena različita. Takođe se može koristiti automatsko ožičavanje po imenu ili se može koristi anotacija @Qualifier. Ovde će biti prikazano rešenje ožičavanja pomoću anotacija, a čitaocu ostaje da implementira i ostale načine.

Kao i u prethodnom zadatku, potrebno je uključiti <context:annotation-config> element i dodati podršku za anotacije. Interfejs ima standardnu implementaciju:

```
package rs.ac.singidunum.fir.pj;

public interface Posao {
	void uradiPosao();
}
```

Kako bi se naglasilo koje zrno bi trebalo biti dodeljeno kojim poljima klase Firma, u klasama Menadzer i Radnik ćemo uvesti @Qualifier notacije: menadzer i radnik:

```
package rs.ac.singidunum.fir.pj;

import org.springframework.beans.factory.annotation.Qualifier;

@Qualifier("menadzer")
public class Menadzer implements Posao {
	public void uradiPosao() {
		System.out.println("Plan rada je napravljen");
	}
}
```

```
package rs.ac.singidunum.fir.pj;
import org.springframework.beans.factory.annotation.Qualifier;

@Qualifier("radnik")
public class Radnik implements Posao{
	public void uradiPosao() {
		System.out.println("Dodeljen posao je izvrsen");
	}
}
```

U klasi Firma je potrebno obezbediti pravilno ožičavanje korišćenjem anotacija na sledeći način:

```
package rs.ac.singidunum.fir.pj;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.beans.factory.annotation.Qualifier;

public class Firma {
	@Autowired
	@Qualifier(“menadzer”)
	private Posao menadzer;
	@Autowired
	@Qualifier(“radnik”)
	private Posao radnik;

	public void napraviPlan() {
		menadzer.uradiPosao();
	}
	public void sprovediPlan() {
		radnik.uradiPosao();
	}
}
```

Fajl beans.xml:

```
<?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:context=”http://www.springframework.org/schema/context”
	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-
4.3.xsd”>


	<context:annotation-config></context:annotation-config>
	<bean name=”radnik” class=”rs.ac.singidunum.fir.
pj.Radnik”></bean>
	<bean name=”menadzer” class=”rs.ac.singidunum.fir.
pj.Menadzer”></bean>
	<bean name=”firma” class=”rs.ac.singidunum.fir.pj.Firma”></
bean>
</beans>
```

Glavna klasa koja testira rad ovog programa zajedno sa ispravnim izlazom je dat u nastavku:

```
package rs.ac.singidunum.fir.pj;

import org.springframework.context.ApplicationContext;
import   org.springframework.context.support.FileSystemXmlApplica-
tionContext;

public class Glavna {
       public static void main(String[] args) {
              ApplicationContext context = new FileSystemXmlApplic
ationContext("beans.xml");
              Firma firma = (Firma) context.getBean("firma");
              firma.napraviPlan();
              firma.sprovediPlan();
              ((FileSystemXmlApplicationContext) context).close();
       }
}
```

Izlaz:

```
Plan rada je napravljen
Dodeljen posao je izvrsen
```

Zadatak 4: Napisati Spring projekat upotrebom Maven-a koji sadrži klase Predmet, Fakultet i Profesor. Polja klase Predmet i Fakultet proizvoljno definisati, a klasa Profesor sadrži polje fakultet na kom profesor radi i polje predmet koje označava predmet koji profesor predaje. Obezbediti automatsko prepoznavanje zrna. Napisati aplikaciju koja testira rad prepoznavanja zrna i ožičavanja.

Rešenje: Pošto se koristi automatsko prepoznavanje zrna, potrebno je dodati je potrebno dodati <context:component-scan> element (opisano u ovom poglavlju). Nakon toga je potrebno postaviti bazni paket pretrage (atribut base-package). U ovom rešenju sve klase su smeštene u paket *rs.ac.singidunum.fir.pj*, pa je on postavljen kao atribut base-package. Ovo se sve može isvršiti u editoru, a generisan beans.xml fajl izgleda ovako:

```
<?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:context="http://www.springframework.org/schema/context"
       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-4.3.xsd">
```

```
        <context:component-scan base-package="rs.ac.singidunum.fir.pj">
        </context:component-scan>
</beans>
```

Klase Predmet i Fakultet će sadržati polja kojima će u konstruktoru biti dodeljene vrednosti ekslicitno kako bi se objekti klase mogli koristiti posle u klasi Profesor.

```
package rs.ac.singidunum.fir.pj;

import org.springframework.stereotype.Component;

@Component
public class Predmet {
        private String naziv;
        private String sifra;
        private int espb;
        public Predmet() {
                this.naziv = "Programski jezici";
                this.sifra = "pj123";
                this.espb = 8;
        }
@Override
public String toString() {
                return "Predmet [naziv=" + naziv + ", sifra=" + sifra
+ ", espb=" + espb + "]";
        }
}
```

```
package rs.ac.singidunum.fir.pj;

import org.springframework.stereotype.Component;

@Component
public class Fakultet {
        private String naziv;
        private String adresa;
        public Fakultet() {
                this.naziv = "FIR";
                this.adresa = "Danijelova 32";
        }
```

```
	@Override
	public String toString() {
		return "Fakultet [naziv=" + naziv + ", adresa=" + ad-
resa + "]";
	}
}
```

U klasi profesor ćemo koristiti automatsko ožičavanje konstruktora:

```
package rs.ac.singidunum.fir.pj;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Component;

@Component
public class Profesor {
	private Predmet predmet;
	private Fakultet fakultet;

	@Autowired
	public Profesor(Predmet predmet, Fakultet fakultet) {
		this.predmet = predmet;
		this.fakultet = fakultet;
	}

	@Override
	public String toString() {
		return "Profesor [predmet=" + predmet + ", fakultet="
+ fakultet + "]";
	}
}
```

Treba napomenuti da u beans.xml fajl nije dodat ni jedan red iako je obezbeđeno automatsko prepoznavanje zrna i ožičavanje.

Aplikacija koja testira rad:

```
package rs.ac.singidunum.fir.pj;

import org.springframework.context.ApplicationContext;
import   org.springframework.context.support.FileSystemXmlApplica-
tionContext;
```

```
public class App {
	public static void main(String[] args) {
		ApplicationContext context = new FileSystemXmlApplic
ationContext(“beans.xml”);
		Profesor prof = (Profesor) context.getBean(“profesor”);
		System.out.println(prof);
		((FileSystemXmlApplicationContext) context).close();
	}
}
```

Izlaz

```
Profesor [predmet=Predmet [naziv=Programski jezici, sifra=pj123,
espb=8], fakultet=Fakultet [naziv=FIR, adresa=Danijelova 32]]
```

Pitanja

1. Koje načine automatskog ožičavanja nudi Spring?
2. Šta je automatsko ožičavanje po imenu?
3. Šta je automatsko ožičavanje po tipu?
4. Na koji način se automatsko ožičavanje po konstruktoru oslanja na automatsko ožičavanje po tipu i imenu?
5. Da li će automatsko ožičavanje po konstruktoru izbaciti grešku ukoliko pronađe dvosmislenost pri automatskom ožičavanju po tipu? Zašto?
6. Da li način automatskog ožičavanja mora biti definisan za svako zrno posebno?
7. Na koji način se može postaviti tip automatskog ožičavanja za sva zrna u kontekstu?
8. Ukoliko se definiše podrazumevani način automatskog ožičavanja za sva zrna u kontekstu, da li je moguće promeniti način za samo neko od zrna?
9. Koji imenski prostor se koristi za anotacije u Spring-u?
10. Gde je moguće koristiti @Autowired anotaciju?
11. Da li su obavezne set metode prilikom direktne anotacije polja klase?
12. Kako se označava ukoliko je automatsko ožičavanje nekog polja opciono?
13. Čemu služe kvalifikatori (@Qualifier anotacija)?
14. Koja anotacija može da zameni @Autowired anotaciju?
15. Šta je razlika između @Autowired i @Inject anotacije?
16. Šta je razlika između @Qualifier i @Named anotacije?
17. Koji je način eliminisanja <bean> elementata iz konfiguracije?
18. Koji element se korisi za automatsko otkrivanje zrna?
19. Kako <context:component-scan> uopšte zna koje klase da registruje kao zrna?
20. Koje su stereotipne anotacije u Springu?
21. Koji Id dobija zrno prilikom automatskog otkrivanja?

VII
Aspektno orijentisano programiranje u Springu

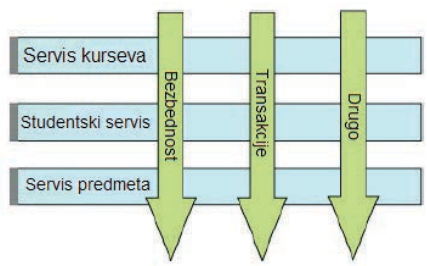
U razvoju softvera, nekoliko aktivnosti se provlači kroz skoro sve aplikacije – logovanje, sigurnost, menadžment transakcija i slično. To jesu bitne aktivnosti, ali da li stvarno objekti aplikacije treba aktivno da učestvuju u njima? Bilo bi bolje da se objekti u aplikaciji fokusiraju na domen svog problema za koji su i dizajnirani, a da nekom drugom ostave upravljanje određenim aktivnostima. Upravo ove aktivnosti, koje se protežu kroz više tačaka u aplikaciji (engl. *cross-cutting concerns*) spadaju u domen aspektno orijentisanog programiranja (AOP).

Ovaj tip aktivnosti je konceptualno potpuno drugačiji od osnovne poslovne logike objekata aplikacije. Međutim, u tradicionalnom programiranju, ove aktivnosti se često direktno ubacuju u kod klasa kojima to nije primarna briga. Postavlja se pitanje da li, na primer, klasa kojoj je glavni zadatak da ažurira neke podatke u bazi, treba da se brine o logovanju ili treba da se fokusira na svoj zadatak, a da brigu o logovanju ostavi nekom drugom. To i jeste glavni cilj AOP – da razdvoji ove aktivnosti koje se provlače kroz više tačaka u aplikaciji od poslovne logike komponenti aplikacije. AOP dakle pomaže u razdvajanju zajedničkih aktivnosti od objekata na koje te aktivnosti treba da deluju. Logovanje je najčešći i najočigledniji primer aktivnosti koja može da se pretvori u aspekt i razdvoji od klasa na koje treba da deluje.

7.1. Koncept AOP i osnovni pojmovi

Aspektno orijentisano programiranje se može posmatrati kao komplement objektno orijentisanog programiranja (OOP), jer pruža drugačiji način posmatranja strukture programa. Osnovna jedinica modularnosti u OOP je klasa, dok je u AOP osnovna jedinica aspekt. Aspekt omogućava modularizaciju aktivnosti koje se protežu kroz više tačaka u aplikaciji, poput logovanja, koje obično obuhvata veći broj objekata programa. Kao što je ranije pomenuto, AOP okvir je jedan od dve glavne komponente Spring okruženja (uz DI). Programer nije u obavezi da koristi AOP u svom projektu ukoliko ne želi, ali se puna snaga Springa dobija upravo zajedničkom primenom AOP i DI. AOP iDI su dve komplementarne tehnike, koje kada se zajedno primene značajno pojednostavljuju kod.

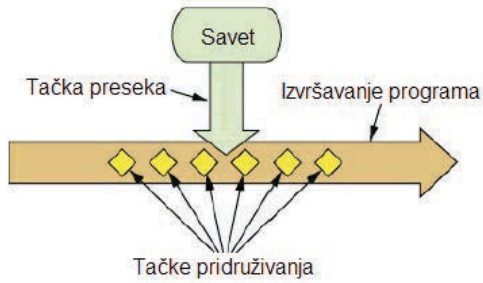
Na slici 7.1. prikazana je vizuelizacija problema aktivnosti koje se protežu kroz više tačaka u aplikaciji. Aplikacija je predstavljena kroz module, poput studentskog servisa, servisa predmeta i kurseva i slično. Svaki modul ima svoju osnovnu funkciju, da pruži odgovarajući servis iz svog domena. Međutim, svaki modul takođe zahteva određenu funkcionalnost koja mu nije primarna briga, poput sigurnosti ili menadžmenta transakcija. Drugim rečima, svim modulima pored koda koji obavlja njihovu primarnu funkcionalnost, treba i kod za ovu zajedničku aktivnost. Kako bi se izbeglo kopiranje istog koda u više klasa, tradicionalni objektno orijentisani pristup u ovakvom slučaju bi bio primena nasleđivanja ili delegacija. Nasleđivanje, međutim, može dovesti do vrlo krhke hijerarhije objekata, jer bi se ista bazna klasa koristila kroz veliki broj modula u aplikaciji. Sa druge strane, delegacija može biti veoma kompleksna i kabasta zbog komplikovanih poziva koji se upućuju delegatu.



Slika 7.1. Aktivnosti koje se protežu preko više modula u aplikaciji

Aspekti nude alternativu nasleđivanju i delegatima koja u najvećem broju slučajeva pruža jednostavnije rešenje i čistiji kod. Upotrebom AOP se i dalje zajednička funkcionalnost definiše na jednom mestu, a zatim se deklarativno definiše kada i gde se ta funkcionalnost primenjuje, bez potrebe menjanja koda klasa na koje se ta funkcionalnost primenjuje. To mesto, gde se zajednička funkcionalnost koja se proteže preko više klasa u sistemu definiše, se naziva aspekt. Aspekti u praksi imaju dve prednosti. Prvo, kompletan kod i logika za neku funkcionalnost, na primer logovanje, može biti modularizovan i smešten u jednu specijalnu klasu – aspekt, umesto da bude raštrkan po svim klasama koje zahtevaju logovanje. Drugo, klase kojima treba logovanje ostaju čiste, jer u tom slučaju sadrže samo kod koji je potreban za njihovu glavnu funkcionalnost, pošto je kod za logovanje izmešten u aspekt.

Kao i veliki broj drugih tehnika programiranja, AOP je razvio svoj sopstveni žargon. Aspekti se najčešće opisuju pomoću pojmova savet (engl. *advice*), tačka preseka (engl. *pointcut*) i tačka pridruživanja (engl. *join-point*). Slika 7.2 prikazuje kako su ovi pojmovi povezani. Na žalost, veliki broj termina koji se koriste u žargonu AOP uopšte nije intuitivan, pa zbog toga ceo koncept deluje znatno složenije nego što jeste. Bilo kako bilo, ti termini se zvanično koriste i moraju se poznavati.



Slika 7.2. Aspekt se primenjuje na jednoj ili više tačaka primene za vreme izvršavanja programa.

Aspekti imaju svoju svrhu – posao odnosno funkciju koju treba da obave. U AOP terminologiji, posao aspekta se zove savet (engl. *advice*). Savet definiše šta aspekt radi i kada to radi. Uz opis posla koji aspekt treba da izvršava, savet definiše i kada taj posao treba da se izvrši – pre poziva metode, posle poziva metode, i pre i posle poziva metode, ili u slučaju da metoda izbaci izuzetak. U Springu, aspekti mogu da rade sa pet vrsta saveta:

- *Before* – savet se izvršava pre poziva metode koja se savetuje.
- *After* – savet se izvršava nakon završetka metode koja se savetuje.
- *After-returning* – savet se primenjuje nakon uspešnog završetka metode koja se savetuje.
- *After-throwing* – savet se izvršava nakon što savetovana metoda izbaci izuzetak.
- *Around* – savet se omotava oko savetovane metode, tako što se deo funkcionalnosti izvršava pre, a deo posle završetka savetovane metode.

U svakoj složenijoj aplikaciji može postojati i više stotina prilika gde se savet može primeniti. Te prilike, odnosno tačke u izvršavanju programa gde se aspekt može uključiti se zovu tačke pridruživanja (engl. *join points*). Ove tačke mogu biti pozivi metoda, izbačeni izuzeci ili promena vrednosti polja neke klase. To su dakle sve tačke na koje se može ubaciti kod aspekta u normalan tok izvršavanja programa, kako bi se dodalo novo ponašanje.

Aspekt ne mora da se primeni na svim mogućim tačkama pridruživanja u kodu. Tačke pridruživanja na koje će se aspekt primeniti se određuju tačkama preseka (engl. *pointcuts*). Tačka preseka dakle određuje tačna mesta gde se aspekt primenjuje. Definicija tačke preseka se poklapa sa jednom ili više tačaka pridruživanja na kojima će se savet utkati (engl. *weaving* – tkanje). Ove tačke preseka se specificiraju na dva načina:

- eksplicitnim nazivima klasa i metoda koje je potrebno savetovati, ili
- regularnim izrazima koji definišu šablone klasa i metoda koje je potrebno savetovati.

Aspekt je zapravo spoj saveta koji treba da se primeni i tačke preseka. Ove dve stvari definišu kompletnu funkcionalnost aspekta – šta treba da radi, kada treba da radi i gde treba da radi.

Utkivanje, odnosno engl. *weaving*, je proces primene aspekata na ciljani objekt pri čemu se kreira proksi oko objekta. Aspekti se utkaju u ciljani objekat na specificiranim tačkama pridruživanja. Utkivanje se može izvršiti u nekoliko tačaka životnog ciklusa ciljanog objekta:

- za vreme kompajliranja – aspekti se utkivaju za vreme kompajliranja ciljane klase. AspectJ *weaving* kompajler radi na ovaj način.
- za vreme učitavanja klasa – aspekti se utkivaju kada se ciljana klasa učita u Java virtuelnu mašinu. Ovo zahteva poseban *ClassLoader*.
- za vreme izvršavanja – aspekti se utkaju u nekom trenutku izvršavanja programa. Tipično će AOP kontejner dinamički generisati proksi objekat oko ciljanog objekta u procesu tkanja aspekata. Na ovaj način funkcioniše Spring AOP.

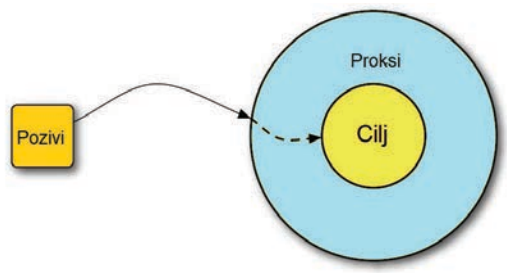
Zbog nezgodne terminologije koja se koristi, koja često nema adekvatan i smislen prevod na srpski, često se i u našem jeziku koriste engleski termini. Na žalost, čak i engleski termini koji opisuju AOP nisu previše smisleni. Sve to čini usvajanje koncepta aspekata znatno težim nego što jeste. Većina stvari će biti razjašnjena u narednim poglavljima. Ono šta za sada treba usvojiti jeste da su tačke pridruživanja sve tačke u toku izvršavanja programa koje su kandidati za primenu saveta aspekta. Tačke preseka definišu na kojim tačkama pridruživanja će se aspekt zaista primeniti.

7.2. Podrška za AOP u Springu

Postoji nekoliko AOP okvira koji se međusobno razlikuju uglavnom u modelima tačaka pridruživanja. Neki okviri dozvoljavaju primenu aspekata na nivou promene vrednosti nekog polja dok drugi dozvoljavaju primenu samo na pozivima metoda. U ovom poglavlju fokus će biti na Springovom okviru za AOP. Odmah na početku se može naglasiti da postoji dosta sličnosti između Spring AOP i AspectJ, pošto se Springov AOP okvir u mnogome oslanja na AspectJ. Ograničenje Springove podrške za AOP ogleda se u tome što je limitirana na presretanje poziva metoda, odnosno ne podržava savete na nivou poziva konstruktora ili promene vrednosti polja klase. Ukoliko presretanje poziva metoda nije dovoljno, preporučuje se implementacija aspekata direktno u AspectJ okviru.

U Spring okruženju, aspekti i saveti se pišu u Javi, odnosno u standardnim Java klasama. To znači da nije potrebno nikakvo dodatno okruženje za razvoj za pisanje aspekata, već se može koristiti standardan IDE u kome se razvija Java projekat. Tačke preseka koje definišu gde će se savet aspekta primeniti se mogu specificirati standardnom XML konfiguracijom ili anotacijama, što bi trebalo da čitaocima već bude poznato iz prethodnih poglavlja.

Spring savetuje objekte za vreme izvršavanja, odnosno aspekti se vezuju (utkaju) u ciljano zrno za vreme izvršavanja tako što se omotaju proksi klasom, kao što je prikazano na slici 7.3. Proksi klasa se omotava oko ciljanog zrna i presreće sve pozive savetovanih metoda, zatim primenjuje logiku aspekta, i nakon toga prosleđuje poziv ciljanom zrnu.



Slika 7.3. Omovatanje zrna proksi klasom prilikom vezivanja aspekata

Spring ne kreira proksi objekat sve dok zrno koje se savetuje ne bude potrebno. Ukoliko se koristi konekst aplikacije (ApplicationContext), proksi objekat će biti kreiran prilikom učitavanja svih zrna iz fabrike zrna. Pošto se u Spring okruženju proksi objekti kreiraju u vreme izvršavanja, nije potreban specijalan kompajler.

Kao što je već napomenuto, Spring podržava samo presretanje metoda, nasuprot drugim okvirima poput AspectJ ili JBoss, koji dodatno omogućavaju postavljanje tačaka preseka na promene vrednosti polja klase i pozive konstruktora. Pošto Spring ne podržava postavljanje tačaka preseka na polja, nije moguće napraviti vrlo precizne savete, poput presretanja ažuriranja vrednosti nekog polja objekta. Bez mogućnosti primene aspekta na poziv konstruktora, nije moguće primeniti savet prilikom instanciranja zrna. U praksi, međutim, presretanje metoda je uglavnom dovoljno za sve potrebe.

7.3. Definisanje tačke preseka i programiranje aspekta (XML)

Tačke preseka definišu mesto gde će se aspekt primeniti. Uz savet aspekta, tačke preseka su osnovni deo svakog aspekta, pa je neophodno razumeti na koji način se implementiraju. U Spring AOP okviru, tačke preseka se definišu pomoću AspectJ izraza. Spring podržava samo podskup svih mogućih izraza za definisanje tačaka preseka koji su dostupni u AspectJ okviru. Neki od dostupnih izraza su prikazani u tabeli 7-1. Pokušaj upotrebe bilo kog nepodržanog AspectJ izraza dovodi do izbacivanja izuzetka tipa IllegalArgumentException.

Tabela 7.1. AspectJ oznake dostupne u Spring okruženju

AspectJ oznake	Kratak opis
args()	Ograničava podudaranje tačaka preseka sa izvršavanjem metoda čiji se argumenti poklapaju sa datim tipovima
@args()	Ograničava podudaranje tačaka preseka sa izvršavanjem metoda čiji su argumenti anotirani sa datim tipovima anotacija
execution()	Podudara se sa tačkama preseka koje označavaju izvršavanje metoda
this()	Ograničava podudaranje tačaka preseka sa onim gde je referenca na proksi zrno datog tipa
target()	Ograničava podudaranje tačaka preseka sa onim gde je ciljani objekat datog tipa
@target	Ograničava podudaranje tačaka preseka sa onim gde klasa objekta ima anotacije datog tipa
within()	Ograničava podudaranje tačaka preseka sa onim koje su unutar zadatih tipova

Ukoliko se detaljnije pogledaju podržane oznake, može se uočiti da samo *execution* oznaka zaista izvršava proveru podudaranja – sve ostale oznake služe da ograniče broj podudaranja. To znači da je *execution* zapravo osnovna oznaka koja se koristi u svim definicijama tačaka preseka, a sve druge oznake samo ograničavaju domet tačke preseka.

Pisanje tačaka preseka biće pokazano na konkretnom primeru. Neka se posmatra projekat sa sledećim podešavanjima unutar pom.xml. U odnosu na standardne zavisnosti, može se uočiti dodatak podrške za aspekte (spring-aspects):

```
<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://ma-
ven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>rs.ac.singidunum.fir.pj</groupId>
  <artifactId>Spring-uvod</artifactId>
  <version>0.0.1-SNAPSHOT</version>
  <dependencies>
              <dependency>
              <groupId>org.springframework</groupId>
              <artifactId>spring-core</artifactId>
              <version>5.0.7.RELEASE</version>
```

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

Neka u projektu postoji klasa Telefon, jednostavna klasa koja simulira funkcionalnosti mobilnog telefona, i ima dve metode, pozovi(), i posaljiPoruku(). Neka je kod klase Telefon dat sa:

```
package rs.ac.singidunum.fir.pj;

public class Telefon {

        public void pozovi () {
                System.out.println("Uspostavljanje poziva");
        }

        public void posaljiPoruku() {
                System.out.println("Slanje poruke");
        }

}
```

Neka je data i jednostavna klasa LogerTelefona, koja treba da loguje rad klase Telefon. Od klase LogerTelefona ćemo kasnije napraviti aspekt, ali ono šta treba uočiti na ovom mestu je da je klasa LogerTelefona najobičniji POJO – ništa u njoj ne ukazuje da će biti iskorišćena kao aspekt. Ova klasa za sada ima dve metode – jedna kojom želimo da logujemo početak poziva, a drugom logujemo kraj poziva. Ovo je inače standardna funkcionalnost logovanja bilo koje aktivnosti.

```
package rs.ac.singidunum.fir.pj;

public class LogerTelefona {

	public void preUspostavljanjaPoziva () {
		System.out.println("Aspekt: pre poziva");
	}

	public void posleZavrsetkaPoziva () {
		System.out.println("Aspekt: posle poziva");
	}

}
```

Konfiguracioni XML fajl beans.xml je definisan na uobičajen način:

```
<?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:context="http://www.springframework.org/schema/con-
text"
	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-
4.3.xsd">

	<bean id="telefon" class="rs.ac.singidunum.fir.
pj.Telefon"></bean>
	<bean id="loger" class="rs.ac.singidunum.fir.
pj.LogerTelefona"></bean>
</beans>
```

Neka postoji i jednotavna App.java klasa u kojoj se demonstrira rad AOP. App.java je definisana na sledeći način.

```
package rs.ac.singidunum.fir.pj;

import org.springframework.context.ApplicationContext;
import org.springframework.context.support.FileSystemXmlApplica-
tionContext;
```

```
public class App {
       public static void main (String[] args){

              ApplicationContext context = new FileSystemXmlApplic
ationContext(“beans.xml”);
              Telefon telefon = (Telefon) context.getBean(“telefon”);
              telefon.pozovi();
              telefon.posaljiPoruku();
              ((FileSystemXmlApplicationContext)context).close();

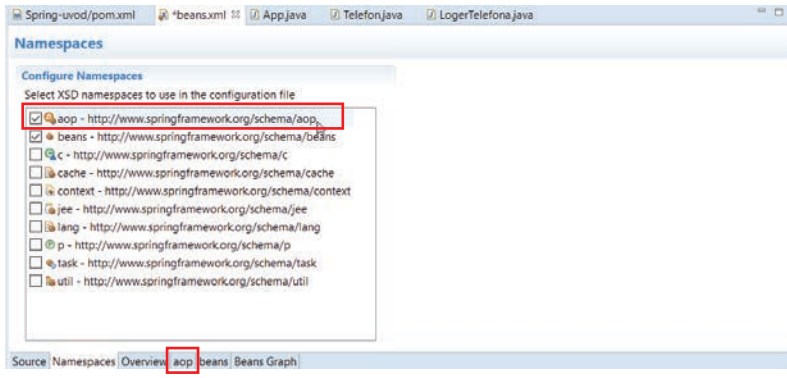
       }
}
```

Ukoliko se program izvrši u ovom obliku, rezultat izvršavanja bi bio sledeći.

```
Feb 02, 2019 7:53:46 PM org.springframework.context.support.Ab-
stractApplicationContext prepareRefresh
INFO: Refreshing org.springframework.context.support.FileSystemXm-
lApplicationContext@bebdb06: startup date [Sat Feb 02 19:53:46 CET
2019]; root of context hierarchy
Feb 02, 2019 7:53:46 PM org.springframework.beans.factory.xml.Xml-
BeanDefinitionReader loadBeanDefinitions
INFO: Loading XML bean definitions from file [C:\Spring\Uvod\beans.
xml]
Uspostavljanje poziva
Slanje poruke
Feb 02, 2019 7:53:46 PM org.springframework.context.support.Ab-
stractApplicationContext doClose
INFO: Closing org.springframework.context.support.FileSystemXmlAp-
plicationContext@bebdb06: startup date [Sat Feb 02 19:53:46 CET
2019]; root of context hierarchy
```

Ono što želimo da uradimo jeste da od klase LogerTelefona napravimo aspekt. Cilj je da se aspekt implementira na takav način da se presretnu pozivi metoda klase Telefon, i da se pomoću aspekta izvrše saveti nad metodama klase Telefon.

Kako bi se dodala podrška za aspekte, neophodno je koristiti Springov aop imenski prostor, koji se može dodati ili ručno u XML fajlu, ili upotrebom editora. Ukoliko se koristi editor, potrebno je u tabu *Namespaces* odabrati aop imenski prostor, čime će se pojaviti još jedan tab (aop) u editoru, kao na slici 7.4. Time se uvode dodatni elementi za aspektno orijentisano programiranje koji se dalje mogu koristiti u konfiguraciji.



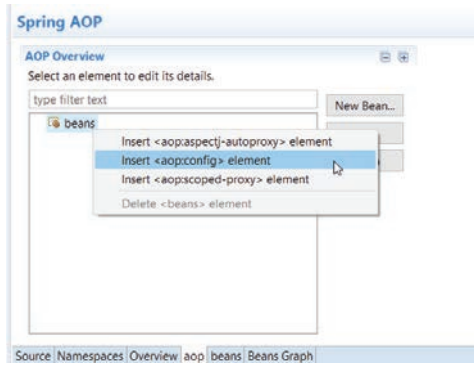
Slika 7.4. Dodavanje aop imenskog prostora u editoru

Nakon dodavanja aop imenskog prostora, na raspolaganju je nekoliko novih konfiguracionih elemenata koji se mogu dodati u XML fajl. Ovi elementi su prikazani u tabeli 7-2.

Tabela 7.2. Aop konfiguracioni elementi

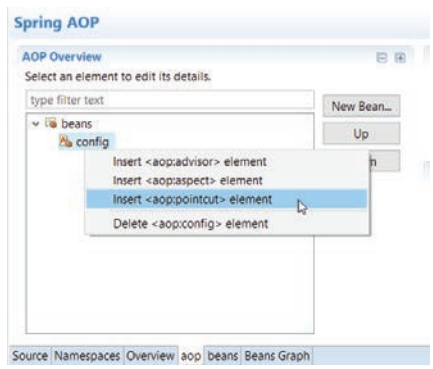
AOP element	Svrha
<aop:config>	Koreni AOP element – skoro svi ostali aop elementi se pišu unutar ovog elementa
<aop:aspect>	Definiše aspekt
<aop:pointcut>	Definiše tačku preseka
<aop:aspectj-autoproxy>	Uključuje podršku za anotacije @AspectJ
<aop:before>	Definiše savet koji treba da se izvrši pre savetovane metode
<aop:after>	Definiše savet koji treba da se izvrši nakon završetka savetovane metode
<aop:after-returning>	Definiše se savet koji treba da se izvrši nakon uspešnog završetka savetovane metode
<aop:after-throwing>	Definiše se savet koji treba da se izvrši u slučaju da savetovana metoda izbaci izuzetak
<aop:around>	Definiše se savet koji se delimično izvršava pre, a delimično posle savetovane metode
<aop:advisor>	Definiše se AOP savetnik

Element <aop:config> je koreni element, i njega je neophodno prvog dodati. Skoro svi ostali elementi se dodaju unutar <aop:config> elementa. Ovaj element se može dodati upotrebom editora, kao što je prikazano na slici 7.5. Naravno, kao i uvek, moguće je pisati elemente i direktno u XML fajlu.



Slika 7.5. Dodavanje korenog aop elementa

Nakon dodavanja <aop:config> elementa, unutar njega se mogu dodati novi aop elementi. Neka je potrebno dodati dve tačke preseka (engl. *pointcut*). Potrebno je kliknuti desnim dugmetom miša na <aop:config> element, i zatim odabrati Insert <aop:pointcut> element, kao na slici 7.6.

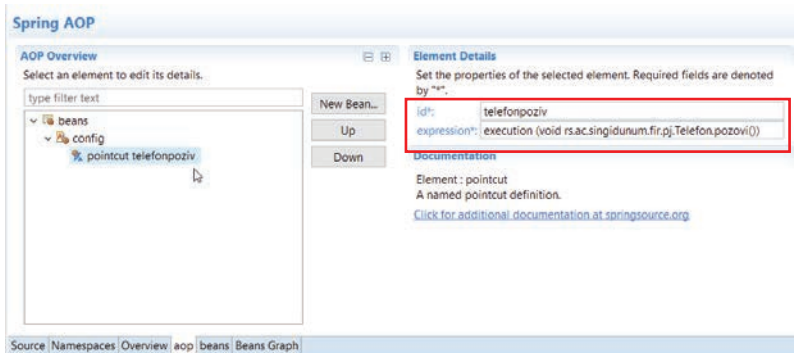


Slika 7.6. Dodavanje elementa koji definiše tačku preseka

Svakoj tački preseka se može dodeliti ime (npr. telefonpoziv), postavljanjem polja Id, i može se dodati izraz u AspectJ sintaksi, kojim se definiše mesto preseka, odnosno metoda koju želimo da presretnemo. U slučaju prve tačke preseka, želimo da presretnemo izvršavanje metode pozovi() u klasi Telefon. U tom slučaju, AspectJ izraz koji definiše presretanje metode pozovi() iz klase Telefon dat je sa:

```
execution (void rs.ac.singidunum.fir.pj.Telefon.pozovi())
```

Konfiguracija prve tačke preseka sa Id *telefonpoziv* prikazana je na slici 7.7.

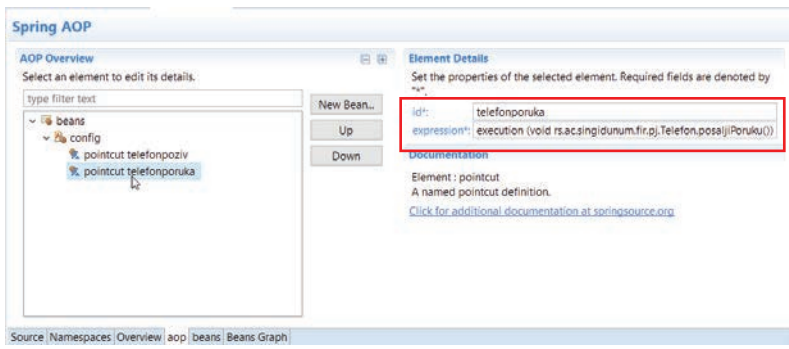


Slika 7.7. Definisanje tačke preseka na izvršavanju metode Telefon.pozovi()

Druga tačka preseka treba da presretne poziv metode posaljiPoruku() iz klase Telefon. AspectJ izraz koji definiše presretanje ove metode dat je sa:

```
execution (void rs.ac.singidunum.fir.pj.Telefon.posaljiPoruku())
```

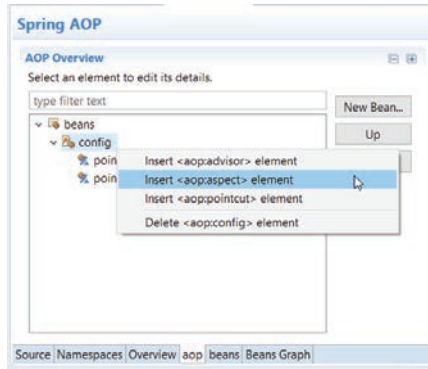
Konfiguracija druge tačke preseka, sa Id *telefonporuka* prikazana je na slici 7.8.



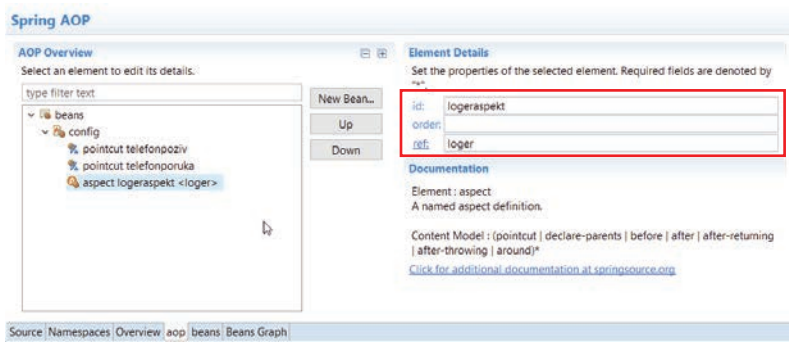
Slika 7.8. Definisanje tačke preseka na izvršavanju metode Telefon.posaljiPoruku()

Na ovaj način dodali smo dve tačke preseka. To su dve tačke u izvršavanju programa u kojima mi želimo da presretnemo normalan tok programa i ubacimo izvršavanje logike aspekta. Sledeći logičan korak u konfiguraciji aspekta jeste dodavanje i definisanje samog aspekta. Potrebno je unutar <aop:config> elementa dodati <aop:aspect> element, kao što je prikazano na slici 7.9.

Nakon dodavanja aspekt elementa, potrebno mu je dodeliti njegov Id, kao i postaviti referencu na zrno koje će pružiti funkcionalnost aspekta (u našem slučaju klasa LogerTelefona će igrati ulogu aspekta, odnosno zrno tipa LogerTelefona sa Id *loger*). Podešavanje aspekta je prikazano na slici 7.10.



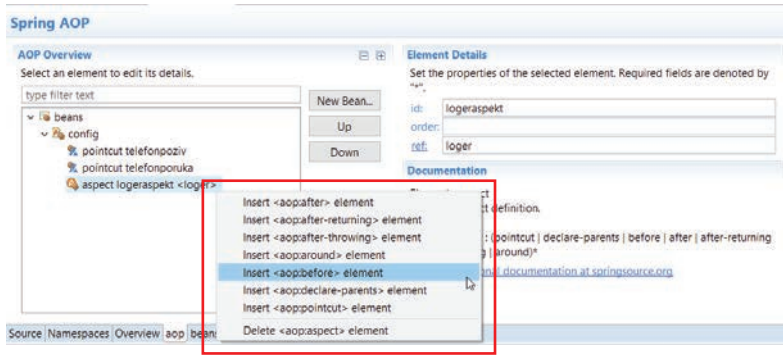
Slika 7.9. Dodavanje aspekta unutar <aop:config> elementa



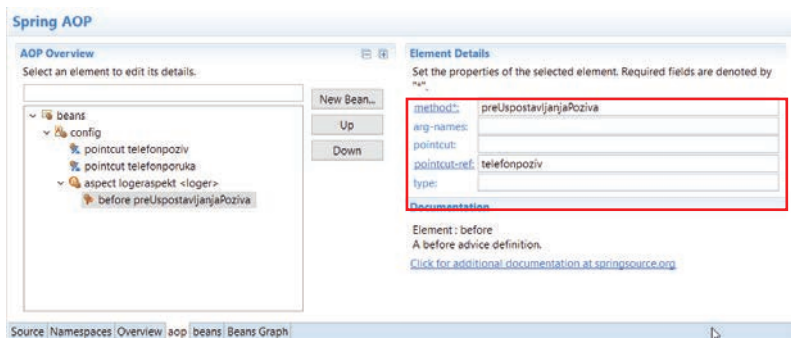
Slika 7.10. Podešavanje aspekta i dodavanje reference
na zrno koje će pružiti funkcionalnost

Do sada smo definisali tačke preseka, odnosno mesta na kojima će se izvršavati dodatna funkcionalnost logovanja. Precizirali smo da su tačke preseka izvršavanje metoda klase Telefon. Definisali smo i aspekt, odnosno definisali smo da će zrno sa Id *loger*, tipa LogerTelefona, pružiti funkcionalnost aspekta. Sada je potrebno definisati i kada će se primeniti savet aspekta. Saveti aspekta su zapravo metode klase LogerTelefona. Unutar <aop:aspect> elementa moguće je dodati jedan od elemenata koji će specificirati da li će se neka od metoda aspekta izvršiti pre ciljne metode, posle ciljne metode, ukoliko ciljna metoda izbaci izuzetak i slično. Desnim klikom na <aop:aspect> element dobijaju se sve moguće opcije, kao što je prikazano na slici 7.11.

Dodajmo za početak jedan <aop:before> element. Potrebno je definisati koja metoda aspekta će se izvršiti (u našem slučaju metoda preUspostavljanjaPoziva() iz zrna loger tipa LogerTelefona). Potrebno je dodati i referencu na tačku preseka – odnosno koju metodu želimo da presretnemo (u našem slučaju odabrana tačka preseka je telefonpoziv). Pošto je element tipa *before*, to znači da će se metoda aspekta izvršiti pre metode koja se presreće. Podešavanje je prikazano na slici 7.12.



Slika 7.11. Podešavanje kada će se savet aspekta izvršiti



Slika 7.12. Podešavanje *before* saveta

Sada imamo implementiran jedan kompletan savet. Ukoliko se pogleda sadržaj beans.xml fajla, može se uočiti ubačeni kod koji daje funkcionalnost aspekta. Kompletna konfiguracija aspekta je u XML fajlu. Nije bila potrebna nikakva izmena klasa Telefon i LogerTelefona. Preciznije, zrno tipa Telefon ni ne zna da je logovano. Kod beans.xml je dat u nastavku.

```
<?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:aop="http://www.springframework.org/schema/aop"
       xsi:schemaLocation="http://www.springframework.org/schema/
beans http://www.springframework.org/schema/beans/spring-beans.xsd
            http://www.springframework.org/schema/aop http://
www.springframework.org/schema/aop/spring-aop-4.3.xsd">

       <bean id="telefon"
class="rs.ac.singidunum.fir.pj.Telefon"></bean>
       <bean id="loger"
class="rs.ac.singidunum.fir.pj.LogerTelefona"></bean>
```

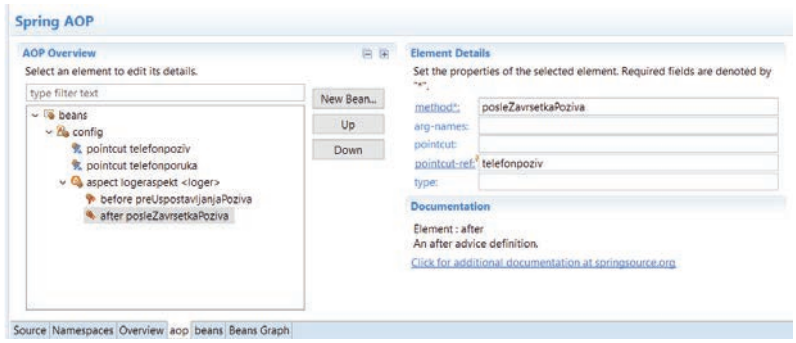
```
      <aop:config>
            <aop:pointcut
                  expression="execution (void rs.ac.singidunum.
fir.pj.Telefon.pozovi())"
                  id="telefonpoziv" />
            <aop:pointcut
                  expression="execution (void rs.ac.singidunum.
fir.pj.Telefon.posaljiPoruku())"
                  id="telefonporuka" />
            <aop:aspect id="logeraspekt" ref="loger">
                  <aop:before method="preUspostavljanjaPoziva"
                        pointcut-ref="telefonpoziv" />
            </aop:aspect>
      </aop:config>
</beans>
```

Iz XML fajla se možda preglednije vidi šta je urađeno do sada. Definisan je aspekt sa id *logeraspekt* koji ima referencu na zrno *loger*, koje je tipa LogerTelefona. Svaki put kada se pozove metoda pozovi() iz klase Telefon, taj poziv će biti presretnut (definisano tačkom preseka sa Id *telefonpoziv*), a pre izvršavanja te metode biće okinut poziv metode aspeta preUspostavljanjaPoziva(). Primer izvršavanja jasno prikazuje da je metoda aspekta umetnuta pre poziva ciljne metode.

```
Feb 02, 2019 10:03:47 PM org.springframework.context.support.Ab-
stractApplicationContext prepareRefresh
INFO: Refreshing org.springframework.context.support.FileSystemXm-
lApplicationContext@bebdb06: startup date [Sat Feb 02 22:03:47 CET
2019]; root of context hierarchy
Feb 02, 2019 10:03:47 PM org.springframework.beans.factory.xml.
XmlBeanDefinitionReader loadBeanDefinitions
INFO: Loading XML bean definitions from file [C:\Spring\Uvod\beans.
xml]
Aspekt: pre poziva
Uspostavljanje poziva
Slanje poruke
Feb 02, 2019 10:03:47 PM org.springframework.context.support.Ab-
stractApplicationContext doClose
INFO: Closing org.springframework.context.support.FileSystemXmlAp-
plicationContext@bebdb06: startup date [Sat Feb 02 22:03:47 CET
2019]; root of context hierarchy
```

Na sličan način se može definisati i <aop:after> element. Njime se definiše metoda aspekta koja će biti izvršena nakon završetka ciljne metode. Ciljna meoda se definiše

preko tačke preseka koja je zakačena na izvršavanje ciljne metode. Na slici 7.13 prikazano je podešavanje <aop:after> elementa. Koristi se ista tačka preseka pošto presrećemo istu metodu kao i u prethodnom slučaju.



Slika 7.13. Podešavanje after saveta

Sadržaj beans.xml fajla nakon dodavanja after elementa je dat u nastavku.

```
<?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:aop="http://www.springframework.org/schema/aop"
       xsi:schemaLocation="http://www.springframework.org/schema/
beans http://www.springframework.org/schema/beans/spring-beans.
xsd
              http://www.springframework.org/schema/aop http://
www.springframework.org/schema/aop/spring-aop-4.3.xsd">

       <bean id="telefon"
class="rs.ac.singidunum.fir.pj.Telefon"></bean>
       <bean id="loger"
class="rs.ac.singidunum.fir.pj.LogerTelefona"></bean>
       <aop:config>
              <aop:pointcut
                     expression="execution (void rs.ac.singidunum.
fir.pj.Telefon.pozovi())"
                     id="telefonpoziv" />
              <aop:pointcut
                     expression="execution (void rs.ac.singidunum.
fir.pj.Telefon.posaljiPoruku())"
                     id="telefonporuka" />
```

```
            <aop:aspect id="logeraspekt" ref="loger">
                  <aop:before method="preUspostavljanjaPoziva"
                        pointcut-ref="telefonpoziv" />
                  <aop:after method="posleZavrsetkaPoziva"
                        pointcut-ref="telefonpoziv" />
            </aop:aspect>
      </aop:config>
</beans>
```

Sada imamo dva saveta našeg aspekta – jedan koji se izvršava pre metode koja se savetuje, a drugi koji se izvršava nakon završetka metode koja se savetuje. Ukoliko se program sada pokrene, rezultat izvršavanja dat je u nastavku:

```
Feb 02, 2019 10:22:16 PM org.springframework.context.support.Ab-
stractApplicationContext prepareRefresh
INFO: Refreshing org.springframework.context.support.FileSystemXm-
lApplicationContext@bebdb06: startup date [Sat Feb 02 22:22:16 CET
2019]; root of context hierarchy
Feb 02, 2019 10:22:16 PM org.springframework.beans.factory.xml.
XmlBeanDefinitionReader loadBeanDefinitions
INFO: Loading XML bean definitions from file [C:\Spring\Uvod\beans.
xml]
Aspekt: pre poziva              <- before – metoda iz zrna loger
Uspostavljanje poziva           <- ciljna metoda iz zrna telefon
Aspekt: posle poziva            <- after – metoda iz zrna loger
Slanje poruke
Feb 02, 2019 10:22:16 PM org.springframework.context.support.Ab-
stractApplicationContext doClose
INFO: Closing org.springframework.context.support.FileSystemXmlAp-
plicationContext@bebdb06: startup date [Sat Feb 02 22:22:16 CET
2019]; root of context hierarchy
```

7.4. Around savet, XML

Osnovni saveti definisani sa elementima <aop:before> i <aop:after> rade odlično, ali imaju određena ograničenja. Na primer, jedno od ograničenja je deljenje informacija između saveta koji se izvršava pre, i saveta koji se izvršava posle ciljne metode, bez upotrebe dodatnih pomoćnih promenljivih. Upotrebom around saveta, koji se definiše sa elementom <aop:around>, može se postići ista stvar kao i sa pojedinačnim before i after savetima, uz veliku prednost da se to može postići samo sa jednom metodom. Around metoda se omotava oko ciljane metode, pa nisu neophodni before

i after saveti. Posmatrajmo ponovo klasu LogerTelefona, u kojoj je dodata upravo jedna takva metoda koja će se koristiti kao around savet. Ovu metodu ćemo zakačiti za izvršavanje metode posaljiPoruku() klase Telefon.

```
package rs.ac.singidunum.fir.pj;

import org.aspectj.lang.ProceedingJoinPoint;

public class LogerTelefona {

	public void preUspostavljanjaPoziva () {
		System.out.println("Aspekt: pre poziva");
	}

	public void posleZavrsetkaPoziva () {
		System.out.println("Aspekt: posle poziva");
	}

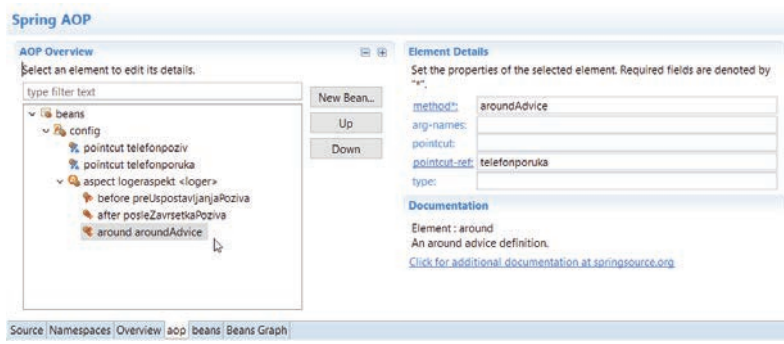
	public void aroundAdvice (ProceedingJoinPoint p){
		System.out.println("Aspekt around: pre slanja poruke");
		long pocetak = System.currentTimeMillis();
		//pokreni izvrsavanje ciljne metode
		try {
			p.proceed();
		} catch (Throwable e) {
			System.out.println(e.getMessage());
		}
		//ciljna metoda je uspesno zavrsena
		long kraj = System.currentTimeMillis();
		long vreme = kraj - pocetak;
		System.out.println("Aspekt around: posle slanja
	poruke");
	System.out.println("Slanje poruke je trajalo: " + vreme + "
	ms");
	}
}
```

Ukoliko se detaljnije pogleda metoda aroundAdvice(), moguće je uočiti nekoliko stvari koje treba dodatno pojasniti. Metoda around se omotava oko ciljne metode. Postavlja se pitanje – kada će se okinuti ciljna metoda? U slučaju before i after saveta, bilo je jednostavno, ciljna metoda se okidala posle saveta (*before* – savet ide pre ciljne metode), odnosno pre saveta (*after* – savet ide posle ciljne metode). Metoda around kao argument prihvata ProceedingJoinPoint parametar. On je neophodan, pošto je

u slučaju around saveta kompletna kontrola pozivanja presretnute metode na programeru. Around savet se okida pre izvršavanja ciljne metode, i u nekom trenutku je potrebno ručno pokrenuti ciljanu metodu i proslediti joj kontrolu, što se postiže pozivom p.proceed(). Pošto može doći do izbacivanja izuzetka, poziv p.proceed() mora biti unutar try-catch bloka.

Kako bi se ilustrovalo deljenje informacija delova koji se izvršavaju pre i posle ciljne metode unutar saveta, meri se vreme trajanja izvršavanja ciljne metode. Uzima se početno vreme pre poziva ciljne metode, a zatim nakon završetka izvršavanja ciljne metode se uzima krajnje vreme, i na osnovu njih sračunava ukupno vreme izvršavanja ciljne metode.

Sam around savet i postavljanje tačke preseka se unutar editora radi na sličan način kao i u slučaju before i after saveta, dodavanjem <aop:around> elementa i podešavanjem tačke preseka i metode aspekta koju je potrebno izvršiti kao savet, kao što je prikazano na slici 7.14.



Slika 7.14. Podešavanje around saveta u editoru

Nakon svih dodatih izmena, beans.xml fajl ima sledeći sadržaj:

```
<?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:aop="http://www.springframework.org/schema/aop"
       xsi:schemaLocation="http://www.springframework.org/schema/
beans http://www.springframework.org/schema/beans/spring-beans.xsd
            http://www.springframework.org/schema/aop http://
www.springframework.org/schema/aop/spring-aop-4.3.xsd">

       <bean id="telefon"
class="rs.ac.singidunum.fir.pj.Telefon"></bean>
       <bean id="loger"
class="rs.ac.singidunum.fir.pj.LogerTelefona"></bean>
       <aop:config>
```

```
            <aop:pointcut
                  expression="execution (void rs.ac.singidunum.
fir.pj.Telefon.pozovi())"
                  id="telefonpoziv" />
            <aop:pointcut
                  expression="execution (void rs.ac.singidunum.
fir.pj.Telefon.posaljiPoruku())"
                  id="telefonporuka" />
            <aop:aspect id="logeraspekt" ref="loger">
                  <aop:before method="preUspostavljanjaPoziva"
                        pointcut-ref="telefonpoziv" />
                  <aop:after method="posleZavrsetkaPoziva"
                        pointcut-ref="telefonpoziv" />
                  <aop:around method="aroundAdvice"
                        pointcut-ref="telefonporuka" />
            </aop:aspect>
      </aop:config>
</beans>
```

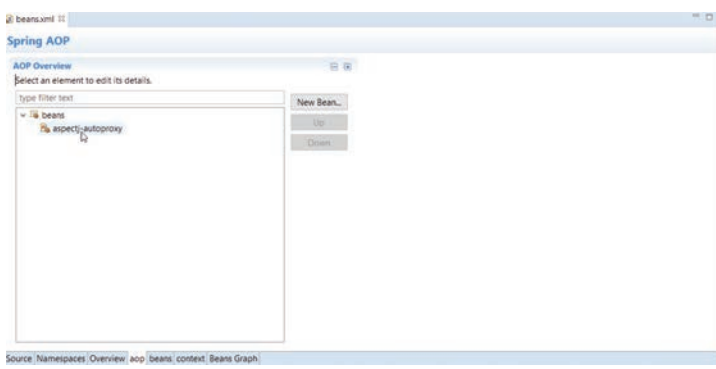
Nakon unetih izmena, ukoliko se pokrene program, dobiće se sledeći rezultat izvršavanja:

```
Feb 02, 2019 11:53:19 PM org.springframework.context.support.Ab-
stractApplicationContext prepareRefresh
INFO: Refreshing org.springframework.context.support.FileSystem-
XmlApplicationContext@bebdb06: startup date [Sat Feb 02 23:53:19
CET 2019]; root of context hierarchy
Feb 02, 2019 11:53:19 PM org.springframework.beans.factory.xml.
XmlBeanDefinitionReader loadBeanDefinitions
INFO: Loading XML bean definitions from file [C:\Spring\Uvod\beans.
xml]
Aspekt: pre poziva
Uspostavljanje poziva
Aspekt: posle poziva
Aspekt around: pre slanja poruke       <- around savet
Slanje poruke                          <- ciljna metoda
Aspekt around: posle slanja poruke     <- around savet
Slanje poruke je trajalo: 10 ms        <- around savet – mereno
vreme
Feb 02, 2019 11:53:19 PM org.springframework.context.support.Ab-
stractApplicationContext doClose
INFO: Closing org.springframework.context.support.FileSystemXm-
lApplicationContext@bebdb06: startup date [Sat Feb 02 23:53:19
CET 2019]; root of context hierarchy
```

7.5. Aspekti sa anotacijama

Kompletna konfiguracija koja se može postići pomoću XML fajla se uglavnom može postići i upotrebom anotacija, kao što smo videli u prethodnim poglavljima. Slična situacija je i sa aspektima. Spring koristi AspectJ model anotacija, koji dodatno olakšava pretvaranje bilo koje klase u aspekt samo dodavanjem nekoliko anotacija. Ova funkcionalnost je poznata pod nazivom @AspectJ. U prethodnom primeru, klasa LogerTelefona sadrži kompletnu funkcionalnost koja je potrebna za aspekt, ali ništa u njoj ne naznačava da je ona zaista aspekt. Zbog toga je bilo potrebno uraditi poprilično posao deklarisanja i konfiguracije saveta i tačaka preseka unutar XML fajla. Kao i u prethodnim poglavljima, upotreba anotacija značajno redukuje količinu XML koda, dodavanjem nekoliko anotacija na odgovarajućim mestima u klasama. To je razlog zbog čega se često i preferira upotreba anotacija (mada je potpuno ravnopravna sa XML konfiguracijom). Posmatra se isti primer sa istim funkcionalnostima kao u prethodnom poglavlju, samo u ovom slučaju su upotrebljene anotacije.

Na početku, uklonjene su sve deklaracije zrna i svih podešavanja vezanih za aspekte iz XML fajla. Potrebno je postaviti <context:component-scan> element, i sve klase anotirati sa @Component kako bi bile automatski otkrivene i deklarisane kao zrna. Osim toga, potrebno je deklarisati autoproksi zrno u Spring kontekstu koje će uključiti @AspectJ anotacije. Spring daje podršku za autoproksi kroz <aop:aspectj-autoproxy/> element koji se nalazi u aop imenskom prostoru. Uz pomoć editora, ovaj element se jednostavno dodaje u okviru aop taba, kao što je prikazano na slici 7.15.



Slika 7.15. Konfigurisanje aspectj-autoproxy elementa

Ukoliko se pogleda XML kod beans.xml fajla, vidi se da je drastično kraći nego u prethodnom poglavlju:

```
<?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:aop="http://www.springframework.org/schema/aop"
       xmlns:context="http://www.springframework.org/schema/context"
       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-4.3.xsd
            http://www.springframework.org/schema/aop http://
www.springframework.org/schema/aop/spring-aop-4.3.xsd">

       <context:component-scan base-package="rs.ac.singidunum.fir.
pj"></context:component-scan>
       <aop:aspectj-autoproxy></aop:aspectj-autoproxy>
</beans>
```

Klase programa takođe moraju biti malo izmenjene. Za početak, u klasi Telefon neophodno je dodati @Component anotaciju koja će omogućiti skeniranje.

```
package rs.ac.singidunum.fir.pj;

import org.springframework.stereotype.Component;

@Component
public class Telefon {

       public void pozovi () {
              System.out.println("Uspostavljanje poziva");
       }

       public void posaljiPoruku() {
              System.out.println("Slanje poruke");
       }

}
```

Sa druge strane, klasa LogerTelefona se mora malo više izmeniti. Osim dodavanja @Component za potrebe automatskog skeniranja, potrebno je dodati i @Aspect anotaciju, kako bismo naznačili da je klasa aspekt. Potrebno je dodati i anotaciju za tačke preseka - @Pointcut, unutar koje se stavlja AspectJ izraz koji definiše podudaranje sa metodama koje je potrebno savetovati. Tačka preseka se obično anotira nad praznom

metodom, koja služi samo da nosi anotaciju. Što se tiče anotacija @Before, @After i @Around, one se postavljaju direktno iznad metoda aspekta koje treba da se pozovu. Unutar zagrada ovih anotacija postavlja se ime metode koja nosi anotaciju za tačku preseka. Nakon unetih promena, klasa LogerTelefona izgleda kao što je prikazano u nastavku.

```
package rs.ac.singidunum.fir.pj;

import org.aspectj.lang.ProceedingJoinPoint;
import org.aspectj.lang.annotation.After;
import org.aspectj.lang.annotation.Around;
import org.aspectj.lang.annotation.Aspect;
import org.aspectj.lang.annotation.Before;
import org.aspectj.lang.annotation.Pointcut;
import org.springframework.stereotype.Component;

@Aspect
@Component
public class LogerTelefona {

	@Pointcut("execution (void rs.ac.singidunum.fir.pj.Telefon.
pozovi())")
	public void telefonPoziv(){
		//dummy metoda, sluzi da nosi anotaciju za pointcut
	}

	@Pointcut("execution (void rs.ac.singidunum.fir.pj.Telefon.
posaljiPoruku())")
	public void telefonPoruka(){
		//dummy metoda, sluzi da nosi anotaciju za pointcut
	}

	@Before("telefonPoziv()")
	public void preUspostavljanjaPoziva () {
		System.out.println("Aspekt: pre poziva");
	}

	@After("telefonPoziv()")
	public void posleZavrsetkaPoziva () {
		System.out.println("Aspekt: posle poziva");
	}
```

```
	@Around("telefonPoruka()")
	public void aroundAdvice (ProceedingJoinPoint p){
		System.out.println("Aspekt around: pre slanja poruke");
		long pocetak = System.currentTimeMillis();
		//pokreni izvrsavanje ciljne metode
		try {
			p.proceed();
		} catch (Throwable e) {
			System.out.println(e.getMessage());
		}
		//ciljna metoda je uspesno zavrsena
		long kraj = System.currentTimeMillis();
		long vreme = kraj - pocetak;
		System.out.println("Aspekt around: posle slanja poruke");
		System.out.println("Slanje poruke je trajalo: " +
vreme + " ms");
	}
}
```

Dati kod se identično izvršava kao i kod u prethodnom pogavlju, gde je korišćen XML za konfiguraciju aspekata. Očigledno se vidi koliko upotreba anotacija olakšava pisanje programa i smanjuje ukupnu količinu koda.

7.6. Definisanje tačaka preseka sa džokerima

Džokeri (engl. *wild cards*) se mogu koristiti za primenu aspekta na više od jedne metode. Neka je klasa Telefon izmenjena tako što je dodato još nekoliko metoda koje preklapaju ime metode posaljiPoruku(). Sve preklopljenje metode se naravno moraju razlikovati u parametrima (odnosno, preciznije rečeno, red, broj i tipovi parametara preklopljenih metoda se moraju razlikovati). Modifikovana klasa Telefon, sa dodatim preklopljenim metodama, prikazana je u nastavku.

```
package rs.ac.singidunum.fir.pj;

import org.springframework.stereotype.Component;

@Component
public class Telefon {

	public void pozovi () {
		System.out.println("Uspostavljanje poziva");
	}
```

```
        public void posaljiPoruku() {
                System.out.println("Slanje poruke");
        }

        public void posaljiPoruku(String tekst){
                System.out.println("Slanje poruke " + tekst);
        }

        public boolean posaljiPoruku (int brojPonavljanja){
                for (int i = 0; i < brojPonavljanja; i++){
                        System.out.println("Slanje poruke");
                }
                return true;
        }

}
```

Koristeći AspectJ sintaksu, moguće je definisati tačke preseka koje će se poklapati sa više od jedne metode. Na primer, ukoliko želimo da definišemo tačku preseka koja treba da se primeni na sve metode klase Telefon čije je ime posaljiPoruku i povratni tip void, bez razmatranja parametara (označava se sa dve tačke unutar zagrada), izraz koji bi definisao takvu tačku preseka bi bio u obliku:

```
execution (void rs.ac.singidunum.fir.pj.Telefon.posaljiPoruku(..))
```

Ovakav izraz bi se primenio na prve dve preklopljene metode u klasi Telefon.

Ukoliko bismo želeli da definišemo tačku preseka koja se primenjuje na sve metode u klasi Telefon, bez obzira na povratni tip, ime metode i broj parametara, može se koristiti znak *:

```
execution (* rs.ac.singidunum.fir.pj.Telefon.*(..))
```

Ukoliko bismo želeli da se definiše tačka preseka koja se odnosi na sve metode svih klasa u projektu, mogao bi da se koristi izraz u obliku:

```
execution (* *.*(..))
```

7.7. Zadaci za vežbu

Zadatak 1: Napisati Spring projekt upotrebom Maven-a koji sadrži klasu Student koja ima polja ime i godine. Implementirati get i set metode kao i metod polaziIspit() koja ispisuje poruku na standardni izlaz da student polaže ispit. Metoda polaziIspit postaviti da bude tačka prekida gde se pre poziva savet klase Logovanje, metoda beforeSavet koja ispisuje poruku da je student prijavio ispit. Posle tačke prekida se poziva metoda afterSavet klase Logovanje.

Rešenje: Prvo ubacujemo sve potrebne zavisnosti u pom.xml. Pošto je potrebno koristiti AOP aspekte, ubacujemo i tu zavisnost što je dodatak u odnosu na zadatke iz prethodnih poglavlja:

```
<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://ma-
ven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>rs.ac.singidunum.fir.pj</groupId>
  <artifactId>Primer71_Logger</artifactId>
  <version>0.0.1-SNAPSHOT</version>
    <dependencies>
      <dependency>
            <groupId>org.springframework</groupId>
            <artifactId>spring-beans</artifactId>
            <version>5.1.4.RELEASE</version>
      </dependency>
      <dependency>
            <groupId>org.springframework</groupId>
            <artifactId>spring-context</artifactId>
            <version>5.1.4.RELEASE</version>
      </dependency>
      <dependency>
            <groupId>org.springframework</groupId>
            <artifactId>spring-core</artifactId>
            <version>5.1.4.RELEASE</version>
      </dependency>
      <dependency>
            <groupId>org.sparingframework</groupId>
            <artifactId>spring-aspects</artifactId>
            <version>5.0.7.RELEASE</version>
      </dependency>
  </dependencies>
</project>
```

Klasa student je standardna Java klasa:

```
package rs.ac.singidunum.fir.pj;

public class Student {
         private Integer godine;
         private String ime;
```

```
        public void setAge(Integer godine) {
           this.godine = godine;
        }

        public Integer getAge() {
           System.out.println("Uzrast : " + godine );
           return godine;
        }

        public void setName(String ime) {
           this.ime = ime;
        }

        public String getName() {
           System.out.println("Ime : " + ime );
           return ime;
        }

        public void polaziIspit() {
                System.out.println("Ime : " + ime + " polaze
ispit");
        }
}
```

Aspekt i tačka prekida su kreirani pomoću editora. Fajl beans.xml:

```
<?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:aop="http://www.springframework.org/schema/aop"
       xmlns:context="http://www.springframework.org/schema/context"
       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-4.3.xsd
            http://www.springframework.org/schema/aop http://
www.springframework.org/schema/aop/spring-aop-4.3.xsd">
   <bean id = "student" class = "rs.ac.singidunum.fir.pj.Student">
      <property name = "name"  value = "Zara" />
      <property name = "age"  value = "11"/>
   </bean>
   <bean id = "Logovanje" class = "rs.ac.singidunum.fir.
pj.Logovanje"/>
```

```
   <aop:config>
      <aop:pointcut
            expression=”execution   (void   rs.ac.singidunum.fir.
pj.Student.polaziIspit())”
            id=”polaganje” />
   </aop:config>
   <aop:config>
      <aop:aspect id=”loger” ref=”logovanje”>
            <aop:before      method=”beforeSavet”      pointcut-
ref=”polaganje” />
            <aop:after       method=”afterSavet”       pointcut-
ref=”polaganje” />
      </aop:aspect>
   </aop:config>
</beans>
```

Glavna klasa:

```
package rs.ac.singidunum.fir.pj;

import org.springframework.context.ApplicationContext;
import   org.springframework.context.support.ClassPathXmlApplica-
tionContext;
import  org.springframework.context.support.FileSystemXmlApplica-
tionContext;

public class App {

      public static void main(String[] args) {
            ApplicationContext context = new FileSystemXmlApplic
ationContext(“beans.xml”);

            Student student = (Student) context.getBean(“student”);

            student.polaziIspit();
            ((FileSystemXmlApplicationContext) context).close();
      }
}
```

Izlaz:

```
Student je prijavio ispit
Ime : Zara polaze ispit
Studentu je upisana ocena
```

Zadatak 2: Modifikovati prethodni zadatak tako da se koristi anotacija za kreiranje tačaka prekida i definisanja saveta.

Rešenje: Klasa Student je identična klasi iz prethodnog zadatka kao i fajl pom.xml. U klasu Logovanje ćemo uvesti potrebnu anotaciju:

```
package rs.ac.singidunum.fir.pj;

import org.aspectj.lang.annotation.Aspect;
import org.aspectj.lang.annotation.Before;
import org.aspectj.lang.annotation.Pointcut;

@Aspect
public class Logovanje {

	/**
	 * Definisemo da tacka preseka selektuje sve metode naseg pa-
keta
	 * Saveti ce biti pozivani za sve metode
	 */
	@Pointcut(“execution(* rs.ac.singidunum.fir.pj.*.*(..))”)
	private void odaberiSve() {
	}


	// Metod koji se izvrsava pre odabrane metode
	@Before(“odaberiSve()”)
	public void beforeSavet() {
		System.out.println(“Krecem sa podesavanja studentskog
profila.”);
	}
}
```

U beans.xml fajlu nema više definicije aspekta:

```
<?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:aop=”http://www.springframework.org/schema/aop”
	xmlns:context=”http://www.springframework.org/schema/context”
	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-4.3.xsd
		http://www.springframework.org/schema/aop http://
www.springframework.org/schema/aop/spring-aop-4.3.xsd”>
```

```
      <aop:aspectj-autoproxy/>

   <bean id = "student" class = "rs.ac.singidunum.fir.pj.Student">
      <property name = "name"  value = "Zara" />
      <property name = "age"  value = "11"/>
   </bean>

  <bean id = "logovanje" class = "rs.ac.singidunum.fir.pj.Logovanje"/>

</beans>
```

Glavna aplikacija:

```
package rs.ac.singidunum.fir.pj;

import org.springframework.context.ApplicationContext;
import    org.springframework.context.support.ClassPathXmlApplica-
tionContext;
import   org.springframework.context.support.FileSystemXmlApplica-
tionContext;

public class App {
       public static void main(String[] args) {
              ApplicationContext context = new FileSystemXmlApplic
ationContext("beans.xml");

              Student student = (Student) context.getBean("student");

              student.getName();
              student.getAge();
              ((FileSystemXmlApplicationContext) context).close();
       }
}
```

Ispravan izlaz:

```
Krecem sa podesavanja studentskog profila.
Ime : Zara
Krecem sa podesavanja studentskog profila.
Uzrast : 11
```

Zadatak 3: Napisati Spring projekat upotrebom Maven-a koji sadrži klasu Profesor i Fakultet. Fakultet ima listu predmeta koje je je potrebno dodeliti nekom profesoru, a Profesor ima polje Fakultet i listu predmeta koje predaje. Potrebno je implementirati metodu dodeliPredmet u klasi Profesor. Ova metoda je tačka prekida gde se pre poziva savet klase BezbednostDodele, metoda preDodele koja ispisuje da proverava da li se dati predmet nalazi u spisku. Posle tačke prekida se poziva metoda posleDodele koja ispisuje poruku da je predmet dodeljen i uklonjen sa spiska.

Rešenje: Fajl pom.xml je isti kao i u prethodnom zadatku. Klase Fakultet, Profesor i BezbednostDodele su implementirane kao standardne Java klase:

```
package rs.ac.singidunum.fir.pj;

import java.util.List;
import org.springframework.stereotype.Component;

public class Fakultet {
	private List<String> predmeti;
	public List<String> getPredmeti() {
		return predmeti;
	}
	public void setPredmeti(List<String> predmeti) {
		this.predmeti = predmeti;
	}
	public String toString() {
		StringBuilder sb = new StringBuilder();
		sb.append("Nedodeljeni predmeti fakulteta: \n");
		for (String predmet : predmeti) {
			sb.append(predmet);
			sb.append("\n");
		}
		return sb.toString();
	}
}
```

```
package rs.ac.singidunum.fir.pj;

import java.util.ArrayList;
import java.util.List;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Component;
```

```
public class Profesor {
	private Fakultet fakultet;
	private List<String> predaje;

	public Profesor(Fakultet fakultet, List<String> predaje) {
		this.fakultet = fakultet;
		this.predaje = predaje;
	}
	public void dodajPredmet(String predmet) {
		predaje.add(predmet);
		fakultet.getPredmeti().remove(predmet);
		System.out.println(this);
		System.out.println(fakultet.toString());
	}
	public String toString() {
		StringBuilder sb = new StringBuilder();
		sb.append("Profesor predaje:\n");
		for (String predmet : predaje) {
			sb.append(predmet);
			sb.append("\n");
		}
		return sb.toString();
	}
}
```

```
package rs.ac.singidunum.fir.pj;

public class BezbednostDodele {
	public void preDodele() {
		System.out.println("\nProvera da li predmet postoji u
listi\n");
	}
	public void posleDodele() {
		System.out.println("\nPredmet je dodeljen profesoru i
uklonjen iz liste predmeta fakulteta\n");
	}
}
```

Kompletan beans.xml fajl je automatski generisan nakon promena koje smo izvršili pomoću editora:

```
<?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:context="http://www.springframework.org/schema/context"
       xmlns:aop="http://www.springframework.org/schema/aop"
       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-4.3.xsd
             http://www.springframework.org/schema/aop http://
www.springframework.org/schema/aop/spring-aop-4.3.xsd">


       <context:component-scan base-package="rs.ac.singidunum.fir.
pj"></context:component-scan>
       <aop:config>
              <aop:pointcut
                     expression="execution (void rs.ac.singidunum.
fir.pj.Profesor.dodajPredmet(String))"
                     id="dodajpredmet" />
              <aop:aspect id="rasporedaspekt" ref="raspored">

              <aop:before method="preDodele" pointcut-
ref="dodajpredmet" />
              <aop:after method="posleDodele" pointcut-
ref="dodajpredmet" />
              </aop:aspect>
       </aop:config>

       <bean id="fakultet" class="rs.ac.singidunum.fir.pj.Fakultet">
              <property name="predmeti">
                     <list>
                     <value>Programski jezici</value>
                     <value>Programiranje</value>
                     <value>Baze podataka</value>
                     </list>
              </property>
       </bean>
       <bean id="profesor" class="rs.ac.singidunum.fir.pj.Profesor">
              <constructor-arg ref="fakultet"></constructor-arg>
              <constructor-arg>
                     <list>
                     </list>
              </constructor-arg>
       </bean>
```

```
      <bean id=”raspored”
            class=”rs.ac.singidunum.fir.pj.BezbednostDodele”>
      </bean>
</beans>
```

Glavna klasa za testiranje:

```
package rs.ac.singidunum.fir.pj;

import org.springframework.context.ApplicationContext;
import   org.springframework.context.support.FileSystemXmlApplica-
tionContext;

public class App {

      public static void main(String[] args) {
            ApplicationContext context = new FileSystemXmlApplic
ationContext(“beans.xml”);
            Profesor prof = (Profesor) context.getBean(“profesor”);

            prof.dodajPredmet(“Programski jezici”);
            prof.dodajPredmet(“Baze podataka”);
            ((FileSystemXmlApplicationContext) context).close();

      }
}
```

Izlaz ovog programa je sledeći:

```
Provera da li predmet postoji u listi

Profesor predaje:
Programski jezici

Nedodeljeni predmeti fakulteta:
Programiranje
Baze podataka


Predmet je dodeljen profesoru i uklonjen iz liste predmeta fakulteta


Provera da li predmet postoji u listi
```

```
Profesor predaje:
Programski jezici
Baze podataka

Nedodeljeni predmeti fakulteta:
Programiranje


Predmet je dodeljen profesoru i uklonjen iz liste predmeta fakulteta
```

Za vežbu se svi zadaci mogu uraditi korišćenjem anotacija i around saveta umesto before i after.

Pitanja

1. Šta je aspekt (u smislu AOP)?
2. Koje su prednosti AOP u odnosu na OOP?
3. Koji su osnovni termini u AOP i koje je njihovo značenje?
4. Sa kojim savetima aspekti mogu da rade u Springu?
5. Na koji način se mogu definisati tačke preseka?
6. Šta je utkivanje (engl. *weaving*) u AOP?
7. U kojim tačkama životnog ciklusa ciljnog objekta se može izvršiti utkivanje?
8. Da li Springov okvir za AOP podržava presretanje:
   - » poziva metoda,
   - » poziva konstruktora,
   - » promene vrednsoti polja klase?
9. Kada se vrši savetovanje objekata u Springu (za vreme kompajliranja, izvršavanja, …)?
10. Koja oznaka služi za proveru podudaranja u tačkama preseka u Spring AOP okviru?
11. Čemu služi element <aop:around>?
12. Koja je prednost korišćenja elementa <aop:around> u odnosu na korśćenje poejdinačnih <aop:before> i <aop:after> elemenata?
13. Kako se mogu definisati aspekti i tačke preseka pomoću anotacija?
14. Navesti neke anotacije koje se korise konfiguracije saveta i tačke preseka.
15. Čemu služe džokeri u Spring AOP okviru?

VIII
Osnove rada sa bazom podataka u Spring okruženju

Nakon upoznavanja sa osnovnim funkcionalnostima Spring okruženja, sledeći korak ka izgradnji stvarnih aplikacija u Springu jeste perzistencija podataka. Skoro svaka ozbiljna aplikacija koristi bazu podataka za perzistenciju. Verovatno je i određeni broj čitalaca do sada već imao kontakata sa bazom podataka. U praksi, rad sa bazom podataka sadrži veći broj koraka koji se moraju ispravno odraditi – inicijalizacija, otvaranje konekcija, rukovanje većim brojem izuzetaka, zatvaranje konekcija i slično. Ukoliko se bilo koji od ovih koraka uradi pogrešno, može doći do korupcije podataka ili u najgorem slučaju potpunog gubitka podataka, što je potencijalna katastrofa za korisnika softvera.

Iz prethodnih poglavlja se može uočiti da Spring teži pojednostavljivanju implementacije softvera i redukciji količine potrebnog koda. Slična stvar važi i za Springovu podršku za perzistenciju podataka. Spring ima bogatu podršku i mogućnosti integracije sa velikim brojem različitih tehnologija za pristup podataka. Spring se može koristiti i za olakšavanje direktnog pristupa bazi pomoću JDBC sa jedne strane, ili za integraciju sa nekim ORM (engl. *Object Relational Mapping*) okvirom poput Hibernate ili JPA (engl. *Java Persistence API*) sa druge strane. U oba slučaja, glavna prednost upotrebe Spring okruženja je u tome što on zaklanja sam pristup podacima, pošto Spring zapravo rukuje pristupom podacima u bazi, a programer može da se posveti rukovanju podacima u okviru aplikacije. Spring u najnovijim verzijama nudi i podršku za novu vrstu baza podataka – NoSQL baze, koje su veoma popularne u poslednje vreme.

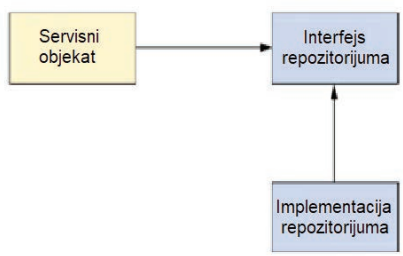
U ovom poglavlju akcenat će biti na tome kako Spring daje podršku za JDBC, koji predstavlja osnovni i direktni pristup perzistenciji u Java aplikacijama.

8.1. Filozofija pristupa podacima u Springu

Većina aplikacija mora da koristi bazu iz koje čita i u koju upisuje podatke, a Spring aplikacije nisu izuzetak. U tradicionalnom programiranju, logika za perzistenciju podataka je često raštrkana svugde po komponentama aplikacije. Što je još gore, ta logika je često vrlo usko povezana sa odabranom tehnologijom perzistencije,

čime se stvara čvrsta sprega između komponenti aplikacije i tehnologije perzistencije. Ukoliko bi, na primer, bilo potrebno u sred razvoja projekta promeniti tehnologiju perzistencije, bilo bi neophodno izmeniti veliki broj klasa, odnosno praktično sve klase koje koriste perzistenciju, i zameniti kod za pristup jednoj tehnologiji kodom za pristup drugoj tehnologiji. Ovo je izuzetno obiman i problematičan zadatak, gde bi često na kraju ispalo da je lakše napisati aplikaciju iz početka, nego prepravljati već postojeći kod da podrži novu tehnologiju. Zbog toga se često u projektima nije ni dozvoljavala promena tehnologije perzistencije kada jednom počne razvoj.

Spring napada ovaj problem iz više uglova. Kako bi se sprečilo da logika za perzistenciju bude raštrkana po celoj aplikaciji, favorizuje se pristup faktorizacije pristupa podacima u jednu ili više komponenti kojima je to glavni zadatak. Ove komponente su poznate pod nazivom DAO (engl. *Data Access Object*) ili repozitorijum. Kako bi se sprečilo da aplikacija bude u čvrstoj sprezi sa određenom tehnologijom perzistencije, ispravno napisani repozitorijumi izlažu svoju funkcionalnost kroz interfejse, kao na slici 8.1.



Slika 8.1. Delegiranje pristupa podacima repozitorijumu

Servisni objekti dakle ne implementiraju pristup podacima, već delegiraju taj posao repozitorijumima. Repozitorijumi su preko interfejsa u labavoj sprezi sa servisnim objektom. Drugim rečima, servisni objekat pristupa repozitorijumu kroz interfejs. To u praksi ima nekoliko praktičnih posledica. Prvo, testiranje servisnih objekata je značajno olakšano, pošto se mogu kreirati *mock* implementacije interfejsa repozitorijuma. Na taj način se testiranje servisnih objekata može izvršiti i bez stvarne konekcije na bazu, čime se ubrzava razvoj jediničnih testova. Drugo, i mnogo bitnije, jeste da se na ovaj način pristupa podacima na način koji ne zavisi od izabrane tehnologije. Konkretna tehnologija pristupa podacima je kompletno izolovana u repozitorijumu, a samo relevantne metode za pristup podacima se izlažu kroz interfejs. To praktično znači da je aplikacija fleksibilna, i da se tehnologija pristupa podacima može zameniti drugom sa minimalnim posledicama po ostatak aplikacije – aplikacija je u labavoj sprezi sa odabranom tehnologijom pristupa. Nasuprot tome, u slučaju da implementacioni detalji tehnologije pristupa procure u druge delove aplikacije, postupak zamene tehnologije pristupa bi bio drastično teži – u ovom slučaju se radi o čvrstoj sprezi aplikacije i tehnologije pristupa.

Potrebno je napomenuti da interfejsi nisu neophodni – oni olakšavaju pisanje koda sa labavom spregom. Kompletna implementacija sloja perzistencije se sakriva iza interfejsa, što je i pristup koji Spring ohrabruje. Međutim, Spring ne zahteva upotrebu interfejsa – moguće je ožičiti zrno repozitorijuma direktno u polje nekog drugog zrna, bez interfejsa između njih.

8.2. Obrada SQL izuzetaka u Springu

Svako ko je radio sa bazama podataka u Javi i ko je pisao JDBC kod zna za noćnu moru pod nazivom SQLException. U pitanju je proveravani izuzetak, odnosno programer je primoran da uhvati i obradi ovakve tipove grešaka. Problem leži u činjenici da SQLException bukvalno samo znači da je došlo do greške u pristupu bazi podataka. Izuzetak nosi vrlo malo informacija o tome šta je pošlo naopako i teško je doći do tih informacija (ko želi da kopa kroz statusne kodove grešaka), a praktično nema nikakvih naznaka šta raditi u tom slučaju i kako rešiti problem.

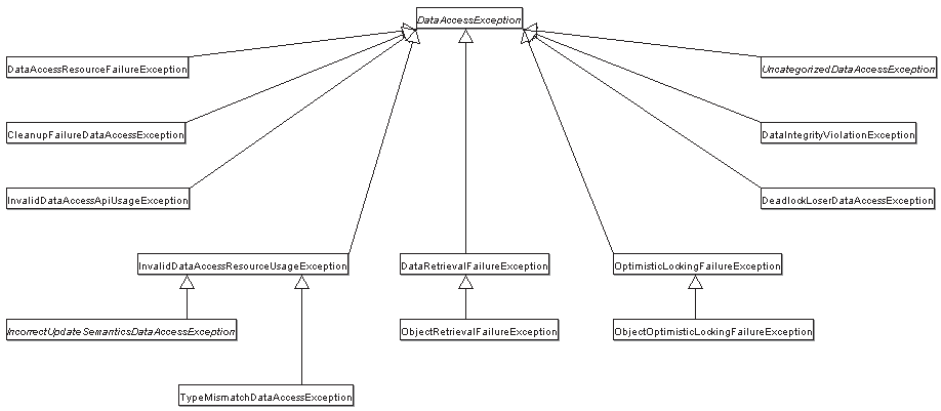
SQLException se izbacuje u širokoj lepezi slučajeva, od kojih su možda najčešći:

- Nije moguće uspešno se povezati sa bazom.
- Upit koji je potrebno izvršiti ima sintaksne greške.
- Tabela (ili kolona) koja se gađa iz upita ne postoji u bazi.
- Narušavanje primarnog ključa prilikom insert operacije.

SQLException se tretira kao jedan izuzetak za sve moguće probleme sa pristupom bazi – umesto da postoje različiti tipovi izuzetka za svaki tipičan problem, SQLException se izbacuje u svim mogućim slučajevima. Čak i najiskusniji programeri najčešće ne znaju kako da rukuju izuzetkom ovog tipa kada se uhvati u catch grani. Većina problema koji okidaju izbacivanje izuzetka tipa SQLException se ne može rešiti unutar catch bloka, jer su indikacija fatalne greške od koje se nije moguće oporaviti. Na primer, ukoliko aplikacija ne može da se poveže sa bazom, najverovatnije ni neće moći da nastavi sa radom. Slično, ukoliko postoji sintaksna greška u upitu, praktično ništa se ne može uraditi u vreme izvršavanja programa. Ako u većini slučajeva nije moguće oporaviti se od SQLException izuzetka, zašto smo onda naterani da ga uhvatimo?

Neki okviri za perzistenciju nude bogatiji izbor izuzetaka. Hibernate npr. nudi dvadesetak različitih tipova grešaka, odnosno svaki specifični problem sa bazom ima svoj odgovarajući izuzetak. Time je rad sa catch granama olakšan, pošto je moguće hvatati samo izuzetke kojima želimo da rukujemo. Problem sa izuzecima u Hibernate okruženju jeste da su oni specifični samo za Hibernate. Kao što smo pominjali ranije, cilj je izolovati detalje o mehanizmu perzistencije, kako bi se sprečila čvrsta sprega aplikacije sa tehnologijom perzistencije. Ukoliko bismo radili sa Hibernate izuzecima u kodu, to bi značilo da su detalji Hibernate okvira procureli u ostatak aplikacije, što želimo da sprečimo.

Spring JDBC pruža hijerarhiju izuzetaka koja rešava većinu problema – u poređenju sa JDBC, Spring sadrži veći broj izuzetaka koji su vezani za specifičan problem. Springovi izuzeci nisu povezani ni sa jednom određenom tehnologijom pristupa podacima. To zapravo znači da će Spring uvek izbacivati konzistentan skup izuzetaka, nezavisno od odabrane tehnologije čuvanja podataka.



Slika 8.2. Hijerarhija DataAccessException izuzetaka

Podskup izuzetaka koje Spring izbacuje prikazan je na slici 8.2 (lista nije potpuna). Spring ima poseban izuzetak za bilo koji slučaj do koga može doći u radu sa bazom podataka. Na ovom mestu potrebno je uočiti dve stvari. Prvo, Spring će uvek izbacivati ovaj isti skup izuzetaka, nevezano za izbor tehnologije perzistencije. Time se obezbeđuje labava sprega aplikacije sa bazom podataka, odnosno omogućava se laka zamena jedne tehnologije drugom.

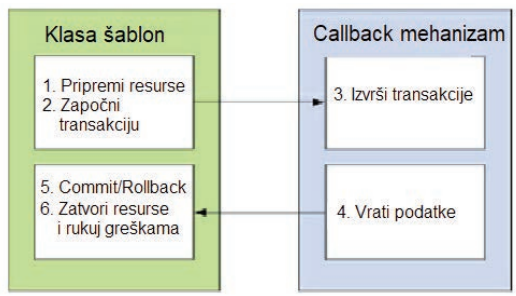
Drugo, što možda nije očigledno na prvi pogled, je da su svi izuzeci izvedeni iz korenog izuzetka DataAccessException. DataAccessException je neproveravani izuzetak, odnosno nije neophodno pisati catch grane za bilo koji izuzetak koji Spring izbacuje. Time programer više nije primoran da piše try-catch blokove za rad sa bazom podataka (od kojih većina catch grana ostane prazna), već može po želji hvatati samo određene tipove izuzetaka koje želi da obradi.

8.3. Šabloni za pristup podacima

Spring koristi *template method* šablon prilikom pristupa podacima. U ovom slučaju, šablon se sastoji od fiksnog skeleta procesa, i promenljivih delova ovog procesa. Nevezano za tehnologiju koja se koristi za čuvanje podataka, određeni koraci su uvek neophodni, poput dohvatanja konekcije ka bazi na početku, i oslobađanja resursa na kraju procesa. To su fiksni delovi procesa pristupa podacima. Sa druge strane,

metode koje se pišu za dohvatanje podataka nisu fiksne. Može se praviti upit za različite podatke, ili ažurirati podaci u bazi na više načina. Ovo su promenljivi delovi procesa pristupa podacima.

Spring nudi klase šablone koji upravljaju fiksnim delom procesa pristupa podacima. Ove klase kontrolišu transakcije, upravljaju resursima i rukuju izuzecima. Od programera se očekuje da implementira promenljivi deo procesa – kreiranje upita, povezivanje parametara, i manipulaciju rezultujućim skupom podataka. Ovaj mehanizam je implementiran kroz *callback* strukturu, kao što je prikazano na slici 8.3.



Slika 8.3. Klasa šablon i callback mehanizam

Spring nudi nekoliko šablona za rad sa bazom. Izbor zavisi od odabrane tehnologije za perzistenciju podataka. Na primer, ukoliko se koristi čist JDBC, može se koristiti JdbcTemplate ili NamedParameterJdbcTemplate. Sa druge strane, ukoliko se koristi JPA ili neki ORM okvir, poput Hibernate, može se odarati HibernateTemplate ili JpaTemplate. U tabeli 8-1 dat je pregled dostupnih šablona uz kratak opis kada se koriste.

Tabela 8.1. Dostupni šabloni u Spring okruženju

Šablon	Za šta se koristi
jca.cci.core.CciTemplate	JCA CCI konekcije
jdbc.core.JdbcTemplate	JDBC konekcije
jdbc.core.namedparam.NamedParameter-JdbcTemplate	JDBC konekcije sa podrškom za imenovane parametre
jdbc.core.simple.SimpleJdbcTemplate	JDBC konekcije (od Spring 3.1. se više ne koristi)
orm.hibernate3.HibernateTemplate	Hibernate 3.x+ sesije
orm.ibatis.SqlMapClientTemplate	iBATIS SqlMap klijenti
orm.jdo.JdoTemplate	Java Data Object implementacije
orm.jpa.JpaTemplate	Java Persistence API entity managers

Može se primetiti da Spring pruža podršku za sve bitne tehnologije za perzistenciju podataka. Najčešće se koriste osnovni JDBC pristup bazi (koji će i biti obrađen u okviru ovog udžbenika) i dva ORM pristupa, Hibernate i JPA (koji prevazilaze okvir ovog udžbenika, pa se u ovom slučaju čitalac upućuje na dodatnu literaturu).

8.4. Implementacija pristupa bazi podataka

Ostvarivanje konekcije sa bazom podataka biće objašnjeno na konkretnom primeru. Baza sa kojom se radi je jednostavna MySQL baza sa samo jednom tabelom, koja je definisana na sledeći način.

```
CREATE DATABASE  IF NOT EXISTS `springtutorial`
USE `springtutorial`;

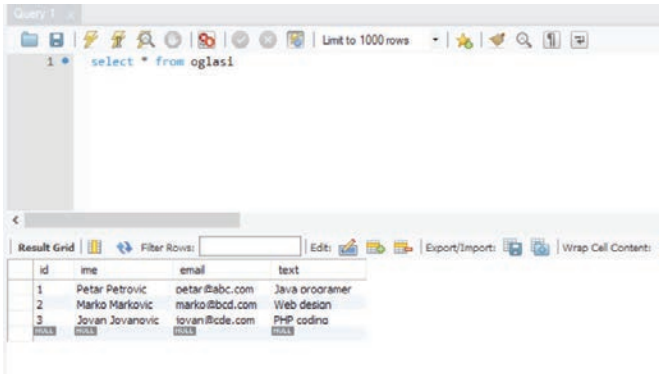
DROP TABLE IF EXISTS `oglasi`;

CREATE TABLE `oglasi` (
  `id` int(11) NOT NULL AUTO_INCREMENT,
  `ime` varchar(100) NOT NULL,
  `email` varchar(60) NOT NULL,
  `text` text NOT NULL,
  PRIMARY KEY (`id`)
)
```

Tabela *oglasi* se nalazi u bazi pod nazivom *springtutorial*. U tabeli se čuvaju oglasi za posao, gde korisnici ostavljaju svoje podatke (ime i email adresu), a u polju text unose opis svojih veština. Osim pomenute tri kolone, postoji i kolona id, koja je primarni ključ i predstavlja jedinstveni id koji jednoznačno određuje svaki pojedinačni red u tabeli. Kako je tabela inicijalno prazna, malom skriptom su uneta tri reda, na sledeći način:

```
INSERT INTO `oglasi` VALUES
(1,'Petar Petrovic','petar@abc.com','Java programer'),
(2,'Marko Markovic','marko@bcd.com','Web design'),
(3,'Jovan Jovanovic','jovan@cde.com','PHP coding');
```

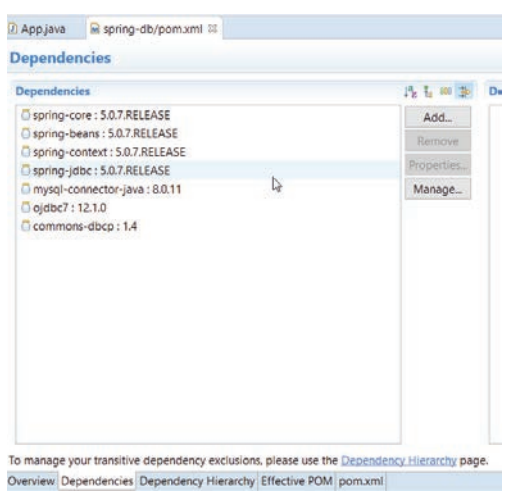
Nakon izvršenih naredbi, u okviru MySQL Workbench okruženja se može videti pregled tabele, kao na slici 8.4.



Slika 8.4. Pregled jedne tabele u bazi u okviru alata MySQL Workbench

U svrhu demonstracije, napravljena je identična baza u Oracle okruženju, kako bi bilo pokazano kako se može zameniti tehnologija perzistencije ukoliko je to potrebno. Sama procedura je identična, pošto je i kompletna ideja Spring okvira da se tehnologija perzistencije podataka sakrije od programera. Spring će u pozadini koristiti odgovarajuće instrukcije u radu sa konkretnom implementacijom baze. Gledano sa strane programera, potpuno je svejedno koja je konkretno tehnologija u pozadini, pošto je unutar aplikacije potrebno samo implementirati logiku. Jedina stvar koja se razlikuje od tehnologije do tehnologije i koju je pri promeni potrebno izmeniti je konektor.

Spring projekat u Eclipse okruženju je podešen na uobičajen način, sa dodatkom nekoliko novih zavisnosti u pom.xml fajlu. Neophodno je uključiti odgovarajuće konektore ka bazi (konektori za MySQL, Oracle i DBCP). Na slici 8.5 prikazane su sve potrebne zavisnosti u grafičkom editoru pom.xml fajla.



Slika 8.5. Potrebne zavisnosti za konektore ka MySQL i Oracle bazama podataka

Nakon unetih dodatnih zavisnosti, izvorni kod 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>ac.rs.singidunum.fir.pj</groupId>
       <artifactId>spring-db</artifactId>
       <version>0.0.1-SNAPSHOT</version>
       <dependencies>
              <dependency>
                     <groupId>org.springframework</groupId>
                     <artifactId>spring-core</artifactId>
                     <version>5.0.7.RELEASE</version>
              </dependency>
              <dependency>
                     <groupId>org.springframework</groupId>
                     <artifactId>spring-beans</artifactId>
                     <version>5.0.7.RELEASE</version>
              </dependency>
              <dependency>
                     <groupId>org.springframework</groupId>
                     <artifactId>spring-context</artifactId>
                     <version>5.0.7.RELEASE</version>
              </dependency>
              <dependency>
                     <groupId>org.springframework</groupId>
                     <artifactId>spring-jdbc</artifactId>
                     <version>5.0.7.RELEASE</version>
              </dependency>
              <dependency>
                     <groupId>mysql</groupId>
                     <artifactId>mysql-connector-java</artifactId>
                     <version>8.0.11</version>
              </dependency>
              <dependency>
                     <groupId>commons-dbcp</groupId>
                     <artifactId>commons-dbcp</artifactId>
                     <version>1.4</version>
              </dependency>
              <dependency>
                     <groupId>com.oracle</groupId>
                     <artifactId>ojdbc7</artifactId>
                     <version>12.1.0</version>
```

```
            </dependency>
      </dependencies>
</project>
```

Osnovna stvar koju je potrebno implementirati u Spring aplikaciji koja radi sa bazom podataka je DAO (engl. *Data Access Object*). To je objekat u kome se definiše logika rada sa bazom i manipulacija objektima iz baze. Kako bismo mogli da na ispravan način manipulišemo podacima, neophodno je napraviti model za podatke koji se nalaze u bazi podataka. Model je Java klasa koja sadrži sva polja koja se nalaze u jednom redu u tabeli u bazi, uz odgovarajuće konstruktore, get i set metode i slično. U našem slučaju, model se naziva Oglas, i predstavlja jedan oglas u tabeli *oglasi*, odnosno jedan red u toj tabeli. Model mora da sadrži sva neohpodna polja kako bi se u njega mogao smestiti jedan red iz tabele.

```
package ac.rs.singidunum.fir.pj;

public class Oglas {
      //polja odgovaraju jednom redu u tabeli oglasi
      private int id;
      private String ime;
      private String email;
      private String text;

      public Oglas () {

      }
      public Oglas(int id, String ime, String email, String text)
{
            this.id = id;
            this.ime = ime;
            this.email = email;
            this.text = text;
      }
      public int getId() {
            return id;
      }
      public void setId(int id) {
            this.id = id;
      }
      public String getIme() {
            return ime;
      }
```

```
	public void setIme(String ime) {
		this.ime = ime;
	}
	public String getEmail() {
		return email;
	}
	public void setEmail(String email) {
		this.email = email;
	}
	public String getText() {
		return text;
	}
	public void setText(String text) {
		this.text = text;
	}
	@Override
	public String toString() {
		return "Offer [id=" + id + ", ime=" + ime + ", email="
+ email + ", text=" + text + "]";
	}

}
```

Objekti klase Oglas služe kao nosioci objekata koje je potrebno sačuvati u bazi podataka, kao i objekata koji se dohvataju iz baze podataka. Sada kada postoji model podataka sa kojim se radi, može se preći na implementaciju same klase DAO.

Za implementaciju klase DAO, potrebno je prvo definisati izvor podataka – engl. *DataSource*. Postoji nekoliko načina za definisanje zrna koje predstavlja izvor podataka unutar Spring aplikacije:

- Izvori podataka definisani pomoću JDBC drajvera
- Izvori podataka koji se dobijaju pomoću JNDI pretrage i otrivanja izvora (eng. *Java Naming and Directory Interface*)
- Izvori podataka koji dobavljaju konekcije iz pula konekcija (engl. *connection pool*).

U ovoj knjizi biće prikazan način za konfiguraciju zrna izvora podataka pomoću JDBC drajvera. Zrna ovog tipa se kreiraju po šablonu – moraju se kreirati i popuniti odgovarajuća polja, koja zavise od tehnologije perzistencije podataka (odnosno razlikuju se podešavanja za MySQL i Oracle bazu). Konkretno, u slučaju konfigurisanja MySQL baze podataka, neophodno je na standardan način kreirati zrno sa Id *dataSource*, koje će imati sledeći oblik:

```
<!-- MySQL Data Source -->
       <bean id="dataSource" destroy-method="close" class="org.
apache.commons.dbcp.BasicDataSource">
            <property name="driverClassName" value="com.mysql.
jdbc.Driver">
      </property>
            <property name="url" value="jdbc:mys
ql://127.0.0.1:3306/springtutorial">
            </property>
      <property name="username" value="miodrag">
      </property>
      <property name="password" value="root">
      </property>
      </bean>
```

Ovo zrno se pravi šablonski. MySQL konekcija zahteva postojanje tačno određenih polja, sa tačno određenim vrednostima:

- *driverClassName* – označava JDBC drajver koji se koristi, u slučaju MySQL treba da ima vrednost `com.mysql.jdbc.Driver`
- *url* – url adresa na kojoj se nalazi baza. 127.0.0.1:3306 označava localhost i port 3306 na kome funkcioniše MySQL, a zatim se navodi i ime baze podataka (u ovom primeru *springtutorial*).
- *username* i *password* – parametri za pristup bazi, naravno zavise od korisničkog imena i lozinke koji su postavljeni na MySQL serveru.

Sada se može napraviti DAO klasa, u koju će biti injektovano zrno tipa DataSource (zrno koje opisuje konekciju sa MySQL bazom). Klasa OglasiDAO će biti anotirana sa @Component, pošto će se koristiti automatsko skeniranje i ožičavanje. Osnova klase OglasiDAO, sa ožičavanjem zrna tipa DataSource, data je u nastavku.

```
package ac.rs.singidunum.fir.pj;

import javax.sql.DataSource;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.jdbc.core.JdbcTemplate;
import org.springframework.stereotype.Component;

@Component("oglasiDao")
public class OglasiDAO {

      private JdbcTemplate jdbc;
```

```
        @Autowired
        public void setDataSource(DataSource jdbc){
                this.jdbc = new JdbcTemplate(jdbc);
        }


}
```

Zrno tipa DataSource se ožičava kroz set metodu klase OglasiDAO, i koristi se za inicijalizaciju šablona koji se koristi – u ovom slučaju upotrebljen je JdbcTemplate. U DAO klasi je potrebno implementirati osnovne CRUD (engl. *Create Read Update Delete*) operacije nad tabelom u bazi podataka. Najosnovnija operacija je čitanje svih zapisa iz tabele oglasi. RowMapper objekat služi da izvuče vrednosti iz rezultujućeg seta podataka i kreira domenski objekat (u ovom slučaju Oglas). Za svaki red koji je vraćen kao rezultat upita, JdbcTemplate poziva mapRow() metodu, kojoj prosleđuje sam vraćeni skup podataka ResultSet, kao i integer vrednost koja označava broj reda. Unutar metode mapRow() implementira se kreiranje objekta tipa Oglas i popunjava odgovarajućim vrednostima iz odgovarajućeg reda iz ResultSet. Opisana implementacija je data u metodi getOglasi().

```
package ac.rs.singidunum.fir.pj;

import java.sql.ResultSet;
import java.sql.SQLException;
import java.util.List;

import javax.sql.DataSource;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.jdbc.core.JdbcTemplate;
import org.springframework.jdbc.core.RowMapper;
import org.springframework.stereotype.Component;

@Component(“oglasiDao”)
public class OglasiDAO {

        private JdbcTemplate jdbc;

        @Autowired
        public void setDataSource(DataSource jdbc){
                this.jdbc = new JdbcTemplate(jdbc);
        }
```

```
	//dohvati sve oglase iz baze
	public List<Oglas> getOglasi () {
		return jdbc.query("SELECT * FROM oglasi", new
RowMapper<Oglas>(){
			public Oglas mapRow(ResultSet rs, int rowNum)
throws SQLException {
				Oglas oglas = new Oglas();
				oglas.setId(rs.getInt("id"));
				oglas.setIme(rs.getString("ime"));
				oglas.setEmail(rs.getString("email"));
				oglas.setText(rs.getString("text"));

				return oglas;
			}
		});
	}
}
```

Kako bismo demonstrirali rad klase OglasiDAO, neka je dat glavni program u klasi App.java, gde se dohvata zrno *oglasiDao*, poziva metoda getOglasi(), koja vraća listu Oglasa. Na kraju, samo se prolazi kroz listu i štampaju svi dovučeni podaci iz baze.

```
package ac.rs.singidunum.fir.pj;

import java.util.ArrayList;
import java.util.List;

import org.springframework.context.ApplicationContext;
import   org.springframework.context.support.FileSystemXmlApplica-
tionContext;
import org.springframework.dao.DataAccessException;
import org.springframework.dao.DuplicateKeyException;
import org.springframework.jdbc.CannotGetJdbcConnectionException;

public class App {

	public static void main(String[] args) {
		ApplicationContext context = new FileSystemXmlApplic
ationContext("beans.xml");

		OglasiDAO   oglasiDAO   =   (OglasiDAO)   context.
getBean("oglasiDao");
```

```
            List<Oglas> oglasi = oglasiDAO.getOglasi();

            for (Oglas oglas : oglasi) {
                  System.out.println(oglas);
            }
            ((FileSystemXmlApplicationContext) context).close();
      }

}
```

Rezultat izvršavanja programa dat je u nastavku, gde se jasno vidi da su sva tri reda koja su inicijalno ubačena u bazu dovučena u aplikaciju i prikazana na standardnom izlazu.

```
Feb 04, 2019 8:49:12 PM org.springframework.context.support.Ab-
stractApplicationContext prepareRefresh
INFO: Refreshing org.springframework.context.support.FileSystemXml
ApplicationContext@4bf558aa: startup date [Mon Feb 04 20:49:12 CET
2019]; root of context hierarchy
Feb 04, 2019 8:49:12 PM org.springframework.beans.factory.xml.Xml-
BeanDefinitionReader loadBeanDefinitions
INFO: Loading XML bean definitions from file [C:\Engineering Software
Lab\Spring DB\beans.xml]
Loading class `com.mysql.jdbc.Driver'. This is deprecated. The new
driver class is `com.mysql.cj.jdbc.Driver'. The driver is automati-
cally registered via the SPI and manual loading of the driver class
is generally unnecessary.
Offer [id=1, ime=Petar Petrovic, email=petar@abc.com, text=Java
programer]
Offer [id=2, ime=Marko Markovic, email=marko@bcd.com, text=Web de-
sign]
Offer [id=3, ime=Jovan Jovanovic, email=jovan@cde.com, text=PHP
coding]
Feb 04, 2019 8:49:14 PM org.springframework.context.support.Ab-
stractApplicationContext doClose
INFO: Closing org.springframework.context.support.FileSystemXmlAp
plicationContext@4bf558aa: startup date [Mon Feb 04 20:49:12 CET
2019]; root of context hierarchy
```

Kao što je ranije navedeno, Spring rukuje svim izuzecima koje konkretna baza izbacuje, i ponovo ih izbacuje kao neproveravane izuzetke (koje programer ne mora obavezno da uhvati). Međutim, ukoliko programer želi, može da uhvati izuzetak i pruži dodatno rukovanje izuzetkom u kodu. Na primer, ukoliko u našem primeru iz bilo kog razloga nije moguće ostvariti konekciju ka bazi, u App.java klasi bi moglo da se doda sledeće:

```
package ac.rs.singidunum.fir.pj;

import java.util.ArrayList;
import java.util.List;

import org.springframework.context.ApplicationContext;
import org.springframework.context.support.FileSystemXmlApplica-
tionContext;
import org.springframework.dao.DataAccessException;
import org.springframework.dao.DuplicateKeyException;
import org.springframework.jdbc.CannotGetJdbcConnectionException;

public class App {

	public static void main(String[] args) {
		ApplicationContext context = new FileSystemXmlApplic
ationContext("beans.xml");

		OglasiDAO oglasiDAO = (OglasiDAO) context.
getBean("oglasiDao");
		try {
			List<Oglas> oglasi = oglasiDAO.getOglasi();

			for (Oglas oglas : oglasi) {
				System.out.println(oglas);
			}
		} catch (CannotGetJdbcConnectionException e) {
			System.out.println("Nije moguce ostvariti
konekciju ka MySQL bazi...");
		} catch (DataAccessException e) {
			System.out.println(e.getMessage());
			System.out.println(e.getClass());
		}
		((FileSystemXmlApplicationContext) context).close();
	}

}
```

Ukoliko se konekcija ka bazi ne ostvari uspešno iz bilo kog razloga, greška se hvata i dobija se sledeći rezultat.

```
Feb 04, 2019 9:33:50 PM org.springframework.context.support.Ab-
stractApplicationContext prepareRefresh
INFO: Refreshing org.springframework.context.support.FileSystemXml
ApplicationContext@3d71d552: startup date [Mon Feb 04 21:33:50 CET
2019]; root of context hierarchy
Feb 04, 2019 9:33:50 PM org.springframework.beans.factory.xml.Xml-
BeanDefinitionReader loadBeanDefinitions
INFO: Loading XML bean definitions from file [C:\Engineering Software
Lab\Spring DB\beans.xml]
Loading class `com.mysql.jdbc.Driver'. This is deprecated. The new
driver class is `com.mysql.cj.jdbc.Driver'. The driver is automati-
cally registered via the SPI and manual loading of the driver class
is generally unnecessary.
Nije moguce ostvariti konekciju ka MySQL bazi...
Feb 04, 2019 9:33:52 PM org.springframework.context.support.Ab-
stractApplicationContext doClose
INFO: Closing org.springframework.context.support.FileSystemXmlAp
plicationContext@3d71d552: startup date [Mon Feb 04 21:33:50 CET
2019]; root of context hierarchy
```

8.5. Uvođenje imenovanih parametara

Imenovani parametri (engl. *named parameters*) se koriste kada je potrebno proslediti parametre SQL upitu. Na primer, ukoliko je potrebno dodati novu metodu u OglasiDAO koja će dohvatati jedan oglas po datom Id, potrebno je u SQL upit proslediti Id koji je želimo da dohvatimo.U tom slučaju, kako bismo parametrima dali imena radi lakšeg prosleđivanja (umesto da ih prosleđujemo po rednom broju parametra), može se koristiti NamedParameterJdbcTemplate umesto prostog JdbcTemplate. Metoda koja dohvata oglas po Id je implementirana na sličan način kao metoda getOglasi() koja vraća sve oglase iz baze. Razlika je da postoji skup parametara MapSqlParameterSource koji ograničavaju broj rezultata koje upit vraća iz baze, u ovom slučaju pretraga se vrši po Id. Svaki pojedinačni parametar za SQL upit se dodaje u obliku params.addValue("id", id). Modifikovana verzija klase OglasiDAO je data u nastavku.

```
package ac.rs.singidunum.fir.pj;

import java.sql.ResultSet;
import java.sql.SQLException;
import java.util.List;

import javax.sql.DataSource;
```

```
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.jdbc.core.RowMapper;
import org.springframework.jdbc.core.namedparam.MapSqlParameter-
Source;
import org.springframework.jdbc.core.namedparam.NamedParameter-
JdbcTemplate;
import org.springframework.stereotype.Component;

@Component("oglasiDao")
public class OglasiDAO {

	private NamedParameterJdbcTemplate jdbc;

	@Autowired
	public void setDataSource(DataSource jdbc) {
		this.jdbc = new NamedParameterJdbcTemplate(jdbc);
	}

	// dohvati sve oglase iz baze
	public List<Oglas> getOglasi() {
		return jdbc.query("SELECT * FROM oglasi", new
RowMapper<Oglas>() {

			public Oglas mapRow(ResultSet rs, int rowNum)
throws SQLException {
				Oglas oglas = new Oglas();
				oglas.setId(rs.getInt("id"));
				oglas.setIme(rs.getString("ime"));
				oglas.setEmail(rs.getString("email"));
				oglas.setText(rs.getString("text"));

				return oglas;
			}

		});
	}

	// dohvati jedan oglas po prosledjenom Id
	public Oglas getOglasById(int id) {
		MapSqlParameterSource params = new
MapSqlParameterSource();
```

```
            params.addValue("id", id);
            return jdbc.queryForObject("SELECT * FROM oglasi
WHERE id = :id", params, new RowMapper<Oglas>() {

                    public Oglas mapRow(ResultSet rs, int rowNum)
throws SQLException {
                            Oglas oglas = new Oglas();
                            oglas.setId(rs.getInt("id"));
                            oglas.setIme(rs.getString("ime"));
                            oglas.setEmail(rs.getString("email"));
                            oglas.setText(rs.getString("text"));

                            return oglas;
                    }

            });
      }

}
```

U glavnom programu se sada može dohvatiti samo jedan red iz baze, na sledeći način:

```
OglasiDAO oglasiDAO = (OglasiDAO) context.getBean("oglasiDao");
Oglas oglas = oglasiDAO.getOglasById(2);
System.out.println(oglas);
```

Za insert novog objekta u bazu, može se takođe koristiti objekat tipa MapSqlParameterSource, kao što je malopre pokazano. Međutim za unos postoji još jedna opcija, koja je više u duhu Springa – kratka i koncizna. Može se koristiti BeanPropertySqlParameterSource, kome se u konstruktoru prosleđuje objekat čija polja se koriste kao parametri u upitu. Osim metode insert, dodata je i metoda za delete, odnosno brisanje jednog reda iz baze podataka, kao i metoda update, za ažuriranje jednog podatka u bazi. Nakon unetih izmena, kompletirana klasa OglasiDAO je data sa:

```
package ac.rs.singidunum.fir.pj;

import java.sql.ResultSet;
import java.sql.SQLException;
import java.util.List;

import javax.sql.DataSource;
```

```
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.jdbc.core.RowMapper;
import org.springframework.jdbc.core.namedparam.BeanPropertySql-
ParameterSource;
import org.springframework.jdbc.core.namedparam.MapSqlParameter-
Source;
import org.springframework.jdbc.core.namedparam.NamedParameter-
JdbcTemplate;
import org.springframework.stereotype.Component;

@Component("oglasiDao")
public class OglasiDAO {

	private NamedParameterJdbcTemplate jdbc;

	@Autowired
	public void setDataSource(DataSource jdbc) {
		this.jdbc = new NamedParameterJdbcTemplate(jdbc);
	}

	// dohvati sve oglase iz baze
	public List<Oglas> getOglasi() {
		return jdbc.query("SELECT * FROM oglasi", new
RowMapper<Oglas>() {

			public Oglas mapRow(ResultSet rs, int rowNum)
throws SQLException {
				Oglas oglas = new Oglas();
				oglas.setId(rs.getInt("id"));
				oglas.setIme(rs.getString("ime"));
				oglas.setEmail(rs.getString("email"));
				oglas.setText(rs.getString("text"));

				return oglas;
			}

		});
	}

	// dohvati jedan oglas po prosledjenom Id
	public Oglas getOglasById(int id) {
		MapSqlParameterSource params = new MapSqlParameter-
Source();
		params.addValue("id", id);
```

```
            return jdbc.queryForObject("SELECT * FROM oglasi
WHERE id = :id", params, new RowMapper<Oglas>() {

                  public Oglas mapRow(ResultSet rs, int rowNum)
throws SQLException {
                        Oglas oglas = new Oglas();
                        oglas.setId(rs.getInt("id"));
                        oglas.setIme(rs.getString("ime"));
                        oglas.setEmail(rs.getString("email"));
                        oglas.setText(rs.getString("text"));

                        return oglas;
                  }

            });
      }

      // obrisi oglas sa datim id
      public boolean delete(int id) {
            MapSqlParameterSource params = new
MapSqlParameterSource("id", id);
            return jdbc.update("DELETE FROM oglasi WHERE id =
:id", params) == 1;
      }

      // insert novi oglas
      public boolean create(Oglas oglas) {
            BeanPropertySqlParameterSource params = new BeanProp
ertySqlParameterSource(oglas);

            return jdbc.update("INSERT INTO oglasi (id, ime,
text, email) VALUES (:id, :ime, :text, :email)",
                        params) == 1;
      }

      // update postojeci oglas
      public boolean update(Oglas oglas) {
            BeanPropertySqlParameterSource params = new BeanProp
ertySqlParameterSource(oglas);

            return jdbc.update("UPDATE oglasi SET ime = :ime,
text = :text, email = :email WHERE id = :id", params) == 1;
      }

}
```

Kompletan listing beans.xml fajla, sa konfiguracijom za MySQL bazu, dat je u nastavku:

```
<?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:context="http://www.springframework.org/schema/context"
       xmlns:tx="http://www.springframework.org/schema/tx"
       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-4.3.xsd
            http://www.springframework.org/schema/tx http://www.
springframework.org/schema/tx/spring-tx-4.3.xsd">

       <context:component-scan base-package="ac.rs.singidunum.fir.pj">
       </context:component-scan>

       <!-- MySQL Data Source -->
        <bean id="dataSource" destroy-method="close" class="org.
apache.commons.dbcp.BasicDataSource">
            <property name="driverClassName" value="com.mysql.
jdbc.Driver"></property>
            <property name="url" value="jdbc:mys
ql://127.0.0.1:3306/springtutorial">
            </property> <property name="username"
value="miodrag"></property> <property
            name="password" value="root"></property>
       </bean>
       <tx:annotation-driven />
       <bean id="transactionManager"
            class="org.springframework.jdbc.datasource.Data-
SourceTransactionManager">
       <property name="dataSource" ref="dataSource"></property>
       </bean>
</beans>
```

Kao što smo na početku ovog poglavlja naveli, Spring teži da olakša rad sa različitim bazama podataka, pošto je tehnologija sakrivena od programera. Na primer, u slučaju da je potrebno promeniti tehnologiju perzistencije, i umesto MySQL baze koristiti Oracle bazu (naravno, podrazumeva se da baze imaju iste šeme – odnosno iste tabele sa istim kolonama), dovoljno je samo promeniti zrno DataSource. Naravno, treba obratiti pažnju da Oracle zahteva drugačije parametre. Izmenjeni beans.xml kod koji podržava rad sa Oracle bazom dat je u nastavku (zrno za konekciju sa MySQL bazom je zakomentarisano).

```
<?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:context="http://www.springframework.org/schema/context"
       xmlns:tx="http://www.springframework.org/schema/tx"
       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-4.3.xsd
              http://www.springframework.org/schema/tx http://www.
springframework.org/schema/tx/spring-tx-4.3.xsd">

       <context:component-scan base-package="ac.rs.singidunum.fir.pj">
       </context:component-scan>
       <!-- Oracle Data Source -->
       <bean id="dataSource" class="oracle.jdbc.pool.OracleData-
Source"
              destroy-method="close">
              <property name="URL" value="jdbc:oracle:thin:@
localhost:1521:orcl">
              </property>
              <property name="user" value="sys as sysdba"></prop-
erty>
              <property name="password" value="root"></property>
       </bean>

       <!-- MySQL Data Source -->
        <!--<bean id="dataSource" destroy-method="close"
class="org.apache.commons.dbcp.BasicDataSource">
              <property name="driverClassName" value="com.mysql.
jdbc.Driver"></property>
              <property name="url" value="jdbc:mys
ql://127.0.0.1:3306/springtutorial">
              </property> <property name="username"
value="miodrag"></property> <property
              name="password" value="root"></property>
       </bean> -->
       <tx:annotation-driven />
       <bean id="transactionManager"
              class="org.springframework.jdbc.datasource.Data-
SourceTransactionManager">
       <property name="dataSource" ref="dataSource"></property>
       </bean>
</beans>
```

Pitanja

1. Šta su DAO (*Data Access Object*) i za šta se koriste u Springu?
2. Koje su praktične posledice pristupanja servisnih objekata repozitorijumu kroz interfejs?
3. Zašto se implementacija sloja perzistencije skriva iza interfejsa i da li je moguće koristiti neki drugi pristup?
4. Zašto je teško rukovati izuzetkom SQLException?
5. Koja je prednost Spring JDBC u odnosu na Hibernate okruženje?
6. Koje šablone Spring nudi za rad sa bazama i u čemu su razlike?
7. Koja je jedina izmena koje je potrebna u Springu ukoliko se promeni tehnologija perzistencije?
8. Koja je osnovna stvar koju je potrebno implementirati u Spring aplikaciji za rad sa bazom podataka?
9. Kako se implementira model za podatke iz baza podataka?
10. Šta je potrebno da se definiše za implementaciju klase DAO?
11. Kako se može definisati zrno koje predstavlja izvor podataka unutar Spring aplikacije?
12. Koja polja mora imati zrno dataSorce za konfiguraciju MySQL baze podataka?
13. Šta je potrebno implementirati u DAO klasi?
14. Da li je u Springu neophodno hvatati izuzetke koje baza izbacuje?
15. Kada se koriste imenovani parametri?
16. Koji je šablon potreban da bi se koristili imenovani parametri?
17. Koji se tipovi objekata mogu koristiti za ubacivanje novog reda u bazu? Koji je više u duhu Springa i zašto?

IX
Uvod u veb servise i osnovna terminologija

U literaturi može da se nađe više definicija veb servisa. U ovom delu udžbenika spomenuto je više različitih definicija iz heterogenih literaturnih izvora.

Veb servisi (engl. *Web service*) su standardizovani komunikacioni medijumi između klijentskih i serverskih aplikacija na WWW-u (Word Wide Web).

Veb servisi obezbeđuju zajedničku (deljenu) platformu koja omogućava različitim aplikacijama, koje su pisane u različitim programskim jezicima i tehnologijama, da komuniciraju međusobno.

Među popularnim protokolima veb servisa ističu se SOAP, WSDL i REST.

SOAP (engl. *Simple Object Access Portocol*) je razvijen kao posrednički jezik (engl. *intermediate language*), sa ciljem da omogući različitim aplikacijama (kreiranim u različitim programskim jezicima i tehnologijama) međusobnu komunikaciju, bez previše uloženog napora u njihov razvoj.

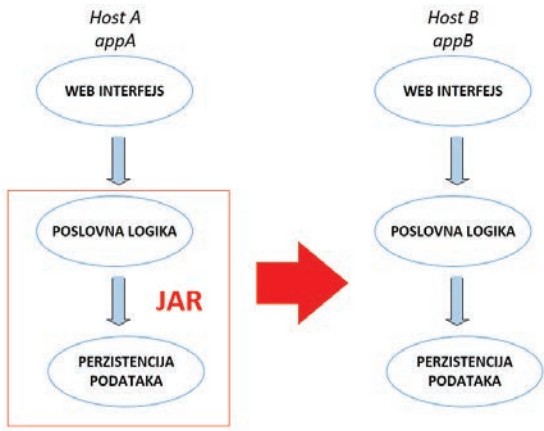
WSDL (engl. *Web Services Description Language*) je XML – bazirani fajl koji daje osnovne informacije klijentskim aplikacijama o tome šta veb servis radi, kao i detaljne informacije o načinu povezivanja na veb servis.

REST (engl. *REpresentational State Transfer*) se koristi za izgradnju veb servisa koji su laki (engl. *lightweight*) u kontekstu konzumiranja računarskih resursa, koji se lako održavaju (engl. *maintanable*) i koji su skalabilni (engl. *scalable*), tj. lako proširivi.

Da bi se bolje razumele prednosti koje proističu iz upotrebe veb servisa u odnosu na standardnu interakciju na nivou aplikacija – aplikacija, može da posluži sledeći primer. Neka se na jednom host-u (hostA) nalazi troslojna veb aplikacija appA koja se sastoji iz slojeva veb interfejsa, poslovne logike (engl. *business logic*) i sloja perzistencije podataka. Neka appA komunicira sa aplikacijom appB koja se nalazi na host-u (hostB). Obe aplikacije su pisane u Java programskom jeziku i komunikacija između njih se vrši na nivou metoda koje su definisane u slojevima poslovne logike i perzistencije podataka.

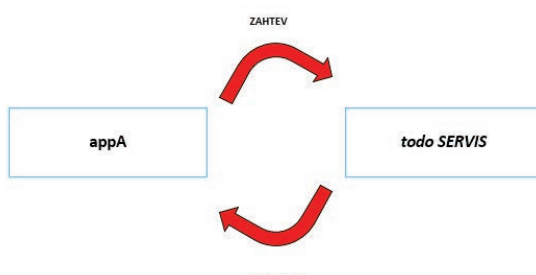
Pod pretpostavkom da se appA modifikuje na nivou poslovne logike i/ili sloja perzistencije podataka, aplikacije appA i appB neće moći da komuniciraju sve dok se promene se ažuriraju i na strani appB. Jedan od načina za usaglašavanje appA i appB je kreiranje JAR fajla na strani appA, koji se zatim importuje u appB. Sam proces kreiranja JAR fajla je mukotrpan i očigledno je da ovo nije racionalno rešenje. Navedeni scenario prikazan je na slici 9.1.

Potrebna je arhitektura, gde jedna aplikacija generiše output koji druge aplikacije mogu da razumeju. Upravo u ovakvim situacijama veb servisi „stupaju na scenu“. Da je appA kreirana kao veb servis, bilo kakve promene u njenoj internoj implementaciji ne bi uticale na mogućnost da neka druga aplikacija komunicira sa njom, zato što bi output appA bio u standardom formatu poput XML-a ili JSON (engl. *JavaScript Object Notation*). Na output s druge strane ne bi uticale interne promene u slojevima poslovne logike i perzistencije podataka.



Slika 9.1. Generisanje JAR-a i import u drugu aplikaciju

Uprošćeni dijagram veb servisa je prikazan na slici 9.2 . Sa slike se vidi da aplikacija appA zahteva odgovarajući resurs (ulsugu) servisa *todo,* koji obrađuje zahtev, generiše odgovor i vraća aplikaciji zahtevani resurs (uslugu).



Slika 9.2. Generisanje JAR-a i import u drugu aplikaciju

Prema definiciji koju je dao WWW konzorcijum (WWW Consortium – W3C), veb servis je softverski sistem koji je dizajniran na način koji omogućava interoperabilnu komunikaciju na nivou aplikacija – aplikacija preko računarske mreže (najčešće Interneta).

Dakle, da bi neka aplikacija bila definisana kao veb servis, ona najmanje mora da zadovolji tri kriterijuma:

1. da bude interoperabilna (engl. *Interoperability*),
2. da omogućava interakcije (komunikacije) na nivou mašina – mašina ili aplikacija – aplikacija i
3. da obavlja komunikaciju preko računarske mreže.

Prvi kriterijum interoperabilnost, odnosi se na mogućnost da heterogene aplikacije, u smislu tehnologija i programskih jezika u kojima su implementirane, međusobno komuniciraju. Dakle, nije važno da li su aplikacije pisane u Java, .NET, Python, ili nekim drugim tehnologijama – sve dok su implementirane kao veb servisi, one će moći da komuniciraju.

Standardne aplikacije su namenjene za interakciju (komunikaciju) na nivou korisnik – aplikacija preko grafičkog ili konzolnog interfejsa, ili na neki drugi način. Da bi aplikacija bila veb servis, ona mora da bude dizajnirana da podržava interakciju na nivou aplikacija – aplikacija (ne korisnik – aplikacija), što ujedno predstavlja drugi kriterijum koji aplikacija mora da zadovoljava da bi mogla da se definiše kao veb servis. Da bi aplikacije mogle da komuniciraju sa drugim aplikacijama, one moraju da generišu output u standardnom formatu, koga sve druge aplikacije razumeju, kao što su XML i JSON, YAML (*YAML Ain't Markup Language*), itd.

Konačno, treći kriterijum kojeg aplikacija treba da zavodoljava da bi upala u kategoriju veb servisa jeste mogućnost obavljanja komunikacije preko računarske mreže. Sve više komunikacija se obavlja preko računarske mreže, kao što je Internet, i bez ovog uslova aplikacija ne bi mogla da ispunjava svoju osnovnu svrhu.

Veb servis je aplikacija koja ne zavisi od platforme. Da bi se omogućila komunikacija koja ne zavisi od platforme (engl. *platform independent communication*) i da bi na primer appA mogla da konzumira usluge veb servisa, moraju se koristiti standardni formati zahteva i odgovora. Međutim, postavlja se pitanje kako aplikacija appA zna koji format zahteva treba da generiše kako bi komunicirala sa određenim veb servisom, i kako da zna u kom formatu da očekuje odgovor od veb servisa.

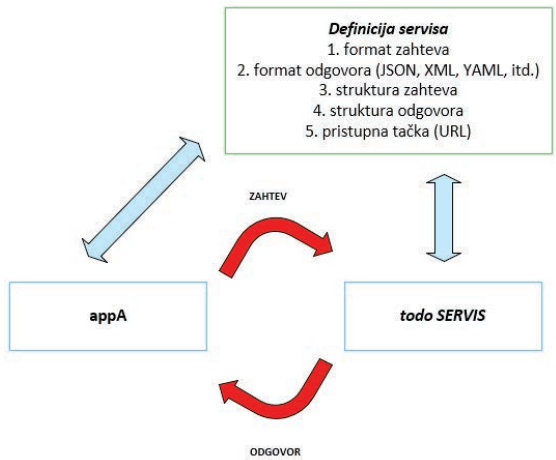
Odgovor na ovo pitanje leži u definiciji servisa (engl. *service definition*). Definiciju servisa obezbeđuje sam veb servis i na ovaj način aplikacija koja želi da konzumira usluge veb servisa dobija informacije o:

1. formatu zahteva,
2. formatu odgovora,
3. strukturi zahteva,
4. strukturi odgovora i
5. pristupnoj tački veb servisa (engl. *Web service endpoint*).

Na osnovu informacije o pristuponoj tački veb servisa, aplikacija klijent će znati kako da pristupi veb servisu. Drugim rečima, aplikacija će znati na kojoj lokaciji se

nalazi željeni veb servis. Pristupna tačka u najvećem broju slučajeva predstavlja URL (*Uniform Resource Locator*) preko koga je veb servis dostupan javnosti (klijentskim aplikacijama koje koriste njegove usluge).

Arhitektura veb servisa sa definicijom servisa vizuelizovana je na slici 9.3.



Slika 9.3. Definicija servisa

Kada se govori o osnovnim pojmovima veb servisa, korisno je spomenuti i transportni medijum, preko koga aplikacija klijent i veb servis komuniciraju. U većini slučajeva, kao transportni medijum se koristi ili Internet, ili redovi (engl. *queues*). U prvom slučaju, klijentska aplikacija i veb servis komunciraju putem HTTP (*Hyper-Text Transfer Protocol*) protokola, osnovnog protokola za veb komunikaciju na Internetu. U ovom slučaju se komunikcija odvija kao bilo koja druga veb komunikacija.

U drugom slučaju (ako se za komunikaciju koriste redovi), koristi se transportni model tzv. redova poruka (engl. *Message Queues – MQ*). U ovom slučaju se komunikacija vrši na sledeći način: korisnik usluga (aplikacija) prvo postavlja poruku u red, zatim provajder usluga (veb servis) uzima poruku iz reda, obrađuje je i poruku sa odgovorom vraća u red i šalje je korisniku usluga.

9.1. Pojam, vrste, arhitektura, osnovne karakteristike i prednosti veb servisa

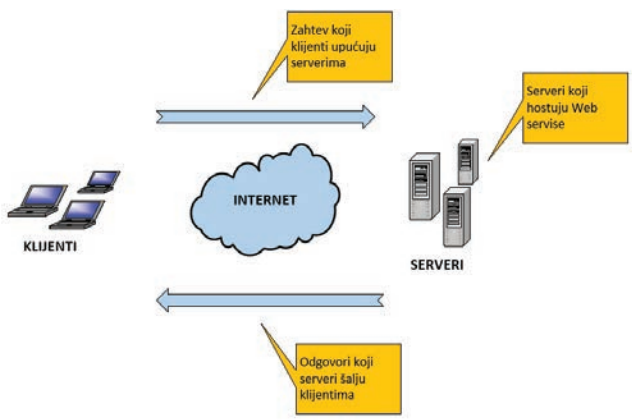
Poslednjih decenija primećuje se da je sve više savremenih poslovnih aplikacija razvijeno u nekoj od veb – baziranih tehnologija, kao što su Java, .NET, Angular JS, Node.js, itd. Sve češće se javlja potreba da heterogene aplikacije međusobno komuniciraju. S obzirom da su aplikacije izgrađene u različitim tehnologijama i programskim jezicima, obezbediti pouzdanu i standardizovanu komunikaciju među njima postaje sve veći izazov.

Veb servisi su nastali kao jedno od mogućih rešenja navednog izazova, koje može relativno lako da se implementira, koje je lako održivo i koje je jeftino u smislu potrošnje računarskih resursa.

Kao što je već rečeno, veb servisi predstavljaju standardizovani medijum koji obezbeđuje i omogućava komunikaciju između heterogenih klijentskih i serverskih aplikacija na veb-u.

Veb servis je softverski modul koji je dizajniran za izvršavanje određenog skupa zadataka. Klijent ima mogućnost da vrši pretragu raspoloživih veb servisa na mreži, najčešće Internetu. Kada klijent pronađe željeni veb servis, on može da ga pozove, kada veb servis klijentu pruža određeni set funkcionalnosti.

Uprošćeni dijagram koji opisuje arhitekturu veb servisa prikazan je na slici 9.4.



Slika 9.4. Uprošćena arhitektura veb servisa

Klijent poziva veb servis slanjem zahteva (engl. *requests*) ka serveru koji hostuje veb servis. Zahtevi se realizuju mehanizmom koji se naziva udaljeno pozivanje procedura (engl. *Remote Procedure Calls – RPC*). Udaljeno pozivanje procedura su pozivi metoda koje se hostuju na relevantnom veb servisu.

Kao primer navodi se Amazon, koji obezbeđuje veb servis koji klijentima pruža informacije o ceni proizvoda koji se prodaju preko ove online platforme. Front-end, ili prezentacioni sloj Amazonove platforme može da bude pisan u bilo kojoj tehnologiji (npr. Java ili .NET), ali bez obzira na to, bilo koja klijentska aplikacija (koja je pisana u bilo kom programskom jeziku) ima mogućnost da komunicira sa Amazonovim veb servisom.

Glavna komponenta svakog veb servisa su podaci, koji se šalju između klijenta i servera. U nekim implementacijama veb servisa za slanje podataka koristi se XML (*Extensible Markup Language*). XML je jedan od standarda za razmenu podataka i informacija na veb-u. Većina programskih jezika ima ugrađene biblioteke koje omoućavaju interpretaciju (prevođenje) XML dokumenata.

Dakle, u mnogim slučajevima, kada aplikacije međusobno komuniciraju, one zapravo koriste XML kao komunikacioni jezik. Na ovaj način se obezbeđuje zajednička platforma za međusobnu komunikaciju aplikacija koje su razvijene u različitim programskim jezicima i tehnologijama.

Veb servisi u nekim slučajevima koriste SOAP protokol za slanje XML podataka. Podaci se šalju pomoću običnog HTTP protokola. U ovom slučaju se za podatke koji se šalju od veb servisa ka klijentskoj aplikaciji koristi termin SOAP poruka (engl. *SOAP message*). SOAP poruka nije ništa drugo, nego XML dokument. S obzirom da je dokument koji se razmenjuje pisan u XML-u, veb aplikacija koja poziva veb servis može da bude razvijena u bilo kom programskom jeziku.

9.1.1. Vrste i osnovne komponente veb servisa

U literaturi se navode dve osnovne vrste veb servisa:

1. SOAP veb servisi
2. RESTful veb servisi

Da bi veb servis bio funkcionalan, određene komponente moraju da se implementiraju, bez obzira na tehnologiju i programski jezik u kojoj je veb servis razvijan.

U slučaju SOAP veb servisa, osnovne komponente su SOAP protokol, WSDL i UDDI. S obzirom da je celo poglavlje posvećeno RESTful veb servisima i da je cela jedna glava udžbenika posvećena primerima RESTful veb servisa, u nastavku je akcenat dat na osnovnim komponentama SOAP veb servisa.

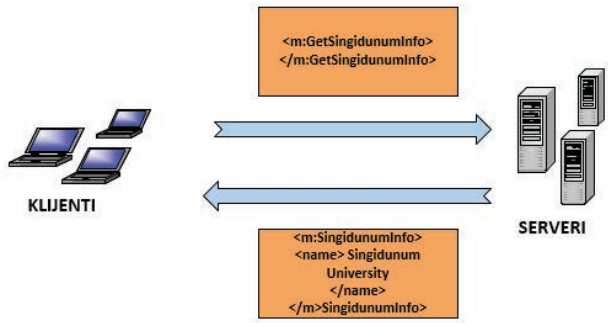
SOAP protokol

SOAP je u literaturi poznat kao protok za razmenu poruka koji je nezavistan od transportnog medijuma (engl. *transport independent messaging protocol*). SOAP se bazira na prenosu XML podataka u formi SOAP poruka. U svakoj SOAP poruci se nalazi XML dokument. Samo struktura (ne sadržaj) XML dokumenta prati specifičan obrazac. Jedna od najboljih strana SOAP-a jeste da razmena podataka i informacija može da se vrši pomoću običnog HTTP protokola, koji predstavlja standardni protokol za komunikaciju na veb-u.

Svaka SOAP poruka se sastoji iz sledećih elemenata:

- poruka počinje tzv. korenim elementom (engl. *root element*), koji je poznat kao <Envelope> element (element koverat). Koreni element je prvi element koji se navodi u XML dokumentu.
- koverat se dalje deli na dva dela. Prvi deo je zaglavlje (engl. *header*), dok je drugi deo telo (engl. *body*).
- u zaglavlju se navode podaci o rutiranju, što su jednostavno rečeno informacije o tome kojem klijentu se šalje XML dokument.
- u telu koverte se nalazi poruka koja se šalje.

Na slici 9.5. dat je jednostavan primer komunikacije pomoću SOAP protokola.



Slika 9.5. Jednostavan primer komunikacije preko SOAP protokola

WSDL jezik

WSDL (*Web Services Description Language*) je, kao što sledi iz samog akronima, jezik koji opisuje veb servis. Klijent neće biti u mogućnosti da konzumira veb servis, ako nema informacije o tome gde se veb servis nalazi. Lociranje veb servisa je jedna od funkcija koju ima WSDL.

Drugo, klijentska aplikacija treba da ima informacije o tome šta veb servis radi (koje funkcije obavlja), kako bi mogla da pozove pravi veb servis (onaj servis koji zadovoljava potrebe klijentske aplikacije). WSDL obezbeđuje i ove informacije, tj. opisuje šta tačno veb servis radi i koje su njegove osnovne funkcionalnosti.

Fajl WSDL-a je još jedan XML-bazirani fajl koji govori klijentskoj aplikaciji gde je veb servis lociran na vebu, koje su njegove funkcionalnosti i kako može da se koristi (konzumira).

Primer WSDL fajla je naveden ispod.

```
<definitions>
 <message name=”SingidunumRequest”>
    <part name=”SingidunumID” type=”xsd:string”/>
 </message>

 <message name=” SingidunumResponse”>
    <part name=”SingidunumName” type=”xsd:string”/>
 </message>
```

```
   <portType name=”Singidunum_PortType”>
      <operation name=”Singidunum”>
         <input message=”tns: SingidunumRequest”/>
         <output message=”tns: SingidunumResponse”/>
      </operation>
   </portType>

   <binding name=”Singidunum_Binding” type=”tns: Singidunum_Port-
Type”>
      <soap:binding style=”rpc”
         transport=”http://schemas.xmlsoap.org/soap/http”/>
      <operation name=”Singidunum”>
         <soap:operation soapAction=”Singidunum”/>
         <input>
            <soap:body
                encodingStyle=”http://schemas.xmlsoap.org/soap/en-
coding/”
               namespace=”urn:examples: Singidunumservice”
               use=”encoded”/>
         </input>

        <output>
            <soap:body
                encodingStyle=”http://schemas.xmlsoap.org/soap/en-
coding/”
               namespace=”urn:examples: Singidunumservice”
               use=”encoded”/>
         </output>
      </operation>
   </binding>
</definitions>
```

Važni aspekti koje treba naglasiti o prikazanoj WSDL deklaraciji su:

- <message>,
- <portType> i
- <binding>.

Element <message> u WSDL definiciji se koristi za definisanje različitih elemenata podataka za svaku operaciju koju veb servis izvršava. U WSDL primeru koji je naveden iznad, definisano je dve poruke koje mogu da se razmene između veb servisa i klijentske aplikacije, i to su u ovom slučaju „SingidunumRequest“ i „SingidunumResponse“ operacije.

Poruka „SingidunumRequest“ sadrži element „SingidunumID“ tipa string. Slično, operacija „TutorialResponse“ definiše element „SingidunumName“ koji je takođe tipa String.

Element <portType> opisuje operaciju koju veb servis može da izvrši, i u prikazanom WSDL primeru naziv ove operacije je Singidunum. Ova operacija preuzima dve poruke, i to jednu input i jednu output poruku.

Elementom <binding> se definiše protokol koji veb servis koristi. U ovom slučaju, definiše se http protokol (**http://schemas.xmlsoap.org/soap/http**). Osim protokola, u ovom elementu se definišu i ostale informacije tele operacije koja može da se izvrši, kao što su prostor imena (engl. *namespace*) i zahtev za kodiranjem poruke.

UDDI standard

UDDI (*Universal Description, Discovery and Integration*) je standard za opisivanje, objavljivanje i otkrivanje veb servisa koje pruža određeni dobavljač usluga (engl. *service provider*). UDDI obezbeđuje specifikacije koje pomažu u hostovanju informacija o veb servisu.

U prethodnoj sekciji je navedeno da WSDL pruža informacije koje otkrivaju šta veb servis radi (koje usluge i operacije pruža). Međutim, postavlja se pitanje kako će klijent na prvom mestu da locira WSDL fajl u kom su navedeni detalji operacija veb servisa. Odgovor na ovo pitanje daje UDDI, koji obezbeđuje repozitorijum za skladištenje WSDL fajlova. Drugim rečima, klijentska aplikacija ima potpun pristup UDDI-ju, koji igra ulogu baze podataka u kojoj se nalaze svi WSDL fajlovi.

U literaturi se često navodi analogija između UDDI-ja i telefonskog imenika. Slično kao što se u telefonskom imenu nalaze osnovne informacije, poput imena, adrese i broja telefona određene osobe, UDDI registar čuva sve relevantne podatke veb servisa.

9.1.2. Prednosti veb servisa

Među brojnim prednostima veb servisa koje se često navode u literaturi, u nastavku su navedene samo neke od njih:

- veće izlaganje poslovnih funkcionalnosti računarskoj mreži, kao što je Internet,
- interoperabilnost (engl. *interoperability*) između heterogenih aplikacija,
- standardizovan protokol koji svi razumeju i
- smanjivanje komunikacionih troškova.

Veće izlaganje poslovnih funkcionalnosti Internetu. veb servis je jedinica koda kojim se upravlja koja obezbeđuje neku vrstu funkcionalnosti klijentskim aplikacijama ili krajnjim korisnicima. Funkcionalnosti se pozivaju pomoću HTTP protokola, što znači da ih klijenti mogu pozvati putem Internet mreže. U poslednje vreme se veliki broj aplikacija hostuje na nekom od servisa Interneta, što još više jača poziciju i svrhu koju veb servisi imaju. Ova činjenica dalje implicira da veb servis može da se nalazi bilo gde na Internetu i da obezbedi određenu funkcionalnost kada se za to ukaže potreba.

Interoperabilnost između heterogenih aplikacija. veb servisi omogućavaju da različite, tehnološki heterogene aplikacije međusobno „razgovaraju" i da dele podatke, kada tehnička specifikacija i programski jezik u kome je aplikacija pisana postaju nevažni. Dakle, umesto da se piše specifičan kod, koji mogu da razumeju samo aplikacije određenog tipa, kada se koriste veb servisi može da se piše generički kod koji razumeju sve aplikacije.

Standardizovan protokol koji svi razumeju. veb servisi koriste standardizovani industrijskih protokol za komuniciranje. Sva četiri sloja veb servisa: transport servisa (engl. *service transport*), XML razmene poruka (engl. *XML messaging*), opis servisa (engl. *service description*) i sloj otkrivanja servisa (engl. *Service discovery*) koriste dobro definisane i poznate protokole u steku veb servis protokola.

Smanjivanje komunikacionih troškova. S obzirom da većina veb servisa koristi HTTP protokol za komunikaciju, mogu da se koriste jeftine Internet komunikacije za implementiranje veb servisa.

9.1.3. Arhitektura veb servisa

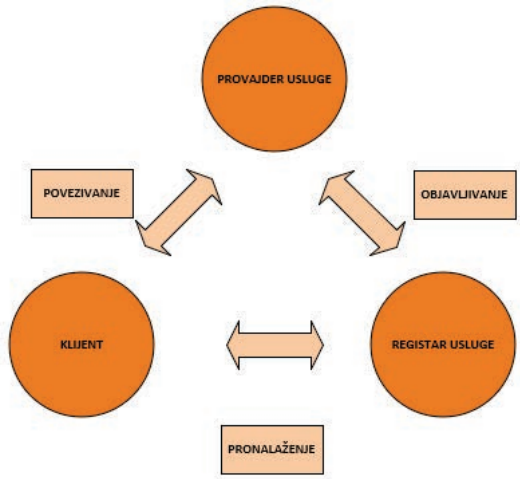
Arhitektura veb servisa sastoji se iz tri posebne uloge: dobavljač (provajder), entitet koji zahteva uslugu (engl. *requestor*) i broker (engl. *broker*).

Entitet koji ima ulogu provajdera kreira veb servis i izlaže ga. tj. omogućava njegovo korišćenje od strane klijenta (klijentske aplikacije), kome je neka od usluga veb servisa potrebna.

Entitet koji zahteva uslugu je klijentska aplikacija kojoj je potrebna neka od usluga koje veb servis pruža. Kao što je već više puta navedeno, klijentska aplikacija može biti bilo koja aplikacija koja je pisana u bilo kojoj tehnologiji (.NET, Java, itd.) i koja zahteva neku od funkcionalnost veb servisa.

Broker je aplikacija koja obezbeđuje pristup UDDI-ju. Kao što je već i navedeno, UDDI omogućava klijentskoj aplikaciji da locira veb servis.

Na slici 9.6. je prikazana interakcija između entiteta koji zahteva uslugu (klijenta), provajdera usluge i registra usluge (UDDI).



Slika 9.6. Interakcija između klijenta, provajdera i registra usluge

9.1.4. Osnovne karakteristike veb servisa

Neke od osnovnih karakteristika veb servisa obuhvataju sledeće: baziranost na XML tehnologiji, slaba povezanost, sinhrone ili asinhrone funkcionalnosti, mogućnost podrške za udaljeno pozivanje procedura i podrška za razmenu dokumenata.

Baziranost na XML tehnologiji. veb servisi koirste XML za reprezentaciju podataka u slojevima prezentacije i transporta. Upotrebom XML standarda se eliminiše potreba da sistem zavisi od specifične mrežne, ili platforme operativnog sistema, zato što XML predstavlja univerzalni standard kojeg svi razumeju.

Slaba povezanost odnosno *labava sprega (*engl. *loosely coupled)*. Termin slaba povezanost se odnosi na to da klijent i veb servis nisu povezani, što znači da ako se na primer implementacija veb servisa vremenom promeni, način na koji klijent poziva usluge veb servisa će ostati isti. Drugim rečima, promene na strani veb servisa ne zahtevaju promene na strani klijenta. Prihvatanje slabo povezane arhitekture softvera olakšava njegovo upravljanje i omogućava jednostavniju integraciju različitih sistema.

Sinhrone ili asinhrone funkcionalnosti. Sinhronizovanost se odnosi na povezivanje klijenta sa izvršavanjem servisa. U slučaju sinhronizovanih operacija klijent čeka dok veb servis završi neku operaciju. Kao primer ovakvog scenarija navode se operacije upisivanja i čitanja iz baze podataka. Ako se podaci čitaju iz jedne baze podataka i odmah nakon toga upisuju u drugu, onda navedene operacije čitanja i pisanja treba sekvencijalno da se izvršavaju.

S druge strane, asinhrone operacije omogućavaju klijentu da pozove veb servis i da nakon toga izvršava paralelno i druge funkcije. Ovo je jedna od najpoželjnijih tehnika koja osigurava da u slučaju izvršavanja neke operacije ostali servisi neće biti zaustavljeni.

*Podrška za udaljeno pozivanje procedura (*engl. *Remote Procedure Calls – RPCs)*. Veb servisi omogućavaju klijentima da pozovu procedure, funkcije i druge metode na udaljenim objektima korišćenjem protokola koji se bazira na XML tehnologiji. Udaljene procedure otkrivaju input i output parametre koje veb servisi moraju da podržavaju.

Podrška za razmenu dokumenata. Jedna od ključnih prednosti XML-a je mogućnost reprezentacije podataka i kompleksnih dokumenata na opšti način koji svi razumeju. Dokumenti mogu biti prosti, kao što je reprezentacija adrese nekog entiteta, do složenih, kao što je reprezentacija cele knjige.

9.2. SOAP veb servisi

Osnovna karakteritstika SOAP veb servisa je da se temelje na SOAP protokolu.

SOAP je XML-bazirani protokol za pristupanje veb servisima pomoću HTTP protokola. SOAP definiše određene specifikacije koje mogu biti korišćene od strane svih aplikacija.

SOAP je razvijen kao posrednički jezik (engl. *intermediate language*) sa ciljem omogućavanja da aplikacije pisane u različitim programskim jezicima mogu međusobno da komuniciraju. Na ovaj način se eliminiše dodatni napor koji bi programeri i developer-i morali da ulože u prilagođavanje aplikacija za međusobnu komunikaciju. SOAP definiše način na koji veb servisi komuniciraju jedni sa drugima i sa klijentskim aplikacijama koje ih pozivaju.

Kao što je već više puta naglašeno do sada, poslednjih decenija, na desktop i serverskim računarima i Internetu širom sveta postoji veliki broj aplikacija koje su pisane u različitim programskim jezicima i pomoću različitih tehnologija (Java, .NET, PHP, Python, itd.). Jedan od ključnih zahteva savremenog računarskog okruženja jeste da aplikacije međusobno komuniciraju i da razmenjuju podatke. Međutim, komunikacija u heterogenim okruženjima može da bude jako kompleksna, gde je potrebno razviti sofisticirani i složen programski kod koji bi podržao ovakve komunikacije.

Jedan od široko rasprostranjenih metoda za „borbu“ protiv heterogenosti i kompleksnosti jeste upotreba XML standarda, kao posredničkog jezika u razmeni podataka između aplikacija. U svaki programski jezik može da se ugradi mogućnost prepoznavanja i razumevanja XML standarda. Međutim, ne postoji standardna specifikacija za korišćenje XML-a od strane različitih programskih jezika za potrebe razmene podataka. Upravo u ovom delu SOAP igra svoju ulogu.

SOAP je dizajniran da radi sa XML-om preko HTTP protokola i implementira neku vrstu specifikacija koje mogu sve aplikacije da koriste.

U nastavku su navedeni detalji o SOAP arhitekturi i specifikaciji.

Prednosti SOAP-a

U nastavku su navedeni neki od razloga široke upotrebe SOAP-a.

Tokom razvoja veb servisa potrebana je neka vrsta jezika kojom će veb servisi da komuniciraju sa klijentskim aplikacijama. SOAP je odličan medijum koji je pre svega razvijen sa ciljem realizacije ove svrhe. Upotrebu SOAP-a preporučuje WWW Konzorcijum (engl. World Wide Web Consortium – W3C), kao vodeće telo u donošenju standarda za veb i veb tehnologije.

SOAP je lagan protokol (engl. *light-weight protocol*) koji se koristi za razmenu podataka između aplikacija. Reč „lagan" se odnosi na zahteve u kontekstu korišćenja računarskih resursa. SOAP je „lagan" zato što se bazira na XML-u, koji i sam spada u lakše jezike za razmenu podataka.

SOAP je dizajniran na način da je nezavistan od platforme i od operativnog sistema. Ovo dalje znači da SOAP protokol može da izvršava bilo koji programski jezik bilo na Windows, Linux, ili nekoj trećoj platformi.

SAOP funkcioniše iznad HTTP protokola, koji je podrazumevani protokol za sve veb aplikacije. Ova činjenica implicira da nisu potrebne dodatne modifikacije da bi veb servisi koji koriste SOAP mogli da se izvršavaju na veb-u.

SOAP gradivni blokovi

U SOAP specifikaciji je definisano nešto što nosi naziv SOAP poruka. SOAP poruka je ono što se šalje veb servisu i klijentskoj aplikaciji. SOAP poruka se sastoji iz više gradivnih blokova, koji su prikazani na slici 9.7.

SOAP poruka je običan XML dokument koji se sastoji iz sledećih komponenti:

- elementa koverta koji služi za identifikaciju XML dokumenta kao SOAP poruke i koji enkapsulira detalje u SOAP poruci. Koverat predstavlja koreni element SOAP poruke.
- elementa zaglavlja u kome se nalaze informacije o zaglavlju. U zaglavlju mogu da se nalaze informacije poput kredencijala za autentifikaciju koje koristi aplikacija koja poziva veb servis i definicije složenih tipova podataka koje koristi SOAP poruka. Po podrazumevanim podešavanjima, SOAP poruka može da se sastoji od parametara prostih tipova kao što su stringovi i brojevi, ali isto tako mogu da se koriste i složeni, objektni tipovi podataka.

Jednostavan primer složenog tipa podataka je prikazan u nastavku.

U slučaju da je potrebno da pošalje strukturirani tip podataka koji predstavlja kombinaciju elemenata „Singidunum Name" i „Singidunum Description", onda bi složeni tip podataka bio definisan, kao što je prikazano u kodu ispod.

Složeni tip podataka se definiše tagom <xsd:complexType>. Svi potrebni elementi ovakve strukture zajedno sa tipovima podataka su zatim definisani kolekcijom složenog tipa podataka.

```
<xsd:complexType>
 <xsd:sequence>
      <xsd:element name=”Singidunum Name” type=”string”/>
      <xsd:element name=” Singidunum Description” type=”string”/>
  </xsd:sequence>
</xsd:complexType>
```



Slika 9.7. Gradivni blokovi SOAP poruke

U elementu tela SOAP poruke nalaze se informacije koje su relevantne za poziv veb servisu, i za odgovor koji veb servis upućuje klijentskim aplikacijama. Drugim rečima, u elementu telo nalaze se podaci koji se razmenjuju između veb servisa i aplikacije koja taj servis poziva.

U nastavku je naveden primer SOAP tela koji koristi složeni tip podataka koji je definisan u sekciji zaglavlja.

```
<soap:Body>
   <GetSingidunumInfo>
           <SingidunumName>Singidunum University</SingidunumName>
                <SingidunumDescription>The biggest private Univer-
     sity          </SingidunumDescription>
   </GetTutorialInfo>
</soap:Body>
```

Struktura SOAP poruke

Kada god klijentska aplikacija pozove veb servis, servis automatski generiše SOAP poruke.

Kada god klijent pozove neku od metoda veb servisa, servis automatski generiše SOAP poruku koja sadrži sve neophodne detalje podataka koje će veb servis da pošalje klijentskoj aplikaciji.

Kao što je već i navedeno, jednostavna SOAP poruka se sastoji iz sledećih elemenata:

- elementa koverta,
- elementa zaglavlja,
- elementa tela i
- opciono elementa greške (engl. *fault element*).

Na 9.8 je dat primer SOAP poruke sa deskriptivnim opisom funkcije koju svaki element ima.

Na prikazanoj slici, elementi su označeni rednim brojevima, gde svaki od označenih elemenata obavlja sledeće funkcije:

1. Prvi deo SOAP poruke je element koverte koji se koristi za enkapsulaciju cele SOAP poruke,
2. Naredni element SOAP poruke je SOAP telo, u kome se nalaze detalji poruke koja se razmenjuje između klijentske aplikacije i veb servisa,
3. U poruci iz primera se nalazi veb servis pod nazivom „SingidunumService“ i
4. „SingidunumService“ prihvata parametar tipa integer, čiji je naziv „SingidunumID“.

Nakon definisanja svega navedenog, SOAP poruka se razmenjuje između veb servisa i klijentske aplikacije.

Iz navedenog primera jasno može da se vidi koje informacije SOAP poruka obezbeđuje klijentu. Prvo, klijent iz SOAP poruke može da vidi naziv (ime) veb servisa, zatim informacije o parametrima koje veb servis očekuje i koji je tip svakog parametra.



Slika 9.8. Primer SOAP poruke

SOAP element koverte

Prvi gradivni element SOAP poruke je SOAP koverat.

SOAP koverat se koristi za enkapsulaciju svih neophodnih detalja SOAP poruka koje se razmenjuju između veb servisa i klijentskih aplikacija.

Element koverat se koristi za označavanje početka i kraja SOAP poruke. Na ovaj način klijentska aplikacija koja poziva veb servis zna kada gde je kraj SOAP poruke.

U nastavku su navedene ključne stvari elementa SOAP koverte:

1. Svaka SOAP poruka mora da ima koreni element Envelope. Prema specifikacijama je apsolutno obavezno da SOAP poruka ima element koverte,
2. Svaki element koverte mora da ima makar jedan element SOAP telo,
3. Ako element koverat ima element zaglavlje, ne sme biti više od jednog zaglavlja i zaglavlje mora da bude prvi element dete elementa koverat i da bude naveden pre elementa telo,
4. Element koverat se menja sa promenom verzije SOAP-a,
5. SOAP procesor v.1.1 generiše grešku kada primi poruku koja sadrži v1.2 imenski prostor elementa koverte,
6. SOAP procesor v.1.2 generiše poruku nepoklapanja verzija (engl. *version mismatch*) kada primi poruku koja ne uključuje imenski prostor v.1.2 elementa koverte.

U nastavku je naveden primer v.1.2 SOAP elementa koverte.

```
<?xml version=”1.0”?>
<SOAP-ENV:Envelope     xmlns:SOAP-ENV=”http://www.w3.org/2001/12/
soap-envelope” SOAP-ENV:encodingStyle=” http://www.w3.org/2001/12/
soap-encoding”>
          <soap:Body>
          <SingidunumWebService xmlns=”http://singidunum.ac.rs/”>
                  <SingidunumID>int</SingidunumID >
             </SingidunumWebService >
         </soap:Body>
</SOAP-ENV:Envelope>
```

SOAP poruka o grešci

Kada klijent pošalje zahtev SOAP veb servisu, klijent može da dobije dva tipa odgovora, i to: uspešan odgovor ili odgovor sa statusom greška.

U normalnim okolnostima, kada ne dođe do greške, veb servis u odgovoru klijentu šalje SOAP poruku. S druge strane, ako dođe do greške, veb servis u odgovoru vraća SOAP poruku o grešci (engl. *SOAP fault message*), sa statusom HTTP 500.

SOAP poruke o grešci se sastoji iz sledećih elemenata:

1. <faultCode> koji predstavlja kod koji opisuje grešku i on može da bude:
   - » SOAP-ENV:VersionMismatch – greška koja se javlja kada je veb servis uočio pogrešni imenski prostor SOAP Envelope elementa.
   - » SOAP-ENV:MustUnderstand – prvo dete Header elementa sa atributom mustUnderstand koji je postavljen na vrednost 1 nije dobro obrađeno.
   - » SOAP-ENV:Client – poruka nema dobru strukturu, ili sadrži netačne informacije.
   - » SOAP-ENV:Server – postoji problem na serveru i poruka ne može biti obrađena.
2. <faultString> je tekstualna poruka koja prikazuje detaljni opis nastale greške.
3. <faultActor> je opcioni element koji predstavlja tekstualni string koji navodi entitet koji je prouzrokovao grešku.
4. <detail> je takođe opcioni element koji opisuje grešku koja je specifična za određenu aplikaciju. Svaka aplikacija može da definiše specifičnu grešku za različite scenarije poslovne logike.

U nastavku je naveden primer poruke o grešci. Poruka se generiše u situaciji kada klijentska aplikacija pokuša da pozove metod SingidunumID u klasi GetSingidunum.

Prikazana poruka o grešci se generiše u slučaju kada tražena metoda ne postoji u definisanoj klasi.

```
<?xml version='1.0' encoding='UTF-8'?>
<SOAP-ENV:Envelope
    xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/enve-
    lope/"   xmlns:xsi="http://www.w3.org/1999/XMLSchema-instance"
    xmlns:xsd="http://www.w3.org/1999/XMLSchema">
    <SOAP-ENV:Body>
        <SOAP-ENV:Fault>
          <faultcode xsi:type="xsd:string">SOAP-ENV:Client</fault-
code>
        <faultstring xsi:type="xsd:string">
                        Failed to locate method (GetSingidunumID)
             in class    (GetSingidunum)
         </faultstring>
       </SOAP-ENV:Fault>
   </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Kada veb servis pokuša da izvrši kod koji je prikazan, generisaće poruku o grešci „Failed to locate method (GetSingidunumID in the class (GetSingidunum).

SOAP model komunikacije

U poslednje vreme se sve vrste komunikacionih aktivnosti sa SOAP-om izvršavaju putem HTTP protokola. Pre SOAP-a, mnogi veb servisi su za potrebe komunikacije koristili standard udaljenog pozivanja procedura (engl. *Remote Procedure Call – RCP*).

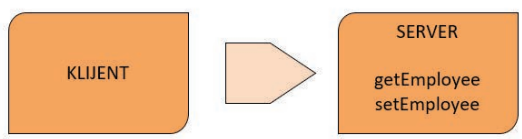
RPC je bio najjednostavniji vid komunikacije, ali uz brojna ograničenja.

U nastavku je razmotren dijagram prikazan na slici 9.9, kako bi se bolje razumela komunikacija između klijenta i veb servisa.

Neka je pretpostavka da server hostuje veb servis koji obezbeđuje dve metode:

- getEmployee – metoda koja prikazuje sve detalje o zaposlenom (engl. employee) i
- setEmployee – metoda koja postavlja vrednosti atributa zaposlenog, kao što su departman u kom zaposleni radi, platu, itd.

U standardnoj RPC komunikaciji, klijent bi jednostavno pozvao navedene metode i u pozivu serveru bi naveo neophodne parametere, i server bi klijentu vratio željeni odgovor.



Slika 9.9. Primer RPC komunikacije

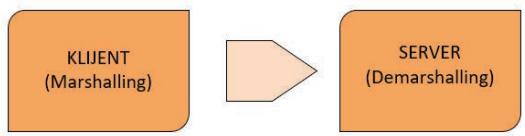
Prikazani RPC model komunikacije ima ozbiljna ograničenja:

1. *Nije nezavisan od programskog jezika*. Server koji hostuje metode je pisan u određenom programskom jeziku i pozivi serveru moraju da budu pisani u tom istom programskom jeziku.
2. *Ne koristi standardizovani protok*. Kada se uputi poziv udaljenoj proceduri, poziv ne koristi standardizovani protokol. Ovo ograničenje je predstavljalo veliki problem jer je standardni protokol za veb komunikaciju HTTP.
3. *Zaštitni zidovi (engl. firewalls)*. S obzirom da RPC pozivi ne koriste standardni protokol, kao što je HTTP, na zaštitnom zidu koji brani server moraju da se otvore dodatni portovi kako bi saobraćaj RPC-a bio omogućen. Po podrazumevanim podešavanjima, skoro svi zaštitni zidovi blokiraju ovakav tip saobraćaja i da bi on bio omogućen potrebno je sprovesti dodatne administrativne aktivnosti.

Da bi se prevazišla navedena ograničenja, SOAP koristi komunikacioni model koji je vizuelizovan na slici 9.10.

Ovakav komunikacioni model se sastoji iz dva koraka:

1. Klijent formatira informacije i sve argumente u skladu sa sintaksom poziva procedure i ubacuje u SOAP poruku, koju šalje serveru kao deo HTTP zahteva. Ovakav proces enkapsulacije podataka u SOAP poruku je u literaturi poznat pod nazivom *Marshalling*.
2. S druge strane, server odmotava poruku koju je poslao klijent, analizira zahtev klijenta i generiše odgovor koji šalje nazad klijentu u formi SOAP poruke. Proces odmotavanja zahteva koji je poslao klijent je u literaturi poznat pod nazivom *Demarshalling*.



Slika 9.10. Primer SOAP komunikacije

Kratak pregled osnovnih informacija o SOAP veb servisima

SOAP je protokol koji se koristi za razmenu podataka između aplikacija koje su razvijene u različitim programskim jezicima i tehnologijama.

SAOP je baziran na XML specifikacijama i funkcioniše pomoću HTTP protokola. Zbog toga se SOAP često koristi za potrebe komuniciranja između veb aplikacija.

Osnovni SOAP gradivni element je SOAP poruka, koja se sastoji iz elementa konverte, zaglavlja i tela.

Element koverte je obavezan element svake SOAP poruke i koristi se za enkapsulaciju svih podataka jedne SOAP poruke.

U elementu zaglavlja se najčešće navode informacije poput informacija koje su važne za autentifikaciju ili definicije složenih tipova podataka.

Element telo je glavni element SOAP poruke. U ovom element se nalaze definicije veb metoda i informacije o parametrima u slučaju potrebe.

9.3. Primer SOAP veb servisa u Visual Studio okruženju

S obzirom da je cela jedna glava udžbenika posvećena primerima RESTful veb servisa kreiranih u Java tehnologiji i Spring Boot razvojnom okviru, u ovom delu udžbenika je dat primer implementacije jednostavnog veb servisa u .NET okruženju i Microsoft Visual Studio 2017 integrisanom razvojnom okruženju.

Kroz analizu praktičnog primera najbolje može da se vidi način na koji se generiše SOAP poruka. U ovom primeru data je implementacija ASMX veb servisa pomoću

Microsoft .NET Framework-a. Microsoft .NET podržava i SOAP verziju 1.1 i verziju 1.2.

ASMX veb servis automatski generiše WSDL dokument. Kao što je već navedeno, WSDL dokument je neophodan kako bi klijentska aplikacija imala uvid u sve funkcionalnosti veb servisa (šta sve veb servis može da uradi).

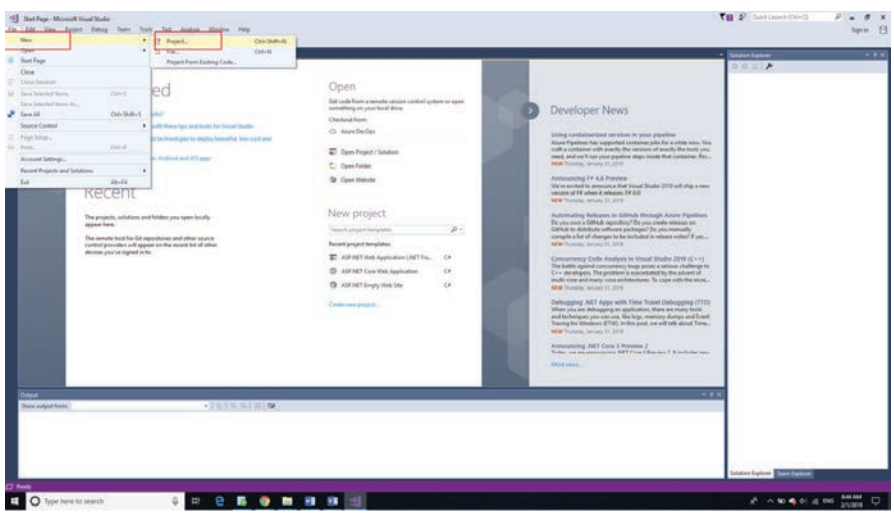
U demonstrativnom primeru datom u ovom poglavlju kreiran je jednostavan veb servis koji vraća podatak tipa String aplikaciji koja konzumira usluge veb servisa. Sam veb servis je hostovan na ASP.NET veb aplikaciji.

U okruženju Visual Studio može se videti i sama SOAP poruka koja se razmenjuje između veb servisa i klijenta koji ga poziva.

Preporuka je da studenti sami probaju da implementiraju ovaj primer i za to im je potreban Visual Studio verzija 2013, 2015 ili 2017.

U vreme pisanja ovog udžbenika bio je aktuelan Visual Studio 2017 i zbog toga je primer implementiran u ovoj verziji.

Korak 1 - Potrebno je pokrenuti Visual Studio i izabrati opciju File -> New -> Project (slika 9.11).



Slika 9.11. Kreiranje SOAP veb servisa – Korak 1

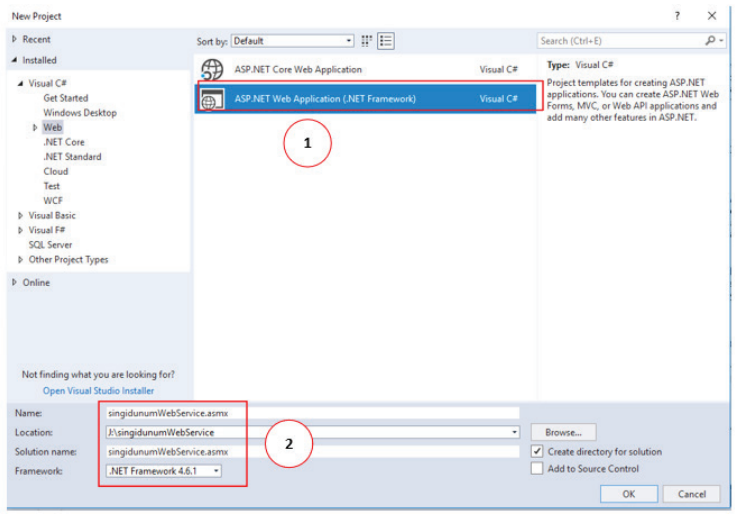
Nakon izbora navedenih opcija, Visual Studio prikazuje okvir za dijalog, gde se definišu specifikacije projekta, što je dato u koraku 2.

Korak 2 – U ovom koraku potrebno je uraditi sledeće:

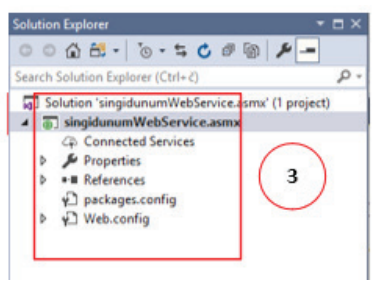
1. potrebno je izabrati C# za veb template ASP.NET veb aplikacije. Projekat mora da bude ovog tipa, kako bi mogao da se implementira veb servis. Visual Studio će sve fajlove i biblioteke koji su neophodni za implementiranje projekta sam da doda.

2. potrebno je dati naziv projektu (u primeru je projekat nazvan singidunumWebService.asmx), izabrati željenu verziju .NET Framework i definisati lokaciju gde će fajlovi projekta da se čuvaju.
3. nakon klika na dugme OK, fajlovi projekta će biti prikazani u komponenti Solution Explorer Visual Studio.

Navedene operacije date su na slikama 9.12 i 9.13.



Slika 9.12. Kreiranje SOAP veb servisa – Korak 2a

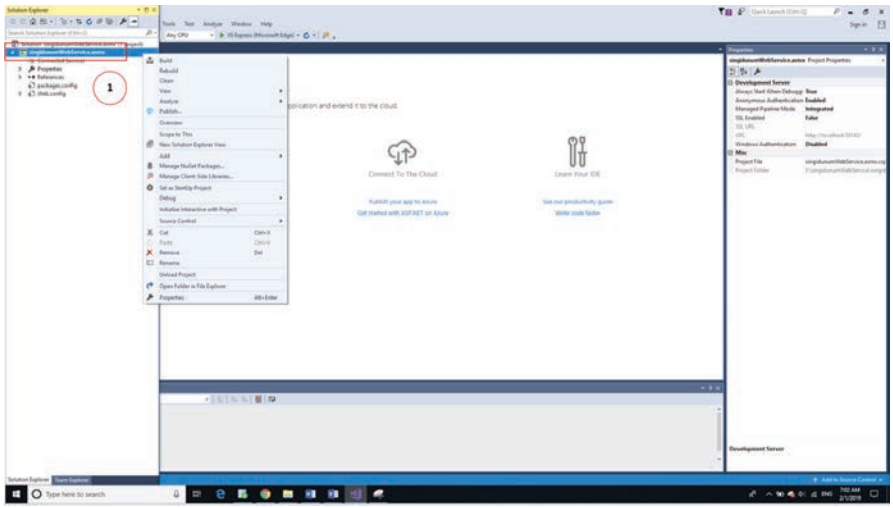


Slika 9.13. Kreiranje SOAP veb servisa – Korak 2b

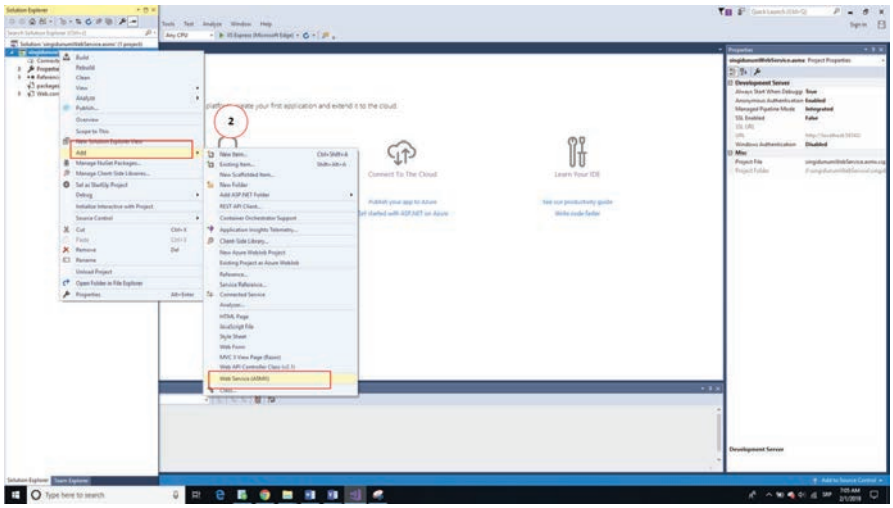
Korak 3 – Dodavanje fajla veb servisa u projekat:

1. desni klik na fajl Projekta
2. izbor opcije add -> Web Service (ASMX)
3. određivanje naziva fajla veb servisa (u ovom primeru fajl je nazvan SingidunumService).

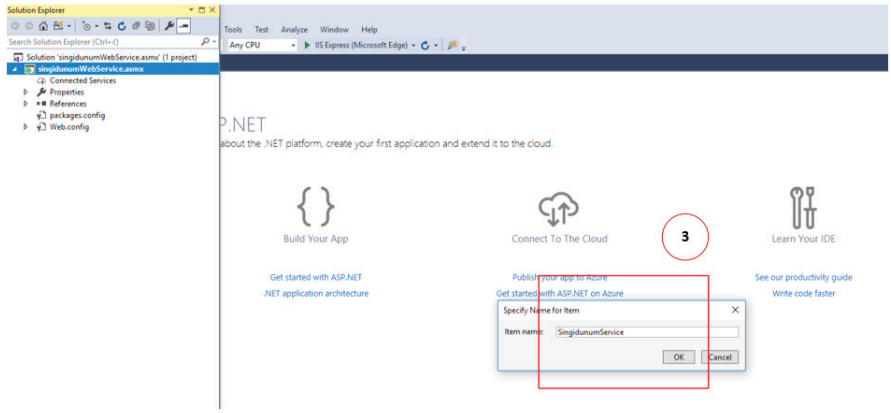
Navedene operacije prikazane su na slikama 9.14, 9.15 i 9.16.



Slika 9.14. Kreiranje SOAP veb servisa – Korak 3a

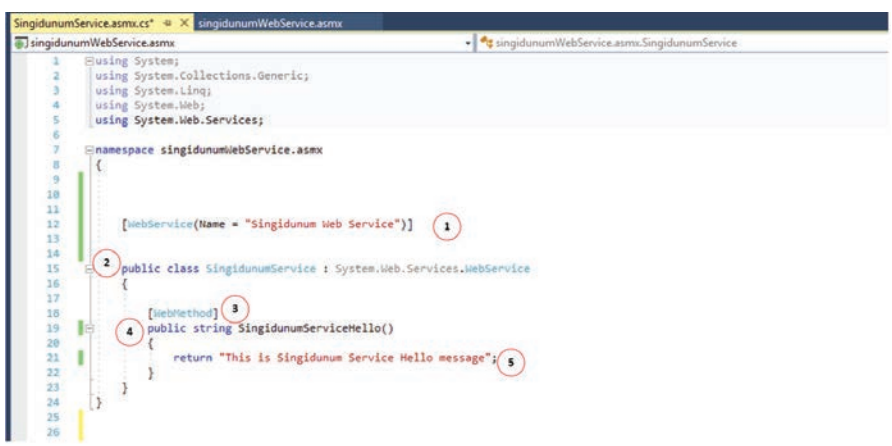


Slika 9.15. Kreiranje SOAP veb servisa – Korak 3b



Slika 9.16. Kreiranje SOAP veb servisa – Korak 3c

Korak 4 – potrebno je dodati kod koji je prikazan na slici 9.17 u fajl Singidunum-Service.asmx.



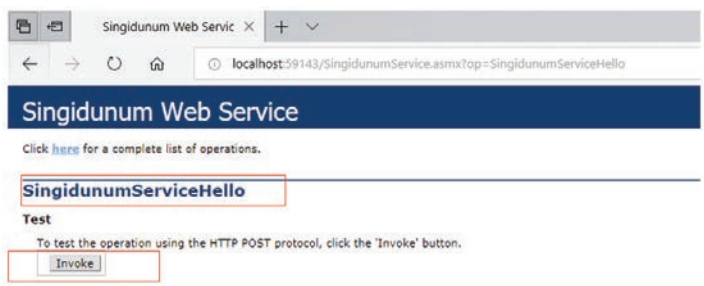
Slika 9.17. Kod fajla SingidunumService.asmx – Korak 4

U nastavku je objašnjen kod koji je dat u fajlu SingidunumService.asmx (slika 9.17).

1. U ovoj liniji koda je definisan naziv veb servisa. Važno je da se definiše naziv veb servisa, zato što na taj način klijentska aplikacija može da poziva veb servis po imenu.
2. Klasa ima funkciju enkapsulacije funkcionalnosti veb servisa. Dakle, klasa obezbeđuje definisanje svih metoda veb servisa koje obezbeđuju određenu funkcionalnost aplikaciji klijentu.

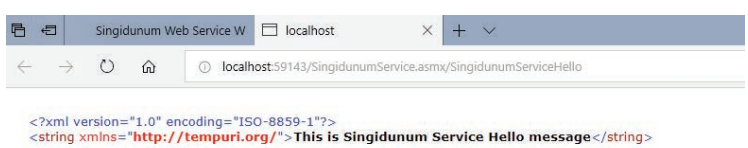
3. [WebMethod] je atribut koji opisuje funkciju veb servisa. U narednom koraku (koraku 4) kreirana je metoda SingidunumServiceHello. Bez atributa [WebMethod] koji se navodi iznad deklaracije funkcije, metodu SingidunumServiceHello ne bi mogla da poziva klijentska aplikacija.
4. U ovoj liniji koda kreirana je metoda SingidunumServiceHello koja vraća jednostavan odgovor tipa String klijentskoj aplikaciji. Ova metoda predstavlja veb servis koji može da koristi klijentska aplikacija.
5. Pomoću naredbe return vraća se podatak („This is Singidunum Service Hello message“) tip String klijentskoj aplikaciji koja poziva veb servis.

Nakon pokretanja aplikacije, u Browser-u se prikazuje veb interfejs, kao što je prikazano na slici 9.18.



Slika 9.18. Interfejs veb servisa

Sa slike 9.18 se jasno vidi da je naziv veb servisa „SingidunumServiceHello“. Klikom na dugme „Invoke“, odgovor veb servisa se prikazuje u veb Browser-u (slika 9.19).



Slika 9.19. Odgovor veb servisa

Kao što je definisano u kodu, odgovor veb servisa je „This is Singidunum Service Hello message“ koji je tipa String.

Visual Studio omogućava prikaz SOAP 1.1 i SOAP 1.2 poruke sa zahtevom i odgovorom, koji se generiše kada klijent pozove veb servis.

SOAP zahtev 1.1 i zahtev 1.2 prikazani su na slikama 9.20 i 9.21, respektivno.

SOAP zahtev 1.1

```
POST /SingidunumService.asmx HTTP/1.1
Host: localhost
Content-Type: text/xml; charset=utf-8
Content-Length: length
SOAPAction: "http://tempuri.org/SingidunumServiceHello"

<?xml version="1.0" encoding="utf-8"?>
<soap:Envelope xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:xsd="http://www.w3.org/2001/XMLSchema" xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
  <soap:Body>
    <SingidunumServiceHello xmlns="http://tempuri.org/" />
  </soap:Body>
</soap:Envelope>
```

Slika 9.20. SOAP zahtev 1.1

SOAP zahtev 1.2

```
POST /SingidunumService.asmx HTTP/1.1
Host: localhost
Content-Type: application/soap+xml; charset=utf-8
Content-Length: length

<?xml version="1.0" encoding="utf-8"?>
<soap12:Envelope xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:xsd="http://www.w3.org/2001/XMLSchema" xmlns:soap12="http://www.w3.org/2003/05/soap-envelope">
  <soap12:Body>
    <SingidunumServiceHello xmlns="http://tempuri.org/" />
  </soap12:Body>
</soap12:Envelope>
```

Slika 9.21. SOAP zahtev 1.2

Objašnjenje koda prikazanog na slikama 9.20 i 9.21 dato je u nastavku.

- Prvi deo SOAP poruke je element konverte. Koverat je element koji enkapsulira sve ostale elemente SOAP poruke.
- SOAP telo je sledeći element u kome se nalaze detalji SOAP poruke.
- Treći element referiše veb servis koji klijent poziva. U ovom slučaju naziv veb servisa je „SingidunumServiceHello“.

SOAP odgovor 1.1 i odgovor 1.2 prikazani su na slikama 9.22 i 9.23, respektivno.

SOAP odgovor 1.1

```
HTTP/1.1 200 OK
Content-Type: text/xml; charset=utf-8
Content-Length: length

<?xml version="1.0" encoding="utf-8"?>
<soap:Envelope xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:xsd="http://www.w3.org/2001/XMLSchema" xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
  <soap:Body>
    <SingidunumServiceHelloResponse xmlns="http://tempuri.org/">
      <SingidunumServiceHelloResult>string</SingidunumServiceHelloResult>
    </SingidunumServiceHelloResponse>
  </soap:Body>
</soap:Envelope>
```

Slika 9.22. SOAP odgovor 1.1

SOAP odgovor 1.2

```
HTTP/1.1 200 OK
Content-Type: application/soap+xml; charset=utf-8
Content-Length: length

<?xml version="1.0" encoding="utf-8"?>
<soap12:Envelope xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:xsd="http://www.w3.org/2001/XMLSchema" xmlns:soap12="http://www.w3.org/2003/05/soap-envelope">
  <soap12:Body>
    <SingidunumServiceHelloResponse xmlns="http://tempuri.org/">
      <SingidunumServiceHelloResult>string</SingidunumServiceHelloResult>
    </SingidunumServiceHelloResponse>
  </soap12:Body>
</soap12:Envelope>
```

Slika 9.23. SOAP odgovor 1.2

U nastavku je dato objašnjenje koda sa SOAP odgovorom 1.1 i odgovorom 1.2 (slika 9.22 i slika 9.23).

1. Prvi deo SOAP poruke sa odgovorom je element koverte. Koverat je element koji enkapsulira sve ostale elemente SOAP poruke.
2. Element SOAP telo je sledeći element u kome se nalaze detalji SOAP poruke.
3. U SOAP odgovoru prikazan je atribut String. Ovaj atribut govori klijentskoj aplikaciji koja je pozvala veb servis da pozvani servis vraća objekat koji je tipa String. Ova informacija je neophodna, zato što klijentska aplikacija mora da zna koji tip podataka da očekuje od veb servisa.

Okruženje Visual Studio takođe omogućava i prikaz HTTP POST zahteva i odgovora.

HTTP POST zahtev prikazan je u kodu ispod.

```
POST /SingidunumService.asmx/SingidunumServiceHello HTTP/1.1
Host: localhost
Content-Type: application/x-www-form-urlencoded
Content-Length: length
```

HTTP POST odgovor je naveden u nastavku.

```
HTTP/1.1 200 OK
Content-Type: text/xml; charset=utf-8
Content-Length: length
<?xml version=”1.0” encoding=”utf-8”?>
<string xmlns=”http://tempuri.org/”>string</string>
```

S obzirom da je u HTTP-u bilo više reči u prethodnim glavama ovog udžbenika, i pod pretpostavkom da su čitaoci upoznati sa HTTP protokolom, objašnjenje prikazanih isečaka koda nije dato.

X
RESTful veb servisi

U savremenim Internet komunikacijama primećen je trend da se sve više koriste REST veb servisi. Kao što je već rečeno, REST je akronim od Representational State Transfer.

REST se koriste za izgradnju veb servisa koji su laki (engl. *lightweight*) u kontekstu korišćenja računarskih resursa, lako održivi (engl. *maintanable*) i skalabilni (engl. *scalable*).

REST se odnosi na arhitekturu servisa. veb servis koji je izgrađen na temeljima REST arhitekture u literaturi se naziva RESTful servis.

Za razliku od SOAP baziranih veb servisa, koji za transport mogu da koriste ili HTTP, ili MQ, ili neki drugi protokol, REST koristi isključivo HTTP protokol za komunikaciju.

S druge strane, dok SOAP koristi isključivo XML za reprezentaciju podataka, REST koristi XML, JSON, YAML, ili pak neki drugi format.

U nastavku su prikazani detaljni RESTful veb servisa.

10.1. Ključni elementi RESTful veb servisa

Od nastanka, veb servisi su se dosta menjali i razvijali. Godine 2002., veb konzorcijum (W3C) je objavio prve definicije WSDL-a i SOAP baziranih veb servisa i brzo je uspostavljen standard implementacije veb servisa.

Godine 2004., veb konzorcijum je objavio definiciju dodatnog standarda veb servisa pod nazivom RESTful. Prethodnih nekoliko godina ovaj standard je postao dosta popularan i danas ga koriste mnogi popularni veb portali, poput Facebook-a i Twitter-a.

REST definiše način pristupa resursima koji se nalaze u određenom okruženju. Na primer, na serveru na Internetu se nalaze važni dokumenti ili multimedijalni fajlovi. U kontekstu REST-a, ovi fajlovi se nazivaju resursi. Ako klijent, na primer veb Browser, hoće da pristupi ovim resursima, klijent treba da pošalje zahtev serveru koji željene resurse hostuje. U ovom primeru REST definiše način na koji klijent može da pristupi ovim resursima (dokumentima i multimedijalnim fajlovima).

Ključni elementi RESTful implementacije su sledeći:

1. *Resursi (engl. resources).* Prvi ključni element RESTful veb servisa je resurs. Neka se pretpostavi da veb aplikacija na serveru čuva podatke o studentima Univerziteta Singidunum i neka je URL na kome se nalazi veb aplikacija http://students.singidunum.ac.rs. Dalje, kako bi se pristupilo podacima (slogovima) studenata pomoću REST-a, može da se izvrši sledeća komanda: http://students.singidunum.ac.rs/student/1. Ovom komandom se zahteva od veb servera da vrati detalje (podatke) o studentu sa rednim brojem (engl. id) 1.
2. *Glagoli zahteva (engl. request verbs).* Pomoću glagola zahteva se opisuje akcija koju klijent želi da izvrši sa resursima. Tako na primer klijentski Browser može da koristi GET glagol kako bi dao instrukcije pristupnoj tački veb servisa da želi da uzme podatke. Osim glagola GET, RESTful veb servisi koriste i druge glagole, poput POST, PUT i DELETE. U ovom slučaju korišćenjem GET glagola nad pristupnom tačkom http://students.singidunum.ac.rs/student/1, klijent naglašava da želi da dobije podatke o određenom studentu. Korisno je da se napomene da glagoli zahteva odgovaraju metodama HTTP protokola.
3. *Zaglavlja zahteva (engl. request headers).* Zaglavlja zahteva se koriste za slanje dodatnih informacija kroz zahtev, kao na primer format odgovora koji klijent očekuje, ili detalji o autorizaciji, i slično.
4. *Telo zahteva (engl. request body).* U pojedinim slučajevima se uz sam zahtev šalju i podaci koji se enkapsuliraju u telu zahteva. Podaci se uglavnom šalju onda kada klijent izdaje POST zahtev. Kroz POST zahtev klijentska aplikacija „govori" veb servisu da želi da doda odgovarajuće resurse na server i u ovom slučaju se u telu zahteva nalaze detalji o resursu koji klijentska aplikacija dodaje.
5. *Telo odgovora (engl. response body).* U telu odgovora se enkapsulira odgovor koji veb servis vraća klijentu. U ovom primeru, na osnovu zahteva http://students.singidunum.ac.rs/student/1, veb servis klijentu vraća podatke o željenom studentu u XML, JSON, ili nekom drugom formatu u telu odgovora.
6. *Status kodovi odgovora (engl. response status codes).* U okviru odgovora koji veb servis vraća klijentu koji ga poziva vraćaju se i kodovi koji opisuju status odgovora. Tako na primer, kod 200 označava da je veb servis uspešno obradio zahtev klijenta i da pritom nije došlo ni do jedne greške.

10.2. RESTful metode

Neka je pretpostavka da se RESTful veb servis nalazi na sledećoj lokaciji: http://students.singidunum.ac.rs/student/.

Kada klijent šalje zahtev RESTful veb servisu iz primera, klijent može da koristi sve standardne HTTP metode (GET, POST, DELETE i PUT).

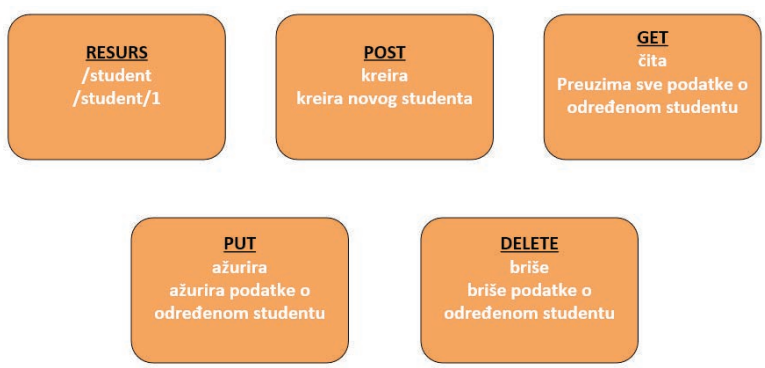
U nastavku su navedeni scenariji korišćenja svakog od navedenih glagola, kada se posmatraju svi objekti (u ovom slučaju lista svih studenata).

1. POST – ovaj zahtev bi se koristio za kreiranje novog studenta korišćenjem RESTful veb servisa.
2. GET – ovaj zahtev bi se koristio za preuzimanje liste svih studenata sa relevantnim podacima korišćenjem RESTful veb servisa.
3. PUT – ovaj zahtev bi se koristio za ažuriranje podataka o svim studentima korišćenjem RESTful veb servisa.
4. DELETE – ovaj zahtev bi se koristio za brisanje podataka o svim studentima korišćenjem RESTful veb servisa.

U nastavku su navedeni scenariji korišćenja svakog od navedenih HTTP glagola, kada se posmatra samo jedan objekat (u ovom slučaju student sa rednim brojem 1).

1. POST – u ovom slučaju ovakav zahtev bi generisao grešku, zato što je student sa rednim brojem 1 već definisan.
2. GET - ovaj zahtev bi se koristio za preuzimanje podataka o studentu sa rednim brojem 1 korišćenjem RESTful veb servisa.
3. PUT - ovaj zahtev bi se koristio za ažuriranje podataka o studentu sa rednim brojem 1 korišćenjem RESTful veb servisa.
4. DELETE – ovaj zahtev bi se koristio za brisanje podataka o studentu sa rednim brojem 1 korišćenjem RESTful veb servisa.

Na slici 10.1 je prikazan dijagram na kom su dati osnovni REST glagoli (GET, POST, PUT i DELETE) sa objašnjenjima funkcija koje obavljaju.



Slika 10.1. Primeri upotrebe GET, POST, PUT i DELETE metoda

10.3. Razlozi za sve veću upotrebu RESTful veb servisa

Među brojnim razlozima za korišćenje RESTful veb servisa, kao neki od njih navode se sledeći:

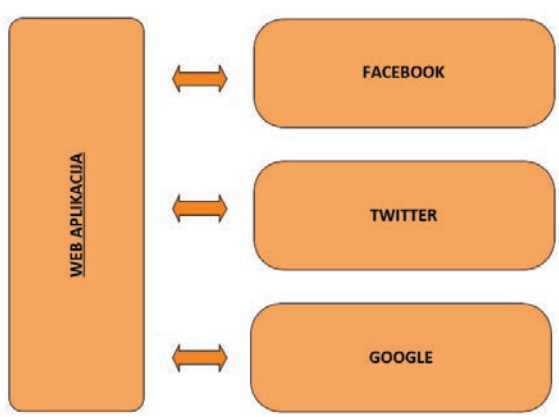
1. heterogenost tehnologija, programskih jezika i okruženja,
2. potreba da svaka aplikacija radi na „uređajima“ i
3. potreba da je svaki servis prilagođen za izvršavanje na klaud okruženju.

Jedan od glavnih razloga za korišćenje RESTful veb servisa u modernoj literaturi se navodi činjenica da se *savremene aplikacije razvijaju u okruženjima različitih tehnologija, programskih jezika i okruženja*. Iz ove činjenice jasno proističu razlozi za upotrebu RESTful veb servisa.

Prvo, veb servisi omogućavaju komunikaciju heterogenih aplikacija. Drugo, uz pomoć RESTful veb servisa, veb aplikacije mogu da se hostuju na različitim okruženjima. Tako na primer, ako su izgrađene u skladu sa REST arhitekturom, aplikacije koje su hostovane na Windows okruženjima i Linux mogu međusobno da komuniciraju i da razmenjuju podatke kao da se nalaze na istoj, homogenoj platformi.

Na kraju, nije važna platforma na kojoj su veb aplikacije hostovane, niti je važan programski jezik u kojoj su razvijane, jedino je važan rezultat, a to je mogućnost komunikacije i razmene informacija. RESTful veb servisi nude upravo ovu mogućnost, koja naročito dolazi do izražaja u uslovima heterogenosti programskih jezika, platformi i tehnologija.

Na slici 10.2 je dat primer veb aplikacije, u čijim je specifikacijama naveden zahtev da podržava komunikaciju sa Facebook, Twitter i Google servisima. Logično je pretpostaviti da ako aplikacija treba da komunicira sa Facebook, Twitter i Google servisima, da ta aplikacija treba da „poznaje“ jezik ovih servisa.



Slika 10.2. Veb aplikacija i komunikacija sa Facebook, Twitter i Google servisima

Ovo dalje znači da je potrebno da se u veb aplikaciju inkorporira jezik za komunikaciju sa svakim od navedenih servisa, što predstavlja „noćnu moru" za svakog programera. Na sreću, Facebook, Twitter i Google javnom Internetu izlažu svoje funkcionalnosti u formi RESTful veb servisa, što omogućava bilo kojoj klijentskoj aplikaciji da poziva ovako definisane veb servisa putem REST-a.

Kao drugi veći razlog za korišćenje RESTful veb servisa u literaturi se navodi *potreba da se skoro svaka veb aplikacija izvršava na uređajima, poput pametnih telefona, netbook računara, pametnih televizora, itd*. S obzirom da REST koristi HTTP protokol, bilo koja aplikacija koja podržava veb komunikaciju može da koristi RESTful veb servise, bez obzira da li je ta aplikacija pisana za Andorid, iOS, ili neku drugu platformu.

Konačno, kao treći veći razlog za upotrebu RESTful veb servisa navodi se činjenica da *se u poslednje vreme skoro sve migrira na klaud okruženje*. Savremene aplikacije se sve više „pomeraju" na Microsoft Azure, Amazon AWS (Amazon Web Services), ili na neko drugo klaud okruženje. Svi veći klaud provajderi (engl. *Cloud Service Providers* – CSPs) obezbeđuju veliki broj API-ja (*Application Programming Interface*) koji se zasnivaju na REST arhitekturi. S obzirom da sve klaud – bazirane arhitekture funkcionišu na osnovu REST principa, optimalno je da se veb servisi razvijaju na osnovu REST arhitekture, kako bi najbolje mogli da koriste klaud usluge.

10.4. RESTful arhitektura, principi i ograničenja

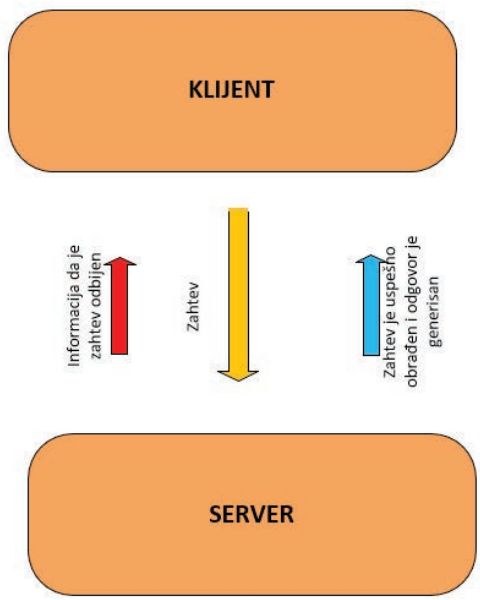
Aplikacija ili arhitektura koja implementira RESTful stil ima sledeća svojstva:

1. stanje i funkcionalnosti su podeljeni na distribuirane resurse, što znači da je svaki resurs dostupan preko standardnih HTTP komandi GET, POST, PUT ili DELETE. Dakle, ako neko želi da preuzme resurs, resurs može da se preuzme izvršavanjem GET zahteva. U slučaju da je potrebno da se izmeni postojeći, ili da se doda novi resurs na server, izvršava se POST zahtev, dok se zahtevom DELETE briše resurs sa servera,
2. arhitektura je tipa klijent/server. RESFful stil podrazumeva tipičnu klijent/server arhitekturu, gde je server obično veb server, dok klijent može da bude bilo klijentska aplikacija koja je instalirana u lokalu, ili je portabilna, bilo tanki klijent (engl. *thin client*) kao što je veb Browser,
3. REST ne pampti trenutno stanje aplikacije (engl. *stateless*). Na primer, ako klijent izvrši naredbu DELETE i izbriše resurs sa servera, informacija o brisanju neće biti prosleđena sledećem zahtevu. Ako klijent želi da proveri da li je resurs izbrisan, klijent mora da izvrši dodatni GET zahtev.
4. arhitektura se sastoji iz više slojeva (engl. *layered architecture*) i
5. podržano je keširanje (engl. *caching enabled*).

Osim navedenog, da bi veb servis dobio epitet RESTful, on mora da bude kompatibilan sa sledećim standardima i karakteristikama: klijent - server arhitektura, odsustvo mehanizma koji pamti stanje, sposobnost keširanja, slojeviti sistem i implementira standardni komunikacioni interfejs.

Navedene karakteristike su detaljnije opisane u nastavku.

RESTful klijent - server je jedan od osnovnih zahteva svake REST arhitekture. Server hostuje RESTful veb servis koji obezbeđuje potrebnu funkcionalnost klijentu. Klijent šalje zahtev veb servisu koji je implementiran na serveru, kada servis ili odbija zahtev, ili ga izvršava i generiše odgovarajući odgovor koji klijentu šalje nazad (slika 10.3).



Slika 10.3. RESTful klijent – server

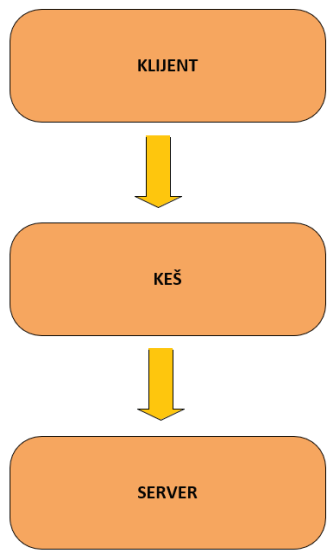
Odsustvo mehanizma koji pamti stanje odnosi se na to da sam klijent mora da osigura da će sve potrebne informacije biti obezbeđene serveru. Ove informacije su neophodne zato što samo na taj način server može da obradi zahtev klijenta i da klijentu pruži odgovor (potrebne funkcionalnosti). Server u ovom slučaju ne treba da čuva nikakve informacije između različitih zahteva klijenta.

Ovakav scenario može da se opiše na sledeći način: klijent postavi pitanje i server mu odgovori. Klijent zatim postavi drugo pitanje. Server se ne seća prvog pitanja i odgovora i odgovara klijentu na drugo pitanje nezavisno od prvog.

Koncept keširanja rešava problem odsustva mehanizma koji pamti stanja koji je opisan. S obzirom na to da je svaka serija zahteva – odgovora između klijenta i servera nezavisna, može da se desi scenario gde nakon određenog vremena klijent ponovo pita server za isti resurs.

Svaki put kada klijent postavi pitanje serveru, razmena pitanja i odgovora između klijenta i servera se odvija preko računarske mreže, što može značajno da poveća mrežni saobraćaj. Mehanizam keširanja se implementira na strani klijenta. Keširanjem se čuvaju zahtevi klijenta koji su ranije već bili slati serveru. Dakle, primenom ovog mehanizma, u situacijama kada klijent postavlja isti zahtev serveru ponovo, umesto da „ide" do servera, klijent „ide" do skladišta za keširanje, gde dobija potrebne informacije. Na ovaj način se štedi u mrežnom saobraćaju i server se manje opterećuje.

Koncept keširanja prikazan je na slici 10.4.



Slika 10.4. Mehanizam keširanja

RESTful je slojeviti sistem, što znači da dodatni nivo (sloj), kao što je međusloj (engl. *middleware layer*) može da se doda između klijenta i servera koji hostuje RESTful veb servis. Međusloj je sloj koji predstavlja sloj poslovne logike.

U pojedinim slučajevima, međusloj može da implementira i dodatni servis sa kim klijent komunicira pre nego što pošalje zahtev veb servisu. Važan zahtev je da međusloj bude transparentan, da se komunikacija između klijenta i servera ne bi remetila.

RESTful veb servis implemenira standardni komunikacioni interfejs koji se bazira na HTTP protokolu. RESTful funkcioniše iznad HTTP veb sloja i koristi sledeće tipove zahteva za interakciju sa resursima na serveru:

- POST – kreira resurs na serveru,
- GET – preuzima resurs sa servera,
- PUT – menja stanje, ili ažurira resurs na serveru i
- DELETE – briše resurs sa servera.

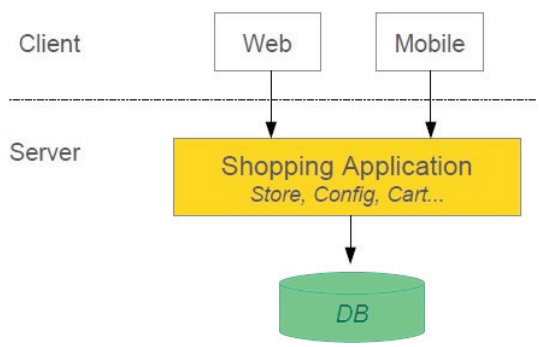
XI
Kratak uvod u Spring Boot Framework

S pogledom na činjenicu da su u narednoj glavi dati primeri RESTful veb servisa koji su implementirani primenom Spring Boot Framework tehnologije, u ovoj glavi je dat kratak pregled osnovnih Spring Boot koncepata.

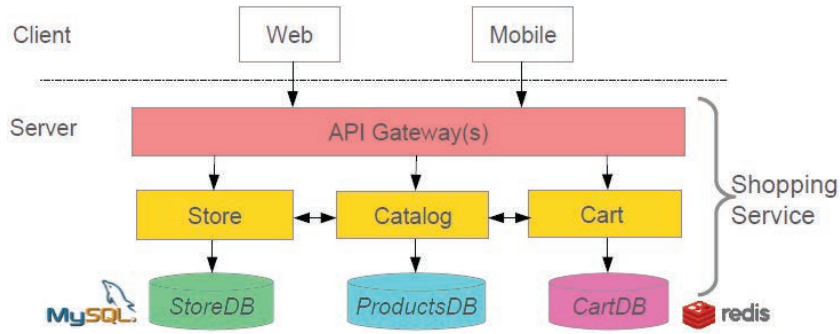
Spring Boot framework je open source radni okvir za razvoj Java aplikacija koje se izvršavaju u backend delu takozvanih enterprise aplikacija, sa ciljem razvoja mikro servisa (engl. *micro services*). Spring Boot je zasnovan na Java programskom jeziku i dizajnerskom obrascu injekcije zavisnosti. Spring Boot je razvijen od strane Pivotal Team-a i koristi se za razvijanje samostalnih Spring aplikacija.

Arhitektura mikro servisa je arhitektura koja dozovljava programerima da razvijaju i da implementiraju servise nezavisno. Svaki servis ima svoje procese koji omogućava laku podrški biznis aplikacijama.

Arhitekture aplikacije za online kupovinu bez i sa mikro servisa prikazane su na slikama 11.1 i 11.2.



Slika 11.1. Arhitektura aplikacije za online kupovinu bez mikro servisa (slika preuzeta sa: http://amilasilva88.blogspot.com)



Slika 11.2. Arhitektura aplikacije za online kupovinu sa mikro servisima (slika preuzeta sa: http://amilasilva88.blogspot.com)

Kao neke od prednosti mikro servisa u literaturi se navode sledeće:

- lakše razvijanje,
- prosta skalabilonst,
- kompatibilni su sa konceptom „kontejnera“,
- minimalna konfiguracija i
- brže vreme razvijanja.

Spring Boot predstavlja dobru platformu za Java developer-e, na kojoj može da se razvija samostalna i proizvoljna Spring aplikacija, koja je prekonfigurisana. Jedino što je potrebno jeste da se aplikacija pokrene. Spring Boot takođe omogućava razvoj aplikacije od tzv. minimalne konfiguracije bez potrebe za implementacijom svih Spring funkcionalnosti.

Na ovaj način developer-i štede svoje vreme i ne opterećuju se sa konfigurisanjem okruženja aplikacije, već se fokusiraju na poslovnu logiku i na funkcionalnosti koje aplikacija pruža.

Među brojnim prednostima Spring Boot-a izdvajaju se sledeće:

- lako razumljivo i razvijanje spring aplikacija,
- povećava produktivnost i
- smanjuje vreme razvoja proizvoda.

Kao neki od ciljeva razvoja Spring Boot aplikacija navode se sledeći:

- izbegavanje kompleksnih xml konfiguracija
- razvijanje personalizovanih aplikativnih proizvoda
- smanjivanje vremena razvoja i korišćenje nezavisne aplikacije i
- lakši način razvijanja aplikacije.

Developer-i Spring Boot-a navode sledeće razloge za korišćenje ove tehnologije:

- nudi fleksibilan način konfiguracije Java Beans-a, XML konfiguratora i transakcionih baza podataka,
- u Spring Boot-u se sve bazira na autokonfiguraciji i ne postoji potreba za manuelnom konfiguracijom
- sadrži Embedded Servlet Container i
- olakšava nezavisno upravljanje

11.1. Konfigurisanje Spring Boot aplikacije

Spring Boot automatski konfiguriše aplikaciju na osnovu zavisnosti koje su dodate u projekat korišćenjem anotacije @EnableAutoConfiguration. Tako na primer, ako se MySQL sistem za upravljanje relacionim bazama podataka nalazi u varijabli putanje projekta (engl. *classpath variable*), i ako u projektu nije konfigurisana ni jedna konekcija sa bazom podataka, Spring Boot će automatski da konfiguriše memorijsku bazu podataka (engl. *in-memory database*).

Ulazna tačka u Spring Boot aplikaciju je klasa u kojoj je definisana @SpringBootApplication anotacija i metoda main().

Spring Boot takođe ima mogućnost skeniranja svih komponenti koje se nalaze u projektu korišćenjem anotacije @ComponentScan.

11.2. Spring Boot Starter projekti

Konfigurisanje i upravljanje zavisnostima predstavlja izazov i težak zadatak, posebno ako su u pitanju veliki projekti. Spring Boot rešava ovaj problem tako što obezbeđuje predefinisani skup zavisnosti u formi tzv. početnih Spring Boot projekata (engl. *Spring Boot Starter Projects*).

Tako na primer, ako je potrebno u projektu da se koristi Spring i JPA za pristup bazi podataka, developer samo treba da uključi spring-boot-starter-data-jpa zavisnost u svoj projekat.

Napominje se da svih Spring Boot Starter projekti imaju prate istu konvenciju imenovanja, gde nazivi svih projekata počinju sa spring-boot-starter-*, gde * označava tip aplikacije (projekta).

U nastavku su navedeni neki od primera Spring Boot Starter projekata.

Spring Boot Starter Actuator se koristi za lakše praćenje i upravljanje aplikacijom (projektom). Kod je prikazan u nastavku.

```
<dependency>
   <groupId>org.springframework.boot</groupId>
   <artifactId>spring-boot-starter-actuator</artifactId>
</dependency>
```

Spring Boot Starter Security se koristi za unapređenje sigurnosti Spring Boot projekta. U nastavku je naveden kod.

```
<dependency>
   <groupId>org.springframework.boot</groupId>
   <artifactId>spring-boot-starter-security</artifactId>
</dependency>
```

Spring Boot Starter Web, čiji je kod prikazan u nastavku, se koristi za kreiranje pristupnih tačaka RESTful veb servisa.

```
<dependency>
   <groupId>org.springframework.boot</groupId>
   <artifactId>spring-boot-starter-Web</artifactId>
</dependency>
```

Spring Boot Statter Thyme Leaf omogućava lakše kreiranje veb aplikacije. Kod je prikazan u nastavku.

```
<dependency>
   <groupId>org.springframework.boot</groupId>
   <artifactId>spring-boot-starter-thymeleaf</artifactId>
</dependency>
```

Spring Boot Starter Test se široko upotrebljava za potrebe kreiranja Test slučajeva (kod je dat u nastavku).

```
<dependency>
   <groupId>org.springframework.boot</groupId>
   <artifactId>spring-boot-starter-test<artifactId>
</dependency>
```

11.3. Osnovne Spring Boot anotacije

U ovom delu ukratko su opisane osnovne Spring Boot anotacije: @EnableAutoConfiguration, @SpringBootApplication i @ComponentScan.

Pomoću mehanizma Spring Boot za automatsku konfiguraciju, Spring aplikacija koja se, na osnovu JAR zavisnosti koji su dodati u projekat, automatski konfiguriše. Kao što je već i navedeno, ako se na primer MySQL sistem za upravljanje relacionim bazama podataka nalazi u varijabli putanje projekta, i ako u projektu nije konfigurisana ni jedna konekcija sa bazom podataka, Spring Boot će automatski da konfiguriše memorijsku bazu podataka.

Da bi se mehanizam automatske konfiguracije omogućio, potrebno je dodati anotaciju @EnableAutoConfiguration, ili @SpringBootApplication u fajl u kome se nalazi main() metoda (u kod glavne klase projekta).

U nastavku je naveden primer koda za automatsku konfiguraciju.

```
import org.springframework.boot.SpringApplication;
import  org.springframework.boot.autoconfigure.EnableAutoConfigura-
tion;

@EnableAutoConfiguration
public class DemoApplication {
   public static void main(String[] args) {
      SpringApplication.run(DemoApplication.class, args);
   }
}
```

Ulazna tačka u Spring Boot aplikaciju je klasa u kojoj je definisana anotacija @SpringBootApplication. U okviru ove klase treba da bude definisana metoda main().

Anotacija @SpringBootApplication obuhvata sledeće mehanizme: automatsku konfiguraciju (engl. *auto-configuration*), skeniranje komponenti (engl. *component scan*) i Spring Boot konfiguraciju (engl. *Spring Boot Configuration*).

Ako se u projekat doda anotacija @SpringBootApplication nema potrebe za dodavanjem anotacija @EnableAutoConfiguration, @ComponentScan i @SpringBootConfiguration.

U nastavku je naveden primer koda.

```
import org.springframework.boot.SpringApplication;
import   org.springframework.boot.autoconfigure.SpringBootApplica-
tion;

@SpringBootApplication
public class DemoApplication {
   public static void main(String[] args) {
      SpringApplication.run(DemoApplication.class, args);
   }
}
```

Spring Boot aplikacija skenira sve binove i deklaracije paketa koje aplikacija koristi.

Da bi Spring Boot mogao automatski da skenira sve komponente projekta potrebno je dodati @ComponentScan anotaciju u odgovarajući fajl Spring Boot projekta.

Ispod je dat primer koda.

```
import org.springframework.boot.SpringApplication;
import org.springframework.context.annotation.ComponentScan;

@ComponentScan
public class DemoApplication {
   public static void main(String[] args) {
      SpringApplication.run(DemoApplication.class, args);
   }
}
```

XII
Primeri i implementacije RESTful veb servisa

U ovoj glavi udžbenika data su tri primera RESTful veb servisa koji su kreirani primenom Spring Boot tehnologije.

Za praćenje i samostalnu implementaciju primera potrebne su sledeće tehnologije i razvojna okruženja:

- JDK 8 ili viša verzija
  URL: https://www.oracle.com/technetwork/java/javase/downloads/
- Eclipse IDE Neon, Oxygen, Photon ili 2018-12
  URL: https://www.eclipse.org/downloads/
- IntelliJ IDEA IDE
  URL: https://www.jetbrains.com/idea/

Primeri su demonstrirani u okruženju Windows operativnog sistema – analogno je i u Linux okruženju.

Za preuzimanje Maven zavisnosti tokom setovanja projekata potrebna je Internet konekcija.

U praksi se koriste makar tri načina za kreiranje Spring Boot aplikacija u Java tehnologiji, i to:

- upotrebom veb alata pod nazivom Spring Boot Initializer,
- alatke konzolnog interfejsa (eng. command line interface – CLI) i
- Maven alata za kreiranje i upravljanje Java projektima u kombinaciji sa integrisanim razvojnim okruženjem kao što su Eclipse ili InteliiJ IDEA.

Preporuka je da osim čitanja studenti i sami implementiraju prikazane primere (RESTful Web servise).

U svim navedenim primerima, projekat se izvršava u lokalu (localhost), na portu 8080, gde se pokreće embedovani Tomcat server.

12.1. Primer 1: Implementacija projekta RESTful veb servisa pomoću Spring Initializr alata u Eclipse razvojnom okruženju

U ovom primeru prikazana je implementacija jednostavnog RESTful veb servisa bez baze podataka korišćenjem Spring Initializr alata u Eclipse razvojnom okruženju.

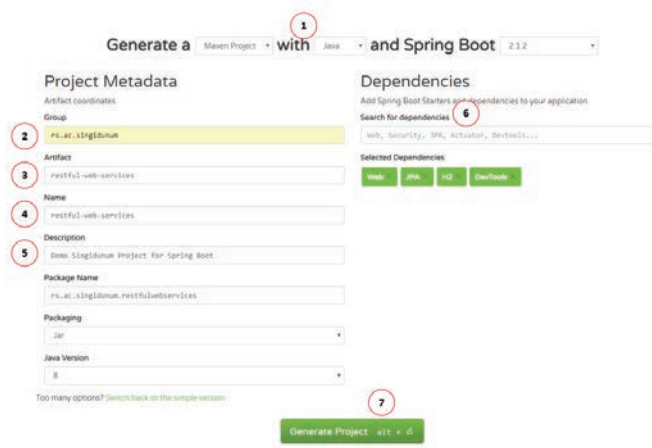
Spring Initializr je alat koji olakšava implementaciju Spring baziranih aplikacija, tako što se na osnovu definisanih zahteva koje će buduća Spring aplikacija imati generiše početni projekat, koji se zatim jednostavno importuje (uvozi) u određeno razvojno okruženje.

Kao što je navedeno, u ovom primeru je prikazano generisanje početnog projekta pomoću Spring Initializr alata, koji je importovan u Eclipse integrisano razvojno okruženje.

Prvo je potrebno otići na zvanični Spring Initializr veb sajt (URL: https://start.spring.io/), gde se konfigurišu osnovna podešavanja Spring projekta RESTful veb servisa kroz sledeće jednostavne korake:

1. Prvi zahtevi su da projekat bude Maven u Java tehnologiji i Spring Boot verziji 2.1.2 (za vreme pisanja ovog udžbenika verzija 2.1.2 je bila poslednja stabilna verzija).
2. Group: rs.ac.singidunum (naziv grupe je isto što i naziv glavnog paketa)
3. Artifact: restful-Web-services (artifact je naziv podpaketa u kome se nalaze osnovne Java klase projekta).
4. Name: restful-Web-services (u ovom polju se definiše naziv projekta).
5. Description: Demo Singidunum Project for Spring Boot (u ovom polju se navodi opis projekta).
6. Dodavanje zavisnosti: veb, JPA, H2, DevTools (vrši se dodavanje Maven zavisnosti u projekat).
7. Klikom na dugme Generate Project generiše se Spring Boot projekat prema željenim specifikacijama koji se preuzima u formi zip arhive.

Screenshot navedenih koraka prikazan je na slici 12.1.



Slika 12.1. Konfigurisanje Spring Initializr projekta

U ovom primeru, radi demonstracije funkcionalnosti Spring Initializr alata, dodate su i neke zavisnosti koje neće biti korišćene u projektu (JPA, H2 i DevTools).

H2 zavisnost se odnosi na memorijsku relacionu bazu podataka koja je pisana u Java programskom jeziku i koja može da se ugradi u bilo koju Java aplikaciju i da se izvršava u klijent-server modu.

Zavisnost DevTools se odnosi na grupu razvojnih alata koji olakšavaju posao programerima. Glavni cilj ove zavisnosti (modula) je unapređenje i pojednostavljenje rada developera na Spring Boot aplikacijama.

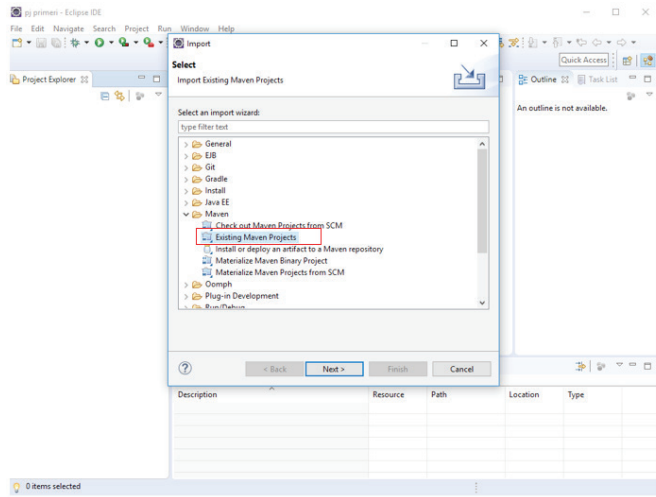
JPA (Java Persistance API) je Java specifikacija za pristupanje, čuvanje i upravljanje podacima između Java objekata/klasa i relacione baze podataka. Drugim rečima, JPA se koristi za objekt-relaciono mapiranje (engl. Object Relational Mapping – ORM), koje omogućava da se podaci konvertuju iz objektnog tipa u relacioni tip, i obrnuto. JPA tehnologija je pokazana u trećem RESTful primeru, gde je implementirana jednostavna baza podataka.

Nakon što se fajlovi projekta u formi zip arhive preuzmu (u ovom primeru naziv zip arhive je restful-Web-services.zip), potrebno je raspakovati ovu arhivu i pokrenuti Eclipse razvojno okruženje.

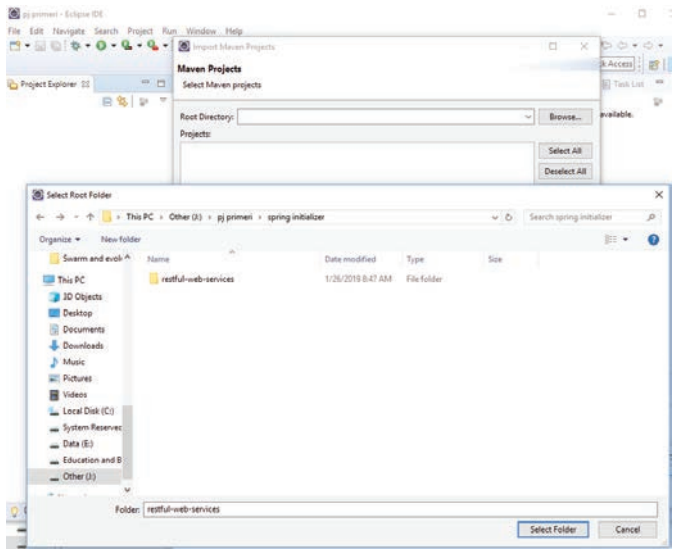
U Eclipse-u je potrebno izabrati opciju import-> Maven -> Existing Maven Project, koja omogućava importovanje postojećeg Maven projekta, i označiti folder u kome se nalazi fajlovi projekta koji je preuzet sa start.spring.io veb sajta. Konačno, klikom na dugme Finish, proces importovanja počinje.

Navedeni koraci prikazani su na slikama 12.2 – 12.4.

Napominje se da je, nakon izvršavanja navedenih koraka, potrebno neko vreme kako bi Eclipse projekat bio inicijalizovan, zato što se tokom ovog procesa preuzimaju i instaliraju sve Maven zavisnosti u lokalni Maven repozitorijum.

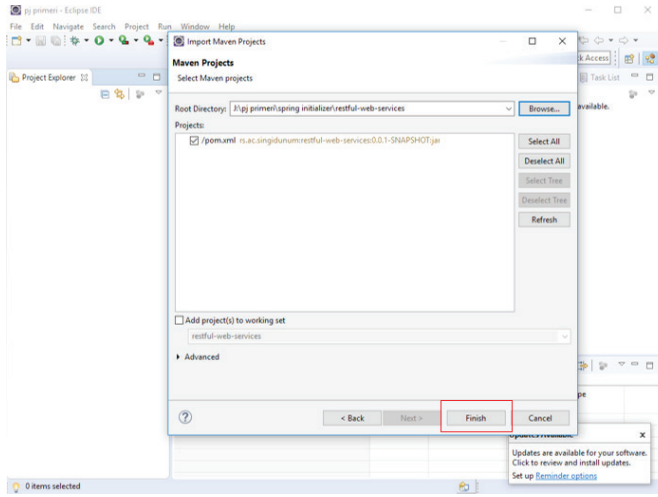


Slika 12.2. Import Maven projekta u Eclipse (1)

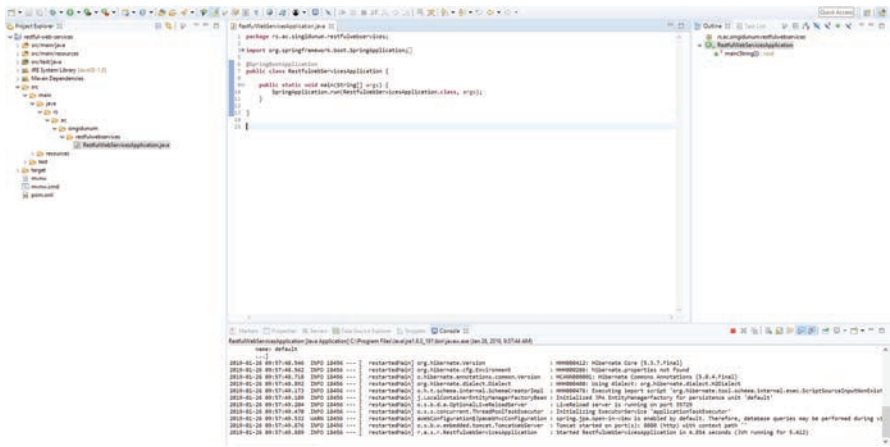


Slika 12.3. Import Maven projekta u Eclipse (2)

Kada se proces importovanja završi, pokretanjem metode main(), koja se nalazi u glavnoj klasi Spring Boot aplikacije (u ovom slučaju to je klasa RestfulWebServicesApplication), REST veb servis se pokreće i ugrađeni Apache Tomcat server se pokreće na podrazumevanom portu 8080 (slika 12.5).



Slika 12.4. Import Maven projekta u Eclipse (3)



Slika 12.5. Pokretanje veb servisa

Osnovna Maven podešavanja

Prema podrazumevani podešavanjima, lokalni Maven repozitorijum se nalazi na sledećoj lokaciji:

- U okruženju Linux/Unix/Mac OS X - ~/.m2/repository
- U okruženju Windows OS – c:\users\{user_name}\.m2\repository

U slučaju da je nepoznata lokacija lokalnog Maven repozitorijuma, u Windows command prompt ili power shell okruženju, je potrebno izvršiti sledeću naredbu, kako bi repozitorijum bio lociran:

```
D:\> mvn help:evaluate -Dexpression=settings.localRepository
[INFO] Scanning for projects...
[INFO]
[INFO] ------------------< org.apache.maven:standalone-pom >-----
--------------
[INFO] Building Maven Stub Project (No POM) 1
[INFO] --------------------------------[ pom ]------------------
--------------
[INFO]
[INFO] --- maven-help-plugin:3.1.1:evaluate (default-cli) @ stand-
alone-pom ---
[INFO]  No  artifact  parameter  specified,  using  'org.apache.
maven:standalone-pom:pom:1' as project.
[INFO]
```

```
G:\MavenRepo
[INFO] ------------------------------------------------------------
--------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------
--------------
[INFO] Total time:  16.302 s
[INFO] Finished at: 2019-01-28T06:28:38+01:00
[INFO] ------------------------------------------------------------
--------------
```

U navedenom primeru, lokalni Maven repozitorijum se nalazi na lokaciji `G:\MavenRepo`.

Izvršavanjem sledeće komande može da se vidi lokacija gde je Maven instaliran:

```
G:\ mvn -version
Apache Maven 3.6.0 (97c98ec64a1fdfee7767ce5ffb20918da4f719f3; 2018-
10-24T20:41:47+02:00)
Maven home: G:\apache-maven-3.6.0\bin\..
Java version: 1.8.0_191, vendor: Oracle Corporation, runtime: C:\
Program Files\Java\jdk1.8.0_191\jre
Default locale: en_US, platform encoding: Cp1252
OS name: "windows 10", version: "10.0", arch: "amd64", family:
"windows"
```

U ovom slučaju je glavni Maven direktorijum (Maven Home) `G:\apache-maven-3.6.0`.

Ako je potrebno promeniti podrazumevani direktorijum Maven repozitorijuma, potrebno je izvršiti promene u fajlu settings.xml, koji se u ovom slučaju nalazi na lokaciji G:\apache-maven-3.6.0\conf\settings.xml.

Izmene u Maven konfiguracionom fajlu settings.xml koje se odnose na lokaciju lokalnog Maven repozitorijuma prikazane su ispod.

```
<settings xmlns="http://maven.apache.org/SETTINGS/1.0.0"
          xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:schemaLocation="http://maven.apache.org/SETTINGS/1.0.0
http://maven.apache.org/xsd/settings-1.0.0.xsd">
  <!-- localRepository
   | The path to the local repository maven will use to store ar-
tifacts.
   |
   | Default: ${user.home}/.m2/repository
    -->
<localRepository>G:/MavenRepo</localRepository>
```

12.1.1. Implementacija pokaznog HelloWorld primera

Cilj pokaznog HelloWorld primera je da se napravi uprošćeni veb servis koji prikazuje jednostavnu poruku na ekranu.

U paketu rs.ac.singidunum.restfulWebservices kreira se jednostavna klasa pod nazivom HelloWorldBean čiji je kod prikazan ispod.

```
package rs.ac.singidunum.restfulWebservices;

public class HelloWorldBean {

    private String message;

    // constructor
    public HelloWorldBean(String message) {
        this.message = message;
    }


    public void setMessage(String message) {
        this.message = message;
    }
```

```
    // We must have getter for this to work!
    public String getMessage() {
        return this.message;
    }

    @Override
    public String toString(){
      return String.format("HelloWorldBean [message=%s]",message);
    }
}
```

Kao što se vidi iz prikazanog koda, klasa HelloWorldBean ima jedan konstruktor kojim se prosleđuje jedan argument (message) koji je tipa String.

Nakon toga pravi se jednostavna kontroler klasa pod nazivom HelloWorldController, koja demonstrira upotrebu HTTP GET metode.

Ideja je da kada se pozove URI REST resursa http://localhost:8080/hello-world da se ispisuje jednostavna pozdravna poruka na ekranu.

Implementacija ove relativno nesofisticirane REST kontroler klase HelloWorldController data je u kodu ispod.

```
package rs.ac.singidunum.restfulWebservices;

import org.springframework.Web.bind.annotation.RequestMapping;
import org.springframework.Web.bind.annotation.RequestMethod;
import org.springframework.Web.bind.annotation.RestController;

@RestController
public class HelloWorldController {

    @RequestMapping(method= RequestMethod.GET,path="/hello-world")
    public String helloWorld(){
        return "Hello World";
    }
}
```

U primeru je korišćena @RequestMapping anotacija koja mapira bilo GET, bilo POST, bilo neki drugi HTTP zahtev na određenu metodu. Drugim rečima, kada se (u ovom slučaju) izda HTTP GET zahtev za resursom čiji je URI /hello-world, izvršava se Java metoda helloWorld() koja rukuje (engl. handle) ovim zahtevom. Zbog toga se u argumentima ove anotacije prvo navodi koji je metod u pitanju, a zatim i URI putanja do željenog resursa.

Umesto @RequestMapping anotacije mogla je da se koristi anotacija @GetMapping (path=”/hello-world”), koja implicitno podrazumeva GET HTTP zahtev.

Pokretanjem veb servisa i pozivanjem URI-ja http://localhost:8080/hello-world prikazuje se pozdravna HelloWorld poruka kao što je prikazano na slici 12.6.



Slika 12.6. Pozdravna HelloWorld poruka

U ovom primeru HTTP GET zahtev vraća običan podatak tipa String. Da bi veb servis vratio Spring bean u podrazumevanom JSON formatu, u klasu HelloWorldController dodaje se sledeća metoda:

```
@GetMapping(path=”hello-world-bean”)
    public HelloWorldBean helloWorldBean()
    {
        return new HelloWorldBean(“Hello World”);
    }
```

Sada se pozivanjem URI-ja http://localhost:8080/hello-world-bean prikazuje HelloWorldBean instanca u JSON formatu kao što je prikazano na slici 12.7.



Slika 12.7. Prikaz HelloWorldBean instance u JSON formatu

Konačno, za kraj pokaznog HelloWorld primera kreirana je još jedna HTTP GET metoda u HelloWorldController klasi, gde se koristi i promenljiva putanje (engl. path variable), na koju je mapirana URl adresa: hello-world-bean/path-variable/.

U nastavku je da kompletan kod HelloWorldController klase nakon navedenih izmena.

```
package rs.ac.singidunum.restfulWebservices;

import org.springframework.Web.bind.annotation.GetMapping;
import org.springframework.Web.bind.annotation.PathVariable;
import org.springframework.Web.bind.annotation.RequestMapping;
import org.springframework.Web.bind.annotation.RequestMethod;
import org.springframework.Web.bind.annotation.RestController;

@RestController
public class HelloWorldController {

    @RequestMapping(method= RequestMethod.GET,path=”/hello-world”)
    public String helloWorld(){
        return “Hello World”;
    }

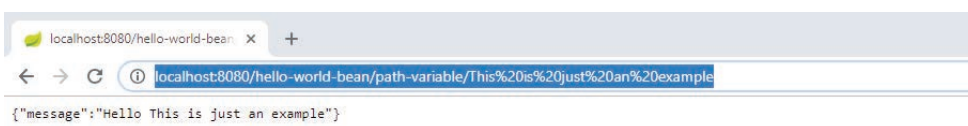
    @GetMapping(path=”hello-world-bean”)
    public HelloWorldBean helloWorldBean()
    {
        return new HelloWorldBean(“Hello World”);
    }

    @GetMapping(path = “hello-world-bean/path-variable/{name}”)
    public HelloWorldBean helloWorldBean(@PathVariable String name)
{

        return new HelloWorldBean(String.format(“Hello %s”, name));
    }

}
```

Pozivanjem URI-ja http://localhost:8080/hello-world-bean/path-variable/This%20is%20just%20an%20example prikazuje se rezultat kao na slici 12.8.



Slika 12.8. Poziv HelloWorldBean sa promenljivom putanje

12.1.2. Implementacija RESTful veb servisa sa listom korisnika

U ovom primeru je pokazni HelloWorldBean primer proširen, gde je implementiran jednostavan RESTful veb servis koji omogućava prikaz i unos liste korisnika sistema.

U primeru su kreirane tri klase i to:

- klasa User koja predstavlja korisnika sistema,
- klasa UserDAOService koja predstavlja klasu koja komunicira sa bazom podataka i
- klasa UserResource koja izvršava funkciju REST kontroler klase.

Kod klase User je dat ispod.

```
package rs.ac.singidunum.restfulWebservices;

import java.util.Date;

public class User {

    private Integer id;
    private String name;
    private Date birthDate;

    public Integer getId() {
        return id;
    }

    public String getName() {
        return name;
    }

    public Date getBirthDate() {
        return birthDate;
    }

    public void setId(Integer id) {
        this.id = id;
    }

    public void setName(String name) {
        this.name = name;
    }

    public void setBirthDate(Date birthDate) {
        this.birthDate = birthDate;
    }
```

```
   //construct
   public User(){

   }

    public User(Integer id, String name, Date birthDate) {
        this.id = id;
        this.name = name;
        this.birthDate = birthDate;
    }

    @Override
    public String toString() {
        return "User{" +
                "id=" + id +
                ", name='" + name + '\'' +
                ", birthDate=" + birthDate +
                '}';
    }
}
```

Kao što se vidi iz prikazanog koda, klasa User je relativno jednostavna klasa koja ima tri polja: id, name i birthDate, odgovarajuće get i set metode za svako polje, kako bi se ispoštovao princip enkapsulacije, i jedan konstruktor bez parametara. U klasi je takođe implementirana i toString() metoda, za potrebe prikazivanja instanci klase User sa svim atributima.

U nastavku je naveden kod klase UserDAOService.

```
package rs.ac.singidunum.restfulWebservices;

import org.springframework.stereotype.Component;

import java.util.ArrayList;
import java.util.Date;
import java.util.List;

@Component
public class UserDAOService {

    private static List<User> users=new ArrayList<>();

    private static int usersCount=3;
```

```
    static {
        users.add(new User(1,"john",new Date()));
        users.add(new User(2,"Monica",new Date()));
        users.add(new User(3,"Petra",new Date()));
    }

    public List<User> findAll(){
        return users;
    }

    public User save(User user){
        if(user.getId()==null){
            user.setId(++usersCount);
        }
        users.add(user);
        return user;
    }

    public User findOne(int id){
        for(User user:users){
            if(user.getId()==id){
                return user;
            }
        }
        return null;
    }

}
```

S obzirom da u primeru nije implementirana baza podataka, u statičkom bloku u ovoj klasi kreirano je tri objekta klase User koja su dodata u generičku kolekciju List<User>. Na ovaj način je simulirano korišćenje tabele u bazi u kojoj se nalaze tri sloga (zapisa), tj. u ovom slučaju tri korisnika (users). Statički blok se izvršava samo i jednom i to kada se prvi put instancira klasa UserDAOService.

U klasi UserDAOService implementirane su sledeće metode:

- metoda findAll() koja vraća sve korisnike iz tabele (u ovom slučaju sve članove kolekcije List<User>),
- metoda save(User user) koja dodaje novog korisnika u tabelu (u ovom slučaju u kolekciju List<User>). Ovde treba primetiti da je za potrebe ove metode kreirano statičko polje usersCount koje je inicijalizovano na 3, zato što su u statičkom bloku dodata tri korisnika u listu. Zatim, prilikom dodavanja svakog novog korisnika, vrednost id atributa novog korisnika je inkrement

statičke promenljive usersCount. Na ovaj način se simulira AUTO_INCREMENT svojstvo primarnog ključa u tabeli baze podataka. Ova metoda nakon dodavanja novog korisnika, korisnika koji je dodat vraća u JSON formatu i

- metoda findOne(int id) pretražuje tabelu korisnika (u ovom slučaju generičku kolekciju List<User>) na osnovu vrednosti id atributa. Ako je željeni korisnik pronađen, metoda ga vraća, a ako nije, onda metoda vraća praznu vrednost (null).

Ispreda same deklaracije klase UserDAOService dodata je anotacija @Component. Ova anotacija služi za interakciju sa Spring-om i na ovaj način klasa UserDAOService postaje Spring bean.

Instanca klase UserDAOService je zavisnost klase UserResource (REST kontroler klase) i primenom anotacije @Autowired ova zavisnost će biti automatski injektovana u klasu UserResource.

U REST kontroler klasi UserResource implementirane su metode koje odgovaraju na HTTP GET i POST zahteve prema sledećoj tabeli.

Tabela 12.1. Metode kontrolne klase UserResource

Tip zahteva	*Naziv metode*	*Pristupna tačka (endpoint, URI)*	*Opis odgovora*
GET	retreiveAllUsers()	<url>/users	Prikazuje sve korisnike
GET	RetreiveUser()	<url>/users/{id}	Vraća određenog korisnika
POST	CreateUser()	<url>/users	Kreira novog korisnika

Kod REST kontroler klase UserResource je prikazan ispod.

```
package rs.ac.singidunum.restfulWebservices;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.http.ResponseEntity;
import org.springframework.Web.bind.annotation.*;
import org.springframework.Web.servlet.support.ServletUriCompo-
nentsBuilder;

import java.net.URI;
import java.util.List;
```

```
@RestController
public class UserResource {

    @Autowired
    private UserDAOService service;


    // retrieves all users
    @GetMapping(path="/users")
    public List<User> retreiveAllUsers(){
        return service.findAll();
    }

    // retrieves one user
    @GetMapping(path="/users/{id}")
    public User retreiveUser(@PathVariable int id){
        return service.findOne(id);
    }

    // creates new user
    @PostMapping(path="/users")
    public ResponseEntity<Object> createUser(@RequestBody User
user){
        User savedUser=service.save(user);
        URI location= ServletUriComponentsBuilder.fromCurrentRe-
quest().path("/{id}").buildAndExpand(savedUser.getId()).toUri();

        return ResponseEntity.created(location).build();
    }
}
```

Iz prikazanog koda se vidi da je pre kreiranja instance klase UserDAOService navedena anotacija @Autowired koja govori Spring-u da kreira ovu instancu i da je automatski injektuje u klasu UserResource.

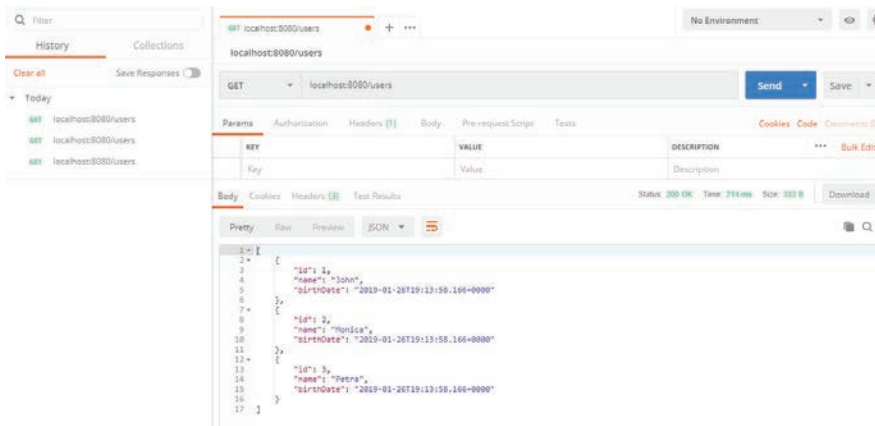
Kada korisnik preko URI-ja <url>/users izda POST zahtev za kreiranje novog korisnika, izvršava se metoda createUser(), koja upisuje novog korisnika u generičku kolekciju List<User> i koja kao rezultat vraća pristupnu tačku (URI) preko koje može da se pristupiti novokreiranom korisniku.

Testiranje veb servisa

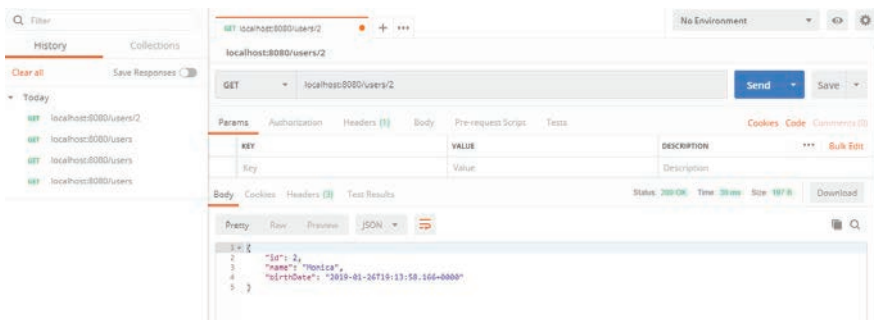
Testiranje implementiranog veb servisa je u ovom slučaju vršeno u lokalu pomoću Postman klijentske aplikacije.

Nakon pokretanja veb servisa pomoću URI-ja http://localhost:8080/users i GET HTTP zahteva prikazuje se lista korisnika u JSON formatu (slika 12.9).

Pozivanjem URI-ja http://localhost:8080/users/2 i GET HTTP zahteva prikazuje se korisnik u JSON formatu koji ima id vrednost 2. Rezultat ovog poziva prikazan je na slici 12.10.



Slika 12.9. HTTP GET zahtev koji prikazuje listu svih korisnika u JSON format

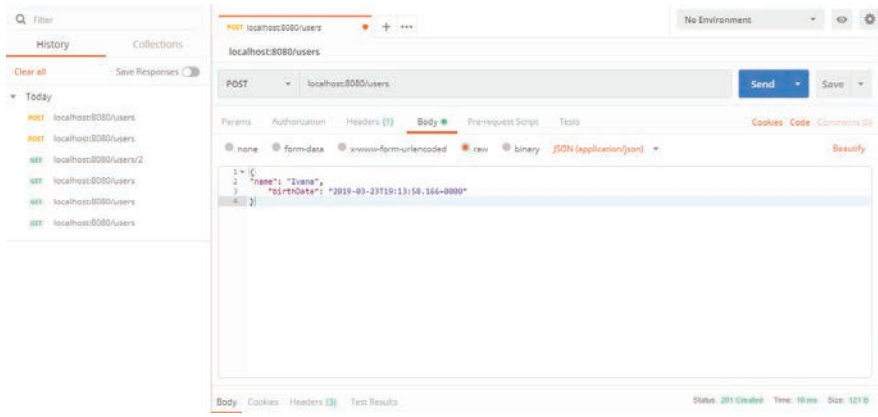


Slika 12.10. HTTP GET zahtev koji prikazuje korisnika sa id brojem 2 u JSON format

Pozivanjem URI-ja http://localhost:8080/users i POST HTTP zahteva kreira se novi korisnik.

Korisnik treba da se unese u JSON formatu, telu (engl. body) HTTP zahteva, ali bez vrednosti id polja, zato što se ova vrednost automatski generiše. Nakon izvršavanja ovog zahteva, ako je korisnik uspešno kreiran, vraća se status 201 Created.

HTTP POST zahtev za kreiranje novo korisnika prikazan je na slici 12.11.

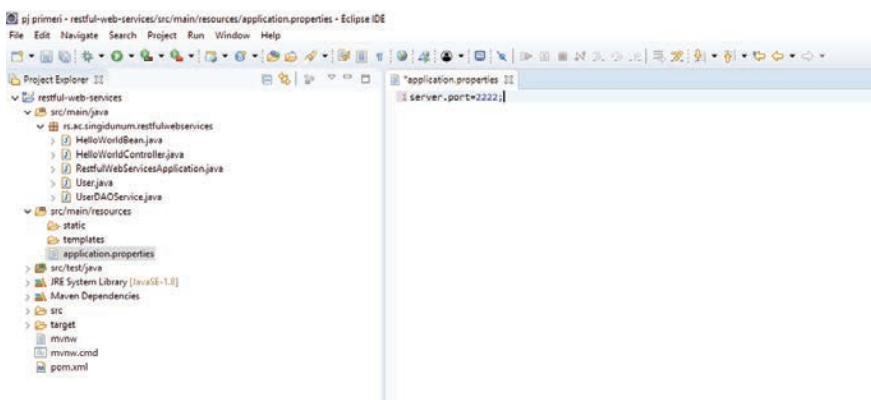


Slika 12.11. HTTP POST zahtev kojim se kreira novi korisnik

Potrebno je da se napomene da će u pojedinim slučajevima biti potrebno da se izmeni podrazumevani port na kome Tomcat server sluša dolazeće klijentske zahteve. Kao što je poznato, podrazumevani Tomcat port je 8080, koji spada u grupu dobro poznatih portova (engl. well-known ports).

Podrazumevana vrednost Tomcat porta može da se izmeni u fajlu /src/main/resources/application.properties, dodavanjem sledeće linije koda:

server.port=xxxx, gde je xxxx željeni broj porta (slika 12.12).



Slika 12.12. Promena vrednosti podrazumevanog Tomcat porta

Potreba za promenom podrazumevanog Tomcat porta se najviše javlja u situacijama kada se testira više veb servisa u lokalu, kada je potrebno sve testirane veb servise pokrenuti istovremeno. U tom slučaju svakom projektu veb servisa se dodeljuje zaseban port Tomcat servera.

12.2. Primer 2: Jednostavni veb servisi za prodavnicu knjiga i video igara

U ovom primeru su prikazane dve jednostavne implementacije veb servisa prodavnice knjiga i video igara.

Oba projekta su inicijalizovana pomoću Spring Boot Initializr alata i slična su kao projekat koji je prethodno prikazan i zbog toga u ovom poglavlju nisu dati detalji implementiranih servisa.

Obe klase (Book i Game) nasleđuju klasu Product koja modelira osnovna svojstva (atribute) svakog proizvoda, kao što su id, name, releaseYear i userScore.

Kodovi oba projekta se nalaze u glavnom paketu rs.ac.singidunum.

Kod klase Product je prikazan u nastavku.

```
package rs.ac.singidunum;

public class Product {
    private String name;
    private int releaseYear;
    private double userScore;
    private int id;

    public Product(int id,String name, int releaseYear, double
userScore) {
        this.id=id;
        this.name = name;
        this.releaseYear = releaseYear;
        this.userScore = userScore;

    }

    public int getId() {
        return id;
    }

    public void setId(int id) {
        this.id = id;
    }

    public String getName() {
        return name;
    }
```

```
    public void setName(String name) {
        this.name = name;
    }


public int getReleaseYear() {
        return releaseYear;
    }

    public void setReleaseYear(int releaseYear) {
        this.releaseYear = releaseYear;
    }

    public double getUserScore() {
        return userScore;
    }

    public void setUserScore(double userScore) {
        this.userScore = userScore;
    }

    @Override
    public String toString() {
        return "name='" + name + '\'' +
                ", releaseYear=" + releaseYear +
                ", userScore=" + userScore +
                ", id=" + id +
                ", ";
    }
}
```

Veb servis za prodaju knjiga (Book Store)

Kod projekta je implementiran u paketu rs.ac.singidunum.BookStore.

Projekat (veb servis) se sastoji iz sledećih klasa:

- Book – klasa koja modelira entitet knjige i koja nasleđuje klasu Product sa osnovnim atributima koji su specifični za knjigu: genre, author i language,
- BookService – klasa koja simulira bazu podataka i koja „komunicira sa bazom podataka“,
- BookResource – klasa koja predstavlja REST kontroler klasu i koja vrši odgovarajuća mapiranja HTTP zahteva i URI-ja na metode koje će zahteve izvršavati i
- BookStoreApplication - glavna klasa Spring Boot aplikacije u kojoj je implementirana main() metodom koja pokreće aplikaciju.

Kod klase Book je dat u nastavku.

```
package rs.ac.singidunum.BookStore;

public class Book extends Product {

    private String author;
    private String genre;
    private String language;

    public Book(int id, String name, int releaseYear, double us-
erScore,
                String author, String genre, String language) {
        super(id, name, releaseYear, userScore);
        this.author = author;
        this.genre = genre;
        this.language = language;
    }

    public String getAuthor() {
        return author;
    }

    public void setAuthor(String author) {
        this.author = author;
    }

    public String getGenre() {
        return genre;
    }

    public void setGenre(String genre) {
        this.genre = genre;
    }

    public String getLanguage() {
        return language;
    }

    public void setLanguage(String language) {
        this.language = language;
    }
```

```
    @Override
    public String toString() {
        return “Book{“ + super.toString() +
                “author=’” + author + ‘\’’ +
                “, genre=’” + genre + ‘\’’ +
                “, language=’” + language + ‘\’’ +
                ‘}’;
    }
}
```

Implementacija klase BookService je navedena ispod.

```
package rs.ac.singidunum.BookStore;

import java.util.ArrayList;
import java.util.List;
import java.util.Random;

import org.springframework.stereotype.Component;

@Component
public class BookService {
    static List<Book> books = new ArrayList<Book>();

    static {
        books.add(new Book(0,”A Game of Thrones”,1996,8.9,”George
R.R. Martin”,”Epic Fantasy”,”English”));
        books.add(new Book(1,”The Alchemist”,1988,7.7,”Paulo Coel
ho”,”Adventure”,”Portuguese”));
        books.add(new Book(2,”The Law of
Success”,1925,8.8,”Napoleon Hill”,”Self Help”,”English”));


    }

    public List<Book> findAll(){
        return books;
    }

    public Book findOne(int id){
        return books.get(id);
    }
```

```
    public Book addBook(Book newBook){
        books.add(newBook);
        return newBook;
    }

    public String deleteBook(int id){
        if (id>=0 && id<books.toArray().length){
            books.remove(id);
            return “OK”;
        }
        return “ID out of range”;
    }

    public Book switchBooks(int id, Book newBook){
        books.remove(id);
        books.add(newBook);
        return newBook;
    }

    public Book anyBook(){
        Random rand = new Random();
        int n = rand.nextInt(books.toArray().length);
        return books.get(n);
    }

    public String authorOfBook(int id){
        return books.get(id).getAuthor();
    }

    public int numberOfBooks() {
        return books.toArray().length;
    }

    public String listOfAuthors(){
        String names = “”;

        for (int i=0; i<books.toArray().length;i++){
            names = names + “ “ + books.get(i).getAuthor() + “ “;
        }
        return names;
    }
}
```

Detalji klase BookResource su navedeni ispod.

```
package rs.ac.singidunum.BookStore;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.Web.bind.annotation.*;

import java.util.List;

@RestController
public class BookResource {

    @Autowired
    private BookService service;

    @GetMapping(“/books”)
    public List<Book> showAllBooks(){
        return service.findAll();
    }

    @GetMapping(“/books/{id}”)
    public Book getBook(@PathVariable int id){
        return service.findOne(id);
    }

    @PostMapping(“/addBook”)
    Book AddNewBook(@RequestBody Book newBook) {
        return service.addBook(newBook);
    }

    @DeleteMapping(“/delete/{id}”)
    public String deleteBook(@PathVariable int id){
        return service.deleteBook(id);
    }

    @PutMapping(“/switch/{id}”)
    public Book switchBooks(@PathVariable int id, @RequestBody
Book   newBook){
        return service.switchBooks(id, newBook);
    }

    @GetMapping(“/randombook”)
    public Book returnAnyBook(){
        return service.anyBook();
    }
```

```
    @GetMapping("/find/{id}")
    public String bookAuthor(@PathVariable int id){
        return service.authorOfBook(id);
    }

    @GetMapping("/numberofbooks")
    public int bookCount(){
        return service.numberOfBooks();
    }

    @GetMapping("/authornames")
    public String allAuthors(){
        return service.listOfAuthors();
    }
}
```

Konačno, kod klase BookStoreApplication je prikazan u nastavku.

```
package rs.ac.singidunum.BookStore;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplica-
tion;
@SpringBootApplication
public class BookStoreApplication {

   public static void main(String[] args) {
      SpringApplication.run(BookStoreApplication.class, args);
   }
}
```

Veb servis za prodaju video igara (Game Store)

Slično kao i u slučaju veb servisa za prodaju knjiga (Book Store), kod projekta je implementiran u paketu rs.ac.singidunum.GameStore.

Projekat (veb servis) se sastoji iz sledećih klasa:

- Game – klasa koja modelira entitet video igre i koja nasleđuje klasu Product sa osnovnim atributima koji su specifični za video igru: platform, publisher i genre,
- GameService – klasa koja simulira bazu podataka i koja „komunicira sa bazom podataka“,
- GameResource – klasa koja predstavlja REST kontroler klasu i koja vrši odgovarajuća mapiranja HTTP zahteva i URI-ja na metode koje će zahteve izvršavati i
- GameStoreApplication - klasa koja predstavlja glavnu Spring Boot klasu sa main() metodom.

Kod klase Game je dat u nastavku.

```
package rs.ac.singidunum.GameStore;

public class Game extends Product {
    private String platform;
    private String publisher;
    private String genre;

    public Game(int id,String name, int releaseYear, double user-
Score,
                String platform, String publisher, String genre)
{
        super(id,name, releaseYear, userScore);
        this.platform = platform;
        this.publisher = publisher;
        this.genre = genre;
    }

    public String getPlatform() {
        return platform;
    }

    public void setPlatform(String platform) {
        this.platform = platform;
    }

    public String getPublisher() {
        return publisher;
    }

    public void setPublisher(String publisher) {
        this.publisher = publisher;
    }

    public String getGenre() {
        return genre;
    }

    public void setGenre(String genre) {
        this.genre = genre;
    }
```

```
    @Override
    public String toString() {
        return "Game{" + super.toString() +
                "platform='" + platform + '\'' +
                ", publisher='" + publisher + '\'' +
                ", genre='" + genre + '\'' +
                '}' ;
    }
}
```

Implementacija klase GameService je prikazana ispod.

```
package rs.ac.singidunum.GameStore;


import java.util.ArrayList;
import java.util.List;
import java.util.Random;

import org.springframework.stereotype.Component;

@Component
public class GameService {
    private static List<Game> games = new ArrayList<>();

    static
    {
    games.add(new Game(1,"Grand Theft Auto IV",2008,9.1,"PS3","Ro
ckstar","Action"));
    games.add(new Game(2,"Red Dead Redemption 2",2018,7.9,"PS4","
ROckstar","Action"));
    games.add(new Game(3,"Half-Life",1998,9.0,"PC","Valve","Shoo
ter"));


    }

    public List<Game> findAll(){
        return games;
    }

    public Game findOne(int id){
        return games.get(id);
    }
```

```
    public Game addGame(Game newGame){
        games.add(newGame);
        return newGame;
    }

    public String deleteGame(int id){
        if (id>=0 && id<games.toArray().length){
            games.remove(id);
            return "OK.";
        }
        return "ID out of range.";
    }

    public Game anyGame(){
        Random rand = new Random();
        int n = rand.nextInt(games.toArray().length);
        return games.get(n);
    }

    public String findPublisher(int publisher){
        for (int i=1; i<=games.toArray().length-1;i++){
            if(games.get(i).getPublisher().equals(publisher))
            { return(games.get(i).toString());
            }
        }
        return ("Not found.");
    }

    public String findPlatform(String platform){
        for (int i=1; i<=games.toArray().length-1;i++){
            if(games.get(i).getPlatform().equals(platform)){
               return (games.get(i).toString());
            }
        }
        return ("Not found.");
    }

    public String findPlatformOfaGame(int id){
        return games.get(id).getPlatform();
    }

    public int numberOfGames() {
        return games.toArray().length;
    }
}
```

Klasa GameResource je navedena u nastavku.

```
package rs.ac.singidunum.GameStore;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.Web.bind.annotation.*;

import java.util.List;

@RestController
public class GameResource {

    @Autowired
    private GameService service;

    @GetMapping(“/games”)
    public List<Game> showAllGames(){
        return service.findAll();
    }

    @GetMapping(“/games/{id}”)
    public Product getGame(@PathVariable int id){
        return service.findOne(id);
    }

    @PostMapping(“/addGame”)
    public Game addGame(@RequestBody Game newGame) {
        return service.addGame(newGame);
    }

    @DeleteMapping(“/delete/{id}”)
    public String deleteGame(@PathVariable int id){
        return service.deleteGame(id);
    }

    @GetMapping(“/anyGame”)
    public Product anyGame(){
        return service.anyGame();
    }

    @GetMapping(“/findplatform/{platform}”)
    public String findPlatform(@PathVariable String platform){
        return service.findPlatform(platform);
    }
```

```
    @GetMapping("/numberOfGames")
    public int numberOfGames(){
        return service.numberOfGames();
    }

}
```

Konačno, implementacija klase GameStoreApplication je u kodu u nastavku.

```
package rs.ac.singidunum.GameStore;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplica-
tion;

@SpringBootApplication
public class GameStoreApplication {

   public static void main(String[] args) {
      SpringApplication.run(GameStoreApplication.class, args);
   }

}
```

Klijentska aplikacija

U nastavku je data implementacija jednostavne klijentske aplikacije koja povezuje dva prikazana veb servisa. Aplikacija implementira relativno jednostavan GUI rađen u Java swing tehnologiji.

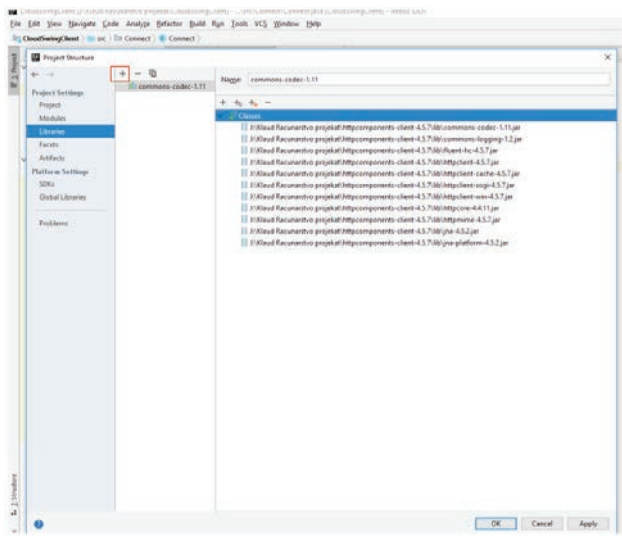
Da bi se uspostavila veza između veb servisa i klijentske aplikacije, potrebno je preuzeti apache httpcomponents-client-4.5.7 biblioteke sa sajta http://hc.apache.org/downloads.cgi. Nakon preuzimanja, ove biblioteke je potrebno dodati u projekat.

U IntelliJ okruženju, biblioteke se dodaju tako što se izabere opcija file -> project structure -> libraries , gde se se klikom na simbol + (plus) dodaju željene biblioteke u projekat (slika 12.13).

Prikazana jednostavna klijentska aplikacija se sastoji iz dve klase, i to:

- klase Connect koja služi za uspostavljanje veze sa veb servisima i
- klasa App koja predstavlja jednostavnu Java swing formu i koja povezuje odgovarajuća polja u formi za URI-je veb servisa.

Obe klase se nalaze u paketu rs.ac.singidunum.client.



Slika 12.13. Dodavanje biblioteka u IntelliJ projekat

Da bi aplikacija funkcionisala, oba RESTful veb servisa treba da budu pokrenuta. S obzirom da su u ovom slučaju oba veb servisa implementirana u lokalu, Tomcat prvog veb servisa je konfigurisan na podrazumevanom portu 8080, dok Tomcat drugog veb servisa osluškuje dolazeće zahteve klijenata na portu 8081.

Kod klase Connect je dat u nastavku.

```
package rs.ac.singidunum.client;

import org.apache.http.HttpResponse;
import org.apache.http.client.methods.HttpPost;
import org.apache.http.entity.StringEntity;
import org.apache.http.impl.client.CloseableHttpClient;
import org.apache.http.impl.client.HttpClientBuilder;

import java.io.BufferedReader;
import java.io.InputStreamReader;
import java.net.HttpURLConnection;
import java.net.URL;


public class Connect {

    public static String get(String urlToRead) throws Exception {
        StringBuilder result = new StringBuilder();
        URL url = new URL(urlToRead);
```

```
        HttpURLConnection conn = (HttpURLConnection) url.openCon-
nection();
        conn.setRequestMethod("GET");
        BufferedReader rd = new BufferedReader(new
InputStreamReader(conn.getInputStream()));
        String line;
        while ((line = rd.readLine()) != null) {
            result.append(line);
        }
        rd.close();
        return result.toString();
    }

public static String post(String urlToRead, String payload)
throws Exception {

        try (CloseableHttpClient client = HttpClientBuilder.cre-
ate().build()) {

            HttpPost request = new HttpPost(urlToRead);
            request.setHeader("User-Agent", "Java client");
            request.setHeader("Content-type", "application/
json");
            request.setEntity(new StringEntity(payload));

            HttpResponse response = client.execute(request);

            BufferedReader bufReader = new BufferedReader(new In-
putStreamReader(
                    response.getEntity().getContent()));

            StringBuilder builder = new StringBuilder();

            String line;

            while ((line = bufReader.readLine()) != null) {
                builder.append(line);
                builder.append(System.lineSeparator());
            }

            return builder.toString();
        }

    }
}
```

Kod klase App je dat ispod.

```
import Connect.Connect;

import javax.swing.*;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;

public class App {
    private JPanel MainPanel;
    private JButton showGames;
    private JTextArea showGamesResult;
    private JButton showNumberOfBooks;
    private JTextArea numberOfBooksResult;
    private JTextArea gameByIdResult;
    private JTextArea gameByIdSearch;
    private JTextArea name;
    private JTextArea releaseYear;
    private JTextArea userScore;
    private JTextArea platform;
    private JTextArea publisher;
    private JTextArea genre;
    private JButton AddGame;
    private JTextArea addgameResult;
    private JButton RandomGame;
    private JTextArea randomGameResult;
    private JTextArea allAuthors;
    private JButton showAllAuthors;
    private JButton showGameByID;
    private JTextArea id;
    public App() {

        showGamesResult.setLineWrap(true);
        showGamesResult.setWrapStyleWord(true);
        showGames.addActionListener(new ActionListener() {
            @Override
            public void actionPerformed(ActionEvent e) {
                showGamesResult.setText("");
                try {
                    showGamesResult.append(Connect.get("http://
localhost:8080/games"));
                } catch (Exception e1) {
                    e1.printStackTrace();
                }
            }
        });
```

```
        numberOfBooksResult.setLineWrap(true);
        numberOfBooksResult.setWrapStyleWord(true);
        showNumberOfBooks.addActionListener(new ActionListener()
{
            @Override
            public void actionPerformed(ActionEvent e) {
                numberOfBooksResult.setText("");
                try {
                    numberOfBooksResult.append(Connect.
get("http://localhost:8080/numberOfGames"));
                } catch (Exception e1) {
                    e1.printStackTrace();
                }
            }
        });

        gameByIdSearch.setLineWrap(true);
        gameByIdSearch.setWrapStyleWord(true);
        gameByIdResult.setLineWrap(true);
        gameByIdResult.setWrapStyleWord(true);
        showGameByID.addActionListener(new ActionListener() {
            @Override
            public void actionPerformed(ActionEvent e) {
                gameByIdResult.setText("");
                String id = gameByIdSearch.getText();
                try {
                    gameByIdResult.append(Connect.get("http://lo-
calhost:8080/games/" + id));
                } catch (Exception e1) {
                    e1.printStackTrace();
                }
            }
        });

        addgameResult.setLineWrap(true);
        addgameResult.setWrapStyleWord(true);
        AddGame.addActionListener(new ActionListener() {
            @Override
            public void actionPerformed(ActionEvent e) {
                addgameResult.setText("");
                   String payload = "{" +
                        "\"id\":\"" + id.getText() + "\"," +
                        "\"name\":\"" + name.getText() + "\"," +
                        "\"releaseYear\":\"" + releaseYear.get-
Text() + "\"," +
```

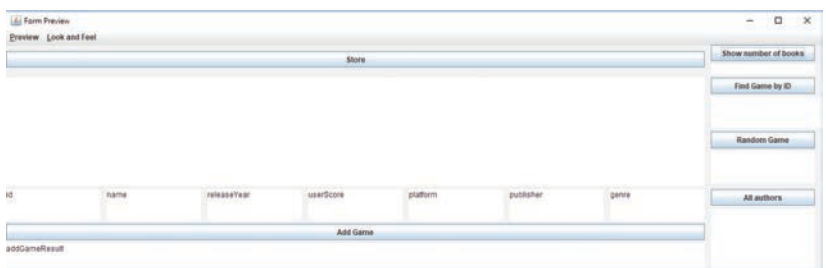
```
                        "\"userScore\":\"" + userScore.getText()
+ "\"," +
                        "\"platform\":\"" + platform.getText() +
"\"," +
                        "\"publisher\":\"" + publisher.getText()
+ "\"," +
                        "\"genre\":\"" + genre.getText() + "\","
+
                        "}";
                try {
                    addgameResult.append(Connect.post("http://lo-
calhost:8080/addGame", payload));
                } catch (Exception e1) {
                    e1.printStackTrace();
                }
            }
        });

        randomGameResult.setLineWrap(true);
        randomGameResult.setWrapStyleWord(true);
        RandomGame.addActionListener(new ActionListener() {
            @Override
            public void actionPerformed(ActionEvent e) {
                randomGameResult.setText("");
                try {
                    randomGameResult.append(Connect.get("http://
localhost:8080/anyGame"));
                } catch (Exception e1) {
                    e1.printStackTrace();
                }
            }
        });
        allAuthors.setLineWrap(true);
        allAuthors.setWrapStyleWord(true);
        showAllAuthors.addActionListener(new ActionListener() {
            @Override
            public void actionPerformed(ActionEvent e) {
                allAuthors.setText("");
                try {
                    allAuthors.append(Connect.get("http://local-
host:8081/authornames"));
                } catch (Exception e1) {
                    e1.printStackTrace();
                }
            }
        });
    }
```

```
    public static void main(String[] args) {
        JFrame frame = new JFrame("App");
        frame.setContentPane(new App().MainPanel);
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.pack();
        frame.setVisible(true);
    }

    private void createUIComponents() {
        // TODO: place custom component creation code here
    }
}
```

Izvršavanjem klijentske aplikacije pokreće se jednostavan GUI kao što je prikazano na slici 12.14 .



Slika 12.14. GUI klijentske aplikacije

12.3. Primer 3: Jednostavan REST API za veb blog postove u IntelliJ IDEA okruženju

U ovom poglavlju prikazane su dve implementacije jednostavnog REST API-ja veb servisa za veb blog u IntelilJ IDEA razvojnom okruženju.

U prvoj implementaciji podaci o blogovima su implementirani u okviru samog koda (engl. hard coded), dok je u drugom primeru, koji predstavlja nadogradnju prvog, veb servis unapređen, gde se koristi baza podataka u kojoj se čuvaju podaci o blogovima.

Kao što je poznato, blog (skraćeno od veb blog) je diskusija ili veb sajt informativnog karaktera koji se objavljuje na WWW (Word Wide Web) servisu Interneta i koji se sastoji od kratkih, često informativnih tekstualnih unosa, koji se nazivaju postovi. Postovi se obično prikazuju u obrnutom vremenskom redosledu, tako da se na vrhu veb strane prvo pojavljuju najsvežiji postovi.

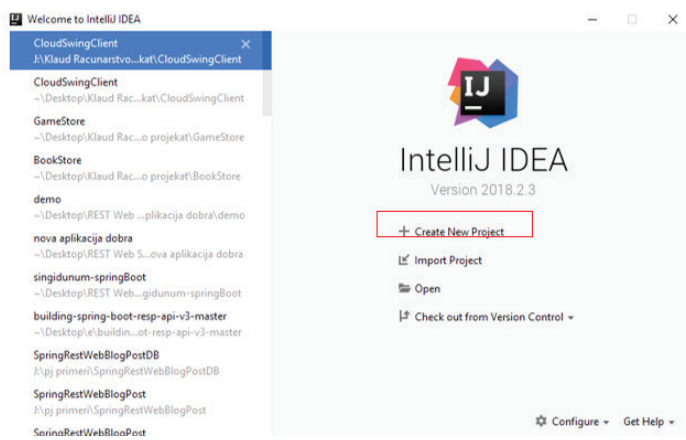
S obzirom da je u pitanju samo ilustrativni primer, u primeru je zanemaren pojam postova (u realnom scenariju svaki blog može da se sastoji od jednog ili više postova) i implementiran je veb servis, gde su samo blogovi uzeti u razmatranje.

12.3.1. Implementacija veb servisa bez baze podataka

Kao što je već i navedeno, u praksi se koriste makar tri načina za kreiranje Spring Boot aplikacija, i to: korišćenjem veb alata pod nazivom Spring Boot Initializer, alatke konzolnog interfejsa (eng. command line interface – CLI) i Maven alata za kreiranje i upravljanje Java projektima u kombinaciji sa integrisanim razvojnim okruženjem kao što su InteliiJ IDEA ili Eclipse IDE.

U ovom primeru ilustrovana je implementacija REST veb servisa jednostavnog veb blog-a korišćenjem Maven alata u IntelliJ IDEA razvojnom okruženju.

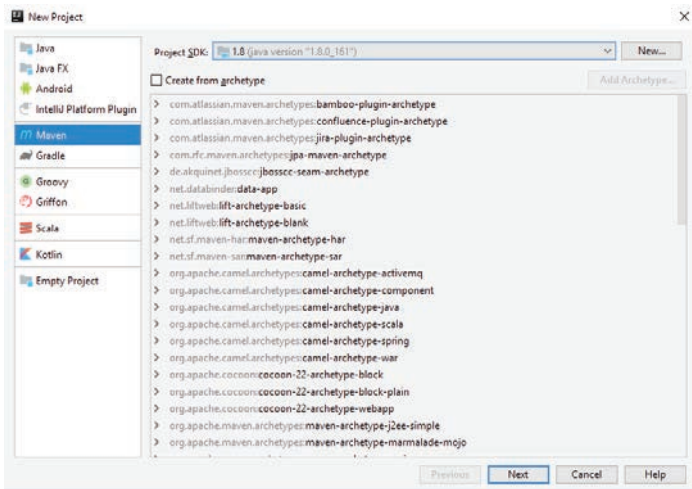
Nakon pokretanja IntelliJ IDEA potrebno je izabrati opciju *Create New Project*, i zatim izabrati Maven za tip projekta (slika 12.15 i slika 12.16).



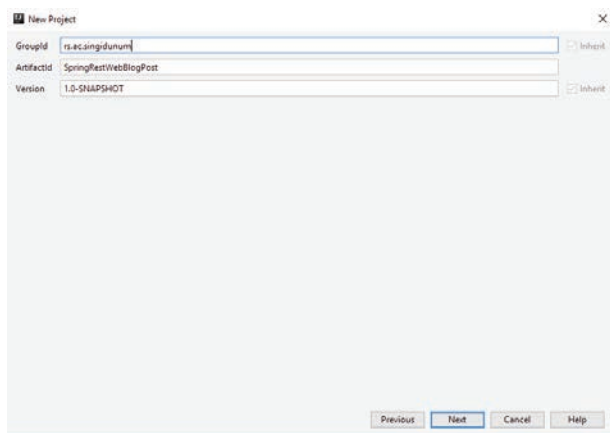
Slika 12.15. *Screenshot* – kreiranje novog projekta u IntelliJ IDEA okruženju

U sledećem koraku je potrebno popuniti osnovne informacije u projektu – *GroupId, ArtifactId* i *Version*. *GroupID* se odnosi na jedinstveno ime organizacije za koju je projekat namenjen (u većini slučajeva, programeri za *GroupID* koriste obrnut naziv kompanijskog domena, na primer rs.ac.singidunum). Polje *ArtifactId* se odnosi na jedinstven naziv projekta, dok se za vrednost polja *Version* zadaje broj verzije projekta.

U ovom primeru postavljene su sledeće vrednosti za osnovne informacije o projektu: za polje *GroupId* postavljena je vrednost *rs.ac.singidunum*, za polje *ArtifactId* vrednost *SpringRestWebBlogPost*, dok je za polje *Version* ostavljena podrazumevana vrednost *1.0-SNAPSHOT* (slika 12.17).



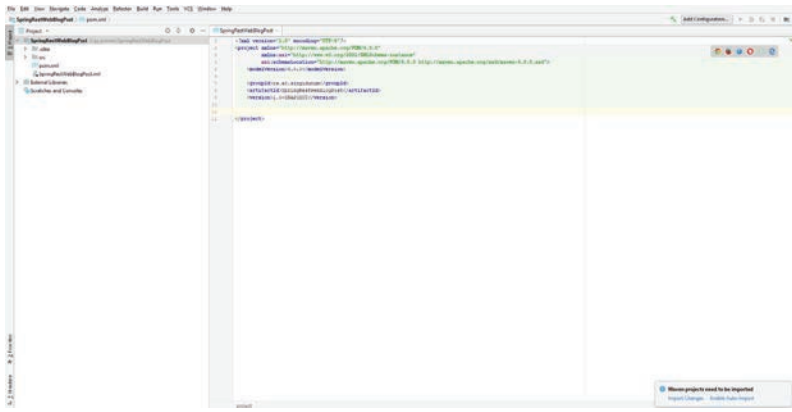
Slika 12.16. *Screenshot* – izbor opcije za Maven projekat u IntelliiJ IDEA okruženju



Slika 12.17. *Screenshot* – osnovne informacije o novom Maven projekat u IntelliJ IDEA okruženju

Nakon definisanja lokacije projekta na skladišnom uređaju, otvara se prazan projekat u kome se samo nalazi Maven *pom.xml* konfiguracioni fajl. Takođe se otvara i prozor sa obaveštenjem da je potrebno da se u projekat „uvezu“ Maven promene, gde je potrebno kliknuti na dugme *Import Changes* ili *Enable Auto-Import* (slika 12.18).

S obzirom da je Maven alat koji upravlja sa zavisnim komponentama Java projekta, u strukturu novokreiranog projekta mogu da se dodaju zavisnosti. Za sada je potreban samo „Spring Boot“, tj. Spring Boot Starter veb projekat, i u *pom.xml* fajl je potrebno dodati kod koji je prikazan u isečku (kod se dodaje pre taga za zatvaranje projekta </project>).



Slika 12.18. *Screenshot* – novi Maven projekat sa *pom.xml* fajlom

```
<parent>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-parent</artifactId>
    <version>2.1.1.RELEASE</version>
</parent>

<dependencies>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-Web</artifactId>
    </dependency>
</dependencies>

<properties>
    <java.version>1.8</java.version>
</properties>

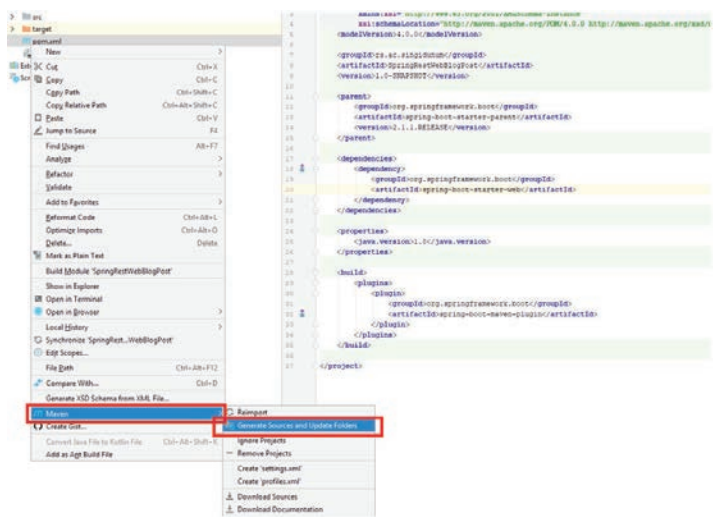
<build>
    <plugins>
        <plugin>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-maven-plugin</artifactId>
        </plugin>
    </plugins>
</build>
```

Nakon dodavanja koda u *pom.xml* fajl, potrebno je ažurirati Maven repozitorijum kako bi zavisnosti bile preuzete sa veb-a (desni klik na *pom.xml* fajl -> Maven -> Generate Sources and Update Folders). Navedni korak je prikazan na slici 12.19.

Tagom <parent> se Maven usmerava da nasledi svojstva od „spring-boot-starter-parent“ projekta, kao što su broj porta, konfiguracije, itd. Navedena podašavanja obezbeđuje Spring.

U okviru taga <dependencies> nalaze se sve zavisnosti projekta. Za sada se koristi samo jedna zavisnost „spring-boot-starter-Web“. U aplikaciji sa svim funkcionalnostima, najčešće se koristi više zavisnosti, kao što su MySQL, Socket, JSON library, itd.

Konačno, u okviru taga <build> nalaze se plugi-inovi i dodaci koji pomažu u samom procesu izgradnje (engl. build) projekta, a tipičan primer je „spring-maven-plugin“.



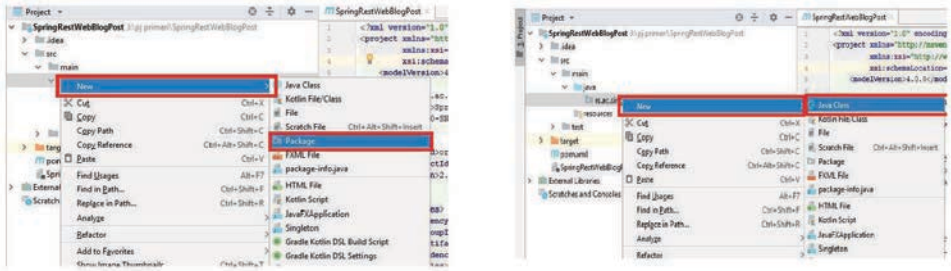
Slika 12.19. *Screenshot* – ažuriranje Maven repozitorijuma

Definisanjem vrednosti taga <version> određuje se verzija Spring Boot-a koja se koristi u projektu. U vreme pisanja ovog udžbenika aktuelna verzija je bila 2.1.1.RELEASE. Trenutna Spring Boot verzija može da se vidi na sledećoj URL adresi: https://mvnrepository.com/artifact/org.springframework.boot/spring-boot-starter.

Kao što važi i za svaki drugi Java projekat, da bi projekat mogao da se pokrene, potrebno je napraviti klasu sa main() metodom. Klasa sa main() metodom treba da bude u okviru paketa *rs.ac.singidunum*. Glavna klasa nosi naziv *MainApplicationClass*. Ovaj naziv je naravno proizvoljan, ali je u ovom primeru korišćen klasičan naziv klase u kojoj se nalazi main() metoda i koji se najčešće koristi u Spring Boot projektima.

Dakle, prvo je potrebno napraviti paket u okviru src/main/java foldera Project Browser-a IntelliJ IDEA okruženja (desni klik na java foder i izbor opcije new -> package).

Kada je paket kreiran onda se u okvru paketa pravi MainApplicaitonClass java klasa (desni klip na naziv paketa -> new -> Java Class). Navedeni koraci prikazani su na slici 12.20.



Slika 12.20. *Screenshot* – kreiranje paketa (levo) i kreiranje klase (desno)

Kod klase MainApplicatioClass je prikazan u listingu.

```
package rs.ac.singidunum;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplica-
tion;

@SpringBootApplication
public class MainApplicationClass {

    public static void main(String[] args) {
        SpringApplication.run(MainApplicationClass.class, args);
    }
}
```

Anotacija @SpringBootApplication mora da se doda ispred deklaracije glavne klase zato što ova anotacija upućuje da se radi o Spring Boot aplikaciji.

Pokretanjem aplikacije (desni klik na kod klase MainApplicationClass -> Run MainApplication…main()) pokreće se aplikacija zajedno sa ugrađenim (engl. embedded) Tomcat serverom na podrazumevanom portu 8080.

Međutim, URL adresa http://localhost:8080/ nije u funkciji zato što još uvek nije napravljena klasa kontroler (engl. controller class) koja upravlja HTTP zahtevima klijenata (slika 2.21).



Slika 12.21. *Screenshot* – http://localhost:8080

Kontroler upravlja svih dolaznim HTTP zahtevima krajnjih korisnika i vraća odgovarajući odgovor. U nekim tehnologijama potrebno je kreirati posebne fajlove za rutiranje (eng. route files) koji mapiraju HTTP zahteve za odgovarajuće kontrolere.

U listngu koda navedenom ispod prikazana je implementacija BlogController klase (fajl BlogController.java se nalazi u paketu rs.ac.singidunum).

Klasa BlogController predstavlja kontroler („C“) u MVC (Model View Control) modelu.

```
package rs.ac.singidunum;

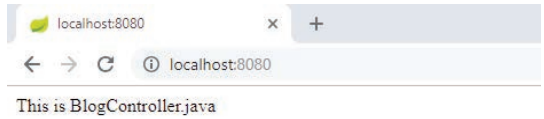
import org.springframework.Web.bind.annotation.RequestMapping;
import org.springframework.Web.bind.annotation.RestController;

    @RestController
    public class BlogController {

        @RequestMapping(“/”)
        public String root() {
            return “This is BlogController.java”;
        }
}
```

Anotacija @RestController upućuje Spring da je klasa BlogController kontroler klasa. S druge strane, anotacijom @RequestMapping(“/”) se definiše da će bilo koji HTTP zahtev (GET, POST, PUT, itd.) koji je upućen ka root-u, tj. zahtev ka URL-u: http://localhost:8080, obraditi metoda root(), a odgovor će biti tipa String.

Pokretanjem aplikacije u lokalu i pozivanjem URL-a http://localhost:8080 veb servis generiše odgovor koji je prikazan na slici 12.22.



Slika 12.22. *Screenshot* – http://localhost:8080

Osim anotacije @RequestMapping, u implementaciji veb servisa često se koriste i anotacije @PostMapping i @GetMapping za upravljanje POST i GET zahtevima, respektivno.

Osnovna funkcija BlogController klase je čuvanje, ažuriranje, brisanje i preuzimanje blogova. Sam blog je opisan klasom Blog u kojoj se nalaze osnovna svojstva (atributi) bloga, i to:

- id – atribut kojim se definiše jedinstveni identifikator svakog bloga (primarni ključ u bazi),
- title – atribut koji definiše naziv bloga i
- content – atribut u kom se čuva sadržaj bloga.

Atributi klase Blog odgovaraju kolonama Blog tabele u bazi podataka. Na slici 12.23 prikazana su svojstva klase Blog i odgovarajuća Blog tabela.



Id	Title	Content
1	First blog	This is just first blog
2	Second blog	This is just second blog
3	Third blog	This is just third blog

Slika 12.23. Atributi klase Blog i Blog tabela u bazi podataka

Kod klase Blog koja se takođe nalazi u paketu rs.ac.singidunum prikazana je ispod.

Iz prikazanog koda se vidi da klasa Blog mora da ima podrazumevani konstruktor (u prikazanoj klasi su implementirana dva konstruktora). Takođe, iz listinga se vidi da je implementirana i toString() metoda koja omogućava prikaz svakog bloga sa svim njegovim atributima.

```
package rs.ac.singidunum;

public class Blog {

    private int id;
    private String title;
    private String content;
```

```
    public Blog() {  }

    public Blog(int id, String title, String content) {
        this.setId(id);
        this.setTitle(title);
        this.setContent(content);
    }

    public int getId() {
        return id;
    }

    public void setId(int id) {
        this.id = id;
    }

    public String getTitle() {
        return title;
    }

    public void setTitle(String title) {
        this.title = title;
    }

    public String getContent() {
        return content;
    }

    public void setContent(String content) {
        this.content = content;
    }

    @Override
    public String toString() {
        return "Blog{" +
                "id=" + id +
                ", title='" + title + '\'' +
                ", content='" + content + '\'' +
                '}';
    }
}
```

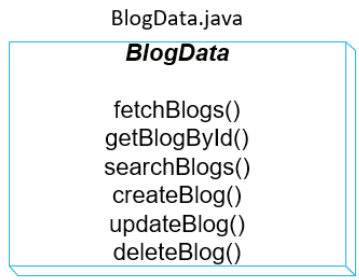
U realnom primeru bi se svaki blog čuvao u tabeli baze podataka (slika 12.23). Međutim, u ovom prvom, uprošćenom primeru, se ne koristi baza podataka, već se funkcionalnosti baze podataka simuliraju java klasom, koja u ovom slučaju ima naziv BlogData.

Klasa BlogData zapravo predstavlja model („M“) u MVC-u.

Osnovne karakteristike klase BlogData su:

- u klasi se nalazi lista hard kodovanih blog postova i
- klasa je dizajnirana korišćenjem singleton dizajn paterna, zbog potrebe da promene budu trajne (perzistentne) preko različitih HTTP zahteva.

Osnovne metode klase BlogData prikazane su na slici 12.24.



Slika 12.24. Metode klase BlogData

Na osnovu samih naziva mogu da se pretpostave i funkcije koje svaka od navedenih metoda obavlja:

- metoda fetchBlogs() prikazuje sve blogove,
- metoda getBlogById() prikazuje određeni blog na osnovu vrednosti id atributa,
- metoda searchBlogs() prikazuje blogove na osnovu sadržaja,
- metoda createBlog() kreira novi blog,
- metoda updateBlog() ažurira određeni blog i
- metoda deleteBlog() briše blog na osnovu vrednosti id atributa.

Kao što je gore navedeno, dizajn klase BlogData odgovara singleton dizajn paternu. Singleton osigurava da ni jedna klasa ne može da ima više od jedne instance. Drugim rečima, ako je određena klasa definisana kao Singleton, onda ta klasa može da ima samo jednu instancu tokom celog životnog ciklusa aplikacije.

Singleton implementacija klase BlogData prikazana je u nastavku.

```
private static BlogData instance = null;
public static BlogData getInstance(){
    if(instance == null){
        instance = new BlogData();
    }
    return instance;
}
```

Svaki put kada je potrebno instancirati klasu BlogData, instanciranje se vrši na sledeći način:

```
BlogData blogData=BlogData.getInstance();
```

Klasu ne treba instancirati kao:

```
BlogData blogData=new BlogData();
```

Implementacija klase BlogData, sa odgovarajućim komentarima, koja se takođe nalazi u paketu rs.ac.singidunum data je u kodu ispod.

```
package rs.ac.singidunum;

import java.util.ArrayList;
import java.util.List;

public class BlogData {
    //placeholder (Java List collection) to store all blogs
    private List<Blog> blogs;

    private static BlogData instance = null;
    public static BlogData getInstance(){
        if(instance == null){
            instance = new BlogData();
        }
        return instance;
    }
// constructor - here we hard code the data
public BlogData(){
    blogs = new ArrayList<Blog>();
    blogs.add(new Blog(1, “Swarm intelligence metaheuristics”,
            “Swarm intelligence simulate the behavior of groups
of organisms in the nature...”));
    blogs.add(new Blog(2, “PSO”,
            “PSO stands for the particle swarm optimization. PSO is
swarm intelligence algorithm that simulates flock of birds or fish...”));
    blogs.add(new Blog(3, “ABC”,
            “ABC stands for the artificial bee colony. ABC is
swarm intelligence algorithm that simulates hives of bees...”));
    blogs.add(new Blog(4, “ACO”,
            “ACO stands for the ant colony optimization. ACO is
swarm intelligence algorithm that simulates group of ants...”));
    blogs.add(new Blog(5, “FA”,
            “FA stands for the firefly algorithm. FA is swarm in-
telligence algorithm that simulates the group of fireflies...”));
}
```

```
// method that returns all blogs
    public List<Blog> fetchBlogs() {
        return blogs;
    }

    // method that returns blog by id
    public Blog getBlogById(int id) {
        for(Blog b: blogs) {
            if(b.getId() == id) {
                return b;
            }
        }
        return null;
    }

    // method that searches blog by text
    public List<Blog> searchBlogs(String searchTerm) {
        List<Blog> searchedBlogs = new ArrayList<Blog>();
        for(Blog b: blogs) {
            if(b.getTitle().toLowerCase().contains(searchTerm.
toLowerCase()) ||
                    b.getContent().toLowerCase().
contains(searchTerm.toLowerCase())) {
                searchedBlogs.add(b);
            }
        }

        return searchedBlogs;
    }

    // method that creates a blog
    public Blog createBlog(int id, String title, String content) {
        Blog newBlog = new Blog(id, title, content);
        blogs.add(newBlog);
        return newBlog;
    }

    // method that updates a blog
    public Blog updateBlog(int id, String title, String content)
{
        for(Blog b: blogs) {
            if(b.getId() == id) {
                int blogIndex = blogs.indexOf(b);
                b.setTitle(title);
                b.setContent(content);
                blogs.set(blogIndex, b);
                return b;
```

```
            }

        }

        return null;
    }

    // method that deletes a blog by id
    public boolean delete(int id){
        int blogIndex = -1;
        for(Blog b: blogs) {
            if(b.getId() == id) {
                blogIndex = blogs.indexOf(b);
                continue;
            }
        }
        if(blogIndex > -1){
            blogs.remove(blogIndex);
        }
        return true;
    }

}
```

Iz prikazanog koda se jasno vidi da klasa BlogData simulira bazu podataka, gde se svi podaci o blogovima (id, title, content) čuvaju u Java ArrayList strukturi podataka.

Drugo što se zapaža jeste da se u samom konstruktoru ove klase, dakle pri samom instanciranju klase, u ArrayList unose podaci o blogovima.

Treća stvar koju je potrebno zapaziti jeste da se brisanje bloga, koje je implementirano u metodi delete(), obavlja izvan for petlje, sa ciljem da se izbegne generisanje izuzetka *java.util.ConcurrentModificationException*.

Nakon što je kreirana klasa BlogData, potrebno je modifikovati kontroler klasu (BlogController), ali je pre toga potrebno razumeti REST pristupne tačke, čiji je pregled dat u Tabeli 12.2.

Osnovne SpringBoot anotacije za upravljanje različitim HTTP zahtevima su sledeće:

- @RequestMapping – koristi se za upravljanje bilo kojim zahtevima,
- @GetMapping – koristi se za upravljanje GET zahtevima,
- @PostMapping – koristi se za upravljanje POST zahtevima,
- @PutMapping – koristi se za upravljanje PUT zahtevima,
- @PatchMapping – koristi se za upravljanje PATCH zahtevima i
- @DeleteMapping – koristi se za upravljanje DELETE zahtevima.

Tabela 12.2. Veb blog REST pristupne tačke

Tip zahteva	*Pristupna tačka*	*Opis odgovora*
GET	<url>/blog	vraća i prikazuje sve blogove
GET	<url>/blog/{id}	vraća i prikazuje jedan blog post koji odgovara {id} vrednosti
POST	<url>/blog/search	pretražuje sve blogove i vraća sve blogove koji odgovaraju parametru pretrage search
POST	<url>/blog	kreira novi blog i vraća kreirani blog
PUT	<url>/blog/{id}	ažurira blog koji odgovara vrednosti {id} parametra i vraća ažurirani blog
DELETE	<url>/blog/blog{id}	Briše blog koji odgovara vrednosti {id} parametra i vraća odgovor tipa boolean

Na primer, metoda koja upravlja POST zahtevima može da se deklariše kao:

```
@RequestMapping(value = “path”, method = RequestMethod.POST)
public typeOfReturn methodName(){...}
```

Ili kao

```
@PostMapping(“path”)
public returnType methodName(){...}
```

Promenljive putanje (engl. path variables) su promeljive koje se navode u URI-ju sa zahtevima za resursima i označavaju se sa @PathVariable. Primer je naveden u kodu ispod.

```
@GetMapping(“/blog/{id}”)
  public typeOfReturn methodName(@PathVariable String id)
  {
...
  }
```

Parametri zahteva (engl. request parameters) su ključ-vrednost parovi podataka koji se navode u URI-ju sa zahtevima za resursima i označavaju se sa @RequestParam. Primer je naveden u kodu ispod.

```
@GetMapping(“/blog”)
  public typeOfReturn methodName(@RequestParam String param1, @
RequestParam String param2)
  {
...
  }
```

HTTP zahtevi tipa POST, PUT i DELETE mogu da sadrže i teret (engl. payload), koji se naziva telo zahteva (engl. request body). U teretu se nalaze podaci koji se čuvaju, ili ažuriraju na strani servera. Teret se uglavnom, u slučaju implementacije REST veb servisa, prikazuje u JSON (JavaScript Object Notation) formatu.

Primer tela zahteva u ovom konkretnom primer dat je u tabeli 12.3.

Tabela 12.3. Primer tela zahteva

Tip zahteva	Pristupna tačka	Telo zahteva (teret)	Funkcija
POST	<url>/blog/search	{“text”:”EHO”}	pretražuje blogove
POST	<url>/blog	{“id”:”6”,”title”:”BAT”,”content”:”BAT is…”}	kreira novi blog
PUT	<url>/blog/{id}	{“id”:”1”,”title”:”ABC”,”content”:”ABC is…”}	ažurira blog sa id-jem {id}
DELETE	<url>/blog/{id}	{}	briše blog sa id-jem {id}

Za potrebe definisanja tela zahteva koristi se anotacija @RequestBody. S obzirom da je telo zahteva par ključ-vrednost, može da se koristi Java apstraktna struktura podataka Map<String,String>.

Primer ovakvog koda je dat ispod.

```
@PostMapping(“/blog”)
public Blog create(@RequestBody Map<String, String> body)
{...
}
```

Za prikazivanje odgovarajućih ključeva i njihovih vrednosti može da se koriste sledeće naredbe:

```
String id = body.get(“id”);
String title = body.get(“title”);
String content = body.get(“content”);
```

Uzimajući sve navedeno u obzir, implementacija modifikovane klase kontroler (BlogController) je prikazana u kodu ispod.

```
package rs.ac.singidunum;

import org.springframework.Web.bind.annotation.*;

import java.util.List;
import java.util.Map;

@RestController
public class BlogController {

    //obtaining singleton instance
    BlogData blogData = BlogData.getInstance();

    // method that returns all blogs
    @GetMapping("/blog")
    public List<Blog> root(){
        return blogData.fetchBlogs();
    }

    // method that returns blog by id
    @GetMapping("/blog/{id}")
    public Blog show(@PathVariable String id){
        int blogId = Integer.parseInt(id);
        return blogData.getBlogById(blogId);
    }

    // method that searches blog by text
    @PostMapping("/blog/search")
    public List<Blog> search(@RequestBody Map<String, String>
body){
        String searchTerm = body.get("text");
        return blogData.searchBlogs(searchTerm);
    }
    // method that creates new blog
    @PostMapping("/blog")
    public Blog create(@RequestBody Map<String, String> body){
        int id = Integer.parseInt(body.get("id"));
        String title = body.get("title");
        String content = body.get("content");
        return blogData.createBlog(id, title, content);
    }
```

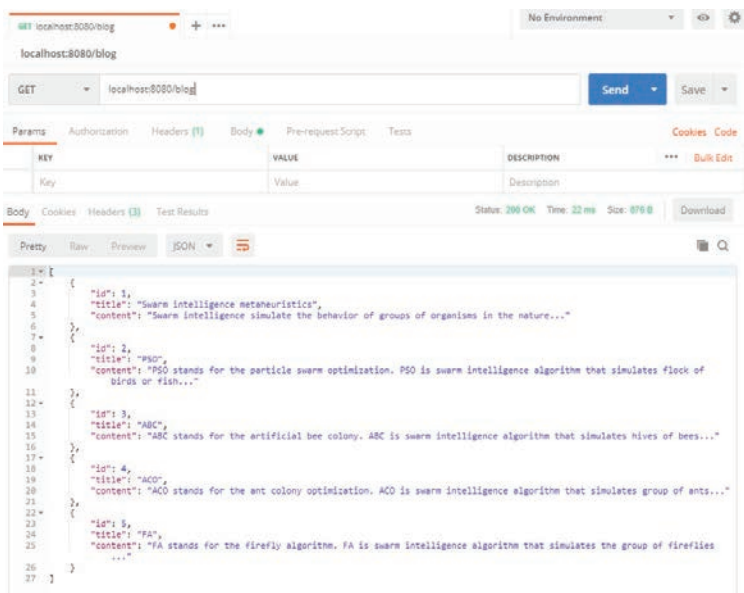
```
    // method that updates existing blog
    @PutMapping("/blog/{id}")
    public Blog update(@PathVariable String id, @RequestBody
Map<String, String> body){
        int blogId = Integer.parseInt(id);
        String title = body.get("title");
        String content = body.get("content");
        return blogData.updateBlog(blogId, title, content);
    }

    // method that deletes a blog by id
    @DeleteMapping("blog/{id}")
    public boolean delete(@PathVariable String id){
        int blogId = Integer.parseInt(id);
        return blogData.delete(blogId);
    }
}
```

Testiranje Blog veb servisa

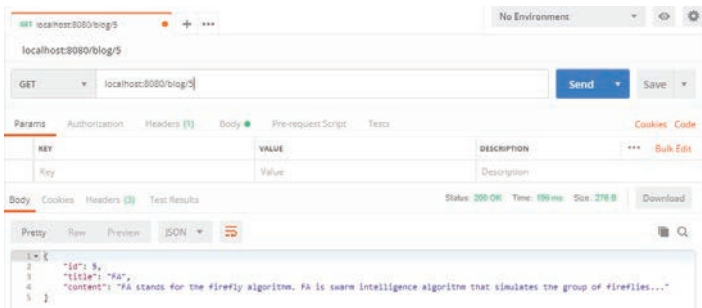
Nakon pokretanja implementiranog veb servisa, testiranje može da se vrši u Postman aplikaciji.

Pozivanjem URI-ja localhost:8080/blog izvršava se metoda root() klase BlogController koja vraća sve blogove u JSON formatu (slika 12.25).



Slika 12.25. Prikaz svih blog postova u JSON format pomoću Postman aplikacije

Pozivanjem URL-a localhost:8080/blog/5 izvršava se metoda show() klase Blog Controller koja vraća željeni blog na osnovu vrednosti parametra id (slika 12.26).



Slika 12.26. Prikaz jednog bloga u JSON format u Postman aplikaciji

Izvršavanje GET zahteva koji su prikazani u prethodna dva primera može da se testira i bez klijentske aplikacije kao što je Postman. Međutim, za testiranje POST zahteva, kao što je kreiranje novog bloga potrebna je klijentska aplikacija, zato što se podaci o novom blogu koji se kreira šalju u telu zahteva, dok se sami podaci formatiraju u JSON formatu.

Da bi se testirao POST zahtev u Postman aplikaciji potrebno je izvršiti sledeće korake:

- prvo je potrebno da se naznači da se radi o POST zahtevu,
- drugo, potrebno je navesti odgovarajuću URI adresu,
- teće, potrebno je izabrati da je format body zahteva JSON i
- konačno, u okviru body opcije je potrebno navesti podatke o novom blogu u JSON formatu.

Svi navedeni koraci prikazani su na slici 12.27.

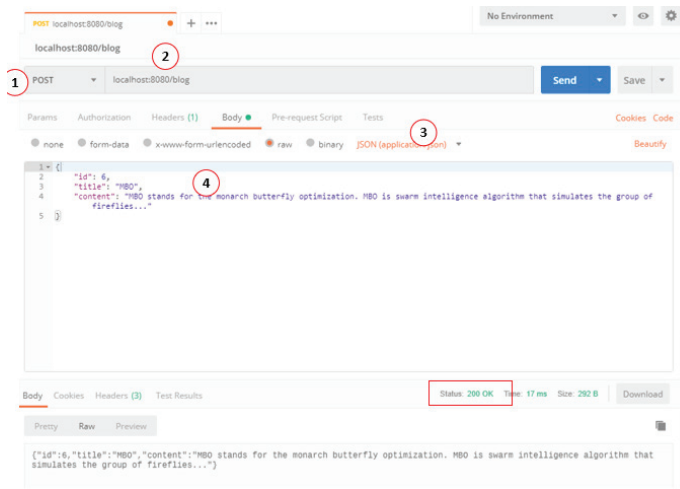
Izvršavanjem POST zahteva pozivanjem URI-ja http://localhost:8080/blog izvršava se metoda create() klase BlogController. Nakon izvršavanja ove metode vraća se HTTP status kod 200, što znači da je POST zahtev uspešno izvršen (slika 12.27).

Na sličan naziv može da se testira i PUT HTTP zahtev. Izvršavanjem PUT zahteva nad URL-om http://localhost:8080/blog izvršava se metoda update() klase BlogController, koja ažurira postojeći blog.

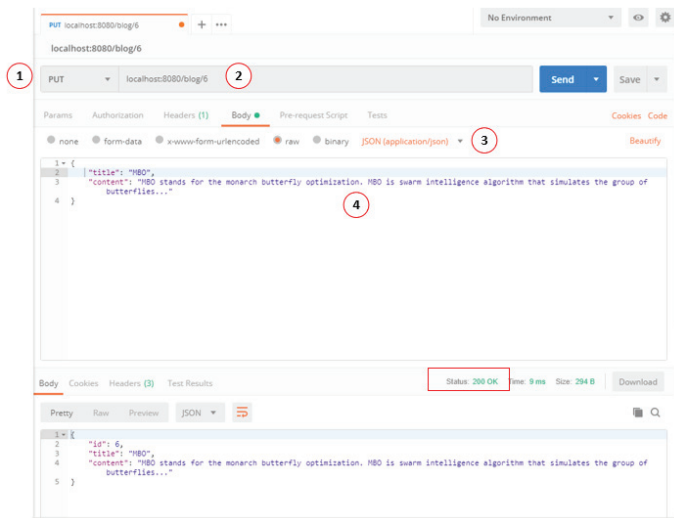
Razlika u odnosu na POST zahtev je taj što se u ovom slučaju id bloga koji treba izmeniti zadaje kao promenljiva putanje u URI-ju, dok se u okviru tela zahteva zadaju samo nove vrednosti za title i content atribute u JSON formatu, bez zadavanje vrednosti za atribut id.

Nakon izvršavanja metode update() vraća se HTTP status kod 200, što znači da je PUT zahtev uspešno izvršen.

Testiranje PUT zahteva u Postman okruženju dato je na slici 12.28.



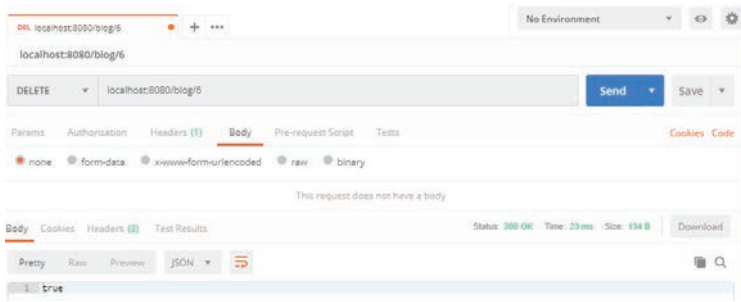
Slika 12.27. Kreiranje novog bloga pomoću POST zahteva u Postman aplikaciji



Slika 12.28. Ažuriranje postojećeg bloga pomoću PUT zahteva u Postman aplikaciji

Na slici 12.29 je dat još jedan primer, gde se briše blog sa id-jem 6 pomoću HTTP DELETE zahteva.

Izvršavanjem DELETE zahteva pozivanjem URI-ja http://localhost:8080/blog/6 izvršava se metoda delete() klase BlogController, koja briše željeni blog.



Slika 12.29. Brisanje bloga pomoću DELETE zahteva u Postman aplikaciji

12.3.2. Implementacija Blog veb servisa sa bazom podataka

U prethodnom primeru data je implementacija Blog veb servisa, gde je na zahtev klijenta odgovarao kontroler, koji je koristio podatke koji su ugrađeni u samom kodu.

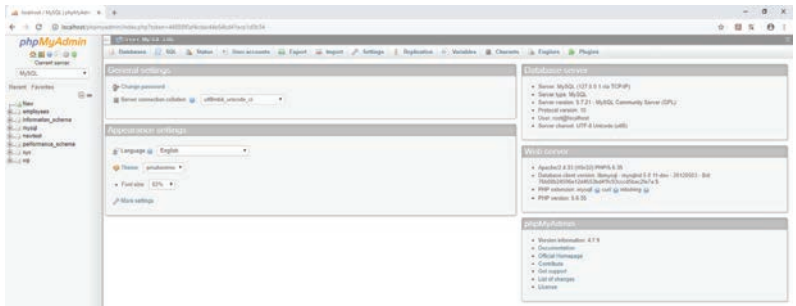
U ovom primeru, veb servis koji je implementiran u poglavlju 12.3.1 je unapređen MySQL bazom podataka i za izvršavanje SQL upita nad bazom korišćena je JPA (Java Persistence Application) tehnologija. JPA je Java biblioteka objekt relacionog mapiranja (engl. object relational mapping – ORM) koja functioniše slično poput Eloquent, Doctrine i LINQ tehnologija.

Da bi primer mogao da se implementira, prvo je potrebno da se instalira MySQL sistem za upravljanje relacionim bazama podataka (engl. Relational Database Management System – RDBMS). MySQL za Windows okruženje može da se preuzme sa sledećeg URL-a: https://dev.mysql.com/downloads/windows/.

Osim MySQL-a kao samostalne komponente, MySQL baza je takođe i integralni deo WAMP (Windows Apache MySQL PHP) i XAMPP (Any Operating System Apache MySQL PHP Perl) paketa. WAMP može besplatno da se preuzme sa sledećeg URL-a: http://www.wampserver.com/en/.

U slučaju korišćenja WAMP ili XAMPP proizvoda, korisniku je na raspolaganju intuitivan i user-friendly veb bazirani interfejs za upravljanje MySQL bazom pod nazivom PHPMyAdmin, kome može da se pristupi preko adrese http://localhost/phpmyadmin, u slučaju da je sistem instaliran u lokalu (slika 12.30).

S druge strane, ako se instalira samo MySQL komponenta, u tom slučaju je raspoloživ veliki broj Desktop aplikacija za upravljanje ovim sistemom, od kojih je najpoznatiji MySQL Workbench, koji takođe besplatno može da se preuzme i koristi (URL: https://www.mysql.com/products/workbench/).



Slika 12.30. PHPMyAdmin veb bazirani interfejs za upravljanje MySQL bazom

Kada se implementira MySQL sistem, potrebno je kreirati bazu podataka za veb servis. U ovom slučaju korišćena je jedna jednostavna tabela pod nazivom Blog, gde se čuvaju osnovni podaci o blogovima (blog id, title i content).

Struktura tabele Blog, koja se u ovom slučaju nalazi u bazi restws, data je na slici 12.31.

Id	Title	Content
1	First blog	This is just first blog
2	Second blog	This is just second blog
3	Third blog	This is just third blog

Slika 12.31. Struktura tabele Blog

Za potrebe ovog primera, Blog tabela je kreirana korišćenjem konzolnog SQL interfejsa (engl. Command Line Interface – CLI). U Windows command prompt-u (ili Windows power shell-u) prvo je potrebno pokrenuti mysql servis izvršanjem komande mysqld (u većini slučajeva se ovaj servis nakon instalacije MySQL-a automatski pokreće nakon butovanja operativnog sistema). Nakon toga, korisnik se loguje na MySQL sistem izvršavanjem jednostavne komande mysql -u root -p . U ovom primeru se podrazumeva da se korisnik loguje sa podrazumevanim korisničkim imenom root bez šifre. Naravno, za eksploataciju veb servisa u realnim, produkcionim okruženjima, poželjno je kreirati posebnog MySQL korisnika za ove potrebe.

Izvršavanje navedenih komandi prikazno je u isečku sa naredbama Windows terminala ispod.

```
G:\>mysqld
G:\>mysql -u root -p
Enter password:
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 38
Server version: 5.7.21 MySQL Community Server (GPL)
Copyright (c) 2000, 2018, Oracle and/or its affiliates. All rights
reserved.
```

```
Oracle is a registered trademark of Oracle Corporation and/or its
affiliates. Other names may be trademarks of their respective
owners.
Type 'help;' or '\h' for help. Type '\c' to clear the current input
statement.
mysql>
```

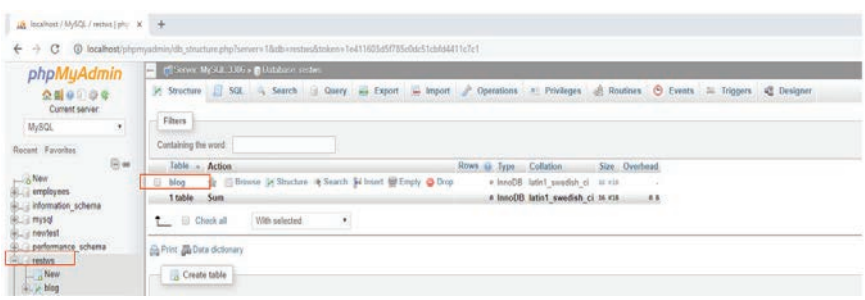
Nakon logovanja na MySQL bazu, izvršavanjem jednostavih SQL komandi CREATE DATABASE i CREATE TABLE kreiraju se baza podataka i tabela za veb servis sa nazivima restws i blog, respektivno.

SQL komande za kreiranje baze podataka i tabele prikazane su ispod.

```
mysql> CREATE DATABASE restws;
Query OK, 1 row affected (0.00 sec)
mysql> USE restws;
Database changed
mysql> CREATE TABLE blog (
    ->   id INT(6) UNSIGNED AUTO_INCREMENT PRIMARY KEY,
    ->   title VARCHAR(500) NOT NULL,
    ->   content VARCHAR(5000) NOT NULL
    -> );
Query OK, 0 rows affected (0.03 sec)
```

Komandom USE se selektuje baza podataka u kojoj se kreira tabela blog. Polje (kolona) pod nazivom id je primarni ključ, čija se vrednost automatski povećava za jedan svaki put kada se novi slog upisuje u tabelu.

Logovanjem preko PHPMyAdmin intefejsa može da se vidi kreirana baza i tabela (slika 12.32).



Slika 12.32. Baza restws i tabela blog

Da bi se SpringBoot aplikacija povezala sa MySQL bazom podataka, potrebno je izvršiti sledeće korake:

- dodati potrebne zavisnosti (npr. MySQL biblioteku) u pom.xml konfiguracioni fajl. Zavisnosti se odnose na Java biblioteke koje su potrebne za povezivanje sa bazom podataka,
- obezbediti parametre za povezivanje na bazu. Ovi parametri najčešće obuhvataju string za povezivanje na bazu (engl. connection string), broj porta MySQL servera, korisničko ime i šifru i
- kreirati klasu koja uspostavlja komunikaciju sa bazom podataka. U literaturi se ova klasa najčešće naziva repozitorijum klasa (engl. repository class).

MySQL sistem se u velikom broju slučajeva osluškuje dolazeće konekcije na podrazumevanom portu 3306. Jedan od načina da se utvrdi na kom portu se MySQL server izvršava jeste da se postavi relativno jednostavan upit pomoću MySQL konzolnog interfejsa, koji je prikazan ispod.

```
mysql> SHOW GLOBAL VARIABLES LIKE 'PORT';
+---------------+-------+
| Variable_name | Value |
+---------------+-------+
| port          | 3306  |
+---------------+-------+
1 row in set (0.00 sec)
```

Drugi način je korišćenje Windows netstat komande koja prikazuje informacije o protokolima i trenutnom korišćenju resursa TCP/IP mreže.

Primer upotrebe netstat komande je naveden u nastavku.

```
G:\>netstat
Active Connections
  Proto  Local Address          Foreign Address        State
  TCP    127.0.0.1:1521         DESKTOP-H5EFMPB:59819  ESTABLISHED
  TCP    127.0.0.1:1524         DESKTOP-H5EFMPB:59820  ESTABLISHED
  TCP    127.0.0.1:51121        DESKTOP-H5EFMPB:51122  ESTABLISHED
  TCP    127.0.0.1:51122        DESKTOP-H5EFMPB:51121  ESTABLISHED
  TCP    127.0.0.1:51123        DESKTOP-H5EFMPB:62204  ESTABLISHED
  TCP    127.0.0.1:59819        DESKTOP-H5EFMPB:1521   ESTABLISHED
  TCP    127.0.0.1:59820        DESKTOP-H5EFMPB:ingreslock  ESTA-
BLISHED
  TCP    127.0.0.1:59889        DESKTOP-H5EFMPB:59890  ESTABLISHED
  TCP    127.0.0.1:59890        DESKTOP-H5EFMPB:59889  ESTABLISHED
```

```
  TCP    127.0. 0.1:61941        DESKTOP-H5EFMPB:61942  ESTABLISHED
  TCP    127.0.0.1:61942         DESKTOP-H5EFMPB:61941  ESTABLISHED
  TCP    127.0.0.1:61943         DESKTOP-H5EFMPB:61944  ESTABLISHED
  TCP    127.0.0.1:61944         DESKTOP-H5EFMPB:61943  ESTABLISHED
  TCP    127.0.0.1:3306          DESKTOP-H5EFMPB:51123  ESTABLISHED
  TCP    192.168.0.13:51624      40.67.255.199:https    ESTABLISHED
```

Iz navedenog primera se vidi da MySQL server osluškuje na localhost-u (adresa povratne petlje 127.0.0.1) na podrazumevanom portu 3306.

Da bi se Spring Boot aplikacija povezana na MySQL bazu podataka potrebna je biblioteka pod nazivom mysql-connector-java. Slično, da bi se u aplikaciji (veb servisu) koristila JPA tehnologija, potrebna je spring-boot-starter-data-jpa biblioteka.

Kod koji je naveden ispod potrebno je kopirati u pom.xml konfiguracioni fajl u okviru taga dependencies.

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-data-jpa</artifactId>
</dependency>
<dependency>
    <groupId>mysql</groupId>
    <artifactId>mysql-connector-java</artifactId>
</dependency>
```

Nakon kopiranja navedenog koda u pom.xml fajl, da bi aplikacija prepoznala dodate Maven zavisnosti, potrebno je ažurirati Maven repozitorijum i pritisnuti dugme import changes (pogledati sliku 12.18).

Nakon dodavanja navedenih zavisnosti, struktura pom.xml fajla bi trebalo da izleda kao što je prikazano ispod.

```
<?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>rs.ac.singidunum</groupId>
    <artifactId>SpringRestWebBlogPost</artifactId>
    <version>1.0-SNAPSHOT</version>

    <parent>
        <groupId>org.springframework.boot</groupId>
```

```
        <artifactId>spring-boot-starter-parent</artifactId>
        <version>2.1.1.RELEASE</version>
    </parent>

    <dependencies>
        <!-- Spring boot dependency -->
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-Web</artifactId>
        </dependency>

        <!-- JPA Data dependency (We use in this example Reposi-
tories, Entities, Hibernate, etc...) -->
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-data-jpa</artifactId>
        </dependency>
        <!-- MySQL connector dependecny-->
        <dependency>
            <groupId>mysql</groupId>
            <artifactId>mysql-connector-java</artifactId>
        </dependency>
    </dependencies>

    <properties>
        <java.version>1.8</java.version>
    </properties>

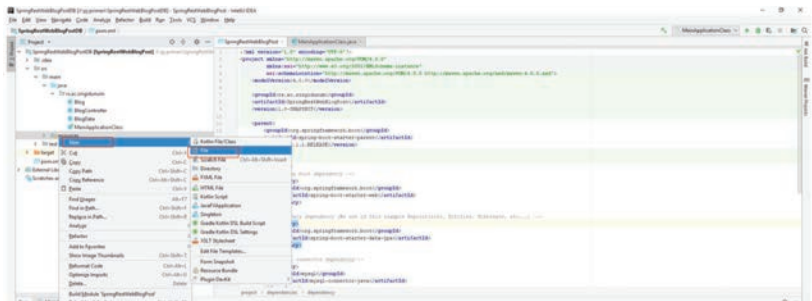
    <build>
        <plugins>
            <plugin>
                <groupId>org.springframework.boot</groupId>
                <artifactId>spring-boot-maven-plugin</artifactId>
            </plugin>
        </plugins>
    </build>

</project>
```

U prikazanom kodu pom.xml fajla korišćeni su komentari koji se označavaju sa parom tagova <!- - i - - > .

Da bi se Spring Boot aplikacija povezala sa bazom podataka, potrebno je da se definiše URL baze podataka, korisničko ime i lozinka (engl. database credentials). Sva navedena podešavanja se konfigurišu u fajlu application.properties koji se nalazi u folderu resources Spring Boot projekta.

Ukoliko navedeni faji ne postoji u Project Explorer-u InteliiJ-a, potrebno ga je kreirati (slika 12.33).



Slika 12.33. Kreiranje fajla application.properties

Linije koda koje je potrebno kopirati u fajl application.properties su navedene ispod.

```
spring.datasource.url=jdbc:mysql://localhost:3306/restws
spring.datasource.username=root
spring.datasource.password=
```

Svojstvo spring.datasource.url definiše MySQL string konekcije, svojstvom spring.datasource.username se zadaje korisničko ime MySQL baze podataka, dok se pomoću vrednosti svojstva spring.datasrouce.password definiše pristupna lozinka MySQL baze podataka.

U prikazanom kodu navedeno je da se MySQL server izvršava na podrazumevanom portu 3306, da je naziv baze restws, da se na bazu loguje sa korisničkim imenom root bez lozinke. Naravno, od konkretne implementacije zavisiće i podešavanja navedenih parametara.

Nakon implementiranja baze podataka, instaliranja zavisnosti i podešavanja konekcije ka bazi podataka, potrebno je kreirati klasu koja će da komunicira sa bazom. Kao što je već rečeno, ovakva klasa se često u literaturi naziva repozitorijum.

Kreiranje repozitorijuma se sastoji iz sledeća dva koraka:

- prvo se kreira interfejs koji nasleđuje klasu JpaRepository. Klasa JpaRepository obezbeđuje osnovne funkcionalnosti za rad sa bazom podataka, kao što su preuzimanje svih zapisa (slogova) iz baze, jednog sloga, ćuvanje, ažuriranje i brisanje slogova, itd.
- drugo, interfejsu se dodaje anotacija @Repository koja govori Spring-u da je u pitanju klasa repozitorijum.

Generička strutkura repozitorijuma data je u kodu ispod.

```
@Repository
public interface{repository name} extends JpaRepository<{Entity
class}, {Data type of Primary key field}>{
… here we define specific queries
}
```

U ovom konrektnom primeru kreira se BlogRepository klasa, čiji je kod prikazan ispod.

U nastavku teksta su data detaljnija objašnjenja o JPA repozitorijumu.

```
package rs.ac.singidunum;

import org.springframework.data.jpa.repository.JpaRepository;
import org.springframework.stereotype.Repository;

import java.util.List;

@Repository
public interface BlogRepository extends JpaRepository<Blog, Inte-
ger> {

    // custom query that is used to search blog post by title or
by the content

    List<Blog> findByTitleContainingOrContentContaining(String
text, String textAgain);

}
```

Klasu Blog koja je kreirana u prethodnom primeru i koja ima atribute koji odgovaraju poljima (kolonima) MySQL tabele je potrebno izmeniti. Svaka instanca (objekat) klase Blog treba da bude jedan unos (red, slog, zapis) MySQL tabele Blog.

Kod izmenjene klase Blog je naveden u nastavku.

```
package                                          rs.ac.singidunum;

import                                    javax.persistence.Entity;
import                              javax.persistence.GeneratedValue;
import                              javax.persistence.GenerationType;
import javax.persistence.Id;
```

```
@Entity
public class Blog {
    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private int id;

    private String title;
    private String content;

    // default constructor
    public Blog() {  }

    // new constructor with only title and content
    public Blog(String title, String content) {
        this.setTitle(title);
        this.setContent(content);
    }

    // constructor with id, title and content
    public Blog(int id, String title, String content) {
        this.setId(id);
        this.setTitle(title);
        this.setContent(content);
    }

    public int getId() {
        return id;
    }

    public void setId(int id) {
        this.id = id;
    }

    public String getTitle() {
        return title;
    }

    public void setTitle(String title) {
        this.title = title;
    }
    public String getContent() {
        return content;
    }
```

```
    public void setContent(String content) {
        this.content = content;
    }

    @Override
    public String toString() {
        return "Blog{" +
                "id=" + id +
                ", title='" + title + '\'' +
                ", content='" + content + '\'' +
                '}';
    }
```

U prikazanom kodu se uočava nekoliko važnih izmena, koje su u daljem tekstu ukratko opisane.

Da bi Spring znao da su instance klase Blog zapisi u tabeli, potrebno je dodati anotaciju @Entity pre deklarisnja klase.

S obzirom da je kolona id primarni ključ u tabeli Blog i da se njenu vrednost automatski generiše MySQL sistem, Spring-u je potrebno da se definiše da je atribut id primarni ključ i da se vrednost atributa id klase Blog automatski generiše korišćenjem anotacija @Id i @GeneratedValue(strategy = GenerationType.AUTO).

U klasi Blog je potrebno implementirati još jedan konstruktor sa atributima title i content. Ovaj konstruktor se koristi kada se šalju podaci kontroleru. Budući da se vrednost atributa id sama generiše, kontroleru ne treba da se prosleđuje ova vrednost.

U pojedinim implementacijama, gde naziv klase ne odgovara nazivu MySQL tabele potrebno je dodati anotaciju @Table(name="Blog"). S obzirom da su u ovom primeru nazivi klase i MySQL tabele isti, ova anotacija nije neophodna.

Slično, u slučajevima gde nazivi atributa klase ne odgovaraju nazivima polja (kolona) MySQL tabele, bilo bi potrebno da se doda sledeća anotacija:

```
@Column(name="title")
private String title;
```

Budući da su u prikazanoj implementaciji nazivi atributa klase Blog isti kao i nazivi kolona MySQL tabele, ova anotacija se ne dodaje u kod.

Automatsko ožičavanje – alternativa Singleton-u

U prethodnom primeru kreirana je klasa BlogData koja je Singleton, što znači da može da ima samo jednu instancu. Kao alternativa Singleton-u može da se koristi rešenje koje obezbeđuje Spring i koje se naziva automatsko ožičavanje (pogledati poglavlje 6).

Ako se ispred deklaracije klase uvede anotacija @Autowired, Spring će automatski razrešiti instancu te klase i injektovaće je u klasu koja ju je deklarisala. Na ovaj način programeri ne treba sami da definišu Singleton instancu.

Tako na primer, u ovom slučaju je potrebno instancirati klasu BlogRepository u klasi BlogController i primenom koncepta automatskog ožičavanja u kodu je potrebno uraditi sledeće:

```
@Autowired
BlogRespository blogRespository;
```

Na ovaj način blogRepository referenca može da se koristi bilo gde u kodu BlogController klase bez manuelnog instanciranja.

Klasa kontroler (BlogController)

Klasu BlogController je potrebno izmeniti. Ova klasa više ne koristi instancu klase BlogData, već instancu tipa BlogRepository.

Modifikovani kod klase BlogController je prikazan ispod.

```
package rs.ac.singidunum;

import org.springframework.beans.factory.annotation.Autowired;
import org. springframework.Web.bind.annotation.*;

import java.util.List;
import java.util.Map;

@RestController
public class BlogController {

    @Autowired
    BlogRespository blogRespository;

    // shows all blogs
    @GetMapping("/blog")
    public List<Blog> root(){
        return blogRespository.findAll();
    }

    // shows a specific blog
    @GetMapping("/blog/{id}")
    public Blog show(@PathVariable String id){
        int blogId = Integer.parseInt(id);
        return blogRespository.findOne(blogId);
    }
```

```
    // search blogs and returns the one that matches
    @PostMapping("/blog/search")
    public List<Blog> search(@RequestBody Map<String, String>
body){
        String searchTerm = body.get("text");
        return blogRespository.findByTitleContainingOrContentConta
ining(searchTerm, searchTerm);
    }


    // creates new blog
    @PostMapping("/blog")
    public Blog create(@RequestBody Map<String, String> body){
        String title = body.get("title");
        String content = body.get("content");
        return blogRespository.save(new Blog(title, content));
    }

    // updates existing blog
    @PutMapping("/blog/{id}")
    public Blog update(@PathVariable String id, @RequestBody
Map<String, String> body){
        int blogId = Integer.parseInt(id);
        // getting blog
        Blog blog = blogRespository.findOne(blogId);
        blog.setTitle(body.get("title"));
        blog.setContent(body.get("content"));
        return blogRespository.save(blog);
    }

    // deletes specific blog
    @DeleteMapping("blog/{id}")
    public boolean delete(@PathVariable String id){
        int blogId = Integer.parseInt(id);
        blogRespository.delete(blogId);
        return true;
    }

}
```

Prva promena u odnosu na implementaciju klase BlogController koja je data u prethodnom primeru (sekcija 12.3.1) jeste korišćenje instance klase BlogRepository umesto klase BlogData i anotacije @Autowired.

Definicije metoda su ostale iste, ali su implementacije promenjene. Važno je da se primeti da instanca blogRepository ima metode findaAll(), findOne(), save i delete(), koje nisu deklarisane u interfejsu BlogRepository. Ova metode obezbeđuje JpaRepository koju BlogRepository interfejs nasleđuje.

Metoda findAll() vraća sve redove iz tabele, što je ekvivalentno sledećem SQL upitu:

```
SELECT * FROM blog;
```

Metoda findOne(parameter) vraća jedan slog koji ima primarni ključ sa vrednošću argumenta parameter, što je ekvivalentno sledećem SQL upitu:

```
SELECT * FROM blog WHERE id=parameter LIMIT 1;
```

Metoda save(blog) čuva jedan unos (slog) u bazu podataka. Ova metoda kreira novi slog ako joj se proslede vrednosti za novi blog, ili ažurira vrednost postojećeg bloga ako joj se proslede vrednosti za postojeći blog koji već postoji u bazi. Izvršavanje ove metode se izvršavaju sledeće SQL naredbe:

```
INSERT INTO blog (title, content) VALUES (blog.title, blog.content);
```

Ili

```
UPDATE blog SET title=blog.title, content=blog.content WHERE
id=blog.id;
```

Metoda delete (parameter) briše zapis (slog) iz tabele na osnovu prosleđene vrednosti id polja (argument parameter), čime se izvršava sledeća SQL naredba:

```
DELETE FROM blog WHERE id=param;
```

Dakle, JPA tehnologija obezbeđuje osnovne CRUD (Create, Retreive, Update, Delete) operacije nad bazom podataka primenom metoda save(), findAll(), findOne() i delete().

Kreiranje korisničkih JPA upita

JPA tehnologija takođe omogućava i kreiranje korisničkih (personalizovanih) JPA upita.

Sintaksa personalizovanih JPA upita je navedena ispod.

```
<return type> findBy<WhatEverWeWantToFindBy>(<SUPPLY PARAMETERS>);
```

U nastavku je dato nekoliko primera korisničkih JPA upita.

```
Blog findByTitle("ACO");
SELECT * FROM blog WHERE title="ACO" LIMIT 1
```

```
List<Blog> findByTitleContaining("ACO");
SELECT * FROM blog WHERE title LIKE "%ACO%"
```

```
List<Blog>  findByTitleContainingOrContentContaining("metaheuristic
s","metaheuristics");
SELECT * FROM blog WHERE title LIKE "%metaheuristics%" OR content
LIKE "%metaheuristics%"
```

U svim navedenim primerima primećuje se da je findBy klauzula ključna reč upita. Nakon ove klauzule navodi se naziv kolone (Title ili Content u ovom slučaju), zatim ograničenja za izvršavanje upita, kao što su Contains, Containing, GreaterThan, LessThan, itd. Takođe se koriste operatori Or i And kako bi se spojila različita ograničenja (slično kao SQL logički operatori OR i AND).

Primeri još nekih JPA upita i ekvivalentnih SQL naredbi navedeni su ispod.

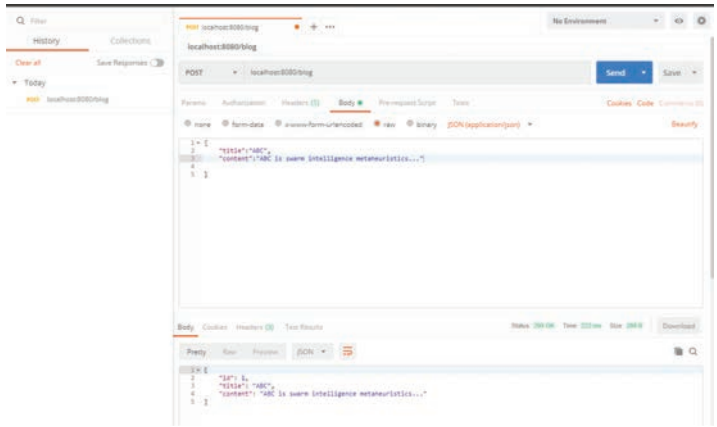
```
JPA1: findByIdGreatherThan(13)
SQL1: WHERE id > 13
JPA2: findByIdGreatherThanAndTitleContaining(13, "BA")
SQL2: WHERE id > 13 AND title LIKE "%BA%"
JPA3:findByIdGreaterThanEqualOrTitle(13, "ABC")
SQL3: WHERE id >= 13 OR title = "ABC"
```

Testiranje RESTful veb servisa veb bloga sa bazom podataka

U nastavku je navedeno nekoliko primera testiranja Blog veb servisa pomoću klijentske Postman aplikacije.

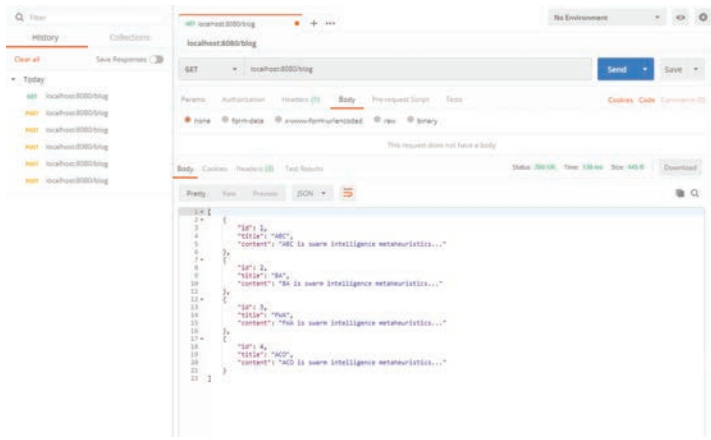
U prvom primeru je prikazan HTTP POST zahtev (URL: localhost:8080/blog), kojim se kreira novi slog tabeli blog (novi blog). Ovaj primer prikazan je na slici 12.34.

Iz primera se vidi da su u telu HTTP zahteva definisane vrednosti samo za title i content polja, dok je vrednost za polje id automatski generisao MySQL.



Slika 12.34. Primer 1: kreiranje novog bloga

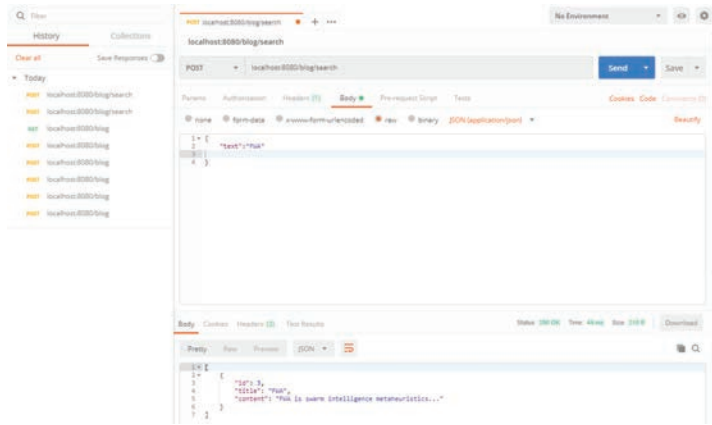
U drugom primeru, koji je dat na slici 12.35, prikazano je izvršavanje HTTP GET zahteva (URL: localhost:8080/blog), kojim se prikazuju svi slogovi iz tabele blog u JSON formatu.



Slika 12.35. Primer 2: prikaz svih blogova iz baze

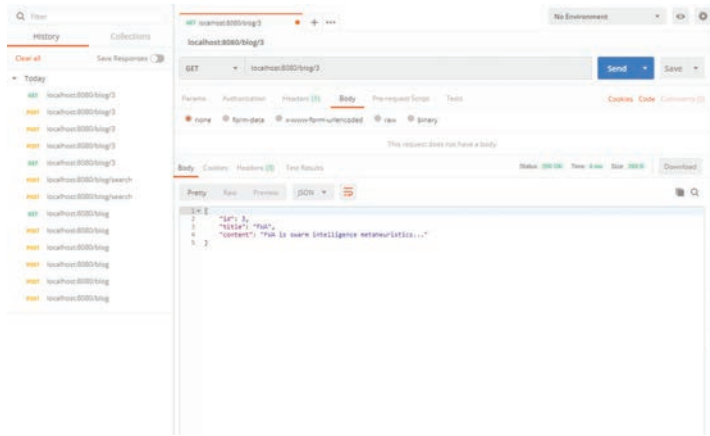
U trećem primeru prikazano je izvršavanje HTTP POST zahteva (URL: localhost: 8080/blog/search) kojim se prikazuju svi slogovi iz tabele blog u JSON format koji zadovoljavaju određene kriterijume pretrage.

Ovaj primer je prikazan na slici 12.36.



Slika 12.36. Primer 3: prikaz blogova koji zadovoljavaju određen kriterijume pretrage

Konačno, u četvrtom primeru dato je izvršavanje HTTP GET zahteva kojim se prikazuje samo jedan blog iz baze na osnovu zadate id vrednosti (slika 12.37).



Slika 12.37. Primer 4: prikaz jednog bloga iz baze

LITERATURA

1. Živković D., *Osnove Java programiranja*, 8. izdanje, Univerzitet Singidunum, 2017.
2. Živković D., *Java programiranje*, 2. izdanje, Univerzitet Singidunum, 2013.
3. Jevremović A., Veinović M., *Internet Tehnologije*, 1. izdanje, Univerzitet Singidunum, 2013.
4. Harold E.R., *Java Network Programming*, 4th edition, O'Reilly, 2013.
5. Walls, C., *Spring in Action*, 4th edition, Manning Publications, 2014.
6. Walls, C., *Spring in Action*, 5th edition, Manning Publications, 2018.
7. Prasanna, D. R., *Dependency Injection, Design patterns using Spring and Guice,* Manning Publications, 2009.
8. Enriques R., Salazar A.C., *RESTful Java Web Services Security*, PACKT Publishing, 2014.
9. Reimer M., *Building RESTful Web Services with Java EE 8: Create Modern RESTful Web Services with the Java EE 8 API*, PACKT Publishing, 2018.
10. Topley K., *Java Web services in a nutshell*, O'Reiley Publishing, 2003.
11. Long J., Bastani K., *Cloud Native Java: Designing Resilient Systems with Spring Boot, Spring Cloud, and Cloud Foundry 1st Edition, O'Reilly Publishing, 2017.*
12. *L. Ramnivas, AspectJ in Action, Second Edition*, Manning Publications, 2009.
13. Hutten D., *Cloud Computing Manipulation, Configuring and Accessing the Applications Online*, Amazon, 2017.
14. D. Gibson, *CompTIA Security+ Get Certified Get Ahead: SY0-501 Study Guide*, CompTIA, 2017.
15. Talal H., Noor H., Zeadally S., Alfazi A., Sheng Q. Z., *Mobile cloud computing: Challenges and future research directions*, Journal of Network and Computer Applications, Elsevier, Vol. 115, 2018, pp. 70-85.
16. Internet izvor: https://spring.io/projects/spring-framework/
17. Internet izvor: Java Socket programming Examples, http://cs.lmu.edu/~ray/notes/javanetexamples/
18. nternet izvor: https://www.w3.org/
19. Internet izvor: https:// developer.mozilla.org/en-US/
20. Internet izvor: https://www.eclipse.org/

CIP - Каталогизација у публикацији - Народна библиотека Србије, Београд

004.42:004.738.5(075.8)
004.438JAVA(075.8)
004.4’2SPRING

ЖИВКОВИЋ, Миодраг, 1982-
Programski jezici : Veb servisi i napredno Java programiranje u Spring okruženju / Miodrag Živković, Nebojša Bačanin Džakula, Eva Tuba. - 2. izd. - Beograd : Univerzitet Singidunum, 2022 (Beograd : Caligraph). - XIII, 439 str. : ilustr. ; 24 cm

Tiraž 700. - Bibliografija: str. 439.

ISBN 978-86-7912-698-6

1. Бачанин Џакула, Небојша, 1983- [аутор] 2. Туба, Ева, 1991- [аутор]

a) Интернет - Програмирање b) Програмски језик «Java» c) Радни оквир «Spring»

COBISS.SR-ID 57449481

© 2022.
Sva prava zadržana. Nijedan deo ove publikacije ne može biti reprodukovan u bilo kom vidu i putem bilo kog medija, u delovima ili celini bez prethodne pismene saglasnosti izdavača.

www.singidunum.ac.rs



Miodrag Živković
Nebojša Bačanin Džakula
Eva Tuba

PROGRAMSKI JEZICI

VEB SERVISI I NAPREDNO JAVA PROGRAMIRANJE U SPRING OKRUŽENJU

Programski jezik Java, kao jedan od najpopularnijih programskih jezika u svetu, svoju popularnost duguje pre svega eleganciji, savremenim konceptima i jednostavnom objektno orijentisanom modelu. Zbog sve veće popularnosti veb aplikacija, fokus programiranja u Javi se pomera sa tradicionalnih desktop aplikacija na mrežno programiranje i servisno orijentisano programiranje, što je ujedno i primarni fokus ovog udžbenika.

Kako bi se razumeo koncept servisno orijentisanog programiranja, potrebno je poznavati paradigme na kojima je zasnovan, uključujući objektno orijentisano programiranje (OOP), klijent – server programiranje i modele komponenti. OOP ima svoje specifične karakteristike, poput nasleđivanja, enkapsulacije i polimorfizma. Servisno orijentisano programiranje se nadgrađuje na OOP, dodavanjem premise da se problem koji se rešava može modelovati u obliku servisa koji objekat pruža ili koristi.

Jedan od najpopularnijih programskih okvira za pisanje veb aplikacija je Spring Framework, koji spada u kategoriju platformi otvorenog koda. Okvir se može koristiti za bilo koje Java aplikacije, ali osnovna primena je u implementaciji veb aplikacija nad Java EE platformom. Spring pored standardnih koncepata OOP uvodi i intenzivno korišćenje šablona injekcije zavisnosti i aspektno orijentisanog programiranja. Spring predstavlja jedno od standardnih rešenja za pisanje serverskih komponenti koda u Javi.

Univerzitet Singidunum