Međimursko veleučilište u Čakovcu

Aplikacija u oblaku i mikroservisi

SKRIPTA

Ver. 1.0

dr. sc. Bruno Trstenjak

Čakovec, 2023.

Autor: dr. sc. Bruno Trstenjak

Recenzenti: Ivan Heđi, v. pred.

mr.sc. Željko Knok, v. pred

Lektor: Sanja Novak, dipl. bibl.

Nakladnik: Međimursko veleučilište u Čakovcu

Za nakladnika: doc.dr.sc. Igor Klopotan, v. pred.

ISBN 978-953-8095-27-6

Sadržaj

1. Poglavlje: Uvod

U ovom poglavlju naučit ćete:

- ✓ Što je to oblak aplikacija
- ✓ Što je to mikroservis i svojstva mikroservis aplikacija
- ✓ Kako je formirana arhitektura virtualnog poslužitelja
- ✓ Osnovne informacije o standardnoj strukturi oblak aplikacije

1. Poglavlje : Uvod

Aplikacije u oblaku (*engl. Cloud*) danas su svakodnevica i prihvaćena strategija u razvoju web aplikacija. Strategija koja omogućava da se kreiraju virtualna okruženja na kojima će se pokretati aplikacije, razni uslužni servisi, a da pri tome korisnik nema potrebe za fizičkom računalnom opremom, ograničenjem prema fizičkoj lokaciji računala te potrebe za održavanjem cijelog sustava. Dostupnost infrastrukture koju nazivamo oblak jedan je od glavnih razloga procvatu ovakvih vrsta aplikacija. Upravo je tehnološki napredak interneta omogućio razvoj nove paradigme u razvoju web aplikacija.

Prvobitne su web aplikacije po karakteru bile „monolitne“, izrađene u jednoj cjelini s kompleksnim načinom nadogradnje i izmjena. Nova filozofija oblak aplikacija temelji se na kontejnerizaciji, mikroservisima i dinamičkom razvoju.

1.1. Osnovni pojmovi oblak tehnologije

Prije početka razvoja aplikacija namijenjenih za izvršavanje u oblak okruženju, potrebno je objasniti određene pojmove vezano za ovu vrstu aplikacija, razlike u pristupu razvoja aplikacija te samu strukturu web aplikacije.

Pojam mikroservisa i mikroservisne arhitekture navodi se kao glavno obilježje specifičnog načina dizajniranja aplikacija koja se temelji na obavljanju specifičnih zadataka, servisa (*engl. Services*). **Mikroservis** predstavlja dio aplikacije koji je namijenjen za obavljanje jednostavnog i ciljanog zadatka. Često se mikroservis povezuje s API sučeljem (*engl. Application Programming Interface*) koje koristi HTTP resurse te omogućuje pristup servisu s udaljenog uređaja. Spajanjem većeg broja ovakvih servisa dobiva se mikroservisna arhitektura web aplikacije. Ukoliko dođe do nekakvih promjena, nije potrebno mijenjati cijelu arhitekturu već samo određeni servis. Time se dobiva na izrazitoj fleksibilnosti web aplikacije i njenom brzom razvoju. Svaki mikroservis moguće je neovisno o drugim servisima postavljati (*engl. Deploy*) na virtualnu okolinu oblak okruženja (*engl. Cloud Framework*).

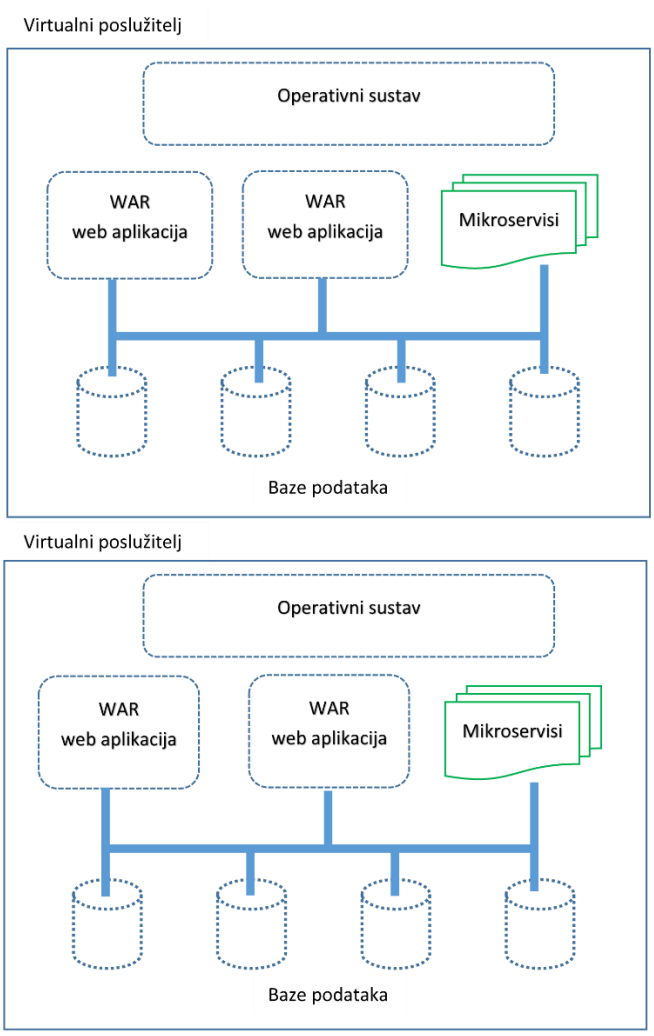
Oblak okruženje predstavlja zbirku razvojnih alata, biblioteka za razvoj i implementaciju web aplikacija koja će se pokretati u oblak okruženju. Trenutno postoje razna razvojna okruženja koja su najčešće povezana uz programski jezik i tehnologiju koja se koristi. Za potrebe ovog kolegija, koristit će se tehnologije i okviri koji su bazirani na Java programskom jeziku.

Pod pojmom **kontejnerizacijom** podrazumijeva se pripremanje web aplikacije i upakiravanje u jedinstvene cjeline. Cjelina koja sadrži, osim same aplikacije, i sve ostale potrebne resurse za pokretanje same web aplikacije. Kontejner je samo asocijacija prema objektu u koji se stavlja sve što je potrebno za rad aplikacije. Ovako pripremljen kontejner, postavlja se na virtualni poslužitelj bez potrebe za dodatnim instaliranjem potrebnih aplikacija. Kontejner koji se postavi na poslužitelj odmah je spreman za pokretanje aplikacije koja se nalazi u njemu. Takav pristup osigurava prenosivost jedne te iste aplikacije na različite platforme. Kontejneri pružaju mehanizam logičkog pakiranja aplikacije i njene radne okoline. Kontejneri su usmjereni više na virtualnoj razini poslužitelja i njegovog operativnog sustava. Osnovne prednosti ovakvog pristupa su:

- dosljedna okolina rada aplikacije,
- mogućnost pokretanja neovisno o vrsti poslužitelja i njegovih hardverskih karakteristika,
- aplikacija ima vlastite resurse za svoj rad neovisno o drugim web aplikacijama.

1.2. Arhitektura oblak okoline i aplikacije

Za korištenje web oblak aplikacije potrebno je uskladiti aplikaciju s oblak okruženjem i arhitekturom virtualnog poslužitelja. Na slici 1. prikazana je standardna struktura i osnovne komponente jednog oblak okruženja i web aplikacije.



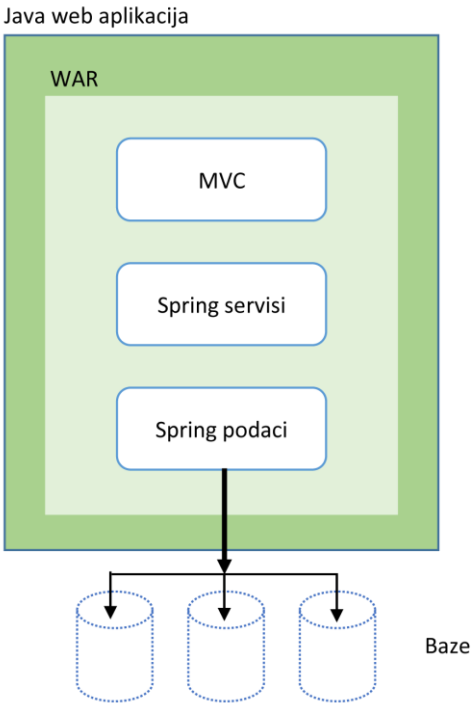
Slika 1. Standardna arhitektura oblak okruženja za različite operacijske sustave

Temelj svakog oblak okruženja čini virtualni poslužitelj. Riječ „virtualni“ znači da korisnik ne zna fizičku lokaciju stvarnog poslužitelja na kojem se pokreće aplikacija. Postoji mogućnost da se na istom virtualnom poslužitelju pokreću različite web aplikacije ili niz mikroservisa. Operativni sustav brine o raspodjeli resursa i o tome da nema štetnog utjecaja jedne aplikacije na drugu aplikaciju. Na svakom virtualnom poslužitelju nalazi se određeni operativni sustav. Operativni sustavi su konfigurirani da podržavaju različite vrste web aplikacija bez obzira na tehnologiju koja je korištena. Najčešće se kompatibilnost rješava pomoću kontejnera o kojima je prije bilo riječi. Na sam virtualni poslužitelj postavljaju se web aplikacije ili mikroservisi. Ovisno o strukturi aplikacija i mikroservisa, realizirano je

njihovo povezivanje s bazama podataka. Baze su uobičajeno postavljene na istom poslužitelju. Ukoliko se radi za dislociranu bazu, tada se koriste određeni protokoli koji omogućavaju razmjenu podataka i pristup podacima. Broj aplikacija i baza ovisi o kapacitetu poslužitelja. Konfiguracija virtualnog poslužitelja može se mijenjati prema potrebi korisnika i opterećenju.

Na slici 2. prikazana je web aplikacija s oznakom WAR (*engl. Web archives*). WAR predstavlja zapakiranu java web aplikaciju. U toj datoteci nalazi se sama web aplikacija kao i sve biblioteke koje aplikacija koristi u svom radu. Ista filozofija koristi se za java mikroservise. Jedina razlika je u tome što su aplikacije mikroservisa znatno manje zapremine u odnosu na klasične web aplikacija.

Standardna java web aplikacija sastoji se od logičko upravljačkog sloja, web sloja za prikaz podataka, raznih servisa te sloja za pristup i upravljanje podacima koji se koriste u komunikaciji s bazama podataka. Na slici 2. prikazana je WAR struktura bazirana na Java tehnologiji i Spring Boot okolini.

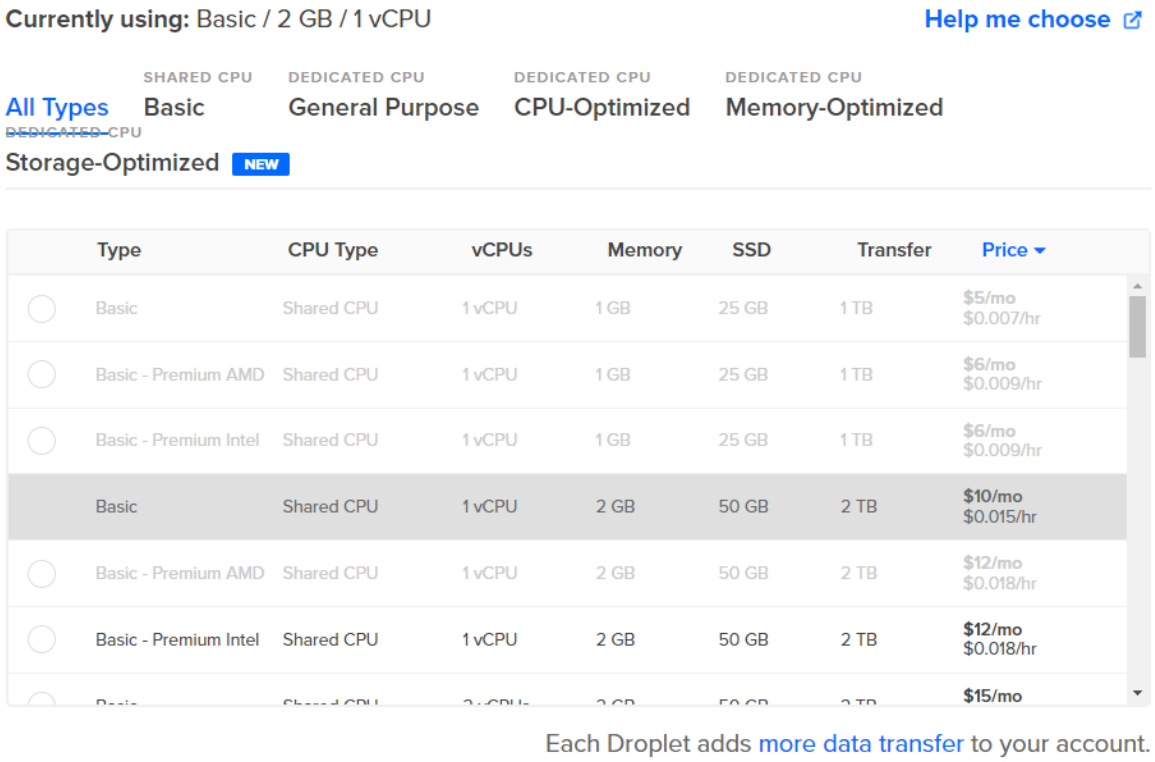


Slika 2. Arhitektura java Spring Boot web aplikacije

U samoj strukturi nalaze se tri komponente. MVC struktura (*engl. Model View Controller*) je jedna od poznatih načina organiziranja web aplikacija. M se odnosi na model podataka koji se koristi za pristup podacima u bazi podataka i njihov prikaz na web stranici aplikacije. V se odnosi na vizualni izgled web stranice. To su najčešće HTML predlošci web stranica s ugrađenim skriptnim naredbama kao što je ***Javascript*** ili ***Thymeleaf*** jezik. C se odnosi na kontroler koji se koristi za organizaciju cijele logike web stranica i rada same aplikacije.

Spring servisi odnose se na razne servise koji se mogu ugraditi u web aplikaciju te mogu biti aktivni bez obzira na stanje aplikacije. Spring podaci su u biti repozitoriji te razna sučelja za pristup i komunikaciju s bazama podataka. U svakoj složenijoj web aplikaciji, obavezno se koristi barem jedna baza podataka koja može biti relacijska ili nerelacijska baza.

Svakom virtualnom poslužitelju dodjeljuje se IP adresa te mu je moguće dodijeliti web domenu za pristup aplikaciji putem web naziva domene. Sama struktura značajki virtualnog poslužitelja određena je osnovnim podacima virtualnog poslužitelja kako je prikazano na slici 3. To su vrsta i jačina procesora, broj procesorskih jezgri koje će aplikacija koristiti, veličina memorije, veličina diska te predviđeni transfer podataka koji će se odvijati između oblak aplikacije i korisnika putem web ili mobilne aplikacije. Kako se virtualni poslužitelji vrlo često iznajmljuju, odabrana konfiguracija donosi i financijske izdatke, na mjesečnoj ili godišnjoj razini. Pojedini operateri daju mogućnost da se plaća samo ono što se utroši od prometa podataka u nekom vremenskom periodu.



Slika 3. Struktura značajki virtualnog poslužitelja

Sve značajke virtualnog poslužitelja mogu se mijenjati prema potrebi, bez da se zaustavlja izvođenje web aplikacije što je izuzetno važno za aplikacije i servise koji se trebaju neprekidno izvršavati. Vrlo često, virtualni poslužitelji se povezuju u različite konfiguracije, klastere i na taj način se osigurava pouzdanost aplikacija i servisa.

2. Poglavlje: Tehnologije izrade aplikacije

U ovom poglavlju naučit ćete:

- ✓ Osnovna svojstva Java programskog jezika
- ✓ Upoznat se s arhitekturom Spring okruženja
- ✓ Upoznati se s osnovnim svojstvima Spring Boot aplikacije
- ✓ Upoznati se s osnovnim programskim alatima koji će se koristiti u izradi aplikacija

2. Poglavlje: Tehnologije izrade aplikacije

Tehnologije koje se koriste u izradi oblak aplikacija vrlo često ovise o povezanosti s programskim jezicima koji se koriste u razvoju aplikacije kao i o karakteru same aplikacije ili servisa. Ukoliko se radi o aplikacijama koje su standardne web aplikacije gdje se koristi baza podataka te određena interakcija s korisnicima uz određenu razinu sigurnosti, mogu se koristiti tehnologije koje su bazirane na Java programskom jeziku.

Naravno, u razvoju složenih aplikacija dolazi do spajanja različitih programskih jezika i tehnologija čime se dobiva kvalitetni proizvod.

2.1. Programski okviri i jezici

U ovom kolegiju, kako je prije navedeno, aplikacije će biti temeljene na Java programskom jeziku. ***Java*** programski jezik je među najpopularnijim jezicima [1] te se koristi u raznim domenama. Jedna od važnih značajki ovog programskog jezika je WORA obilježje (*engl. Write Once, Run Anywhere*), što znači da je kod prenosiv na različite platforme i uređaje, bez potrebe za ponovnim prevođenjem. Java aplikacija se najčešće prevodi u oblik koji se naziva ***bytecode***, koji se kasnije može izvršavati na bilo kojem Java virtualnom stroju (*engl. Java Virtual Machine JVM*). Kada se spomene Java aplikacija, to predstavlja općenit pojam jer se aplikacije u tom jeziku mogu razvijati za potrebe ugrađenih (*engl. embedded*) aplikacija pa do poslužiteljskih naprednih aplikacija (*engl. Server enterprise application*).

Spring je aplikacijski programski okvir (*engl. Spring framework*) koji pruža sveobuhvatnu infrastrukturnu podršku za razvoj Java aplikacija. Temeljna ideja Springa je da se pojednostavi razvoj aplikacija u Javi. Opremljen je nekim lijepim značajkama kao što je ubrizgavanje ovisnosti te uključivanje raznih gotovih modula u aplikaciju i time skraćivati vrijeme razvoja same aplikacije. U Spring okruženju uveden je pojam ***kompozicije aplikacije IoC*** (*engl. Inversion of Control*), a odnosi se formaliziranju načina kompozicije razdvojenih komponenti u potpunu aplikativnu cjelinu.

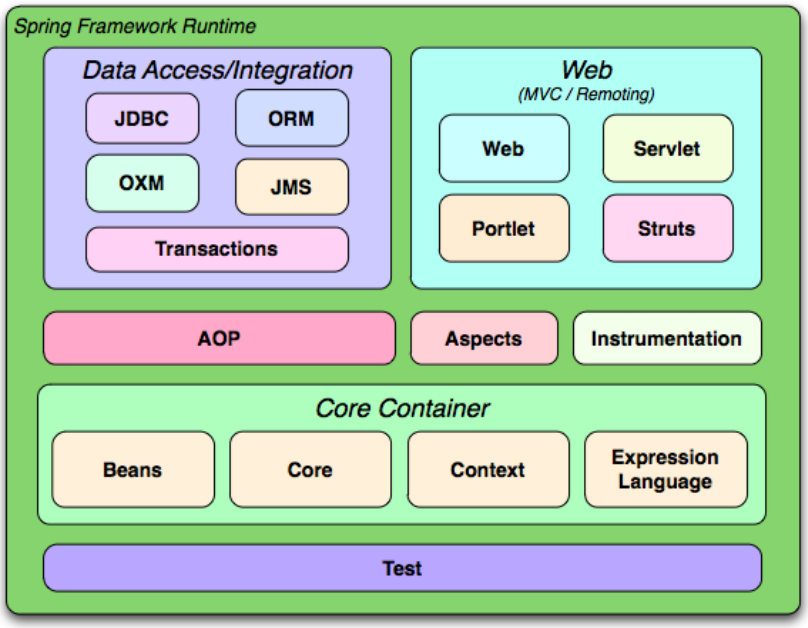
Moduli su cjeline od kojih se sastoji Spring okvir i koji su grupirani u sljedeće grupe: Core Container, Data Access/Integration, Web, AOP (*engl. Aspect Oriented Programming*), Instrumentation, Messaging, Test. Na slici 4. prikazana je arhitektura Spring razvojnog okruženja[2]. Spring okolina se sastoji od dvadesetak različitih modula, a detaljne informacije i pojedinačna svojstva svakog modula mogu se pronaći na poveznici i službenim stranicama Spring projekta.

Spring Boot je aplikacijski okvir koji se temelji na Spring okviru i proširuje ga s dodatnim mogućnostima. Spring Boot pruža dobru platformu za Java programere za razvoj samostalne i proizvodne Spring aplikacije koje se mogu jednostavno pokrenuti. Možete započeti s minimalnim konfiguracijama bez potrebe za cijelom postavkom Spring konfiguracije. Spring Boot skenira dostupne ovisnosti i određuje

[1] TIOBE indeks programskih jezika: https://www.tiobe.com/tiobe-index/

[2] Arhitektura Spring razvojnog okruženja: https://docs.spring.io/spring-framework/docs/3.0.x/spring-framework-reference/html/overview.html

koje elemente u aplikaciji je potrebno kreirati da bi se ostvarile željene funkcionalnosti. Programer ne mora više trošiti vrijeme za pisanje ponovnog koda već se može usredotočiti u razvoj poslovne logike.



Slika 4. Prikaz arhitekture Spring razvojnog okruženja

Spring Boot donosi niz prednosti za programere:

- automatska konfiguracija – osigurava automatsko stvaranje konfiguracija za funkcionalnosti koje su zajedničke brojnim Spring aplikacijama,
- starter ovisnost – uključuje starter ovisnost za određene funkcionalnosti, aplikacija dobiva sve potrebne biblioteke odgovarajućim verzijama,
- komandna linija – opcionalna funkcionalnost koja omogućuje pisanje cijele aplikacije samo pomoću aplikacijskog koda bez tradicionalnog prevođenja,
- aktuator – daje uvid u pozadinu rada pokrenute Spring Boot aplikacije,
- produktivnost – skraćuje se vrijeme razvoja aplikacija,
- pojednostavljeno postavljanje aplikacije,
- jednostavna skalabilnost - svojstvo aplikacije da obrađuje sve veću količinu posla dodavanjem resursa u sustav,
- kontejnerska kompatibilnost – jednostavno pakiranje cijele aplikacije u kontejner okolinu,
- minimalna konfiguracija – jednostavno postavljanje konfiguracije aplikacije i automatsko povezivanje konfiguracijskih parametara s pojedinim komponentama aplikacije.

Automatska konfiguracija je proces koji na temelju trenutnih uvjeta u kojima se aplikacija nalazi, odlučuje koja će se Spring konfiguracija primijeniti. Spring Boot automatski konfigurira vašu aplikaciju na temelju ovisnosti koje ste dodali projektu pomoću ***@EnableAutoConfiguration*** anotacije. Na primjer, ukoliko u aplikaciji koristite MySQL bazu podataka, definirali ste podatkovne modele, ali niste ručno uredili samu bazu, tada Spring Boot automatski provodi konfiguriranje bazu podataka prije pokretanja aplikacije. Sinkronizira karakteristike podataka i modela koje ste naveli u aplikaciji s karakteristikama tabela i atributa u samoj bazi.

Za pojašnjenje koncepcije Spring Boot aplikacija potrebno je objasniti još nekoliko pojmova koji su usko povezani s oblak aplikacijama.

Baze podataka sastavni su dio svake oblak aplikacije. Od baza podataka možemo razlikovati relacijske baze i nerelacijske baze. Relacijske baze su standardne baze koje se sastoje od niza tablica među kojima su definirane razne međusobne relacije i povezanost. Veoma često, ***MySQL*** je relacijska baza podataka koja se vrlo često koristi u web aplikacijama. Osim te baze, u projektima razvijenih na temelju Spring okruženja koristi se i ***H2*** baza podataka. Baza je napisana isključivo u Java programskom jeziku, otvorenog je koda. Može se koristiti na tri načina: kao ugrađena baza u neki sustav ili uređaj te je dostupna samo Java virtualnom stroju tog uređaja. Drugi način je korištenje te baze u načinu klijent-poslužitelj što je standardni način rada aplikacije. Baza se nalazi na poslužitelju te više klijenata putem određenih protokola pristupaju podacima. Treći način je kada se baza koristi u mješovitom pristupu. Pristupu podacima je lokalni ili udaljeni (*engl. remote*).

Hibernate je programski okvir za objektno-relacijsko mapiranje (ORM) podataka. Namjena tog okvira je provođenje poslovne logike bazirane na objektima u podatke u relacijsku bazu podataka. Osnovna ideja je da se objekti i njihovi podaci jednostavno pretvaraju u mapirane tablice baze podataka. Najčešće je klasa ta koja opisuje neki objekt mapiran u jedan red neke tablice podataka. Kako sam Spring ne nudi ORM za tu potrebu koristi se Hibernate i JPA (*engl. Java Persistence API*).

JPA pruža Java programerima mogućnost ORM za upravljanje relacijskim podacima u Java aplikacijama. Java Persistence sastoji se od četiri područja: aplikacijskog sučelja za pristup bazi podataka te komunikaciju s bazom, SQL jezik upita, ORM i korištenje EntityManager i meta podataka za razne operacije baze bez potrebe za pisanjem SQL naredbi.

Gradle je alat za automatizaciju izgradnje aplikacije (*engl. build automation*) otvorenog koda. Gradle omogućuje automatizaciju svih postupaka u procesu izgradnje aplikacije, povezivanje potrebnih biblioteka. Ovaj alat omogućuje upravljanje ovisnim datotekama (*engl. dependencies*) u procesu izgradnje i provođenja aplikacije. Gradle se brine o dohvaćanju svih potrebnih datoteka i biblioteka te njihovo pravilno povezivanje.

Apache Maven je softverski alat za upravljanje projektima. Njegov rad temelji se na konceptu modela projektnog objekta ***POM*** (*engl. Project Object Model*). Napisan je u Java programskom jeziku. Može upravljati izgradnjom projekta, izvještavati i dokumentirati iz središnjeg dijela informacija.

POM je XML datoteka u projektu oblak aplikacije u kojoj se nalaze sve informacije vezane za karakter rada aplikacije i način konfiguriranja aplikacije. Ova XML datoteka nalazi se u glavnoj mapi projekta i na temelju nje provodi se prevođenje izvornog koda aplikacije.

Docker je otvorena platforma za razvoj, isporuku i pokretanje aplikacija. Omogućuje da odvojite svoje aplikacije od svoje infrastrukture tako da možete brzo isporučiti softver. S Dockerom možete upravljati svojom infrastrukturom na isti način na koji upravljate svojim aplikacijama.

2.2. Programski alati

Za cjelokupni razvoj oblak aplikacija potrebno je upotrijebiti razne programske alate. Iz same arhitekture oblak aplikacije vidljivo je da se aplikacija sastoji od niza komponenti. Praktički, za svako od

komponenata potrebno je koristiti neki od alata. U ovom kolegiju za razvoj Spring Boot projekata koristi će se sljedeći programski alati:

- ***Eclipse IDE*** (*engl. Integrated Development Environment*) je razvojno okruženje koje je podržano od velikog broja dodataka za razne programske jezike i alate. Eclipse je uglavnom pisan u Javi. Dodavanjem raznih dodataka (*engl. plug-ins*) pretvara se u alat za razne programske jezike. Veoma je popularan u zajednici otvorenog koda. Koristit će u razvoju projekta za pisanje osnovnog koda oblak aplikacije.
- ***MySQL Workbench*** je vizualni alat za dizajn baze podataka koji integrira SQL razvoj, administraciju, dizajn, kreiranje i održavanje baze podataka u jedinstveno integrirano razvojno okruženje za MySQL sustav baze podataka. Koristit će se za pohranu podataka u oblak okruženju.
- ***Angular JS*** je JavaScript okvir otvorenog koda za razvoj aplikacija na jednoj stranici (*engl. Single Page Application*). Održava ga Google zajednica, a koristi se za razvoj korisničkog sučelja web aplikacije (*engl. Front-end*). Podržava razvoj web aplikacija baziranih na MVC konceptu.
- ***Spring initializr*** je web aplikacija koja omogućuje korisniku da generira strukturu projekta nove Spring Boot aplikacije. Ne generira nikakav programski kod, ali generira osnovnu strukturu projekta, Maven ili Gradle specifikaciju sa svim bibliotekama potrebnim za izradu programskog koda.
- ***Apache Tomcat*** aplikacijski poslužitelj zadužen je za smještaj Spring Boot aplikacije. Postavlja se na oblak poslužitelj i omogućava da u njega postavite veći broj oblak aplikacija. Aplikacijski poslužitelj brine o načinu pokretanju aplikacija, potrebnim memorijskim resursima te upravlja svim ostalim potrebnim resursima. Kontrolira ispravnost izvođenja aplikacija.

Navedeni alati nisu jedini alati koji se mogu koristiti za razvoj oblak aplikacija. Odabir alata koji će se koristiti prilikom procesa razvoja najčešće ovisi o tehnologijama na koje se oslanja razvoj aplikacije. Često, na zahtjev naručitelja aplikacije, odabire se osnovni programskim jezik koji će se koristiti za pisanje koda aplikacije. Zbog toga je moguća i neka druga kombinacija razvojnih.

Alati su ipak samo oruđe u rukama dobrog programera da se dođe do zacrtanog cilja.

3. Poglavlje: Izrada prve Spring Boot aplikacije

U ovom poglavlju naučit ćete:

- ✓ Kako inicijalizirati Spring Boot projekt
- ✓ Korištenje alata za inicijalizaciju aplikacije
- ✓ Što je ovisnost i način kako se ona dodjeljuje budućoj aplikaciji
- ✓ Osnovna struktura Spring Boot aplikacije
- ✓ Struktura glavnih klasa za prikaz početne web stranice

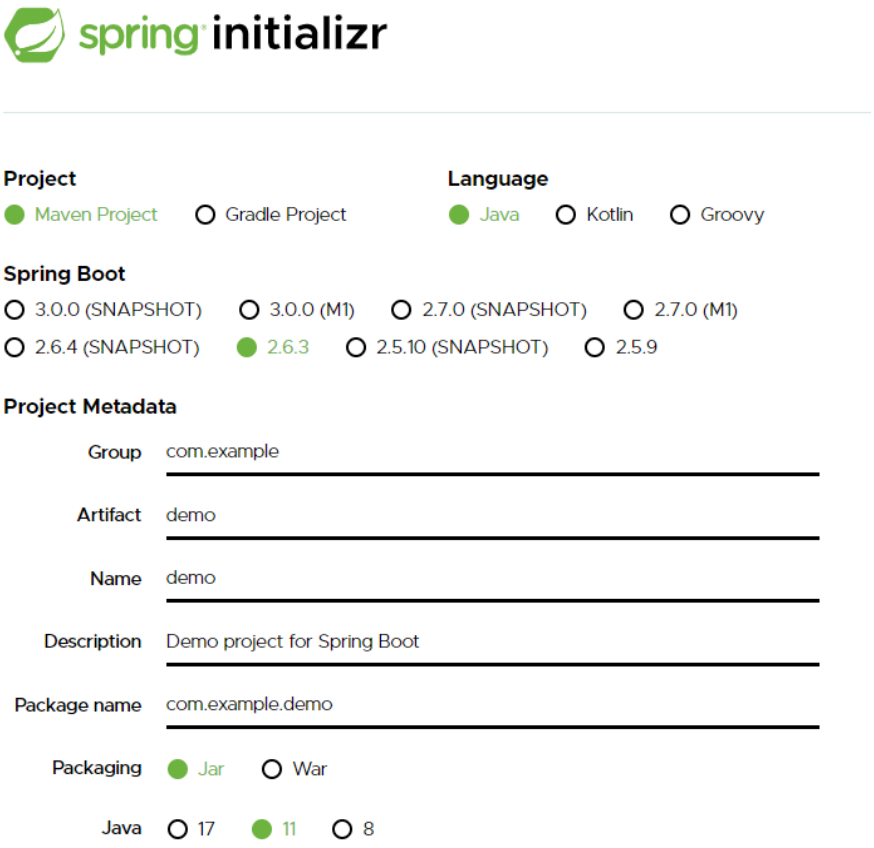
3. Poglavlje: Izrada prve Spring Boot aplikacije

Razvoj Spring Boot aplikacije moguće je započeti na nekoliko načina. Postoje alati koji donekle ubrzavaju razvoj. Međutim, potrebno je detaljno poznavati strukturu aplikacije da se točno zna koji dio projekta je generiran alatom. Da se zna gdje tražiti eventualno pogrešno generirani dio koda ili dio koda kojeg kasnije želite nadograditi.

Najjednostavniji način je korištenje Spring initializer koji će postaviti potrebnu strukturu budućeg projekta.

3.1. Inicijalizacija Spring Boot projekta

Alatu Spring initializr se može pristupiti na web lokaciji[3] . Na početnoj stranici prikazuju se osnovne karakteristike budućeg projekta i aplikacije. Na slici 5. prikazan je izgled web stranice alata.



Slika 5. Izgled početne stranice Spring initializra

Na lijevoj stranici ovog alata nalaze se elementi koji definiraju osnovnu strukturu buduće aplikacije. Korisnik se treba odlučiti da li će projekt za svoju strukturu koristiti Maven ili Gradle vrstu zapisa. Korisniku se predlaže da se odabere Maven struktura koja se i najčešće koristi u Spring Boot projektima. Gradle struktura se uglavnom koristi kod mobilnih aplikacija, međutim moguće je odabrati i tu opciju.

[3] Spring initializr - https://start.spring.io/

Nakon toga, korisnik treba odlučiti koji će programski jezik koristiti za pisanje programskog koda. U ponudi su Java, Kotlin i Groovy. Svi imaju svoje prednosti. Za potrebe ovog kolegija odabrat će se Java programski jezik. Poslije odabira programskog jezika, korisnik treba odlučiti koju verziju Spring Boot aplikacije će koristiti buduća aplikacija. Na slici je vidljivo da je ponuđena verzija 2.6.3 kao srednje rješenje.

Za kreiranje projekta potrebno je odrediti naziv grupe kojoj će pripadati projekt kao i naziv osnovnog paketa aplikacije. Poznato je da Java aplikacije imaju organizirane biblioteke u paketima za razliku od nekih drugih programskih jezika gdje su biblioteke organizirane u imenike i slične strukture.

Parametar Packaging određuje način na koji će buduća oblak aplikacija biti pakirana za rad u oblak okruženju. U ponudi su opcije ***JAR i WAR***. JAR je standardno pakiranje Java aplikacija i predstavlja standardnu arhivu u koju se pakiraju sve potrebne biblioteke i izvorni kod aplikacije. WAR predstavlja datoteku u kojoj je aplikacija zapakirana i prilagođena upravo za web okruženje. Za potrebe razvoja projekta koristit će se ***WAR*** opcija.

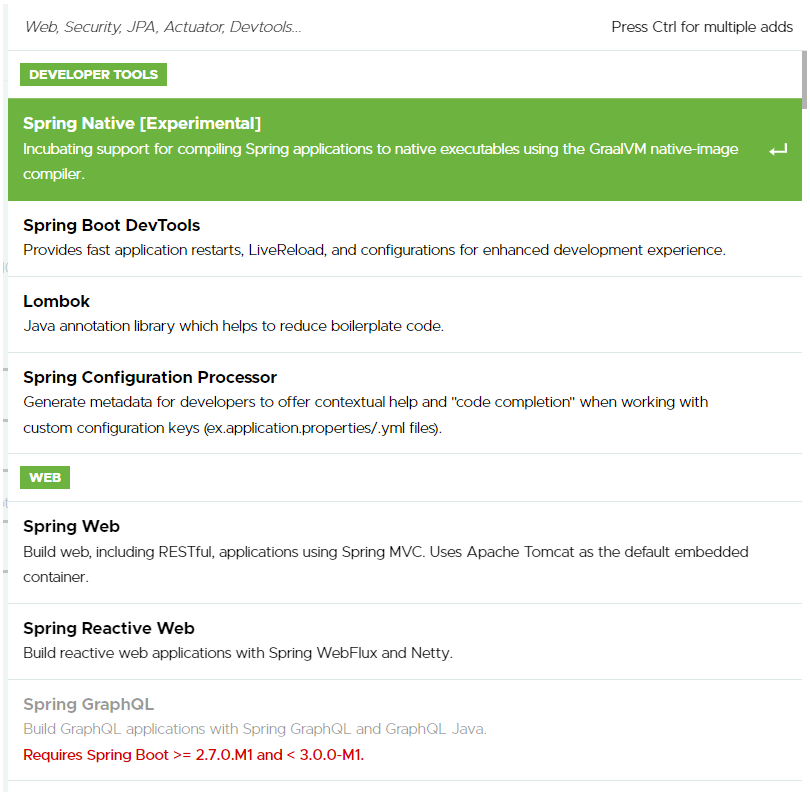
Parametar Java odnosi se na opciju koja definira koju Java verziju jezika će se koristiti. Java 8 predstavlja verziju 1. 8 koja je postala standard i provjereno okruženje. Novije verzije Jave su 11 i 17. Veći broj označava noviju verziju Jave. Korisnik može odabrati bilo koju verziju. Odabir verzije imat će utjecaja na kasniji odabir različitih biblioteka koje će se koristiti u aplikaciji.

S desne strane web stranice nalazi se dio koji označava ovisnost budućeg projekta, karakter buduće aplikacije. O čemu će aplikacija „ovisiti“, drugim riječima što ćemo sve uključiti u početnu aplikaciju. Odabirom ovisnosti u Maven XML datoteku dodavat će se struktura i način generiranja projekta. Pritiskom na ***Add dependecies*** otvara se prozor u kojem su ponuđeni razni dodaci aplikaciji.

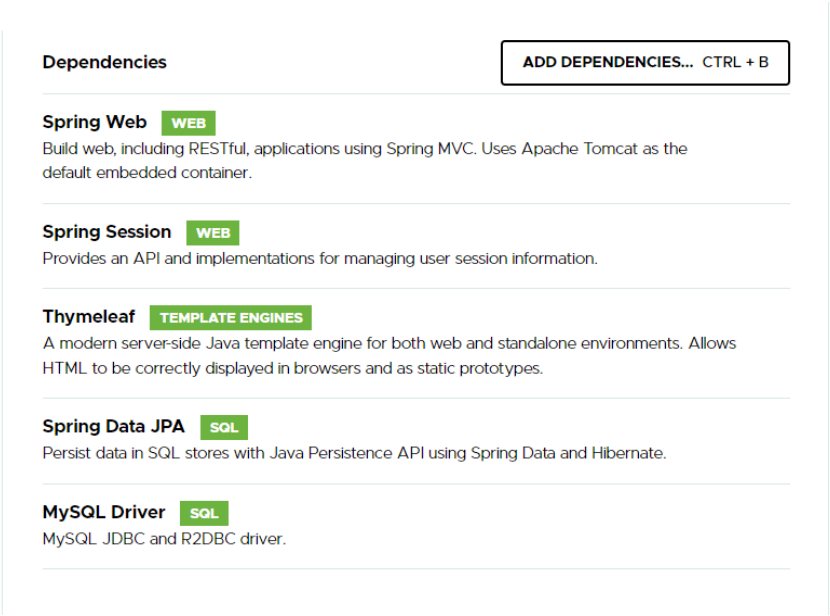
Ti dodaci su grupirani prema karakteru funkcionalnosti koje će aplikacija koristiti. Na slici 6. prikazan je prozor s ponuđenim dodacima. Zelenom bojom su označeni nazivi grupa, pa je tako na slici izdvojena WEB grupa dodataka. U toj grupi može se vidjeti da je ponuđen Spring Web dodatak koji će omogućiti pisanje RESTful aplikacije i Spring MVC strukture aplikacije. Ukoliko se radi o oblak aplikaciji, ovaj dodatak je obavezan. Klikom na bilo koji dodatak formira se popis ovisnosti buduće aplikacije. Ta ovisnost označava da će prilikom generiranja biti dodane biblioteke u samu aplikaciju. Prilikom prevođenja aplikacije, Spring će automatski uključiti i povezati programski kod i sve biblioteke.

Na slici 7. prikazan je primjer odabira ovisnih dodataka u aplikaciju, tako se na popisu nalaze sljedeće ovisnosti:

- ***Spring Web*** za pisanje web aplikacije,
- ***Spring Session*** za upravljanje sesijama web aplikacije,
- ***Thymeleaf*** za izradu web stranica uz upotrebu objekata,
- ***Spring Data JPA*** za korištenje Hibernate ORM modela,
- ***MySQL Driver*** za uključivanje upravljačkog programa za pristup MySQL relacijskoj bazi.



Slika 6. Dodavanje ovisnih biblioteka u aplikaciju

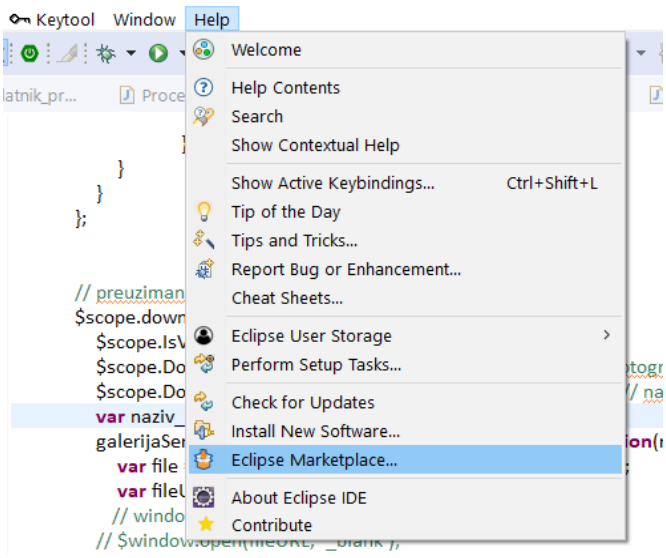


Slika 7. Popis ovisnih biblioteka koje će se koristiti u aplikaciji

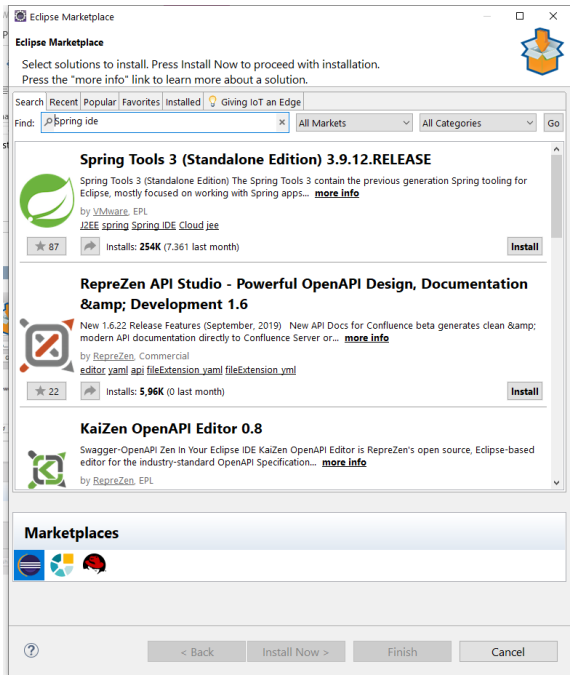
Sva uključena ovisnost tijekom razvoja projekta i aplikacije može se nadograditi i izmijeniti. Može se maknuti pojedina ovisnost ili modificirati s nekom drugačijom verzijom ovisnosti.

Pritiskom na gumb ***Generate*** započinje generiranje novog Spring Boot projekta. Alat formira projekt te ga zapakira i pošalje na lokalno računalo korisnika. Inicijalizaciju projekta i aplikacije moguće je izvršiti i pomoću Eclipse IDE aplikacije. Za to je potrebno da se u Eclipse uključi Spring Boot dodatak

(*engl. Plug-ins*). Dodaci se u Eclipse aplikaciji dodaju u izborniku ***Eclipse Marketplace*** kako je prikazano na slici 8.



Slika 8. Dodavanje novih alata u Eclipse aplikaciju

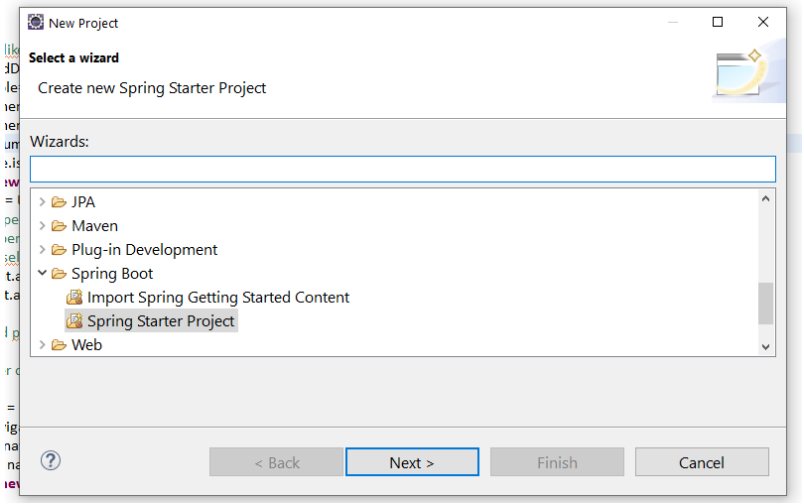


Slika 9. Dodavanje Spring Boot alata u Eclipse IDE

Eclipse marketplace je prostor na kojem se nalaze razni alati i dodaci Eclipse aplikaciji. Nakon odabira otvara se prozor s ponuđenim dodacima kao što je prikazano na slici 9. Korisnik upiše traženi pojam ili naziv dodatka. Klikom na gumb ***Install*** pokreće se instaliranje dodatka u Eclipse. Nakon toga, korisniku se omogućuje inicijalizacija Spring Boot aplikacije putem lokalnog računala.

Kreiranje projekta i aplikacije započinje odabirom izbornika ***Novi projekt*** te odabirom vrste projekta Spring Boot kako je prikazano na slici 10. Postupak definiranja ovisnosti buduće aplikacije veoma je

sličan prije opisanom postupku. Nakon završetka kreiranja projekta, na radnoj površini alata prikazuje se kreirani projekt i sama struktura aplikacije.



Slika 10. Kreiranje novog Spring Boot aplikacije i projekta

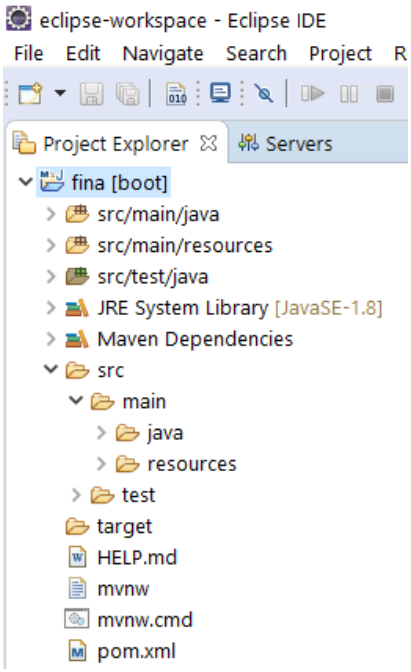
3.2. Struktura projekta i aplikacije

Zapakiranu aplikaciju koja je preuzeta s interneta, potrebno je staviti u mapu gdje će se razvijati aplikacija. Nakon raspakiravanja dobiva se sljedeća struktura, prikazano na slici 11.

Na ovom primjeru vidljivo je da se projekt naziva ***fina*** te da se sastoji od nekoliko mapa i nekoliko datoteka. Njihove uloge u projektu su sljedeće:

- ***scr*** je mapa u kojoj se nalazi izvorni kod aplikacije,
- ***main*** je mapa u kojoj se nalaze dvije podmape. Podmapa ***java*** sadrži Java programski kod kompletne aplikacije. Mapa ***resources*** sadrži datoteke koje su zadužene za formiranje web stranica, HTML kod i kompletni front-end web i oblak aplikacije. Ukoliko se radi samo o mikroservisima koji u pozadini izvršavaju određene servise, ova mapa je prazna,
- ***test*** je mapa u kojoj se stavljaju testne klase i metode s ciljem testiranja pojedinih komponenti aplikacije,
- ***mvnw*** je Maven datoteka u kojoj su definirane postavke za pokretanje oblak aplikacije
- ***pom.xml*** je datoteka u kojoj su navedene sve ovisnosti projekta. Koje sve biblioteke je potrebno preuzeti kod prevođenja aplikacije, koju verziju Java prevoditelja će se koristiti i još dosta informacija bitnih za prevođenje aplikacije i formiranja WAR datoteke,
- ***Maven Dependencies*** je mapa u kojoj se nalaze sve biblioteke (jar datoteke) koje su automatski preuzete s interneta, a potrebne za rad aplikacije,
- ***JRE*** (*engl. Java Runtime Environment*) mapa u kojoj se nalaze standardne Java biblioteke potrebne za kreiranje Java virtualne mašine (engl. Java Virtual Machine JVM) za pokretanje aplikacije,

- ***src/main/java*** nalaze se prevedene klase izvornog koda zapakirane u pakete pogodne za izvršavanje,
- ***src/main/resources*** nalaze se resursi potrebni za formiranje HTML stranica aplikacije.



Slika 11. Struktura Spring Boot aplikacije

3.3. Maven

Maven je XML datoteka u kojoj se nalaze informacije o bibliotekama o kojim ovisi ispravnost rada buduće aplikacije. Kod 1 prikazuje sadržaj ***pom.xml*** datoteke koja je generirana prilikom inicijalizacije projekta putem Spring initializr. U kodu je pomoću xml tagova definirani pojedini elementi strukture.

Između tagova ***<dependencies> .... </dependencies>*** nalaze se podaci o ovisnim bibliotekama. Ovisno što se očekuje od aplikacije, ovaj dio se može proširivati i uređivati nakon inicijalizacije samog projekta. U primjeru koda vidljivo je da projekt koristi verziju Java 1.8 te Spring Boot verziju 2.6.3. Rezultat ovih ovisnosti rezultirat će preuzimanjem jar datoteka.

```
<?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
https://maven.apache.org/xsd/maven-4.0.0.xsd">
        <modelVersion>4.0.0</modelVersion>
        <parent>
                <groupId>org.springframework.boot</groupId>
                <artifactId>spring-boot-starter-parent</artifactId>
                <version>2.6.3</version>
                <relativePath/> <!-- lookup parent from repository -->
        </parent>
        <groupId>hr.spring</groupId>
        <artifactId>fina</artifactId>
        <version>0.0.1-SNAPSHOT</version>
```

```
    <name>fina</name>
    <description>Spring Boot projekt </description>
    <properties>
        <java.version>1.8</java.version>
    </properties>
    <dependencies>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-data-jpa</artifactId>
        </dependency>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-thymeleaf</artifactId>
        </dependency>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-web</artifactId>
        </dependency>
        <dependency>
            <groupId>org.springframework.session</groupId>
            <artifactId>spring-session-core</artifactId>
        </dependency>

        <dependency>
            <groupId>mysql</groupId>
            <artifactId>mysql-connector-java</artifactId>
            <scope>runtime</scope>
        </dependency>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-test</artifactId>
            <scope>test</scope>
        </dependency>
    </dependencies>
    <build>
        <plugins>
            <plugin>
                <groupId>org.springframework.boot</groupId>
                <artifactId>spring-boot-maven-plugin</artifactId>
            </plugin>
        </plugins>
    </build>
</project>
```

Kod 1. Struktura Maven datoteke Spring Boot aplikacije

3.4. Pokretanje aplikacije

Za pokretanje svih aplikacija uvijek se koristi jedna ulazna točka, tako je slučaj i kod Spring Boot aplikacije. Početna točka za pokretanje aplikacije je glavna metoda (*eng. main method*). Kod 2. prikazuje izgled glavne metode.

```
package hr.spring.fina;


import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;


@SpringBootApplication
```

```
public class FinaApplication {


	public static void main(String[] args) {
		SpringApplication.run(FinaApplication.class, args);
	}
}
```

Kod 2. Glavna metoda za pokretanje aplikacije

Na početku se nalazi naziv paketa kojemu pripada ova klasa. Anotacija ***@SpringBootApplication*** označava da se radio o glavnoj klasi ***FinaApplication***. Anotacija su naredbe koje prevoditeljima daju dodatne informacije. Svaka anotacija započinje s karakterom ***@***. Iznad anotacije nalaze se naredbe za uključivanje potrebnih biblioteka. U aplikaciji je moguće koristiti veći broj različitih anotacija čime se osigurava dodatna kontrola kod kreiranja i prevođenja aplikacije. Također, moguće je koristiti ***@ComponentScan anotaciju*** koja osigurava da se prije svakog pokretanja aplikacije pokrene skeniranje svih njenih komponenti i procijeni njihova ispravna međusobna povezanost. Na taj način, umanjuje se mogućnost da aplikaciji nedostaju pojedine komponente ili da su komponente međusobno nekompatibilne uslijed raznih izmjena i nadogradnji. Također, moguće je koristiti ***@EnableAutoConfiguration*** anotaciju koja omogućuje da Spring Boot sam optimalno konfigurira cijelu aplikaciju.

3.5. Ostale osnovne komponente aplikacije

Da bi aplikacija bila upotrebljiva u oblak okolini, bilo da je to MVC web aplikacija ili mikroservis, potrebno je kreirati još nekoliko komponenti. Ukoliko se radi o MVC web aplikaciji, potrebno je dodati klasu kontroler u kojoj će se odvijati kompletna logika aplikacije, podatkovni modeli za zapisivanje podataka te HTML komponente za prikaz sadržaja na web pregledniku. Ukoliko se radi o REST servisu tada nema potrebe za HTML komponentama. Kod 3. prikazuje primjer klase kontrolera za prikaz inicijalne web stranice (index.html).

U ovom primjeru koda, potrebno je istaknuti nekoliko elementa. Sama klasa naziva se ***WebController***, naziv klase može biti proizvoljan. Karakter klase i njena namjena definirana je anotacijom ***@Controller***, što označava da je to klasičan web kontroler. Klasa ima samo jednu metodu pod nazivom ***pocetak***. Iznad naziva metode nalazi se ***@RequestMapping*** anotacija koja označava putanju do web stranice. Sve putanje za pristup web stranicama su mapirane i one označavaju put domene za prikaz stranice. U ovom primjeru to je putanja za početak aplikacije ili web stranicu ***index.html***.

```
package hr.spring.fina.controllers;

@Controller
public class WebController {

 @RequestMapping(value ={"/", "/index"}, method = RequestMethod.GET)
 public ModelAndView pocetak(HttpSession request) {
```

```
        ModelAndView retVal = new ModelAndView();
        retVal.setViewName("index");
        return retVal;
    }
}
```

Kod 3. Kontroler za prikaz web HTML stranice

U kontroleru je navedena ***GET*** metoda pristupa ovom dijelu aplikacije, što označava da se od ove metode očekuje da vratiti određene podataka. U ovom slučaju, radi se o prikazu web stranici čiji se HTML predložak naziva index.

Sama web stranica koju prikazuje aplikacija nalazi se u mapi ***resources*** pod nazivom index.html. Kod 4. prikazuje primjer osnovne html datoteke. Naredbom ***setViewName*** u kontroleru, daje se instrukcija da kontroler prikaže upravo HTML predložak pod nazivom index. U ovom dijelu moguće je navesti bilo koji drugi predložak i time će se prikazati drugačija web stranica. Također, u kontroleru je moguće dodati i određene naredbe na razini poslužitelja. Naredbe za evidentiranje sesija događaja, naredbe za zapisivanje raznih podataka o vremenu pristupa stranici (radi statističke analize) ili bilježenje podataka o klijentskom računalu od kojeg je došao zahtjev za prikaz stranice.

```
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
<html>
 <head>
  <meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-2">
 <title>FINA</title>
 </head>
 <body>
 </body>
</html>
```

Kod 4. HTML stranica početka web aplikacije

4. Poglavlje: Konfiguracija, anotacija i umetanje zavisnosti

U ovom poglavlju naučit ćete:

- ✓ Što je to anotacija
- ✓ Vrste anotacije i kako se koriste u aplikaciji
- ✓ Što je to IoC i DI
- ✓ Kako konfigurirati buduću Spring Boot aplikaciju

4. Konfiguracija, anotacija i umetanje zavisnosti

U cilju bržeg razvoja web aplikacija te smanjenja mogućnosti za pogreškama prilikom konfiguriranja aplikacije i spajanja raznih komponenti u jednu cjelinu, Spring Boot je uveo anotaciju i postupak automatskog konfiguriranja aplikacije. Promjenom bilo koje komponente, klase ili modula, Spring okruženje automatski provodi testiranje kompletne aplikacije te ispravnost međusobne povezanosti komponenti. Na taj način Spring detektira razne upitne situacije kreiranja objekata. Ukoliko utvrdi određenu sumnjivu situaciju, ne dozvoljava pokretanje aplikacije.

4.1. Anotacija

Spring Boot anotacija je oblik ***meta podataka*** koji pružaju podatke o programu. Drugim riječima, bilješke se koriste za pružanje dodatnih informacija o programu. To nije dio aplikacije koju razvijamo. To nema izravan učinak na izvršavanje koda koji označavaju. Ne mijenja karakter prevedenog programa.

U Spring Boot-u sve se samokonfigurira, a uz pomoć različitih anotacija ne moramo brinuti o međuovisnostima unutar klasa. Spring Boot omogućuje programeru da razvija aplikaciju bez da se brine o previše tehničkih stvari. Spring okvir umjesto korisnika odrađuje kompliciranu posao povezivanja, kontrole i redefiniranja pojedinih komponenti. Ako odlučimo Spring Boot koristiti uz Maven dostupni su nam mnogi paketi za razvoj koji se jednostavno uključuju u projekt kroz ***pom.xml*** datoteku.

Anotacije se mogu grupirati prema njihovim ulogama i važnosti u konstrukciji aplikacije. Svaka anotacija koja se nalazi u programskom kodu započinje s karakterom ***@***.

Osnovne anotacije:

- ***@SpringBootApplication*** - omogućava skeniranje i povezivanje svih komponenti i klasa korištenih u projektu te za korištenje automatske konfiguracije. Nalazi se u klasi glavnog programa i predstavlja kombinaciju tri anotacija: ***@EnableAutoConfiguration***, ***@ComponentScan***, i ***@Configuration***. Kod 5. prikazuje primjer korištenja ove anotacije.

```
package hr.spring.fina;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
@SpringBootApplication
public class FinaApplication {
        public static void main(String[ ] args) {
                SpringApplication.run(FinaApplication.class, args);
        }
}
```

Kod 5. Upotreba anotacije SpringBootApplication

- ***@EnableAutoConfiguration*** anotacija daje upute Spring okolini da provede automatsko konfiguriranje cijele aplikacije. Anotacija je ugrađena u novije verzije Spring Boot aplikacije u prijašnju anotaciju.
- ***@Configuration*** anotacija koristi se na razini klasa. Klasa koja je označena s tom anotacijom koristi Spring okvir u kreiranju bean objekata (*engl. Bean*) kao izvor definicija. U kodu 6. je prikazan način kako se konfigurira klasa i objekti tipa Student.

```
package hr.spring.fina.models;

import org.springframework.context.annotation.Configuration;

@Configuration
public class Student {

   private String ime;
   private String prezime;
   private String adresa;
   // get & set metode ...

}
```

Kod 6. Primjer anotacije za konfiguraciju aplikacije

- ***@ComponentScan*** je anotacija koja se koristi kada želimo prije pokretanja i kreiranja aplikacije da se provede skeniranje uključene ovisnosti i svih komponenti aplikacije. Provjerava se međusobna kompatibilnost i ovisnost. Na primjer, da li su klase podatkovnog modela ispravno povezane s tablicama baze podataka, provjerava se razina sigurnosti aplikacije i slične stvar.

Anotacije komponenata:

- ***@Component*** je anotacija koja je realizirana na razini klasa. Označava Java klasu kao ***bean*** objekt. Spring okvir skuplja komponente te ih stavlja u kontekst (*engl. context*) same aplikacije. Na taj način takvim komponentama omogućuje se ovisno uključivanje u razne dijelove aplikacije.
- ***@Service*** anotacije predstavlja alternativu @Component anotaciji i označava namjeru da se pojedine klase koriste kao dio servisa. Servis predstavlja sučelje pomoću kojeg se odvijaju razne logičke operacije na razini aplikacije. Primjer je korištenje servisa za pristup bazi podataka u kojem su definirane sve CRUD (*engl. Create Read Update Delete*) operacije. Kod 7. prikazuje servis zadužen za upravljanje podacima o studentima. Na početku koda nalazi se oznaka sučelje *(engl. interface*) u kojem se navode metode koje su podržane u budućem servisu. Sama implementacija svake od navedenih metoda vrši se u klasi servisa. U donjem dijelu koda nalazi se anotacija, odmah iznad naziva servisa ***StudentServiceImpl***.

```
package hr.spring.fina.services;

import hr.hrcity.models.Student;

public interface StudentService {

        Student createStudent(Student student);

        Student updateStudent(Student student);

        Iterable<Student> getAllStudents();

        Iterable<Student> getAllStudentsAktivni();

        Student getStudentById(long studentId);

        void deleteStudent(long id);

        Student getStudentImePrezime(String ime , String prezime);

 }
}


package hr.hrcity.services;


import java.util.Optional;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.data.domain.Page;
import org.springframework.data.domain.PageRequest;
import org.springframework.stereotype.Service;
import org.springframework.transaction.annotation.Transactional;


import hr.hrcity.exceptions.ResourceNotFoundException;
import hr.hrcity.models.Student;
import hr.hrcity.repository.StudentRepository;


@Service
public class StudentServiceImpl implements StudentService {
...
}
```

Kod 7. Primjer definiranja servisa za CRUD operacije

- ***@Repository*** anotacija označava repozitorij baze podataka. Anotacija se koristi na razini klase. Repozitorij predstavlja klasu koja omogućava direktan pristup bazi podataka. Klasu nazivamo DAO klasa (*engl. Data Access Object*). Klasa povezuje podatkovni model, tablicu baze podataka koju koristi model te moguće SQL naredbe za upravljanje podacima.
- ***@Controller*** anotacija označava web kontroler za web MVC Spring Boot aplikaciju. Kontroler predstavlja logiku same web aplikacije te određuje web stranicu koja će biti prikazana na

temelju zaprimljenog zahtjeva. Kontroler može koristiti nekoliko načina zaprimanja HTTP zahtjeva te omogućuje korisniku da šalje podatke koje će web aplikacija koristiti. U samoj aplikaciji može biti implementiran veći broj web kontrolera. Oni se najčešće odnose na pojedini modul aplikacije te na taj način omogućuju bolju organizaciju koda kao i naknadno efikasnije održavanje same aplikacije.

- ***@RestController*** anotacije koriste se za definiranje REST (*engl. REpresentational State Transfer*) kontrolera kojim se omogućuje udaljeni pristup oblak aplikaciji. Pristup podacima od strane mobilnih uređaja, web stranica i raznih uređaja. Kod 8. prikazuje definiranje klase REST kontrolera za pristup podacima studenata. Ispod anotacije nalazi se ***@RequestMapping*** anotacija kojom se određuje mapiranje putanje poziva pojedine metode u kontroleru. U primjeru je navedena metoda ***getAllStudents*** koja kao rezultat vraća popis svih studenata.

```
package hr.spring.fina.models;

@RestController
@RequestMapping("ABC")
public class StudentController {


  @GetMapping("/sviStudenti")
   public ResponseEntity < Iterable < Student >> getAllStudents() {
      return ResponseEntity.ok().body(this.studentService.getAllStudents());
   }
}
```

Kod 8. Primjer REST kontrolera

- ***@Required*** anotacija primjenjuje se na razine klase modela, za metode postavljanja podataka. Anotacija označava da atribut u modelu treba biti obavezno popunjen. U kodu 9. prikazan je primjer korištenja anotacije u metodi ***setOcjena***.

```
package hr.spring.fina.models;

public class Ocjena   {
  private Float ocjena;
 @Required
  public void setOcjena(Float ocjena)  {
    this. ocjena = ocjena;
  }
 public Float getOcjena()  {
  return ocjena;
 }
}
```

Kod 9. Primjer korištenja Required anotacije

- ***@Autowired*** anotacija omogućuje automatsko povezivanje Spring bean objekata s klasom kontrolera ili repozitorija. Spring vodi računa o cijelom postupku inicijalizacije objekta, svih varijabli i metoda. U primjeru koda 10. anotacija je stavljena za potrebu formiranja objekta

repozitorija. Na taj način osigurano je ispravno pozivanje repozitorija prije pozivanja metoda za pristup bazi podataka.

```
package hr.spring.fina.controllers;

@RestController
@RequestMapping("ABC")
public class StudentController {


 @Autowired
 private final StudentRepository repository;


 @GetMapping("/sviStudenti")
  public ResponseEntity < Iterable < Student >> getAllStudents() {
      return ResponseEntity.ok().body(this.studentService.getAllStudents());
   }
}
```

Kod 10. Primjer korištenja Autowired anotacije

Anotacije zahtjeva i odgovora (*engl. Request Responses*):

- ***@GetMapping*** anotacija preslikava HTTP GET zahtjeve na specifičnu metodu kontrolera. Koristi se za poziv kontrolera i web servisa za prikaz podataka. Ukoliko se anotacija koristi u kontroleru tada rezultat predstavlja web stranica, a ako se radi o REST kontroleru rezultat su podaci iz baze podataka.
- ***@PostMapping*** anotacija preslikava HTTP POST zahtjev na specifičnu metodu kontrolera, pri tome se šalje podaci za obradu.
- ***@PutMapping*** anotacija preslikava HTTP PUT zahtjev na specifičnu metodu kontrolera, za mijenjanje podataka u bazi podataka.
- ***@DeleteMapping*** anotacija preslikava HTTP DELETE zahtjev na specifičnu metodu kontrolera koja je namijenjena za brisanje podataka u bazi podataka.
- ***@RequestBody*** anotacija povezuje HTTP zahtjev sa specifičnim objektom (podatkom) kao ulazi parametar u metodu kontrolera.
- ***@PathVariable*** anotacija koristi se za definiranje varijable koja se koristi u HTTP zahtjevu te se prosljeđuje metodi kontrolera. Pogodna je za korištenje kod REST kontrolera.

Anotacije testiranja:

- ***@ SpringBootTest*** anotacija koja označava testnu klasu s metodama za testiranje pojedinih komponenti aplikacije.
- ***@Test*** anotacija koja metodu za testiranje pojedinačne metode u aplikaciji.
- ***@MockBean*** anotacija za stvaranje privremene verzije usluge za testiranje. Kreiranje ***Mock*** objekta.
- ***@Validated*** anotacija koristi se za provjeru (validaciju) objekata, njihov integritet i ispravnost.

Postoji veliki broj drugih anotacija koje se mogu koristiti u aplikaciji, za razne namjene. Više informacija o njima mogu se pronaći na službenim stranicama Spring Boot projekta i službenoj dokumentaciji.

4.2. Umetanje zavisnosti

Kada se govori o pojmu ***umetanja zavisnosti*** (*engl. Dependency Injection DI*), potrebno je objasniti što je zavisnost kada se govori o objektnom programiranju. Zavisnost je veza između dva i više objekata u kojoj je jednom objektu, za svoju implementaciju, potreban neki drugi objekt. Na slici 12. prikazana je zavisnost objekata. Prikazana su dva objekta: objekt A i objekt B. Objekt B je zavisan, Objektu A potreban je objektu B da bi radio svoj posao, stoga objekt B postaje zavisnost. Ukoliko se promatra ovaj primjer u kontekstu oblak servisa i web aplikacije, tada klasa A ovisi o servisu B. Servis može biti komponenta u Spring aplikaciji. Servis B treba biti spreman i dostupan u vrijeme izvršavanja kontrolera A.

Slika 12. Zavisnost objekta

Previše zavisnosti u kodu dovodi do problema koji nazivamo ***"teška zavisnost"*** (*engl. hard dependency*), koji čini naš kod teškim za ponovnu iskoristivost, teškim za testiranjem te težim za održavanjem.

Umetanje zavisnosti je samo jedna od tehnika pomoću koje izbjegavamo zavisnost. To je tehnika u kojoj jedan objekt osigurava zavisnosti drugom objektu. Ovaj koncept koji stoji iza umetanja zavisnosti u programiranju naziva se ***"inverzija kontrole"*** (*engl. Inversion of Control IoC*).

IoC je princip u softverskom inženjerstvu koji prenosi kontrolu nad objektima ili dijelovima aplikacije razvojnom okviru. Prema ovom konceptu, objekti ne trebaju kreirati druge objekte od kojih zavise već te objekte dobivaju eksterno. Spring okvir brine se za kreiranje zavisnih objekata. U primjeru koda 11. vidljiva je razlika između klasičnog kreiranja objekata i zavisnog objektnog programiranja. U primjeru

je prikazan kontroler koji za svoj rad koristi određeni servis. U samom konstruktoru kontrolera prosljeđuje se implementirani servis i provodi se inicijalizacija servisa.

Ukoliko želimo ostvariti osiguravanje da servis bude implementiran prije nego kontroler započne sa svojim radom, moguće je koristiti prije navedenu anotaciju. U konkretnom primjeru koristit će se ***@Autowired*** anotacija. Ona će osigurati da Spring Boot osigura sve preduvjete za kreiranje servisa. Ukoliko se pojave uvjeti u kojima nije moguće kreirati servis, Spring Boot okvir generira iznimku i sprječava da se takav nekompletan kontroler pokrene. Na taj način onemogućen je neispravan rad aplikacije. U kodu 12. prikazan je primjer kreiranja servisa pomoću anotacije.

```
package hr.spring.fina.controller;

public class ConstructorInjectionClient {

  private Service service;

 /**
 * Konstruktorski princip umetanja zavisnosti: Objekt se kreira
 * konstruktorom čiji su parametri svi interfejsi servisa od kojih zavisi, a
 * prilikom pozivanja konstruktora prosljeđuje se konkretni tipovi servisa.
 */
 public ConstructorInjectionClient(Service service) {
   this.service = service;
 }

}
```

Kod 11. Primjer zavisnog konstruktora

```
package hr.spring.fina.controller;

public class ConstructorInjectionClient {

  @Autowired
  private Service;


}
```

Kod 12. Primjer IoC pomoću anotacije

Wired predstavlja ožičenje ili u pogledu objekta njihova povezanost. ***@Autowired*** omogućuje Spring Boot aplikaciji da automatski razriješi ovisnosti između suradničkih objekata (bean-a) pregledom definiranih objekata.

Spring Boot omogućuje nekoliko načina povezivanja objekta pomoću tri osnovne anotacije.

Anotacija @Resource :

Ova anotacija omogućuje povezivanje različitih resursa koje koristi aplikacija. Najčešće se to odnosi na datoteke koje se nalaze u projektu i okruženju gdje se aplikacija izvršava. U primjeru koda 13. prikazan je jedan od primjera korištenja anotacije. Pozivom anotacije i navođenje imena resursa, povezuje se resurs s objektom klase File. U prvom dijelu primjera, poziva se anotacije u kojoj je specificiran naziv resursa. U drugoj klasi naveden je Bean objekt s tim nazivom kojeg anotacija poziva. Sama metoda poziva kao rezultat vraća objekt ***File***. U konkretnom primjeru to se odnosi na resurs datotekaTest.txt unutar projekta. Anotacija @Resource povezuje varijablu *mojaDatoteka* s resursom i osigurava kreiranje samog objekta klase File.

```
package hr.spring.fina.controller;

public class ConstructorInjectionClient {

   @Resource(name="namedFile")
   private File mojaDatoteka;

 ...

}

@Configuration
public class ApplicationContextResourceNameType {

   @Bean(name="namedFile")
   public File namedFile() {
      File datoteka = new File("datotekaTest.txt");
      return datoteka;
   }
}
```

Kod 13. Primjer upotrebe @Resource anotacije

Anotacija @Inject :

Za razliku od prijašnjeg načina uključivanja, *@Inject* anotacija povezivanje provodi prema nazivu, takozvana atributna zavisnost (*engl. Arbitrary Dependency*). Primjer koda 14. prikazuje primjenu korištenja ove anotacije. U prvom dijelu koda navedena je komponenta aplikacije, klasa ArbitraryDependency. U donjem dijelu koda navodi se Bean objekt koji kreira objekt prije navedene klase. U samom kontroleru kreira se objekt koji se anotacijom povezuje. Ova anotacija sastavni je dio Java ***Contexts and Dependency Injection*** (CDI) prvi puta objavljenog u Java 6 inačici.

```
@Component
public class ArbitraryDependency {

   private final String label = "Arbitrary Dependency";

   public String toString() {
      return label;
```

```
    }
}

public class ConstructorInjectionClient {

  @Inject
  private ArbitraryDependency fieldInjectDependency;
  ....

}

@Bean
public ArbitraryDependency injectDependency() {
   ArbitraryDependency injectDependency = new ArbitraryDependency();
   return injectDependency;
}
```

Kod 14. Primjena @Inject anotacije

Anotacija @Autowired :

Ova je anotacija već prije opisana. Iznad objekta jednostavno se postavi anotacija, a okvir sam kreira Bean objekt i povezuje ga s varijablom i budućim objektom. Ovaj se način vrlo često koristi kod povezivanja raznih servisa u kontroleru. Primjer koda 15. prikazuje kreiranje REST kontrolera u kojoj se koristi servis za pristup bazi podataka radi raznih SQL operacija. Na početku primjera nalazi se kontroler u kojem se koristi anotacija i povezivanje. Povezuje se servis StudentService i kreira se Bean ili objekt u samom klasi REST kontrolera. Okvir provjerava ispravnost povezanosti samog servisa s njegovom implementacijom. Okvir vrši alokaciju potrebnog memorijskog prostora.

```
package hr.spring.fina.controllers;

@RestController
@RequestMapping("ABC")
public class StudentController {

  @Autowired
  private StudentSrvis studentServis;
 ...
}

package hr.spring.fina.services;

public interface StudentService {
        Student createStudent(Student student);
        Student updateStudent(Student student);
        Iterable<Student> getAllStudents();
        void deleteStudent(long id);
        Student getStudentImePrezime(String ime , String prezime);
}
```

Kod 15. Primjer korištenja @Autowired anotacije u REST kontroleru

4.3. Konfiguracija aplikacije

Konfiguracija omogućuje da se prije pokretanje aplikacije osiguraju potrebni resursi, postave sigurnosni preduvjeti za pokretanje aplikacije te da se definiraju određena svojstva same aplikacije. Vrlo često u praksi se koriste posebno kreirane konfiguracijske klase koje se navodi u glavnoj klasi aplikacije, kako je prikazano u kodu 16.

```
@SpringBootApplication
@EnableConfigurationProperties(ConfigProperties.class)
public class EnableConfigurationDemoApplication {
   public static void main(String[] args) {
      SpringApplication.run(EnableConfigurationDemoApplication.class, args);
   }
}
@Configuration
@ConfigurationProperties(prefix = "mail")
public class ConfigProperties {
   private String hostName;
   private int port;
   private String from;
   // standard getters and setters
}
```

Kod 16. Konfiguriranje Spring Boot aplikacije

U kodu je navedena ***@EnableConfigurationProperties*** anotacija koja označava klasu u kojoj se nalaze instrukcije za konfiguraciju. U primjeru ove klase, označeni su atributi za slanje maila. Tijekom pokretanja aplikacije provodi se automatska konfiguracija. U tom procesu moguće je postaviti uvjete koji trebaju biti ispunjeni da se dio modula uključi ili ne uključi.

Za pokretanje automatskog konfiguriranja aplikacije potrebno je staviti ***@EnableAutoConfiguration*** ili ***@SpringBootApplication*** anotaciju u glavnoj klasi programa. Osim automatske konfiguracije aplikacije, moguće je postaviti i ***uvjetnu konfiguraciju*** (*engl. conditional configuration*). Uvjetna konfiguracija je konfiguracija koja je dostupna u aplikaciji samo ako je zadovoljen određeni uvjet. U primjeru koda 17. prikazana je uvjetna konfiguracija. U tom kodu stavljen je uvjet kreiranja servisa ukoliko je dostupna i ispravna klasa CRUD_baza, u suprotnom slučaju servis se ne kreira.

```
@ConditionalOnClass(CRUD_baza.class)
public MyService myService() {
...
}
```

Kod 17. Primjer uvjetne konfiguracije

5. Poglavlje: Thymeleaf

U ovom poglavlju naučit ćete:

- ✓ Što je to Thymeleaf
- ✓ Način integracije u Spring Boot aplikaciju
- ✓ Upotreba Thymeleafa u kontroleru
- ✓ Upotreba Thymeleafa u web aplikaciji

5. Thymeleaf

5.1. Što je to Thymeleaf

Thymeleaf u prijevodu znači majčina dušica. U stvarnosti predstavlja moderni Java mehanizam predložak za obradu i stvaranje HTML stranica i web Spring Boot aplikacija. Sam jezik pripada skriptnoj skupini jezika koji se izvršava na strani poslužitelja, a kao rezultat dobivamo standardne HTML web stranice. Omogućuje prenošenje podataka, atributa i modela podataka iz web kontrolera Spring Boot aplikacije kao i slanje podataka putem web forme na web stranici. Omogućuje razvoj web stranica u kojima se kombiniraju HTML naredbe i naredbe samog Thymeleaf skriptnog jezika.

Thymeleaf može obraditi šest vrsta predložaka, a svaki od njih naziva se Template Mode:

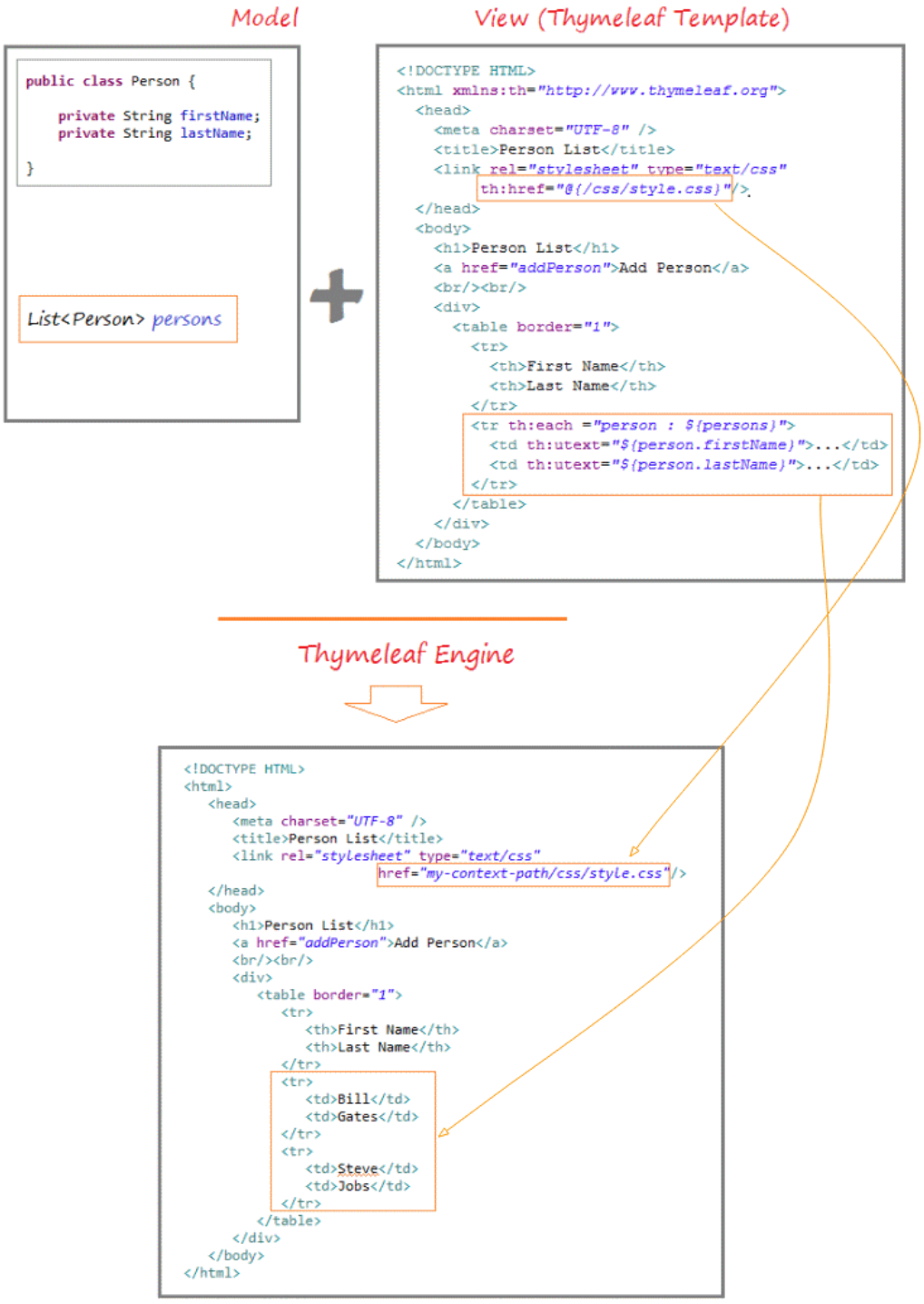
- XML
- Valid XML
- XHTML
- Valid XHTML
- HTML5
- Legacy HTML5

Thymeleaf mehanizam (*engl. Engine*) zadužen je za analizu (*engl. parse*) skriptnih naredbi koje se nalaze u datoteci te formiranje HTML predloška iz kojeg se kasnije formira web stranica. Na slici 13. prikazan je proces stvaranja web stranice. Podaci koji se prikazuju standardni su dio podatkovnog modela. U ovom primjeru to su podaci o osobi klase Person. Iz web kontrolera se prikupe podaci o osobama u obliku liste ***List<Person> persons***.

U Thymeleaf predlošku (*engl. Thymeleaf Template*) nalaze se izmiješane HTML i Thymeleaf naredbe. Thymeleaf skriptne naredbe su uokvirene. Prva naredba odnosi se na uključivanje css datoteke u HTML stranicu. Naredba ***th:href="@{/css/style.css"*** daje upute da se kod kreiranja predloška ova naredba zamijeni s putanjom te datoteke u HTML stranici. Drugi okvir predstavlja naredbu koja je u stvari programska petlja. Petlja prelazi po listi podataka koja je poslana od strane web kontrolera. Petlja generira redove u budućoj HTML tablici gdje će u prvoj koloni tablice biti prikazano ime osobe, a u drugoj koloni njegovo prezime.

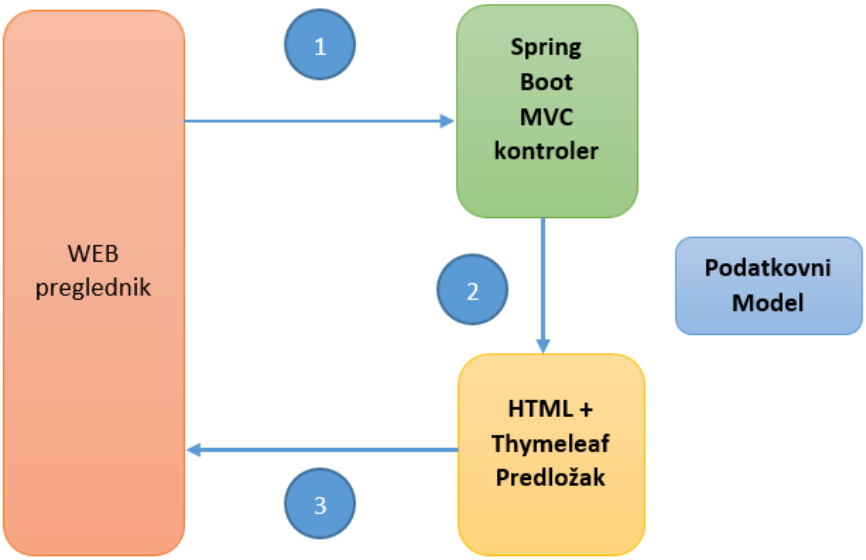
Ovaj Thymeleaf predložak šalje se u Thymeleaf prevodilac i mašinu za skriptnu analizu i generiranje HTML predloška. Nakon generiranja dobiva se HTML predložak kako je prikazano u ovom primjeru. Iz ovog primjera vidljivo je da su podaci zaštićeni te da se sama programska logika izvršava na strani poslužitelja te da korisnik nema uvid i pristup izvoru podataka. Na slici 14. prikazan je proces kreiranja predloška prema koracima izvođenja. U prvom koraku šalje se zahtjev od strane web preglednika za prikaz određene web stranice. Zahtjev se šalje u web kontroler Spring Boot aplikacije. U drugom koraku ovaj zahtjev prihvaća web kontroler koji određuje koji HTML & Thymeleaf predložak će se koristiti za prikaz web stranice. U tom ***koraku 2,*** uključuje se podatkovni model čiji podaci se žele

prikazati na web stranici. Sve zajedno se analizira te poslužitelj generira HTML predložak koji je u stvari sama web stranica koju treba web preglednik prikazati korisniku aplikacije. Formirani predložak kao odgovor na zahtjev šalje se web pregledniku i korisnik dobiva na svoj zaslon izgled web stranice što predstavlja ***korak 3***.



Slika 13. Princip korištenja Thymeleaf predloška za stvaranje web stranice[4]

[4] Izvor slike - https://medium.com/thymeleaf-basics-concepts/thymeleaf-1ef952db0740



Slika 14. Princip kreiranje HTML predloška

5.2. Integracija Thymeleaf u Spring Boot aplikaciju

Ukoliko u aplikaciji imamo potrebe za korištenje Thymeleaf naredbi, potrebno je u Spring Boot projekt integrirati biblioteku ***spring-boot-starter-thymeleaf***. U ***pom.xml*** datoteku potrebno je navesti ovisnost projekta o toj biblioteci. U kodu 18. prikazan je kod datoteke za uključivanje potrebne biblioteke. Nakon uključivanja ovih naredbi projekt samostalno preuzima sve potrebne biblioteke i uključuje ih u Spring Boot projekt.

```
<dependency>
   <groupId>org.springframework.boot</groupId>
   <artifactId>spring-boot-starter-thymeleaf</artifactId>
   <version>2.3.3.RELEASE</version>
</dependency>
```

Kod 18. Uključivanje Thymeleaf biblioteke u Spring Boot projekt

5.3. Korištenje podatkovnih modela i prikaz podataka

Thymeleaf kao skriptni jezik koji se izvodi na razini poslužitelja, ima mogućnost preuzimanja i slanje podataka prema kontrolerima u samoj Spring Boot aplikaciji. Sve naredbe koje se odnose na Thymeleaf započinju s prefiksom ***th:*** što Thymeleaf analizatoru daje naredbu da se radi o skriptnoj naredbi koju je potrebno prevesti i generirati zamjenski HTML kod.

Prikaz teksta

Sama sintaksa naredbi i tag atributa je tipa ***th:text***. Nakon toga dolazi tekst ili atribut koji je povezan s tekstom koji želimo prikazati na web stranici. U primjeru koda 19. prikazana je naredba za prikaz pozdravnog teksta. U nastavku naredbe za ispis teksta dolazi naziv atributa čiji tekst želimo ispisati. U

ovom slučaju taj atribut ili svojstvo nalazi se zapisano u datoteci s ekstenzijom ***.properties*** . Sintaksa ove naredbe je ***th:text = #{...}*** .Ovakav pristup moguće je koristiti ukoliko želimo da aplikacija podržava višejezičnost. U kodu je prikazana naredba za prikaz teksta u *** *** HTML tagu.

```
<p th:text="#{home.pozdrav}">Pozdrav!</p>
```

Kod 19. Naredba za prikaz teksta u Thymeleaf skripti

Datoteke sa svojstvima stavljaju se u projektu u mapu ***/WEB-INF/templates/***home.properties. U ovom dijelu projekta nalaze se svi elementi koji se koriste za formiranje web stranica. Kod 20. prikazuje sadržaj datoteke i atribute.

```
home.pozdrav= Dobro došli na našu web stranicu
home.poduzece = FINA
home.adresa=Zagreb
```

Kod 20. Primjer sadržaja datoteke svojstava i konstanti

Prikaz modela

Podaci iz modela ili objekata dolaze od strane web kontrolera u samoj aplikaciji. Ti podaci se deklariraju kao atributi koji se šalju u Thymeleaf predložak. Sintaksa naredbe za prikaz sadržaja nekog atributa je sljedeća: ***th:text= " ${ime_atributa} "*** . U kodu 21. prikazan je način prikaza sistemskog vremena koje je trenutno na poslužitelju. U kontroleru se odredi sistemsko vrijeme i kao atribut pod nazivom se pošalje u predložak.

```
@Controller
public class IndexController
{

  @RequestMapping(value="/", method=RequestMethod.GET)
  public ModelAndView home()
  {
    ModelAndView retVal = new ModelAndView();
    retVal.setViewName("index"); // definiranje naziva predloška

    return retVal;
  }

  @RequestMapping(value = "/kontakt", method = RequestMethod.GET)
   public ModelAndView kontakt(Model model) {

      ModelAndView modelAndView = new ModelAndView();
      modelAndView.setViewName("kontakt");

      model.addAttribute("sistemskoVrijeme", dateFormat.format(new Date()));

      return modelAndView;
  }
```

```
   @RequestMapping(value = "/popis", method = RequestMethod.GET)
    public ModelAndView popis(Model model) {

        ModelAndView modelAndView = new ModelAndView();
        modelAndView.setViewName("popis");

        List<Student> popis = new ArrayList<Student>();
        // punjenje podataka
        model.addAttribute("popisStudenata",popis);

        return modelAndView;
    }

}

@Entity
public class Student implements Serializable {
    @Id
    @GeneratedValue(strategy=GenerationType.IDENTITY)
     private long id_student;

     private String prezime;

     private String ime;

}
```

Kod 21. Web kontroler za formiranje modela podataka

```
// 1. prikaz sistemskog vremena
Trenutno sistemsko vrijeme na poslužitelju <span th:text="${ sistemskoVrijeme }" />

// 2. prikaz podataka o studentima
<tbody>
   <tr th:each="student: ${ popisStudenata }">
      <td th:text="${student. prezime }" />
      <td th:text="${student. ime }" />
   </tr>
</tbody>
```

Kod 22. Thymeleaf predložak za web stranicu

U ovom primjeru u kodu 22. prikazane su tri metode u web kontroleru. Prva metoda je metoda koja prikazuje prvu (index) stranicu. Metoda se naziva ***home*** i u svojem tijelu ima objekt klase ***ModelAndView***. Ova klasa je predviđena za povezivanje predložaka i podataka sa stvarnom HTML stranicom koja se prikazuje u web pregledniku. S naredbom ***setViewName("index");*** određuje se koji će se predložak prikazati za zahtjev koji je odabran u kontroleru i aplikaciji. U zaglavlju metode nalazi se putanja ili mapa za prikaz web stranice. U ovom primjeru navodi se da će putanja biti ***"/"*** što predstavlja početnu web stranicu neke domene.

Druga metoda je metoda ***kontakt***. Metoda je zadužena za prikazivanje podataka o kontakt podacima. Naredba ***modelAndView.setViewName("kontakt");*** određuje predložak koji se prikazuje. Predložak je datoteka pod nazivom kontakt i nalazi se u dijelu projekta ***/WEB-INF/templates/*** kako je u prijašnjem poglavlju objašnjeno. Naredbom ***model.addAttribute("sistemskoVrijeme",***

dateFormat.format(new Date())); određuje se trenutno sistemsko vrijeme, a ta informacija se prosljeđuje u predložak pod nazivom ***atributa*** "sistemskoVrijeme". U kodu 22. prikazana je upotreba te informacije. S naredbom ***th:text="${ sistemskoVrijeme }"*** prikazuje se na web stranici trenutno vrijeme na poslužitelju.

Treća metoda ***popis*** predviđena je za prikaz popisa studenata s pripadajućim podacima. U metodi poziva se servis za pristup podacima tablice Student. Formira se popis objekata koji se dodjeljuje atributu pod nazivom ***popisStudenata*** . Sama klasa Student je entiteta čiji početni kod je prikazan ispod koda kontrolera. U predlošku se koristi više skriptnih naredbi. Naredba ***th:each*** je zadužena da se realizira programska petlja u kojoj skripta pristupa popisu podataka o studentima. Automatski formira redove u HTML tablici. Broj redova ovisi o broju objekata studenata iz baze podataka.

5.4. Uvjetne naredbe i naredbe ponavljanja

Kako je Thymeleaf skriptni jezik i neka zamjena za programski jezik, u svojem popisu naredbi ima i naredbe grananja, uvjetne naredbe te razne naredbe ponavljanja za automatizaciju generiranja HTML naredbi koje ovise o količini podataka.

Naredba if and unless

Ovo je naredba za uvjetnu provjeru vrijednosti podatkovnih modela. Sintaksa naredbe je ***th:if="${uvijet}"*** . Ukoliko je postavljeni uvjet istiniti (*engl. TRUE*) prikazuje se tekst atributa. Naredba ***th:unless*** je nadogradnja if naredbe. Ona radi malo drugačije. Naredbom u prijevodu označava „osim ako“, primjer naredbe: ***<span th:unless="${student.spol} == 'M'" th:text="Ž" />*** Naredba ispisuje slovo Ž, osim ako atribut modela Student.spol nema vrijednost 'M'.

Naredba switch and case

To je naredba koja zamjenjuje više if naredbi, a sintaksa naredbe je ***th:switch*** . U kodu 23. prikazana je upotreba naredbe za provjeru spola studenta. U slučaju (*engl. case*) da je oznaka spola 'M' ispisuje se tekst Muški, a u slučaju da spol ima vrijednost Ž ispisuje se tekst Ženski.

```
<td th:switch="${student.spol}">
   <span th:case="'M'" th:text="Muski" />
   <span th:case="'Ž'" th:text="Ženski" />
</td>
```

Kod 23. Primjer korištenja switch-case naredbe

Naredba ponavljanja

Veoma često imamo potrebu da se na web stranici automatski generira sadržaj neke tablice i drugi oblik podataka. To se radi s naredbom ponavljanja th:each koja pristupa listi podataka te čita objekte i prikazuje vrijednosti njihovih atributa. U primjeru koda 24. prikazan je ispis podataka o studentima. Pri tome se provjeravaju određeni podaci i na temelju njihove vrijednosti formira se konačan tekst web stranice. U ovom primjeru prikazana je kombinacija već ranije spomenutih naredbi.

```
<tr th:each="student: ${popisStudenata}">
   <td th:text="${student.id_student}" />
   <td th:text="${student.ime}" />
   <td th:switch="${student.spol}">
       <span th:case="'M'" th:text="Muški spol" />
       <span th:case="'Ž'" th:text="Ženski spol" />
  </td>
</tr>
```

Kod 24. Primjer upotrebe th:each naredbe

Naredba slanja podataka

Kada se žele poslati podaci u web kontroler to se najčešće izvršava pomoću standardne HTML forme. U samom zaglavlju forme određuje se mapa akcije koja će se pokrenuti pritiskom na gumb za slanje podataka te vrsta objekta koji se koristi za upis podataka. U kodu 25. prikazan je jednostavni primjer prijave korisnika.

```
<form th:action="@{/prijava}" id="login-form" method="post" th:object="${loginData}" >
   <div class="input-box" title="Upišite korisničku oznaku">
   <div class="error">
     <p th:if="${loginError}" class="errorText">Greška: Neispravna korisnička oznaka ili zaporka.</p><br>
   </div>
  <div class="input-box" title="Upišite korisničku oznaku">
    <label class="entryText" for="username" aria-label="Upišite korisničku oznaku " >KORISNIČKA OZNAKA
    </label>
    <input th:field="*{email}" >
  </div>

  <div class="input-box" title="Upišite zaporku ">
    <label class="entryText" for="password" aria-label="Upišite zaporku >ZAPORKA
    </label>
    <input th:field="*{lozinka}" type="password" aria-label="Upišite zaporku " >
  </div>
  <button type="submit" class="login-btn" aria-label="Kliknite ovdje za prijavu u sustav" >PRIJAVA</button>
</form>
```

Kod 25. Primjer predloška za prijavu korisnika

Naredba ***th:action*** definira element koji je zadužen za pokretanje određene aktivnosti. U formi to je aktivnost slanja podataka, a mapa slanja označena je u nastavku naredbe ***="@{/prijava}"***. U zaglavlju forme navedeno je da se koristi objekt u koji će se zapisivati podaci. Naredba ***th:object="${loginData}"*** definira da se koristi objekt klase loginData. Klasa ima dva podatka, korisničko ime i zaporku.

U formi se nalaze Thymeleaf naredbe za upis podataka. Naredbom th:field daje se instrukcija za upis podatka u polje objekta. Podatak se upisuje u atribut pod nazivom email, a naredba je ***th:field="*{email}"*** . U naredbi se koristi oznaka * koja označava referencu na objekt i njegov atribut. U samoj formi imamo dva takva polja, a na kraju forme nalazi se gumb tipa **submit** koji je zadužen za pokretanje aktivnosti u formi. Kada se pritisne gumb, pokreće se aktivnost i poziva se metoda u web

kontroleru koji je mapiran s adresom ***"@{/prijava}"***. Kod 26. prikazuje metodu u web kontroleru koji prihvaća zahtjev.

```
@RequestMapping(value = "/prijava", method = RequestMethod.POST)
  public ModelAndView prijavaPost(Model model, @ModelAttribute LoginData logData, HttpSession
request)
 {
     // čitanje podatka o registraciji - super
    User user = this.userService.findByUserLogin(logData.getEmail(),logData.getLozinka());

     ModelAndView retVal = new ModelAndView();
     if (user == null ) {
          String messageError ="Greška: Neispravna korisnička oznaka ili zaporka.";
          model.addAttribute("loginError2", messageError);
          retVal.setViewName("/prijava");
     } else if (user.getId_user() <= 0 ) {
          String messageError ="Greška: Neispravna korisnička oznaka ili zaporka.";
          model.addAttribute("loginError2", messageError);
          model.addAttribute("loginError", true);
          retVal.setViewName("/prijava");
     }
     else if (user.getId_user() >0) {
          // sve je ispravno
          retVal.setViewName("redirect:/");
     }
     return retVal;
  }
```

Kod 26. Primjer metode za prihvat podataka iz forme registracije

U metodi se preuzima objekt ***logData*** u kojem se nalaze podaci upisani u formi. Pomoću servisa ***userService*** traže se podaci o korisniku iz baze podataka. Ukoliko podataka o tom korisniku nema, metoda ponovno vraća sve podatke u formu za registraciju uz poruku o pogrešci. Ukoliko je pronađen korisnik s tim pristupnim podacima, metoda preusmjerava kontrolu na početnu stranicu aplikacije gdje korisnik može početi koristiti aplikaciju.

6. Poglavlje: Povezivanje aplikacije s bazom podataka

U ovom poglavlju naučit ćete:

- ✓ Na koji način se ostvaruje povezivanje s bazom podataka
- ✓ JPA anotacija i način upotrebe
- ✓ Hibernate anotacije i način upotrebe
- ✓ Primjer zajedničkog korištenja JPA i Hibernate za pristup bazi podataka

6. Povezivanje aplikacije s bazom podataka

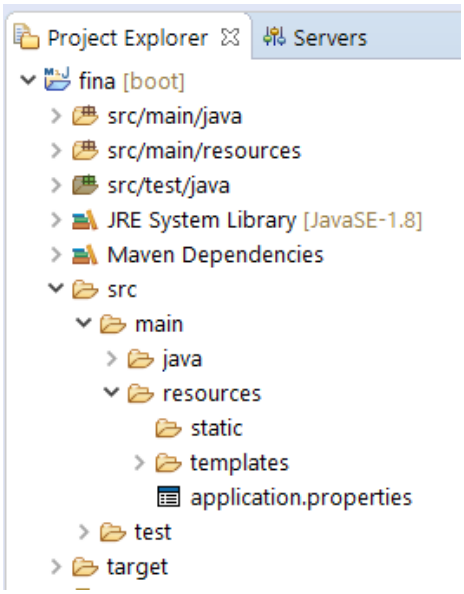
Kako je istaknuto u prijašnjem poglavlju, u Spring Boot aplikacijama koristi se ORM mapiranje kao temelj za upravljanje i pristup podacima u bazama podataka. Za to se koriste dvije osnovne tehnologije, a to su JPA i Hibernate. Spring Boot podržava rad sa svim vrstama baza podataka, relacijskim i nerelacijskim, ovisno o potrebi aplikacije.

6.1. Povezivanje aplikacije s bazom podataka

Prije upotrebe bilo koje baze podataka potrebno je osigurati sigurno povezivanje aplikacije s bazom podataka. U većini slučajeva sama baza podataka najčešće se nalazi instalirana na samom oblak poslužitelju na kojem se izvodi aplikacija.

Povezivanje aplikacije s bazom podataka moguće je na nekoliko načina. Prvi način je statičko povezivanje u kojem se u nekoj datoteci upisuju pristupni podaci za povezivanje. Drugi način je dinamičko povezivanje gdje se u posebnim klasama definiraju podaci za povezivanje. Podaci se uvijek koriste prilikom pokretanja same aplikacije. I jedan i drugi način je ispravan, a odabir ovisi o razvojnoj filozofiji same aplikacije.

Za statičko povezivanje u Spring Boot aplikacijama koristi se datoteka ***application.properties*** koja je sastavni dio svake aplikacije. Datoteka se nalazi u mapi resources kako je prikazano na slici 15. U toj datoteci upisuju se postavke koje želimo da se odnose na rad aplikacije, pristup resursima ili razne lozinke za slanje emaila, oznake portova pokretanja aplikacije, sigurnosne postavke same aplikacije.



Slika 15. Položaj datoteke application.properties u Spring Boot projektu

U kodu 27. prikazan je sadržaj datoteke i konkretna naredba za povezivanje aplikacije s MySQL relacijskom bazom podataka. U kodu se nalazi nekoliko naredbi. Prva naredba odnosi se na oznaku sufiksa web stranica koje će se koristiti u web aplikaciji. Spring Boot podržava razne tehnologije za generiranje web stranica kao što su JSP (engl. Java Server Pages), Thymeleaf, JavaScript i druge. Kao

rezultat generiranja stranica i prikaz putem web kontrolera je klasična HTML web stranica. U promjeru koda postavljeno je svojstvo da će server prikazivati web sadržaj koristiti standardne HTML stranice.

Ispod te naredbe nalazi se naredba za povezivanje aplikacije s bazom podataka. Naredbom ***spring.datacource.url*** definira se putanja povezivanja aplikacije s bazom. U toj naredbi stavljeni su razni parametri povezivanja kojima se određuje način povezivanja, razina sigurnosti, oznaka vremenske zone koja će biti podržana u bazi. To je veoma važno za pravilnu interpretaciju podataka koji su zapisani u samoj bazi podataka. Posebno se to odnosi na karaktere koji su karakteristični za jezik koji se koristi u komunikaciji s korisnicima.

```
spring.mvc.view.suffix=.html
spring.datasource.url=jdbc:mysql://localhost:3306/fina?autoReconnect=true&useSSL=false
        &zeroDateTimeBehavior=convertToNull
        &useUnicode=yes&characterEncoding=UTF-8
        &useJDBCCompliantTimezoneShift=true
        &useLegacyDatetimeCode=false&serverTimezone=Europe/Berlin
spring.datasource.username=korisnicko_ime
spring.datasource.password=lozinka

server.port=9040
```

Kod 27. Primjer naredbe za povezivanje aplikacije s MySQL bazom podataka

Ispod naredbe za povezivanje nalaze se naredbe koja određuju autorizaciju pristupa bazi podataka. Prva naredba odnosi se na korisničko ime koje ima pravo pristupa bazi. Drugi podatak je sama lozinka pristupa. Ovi podaci definiraju se na razini same baze podataka kao i koje ovlasti ima korisnik s tim pristupnim podacima. U datoteci je moguće koristiti naredbe za povezivanje na više od jedne baze podataka. U tom slučaju potrebno je postaviti dodatne informacije kao što je prikazano u kodu 28.
Nakon oznake lozinke nalazi se naredba koja određuje na kojem portu poslužitelja će se izvršavati aplikacija. To se može koristiti iz sigurnosnih razloga ili ukoliko imate više aplikacija postavljenih na jedan te isti poslužitelj. Naredbom određujete koja će se aplikacija izvoditi na kojem portu te na taj način izbjegavate koliziju aplikacija.
U kodu 28. nalaze se dvije relacijske baze. Prva je baza koja ima putanju spring.datasource.url, a druga je spring.secoundDatasource.url. U ovom primjeru to je standardna Oracle baza podataka. Ispod koda za povezivanje nalazi se kod Java klase u kojoj se definiraju bean objekti gdje se definiraju prefiksi za povezivanje. Anotacija ***@ConfigurationProperties*** određuje da se naredba odnosi na konfiguraciju svojstava gdje se navodi naziv prefiksa. Nakon toga u samoj metodi se kreira objekt ***DataSource*** koji označava objekt izvora podataka.

```
#prva db
spring.datasource.url = [url]
spring.datasource.username = [username]
spring.datasource.password = [password]
spring.datasource.driverClassName = oracle.jdbc.OracleDriver

#druga db ...
spring.secondDatasource.url = [url]
spring.secondDatasource.username = [username]
spring.secondDatasource.password = [password]
spring.secondDatasource.driverClassName = oracle.jdbc.OracleDriver



== Java klasa ==
@Bean
@Primary
@ConfigurationProperties(prefix="spring.datasource")
public DataSource primaryDataSource() {
   return DataSourceBuilder.create().build();
}

@Bean
@ConfigurationProperties(prefix="spring.secondDatasource")
public DataSource secondaryDataSource() {
   return DataSourceBuilder.create().build();
}
```

Kod 28. Primjer korištenja više baza podataka u istoj aplikaciji

Ukoliko postoji potreba za korištenjem nerelacijske baze podataka kao što je MongoDB, povezivanje je veoma slično, samo je sintaksa naredbi nešto izmijenjena. Kod 29. prikazuje primjer povezivanja aplikacije s MongoDB bazom. U primjeru koda prikazano je nekoliko načina povezivanja. Prvi primjer pokazuje jednostavno povezivanje gdje se u jednoj naredbi definira korisničko ime i lozinka kao i naziv same baze podataka. Drugi primjer prikazuje nešto detaljnije definiranje načina povezivanja u kojem se deklarira i oznaka porta na kojem će baza raditi. Vidljivo je da je port na kojem radi ova baza 27017, što je različito od porta gdje radi MySQL baza koji iznosi 3306. Treći primjer, koji također u jednoj naredbi definira sva najbitnija svojstva za pristup bazi, demo.

```
# 1. primjer
spring.data.mongodb.uri=mongodb+srv://<username>:<pwd>@<cluster>.mongodb.net/mygrocerylist
spring.data.mongodb.database=mygrocerylist
```

```
# 2. primjer
spring.data.mongodb.host=127.0.0.1
spring.data.mongodb.port=27017
spring.data.mongodb.authentication-database=admin
spring.data.mongodb.username=<username specified on MONGO_INITDB_ROOT_USERNAME>
spring.data.mongodb.password=<password specified on MONGO_INITDB_ROOT_PASSWORD>
spring.data.mongodb.database=<the db you want to use>

# 3. primjer
spring.data.mongodb.uri=mongodb://user:pass@localhost:27017/demo
```

Kod 29. Primjer naredbi za povezivanje aplikacije s MongoDB

Za razliku od statičnog načina povezivanja s bazom podataka, za dinamično povezivanje potrebno je kreirati Java klasu koja će se pomoću anotacije označiti da se radi o konfiguracijskoj klasi. Sama aplikacija pri tome kreira bean objekt klase ***DataSource*** koja označava izvor podataka. U kodu 30. prikazan je primjer dinamičnog kreiranja veze s relacijskom MySQL bazom podataka.

U kodu su dva načina povezivanja. Prvi primjer prikazuje samo kreiranje objekta klase DataSource. Objekt se kreira pomoću klase ***DriverManagerDataSource*** kojem se kasnije postavljaju parametri rada. Nakon kreiranja objekta određuje se upravljačka biblioteka za konkretnu bazu, naziv baze te pristupni podaci.

U drugom primjeru, koji je nešto kompleksniji, anotacijom se označava da se radi o konfiguracijskoj klasi. Nakon toga, navodi se naziv bean objekta koji se povezuje s izvorom podataka. Također, koristi se klasa ***DataSourceBuilder*** koja gradi budući bean objekt. U toj naredbi navode se osnovi parametri za pristup bazi te putanja do baze podataka. Određuje se objekt koji će koristiti kreiranu vezu s bazom. U aplikaciji će se definirati pristup bazi putem bean objekta pod nazivom „jdbcCustom“ jer je tako određen u samoj klasi.

Kasnije, u dijelu koda označenog s ***@Componet*** anotacijom, poziva se bean objekt te se koristi za pokretanje SQL naredbe. U kodu se s ***@Qualifier("jdbcCustom")*** anotacijom povezuje se bean objekt s budućim objektom i naredbom za pristup tablici Student. Formira se upit (*engl. Query*) za selekciju podataka os studentima.

```
# 1. primjer
DriverManagerDataSource dataSource = new DriverManagerDataSource();
dataSource.setDriverClassName("com.mysql.jdbc.Driver");
dataSource.setUrl("jdbc:mysql://host:port/database");
```

```
dataSource.setUsername("korisnickoIme");
dataSource.setPassword("lozinka");
# 2. primjer
@Configuration
public class DataSourceConfig {
   @Bean(name="dsCustom")
   public DataSource dataSource() {
      return DataSourceBuilder
            .create()
            .username("root")
            .password("123456")

.url("jdbc:mysql://localhost:3306/test?useSSL=false&useUnicode=true&characterEncoding=UTF-
8")
            .driverClassName("com.mysql.jdbc.Driver")
            .build();
   }

   @Bean(name = "jdbcCustom")
   @Autowired
   public JdbcTemplate jdbcTemplate(@Qualifier("dsCustom") DataSource dsCustom) {
      return new JdbcTemplate(dsCustom);
   }
# upotreba
@Component
public class StudentDao {
   @Autowired
   @Qualifier("jdbcCustom")
   private JdbcTemplate jdbcTemplateObject;

   public Student getStudent(Integer id) {
      String SQL = "select * from tbl_student where id = ?";
      Student = jdbcTemplateObject.queryForObject(SQL,
            new Object[]{id}, new StudentMapper());
      return student;
   }
```

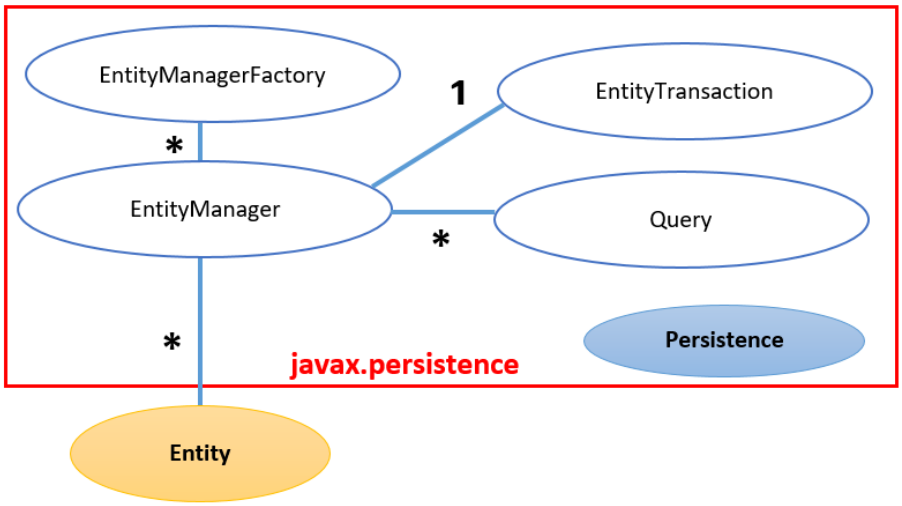
Kod 30. Primjer dinamičnog povezivanja aplikacije s bazom podataka

6.2. JPA / Hibernate anotacija

Naredbe JPA principa pristupa podataka temelji se na primjeni anotacija kao i cijeli koncept Spring Boot aplikacije. JPA je mehanizam za pohranu objekata kao relacijskih entiteta. JPA arhitektura sadrži sljedeće klase:

- ***Persistence*** je klasa koja sadrži statičke metode za dobivanje ***EntityManagerFactory*** instance,
- ***EntityManager*** je sučelje za kontrolu SQL operacija na bazi podataka,
- ***Entity*** predstavlja statični objekt u bazi, najčešće je to povezano s tablicom entiteta. Atributi u tablici podudaraju se s deklaracijom klase Entity. Zapis jedne entitete jednak je zapisu (*engl. record*) jednog reda u tablici baze,
- ***Persistence Unit*** predstavlja sve klase entiteta,
- ***EntityTransaction*** predstavlja odnos jedan na jedan (*engl. one-to-one*) s EntityManager klasom. Svaka SQL operacija provodi se uz pomoć ove klase,
- ***Query*** je instanca kojom se provodi SQL upit.

Na slici 16. prikazane su međusobne relacije između navedenih klasa.



Slika 16. Prikaz relacija između JPA klasa

Relacije među klasama su sljedeće:

- relacija EntityManager & EntiyTransaction: jedan na jedan (*engl. one-to-one*),
- relacija EntityManageFactory & EntiyManager: jedan-prema-više (*engl. one-to-many*),
- relacija EntityManager & Query: jedan na više,
- relacija EntityManager & Entity: jedan na više.

Za uključivanje JPA u Spring Boot aplikaciju potrebno je u pom.xml datoteci definirati zavisnost i uvesti potrebne biblioteke. U primjeru koda 31. prikazana je instrukcija koja uključuje JPA u projekt.

```
<dependency>
```

```
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-data-jpa</artifactId>
   <version>2.2.2.RELEASE</version>
</dependency>
```

Kod 31. Uključivanje JPA ovisnosti u aplikaciju koristeći Maven datoteku

Za početak korištenja JPA i mapiranog pristupa bazama podataka, potrebno je definirati ***entitetu***. Entiteta je klasa u kojoj se definira struktura pojedinačne tablice te njena povezanost i relacije s drugim tablicama. Koristi se ***@Entity*** anotacija koja se nalazi iznad naziva klase. Toj anotaciji moguće je pridodati i ***@Table*** anotaciju koja eksplicitno određuje naziv tablice koju će taj entitet koristiti.

U primjeru koda 32. prikazana je upotreba anotacija za entitet Student. Navedeno je da se tablica u bazi podataka također naziva Student. Ukoliko se ne koristi @Table anotacija, tada Spring okvir sam kreira tablicu automatski prilikom prevođenja aplikacije, prije samog pokretanja. Naziv tablice u tom slučaju jednak je nazivu klase entitete. Također, ukoliko tablica već postoji u bazi, a u međuvremenu je došlo do određenih promjena, Spring okvir sam vrši nadogradnju tablice prema karakteristikama klase Entity.

U samoj klasi nalazi se popis atributa (kolona) koji će u tablici biti predstavljeni kao njezine kolone. Naravno, svakom atributu određene su get/set metode pristupa.

U nastavku koda korištene su i neke druge anotacije. Anotacija ***@Column*** označava da je navedeni atribut klase kolona u tablici. Ukoliko imamo potrebe za jedinstvenim atributom po kojem će se podaci moći indeksirati, iznad naziva atributa stavlja se anotacija ***@Id***. Toj anotaciji dodana je i ***@ GeneratedValue*** anotacija koja označava da će se vrijednost automatski generirati, tipa ***AUTO NUMBER***.

Anotacija ***@NotEmpty*** osigurava da prilikom upisa podataka u tablicu, podatak ove kolone ne smije biti bez vrijednosti.

Anotacija ***@Temporal*** koristi se kada želimo da se tip Date, Time i TimeStamp u Javi pravilno prevede u SQL podatkovni tip. Na taj se način osigurava pravilan transfer podataka iz aplikacije u bazu i obrnuto. Zajedno s tom anotacijom može se i dodati ***@DateTimeFormat*** anotacija koja definira format zapisa podatka. Tako je na ovom primjeru korišten format zapisa yyyy-MM-dd hh:mm:ss što je karakterističan zapis za datuma i vremena za MySQL bazu podataka.

```
@Entity
@Table(name = "STUDENT", schema = "STORE")
public class Student {
        @Id
         @GeneratedValue(strategy = GenerationType.IDENTITY)
         private long id_student;
```

```
@Column(nullable = false)
@NotEmpty
private String ime_prezime;

@Column
@Temporal(TemporalType.TIMESTAMP)
@DateTimeFormat(pattern = "yyyy-MM-dd hh:mm:ss")
private Date datum_upisa;

@Lob
private byte[] slika;
```

Kod 32. Primjer korištenja anotacije @Entity

Java gotovo nema ograničenja vezanih za rad s memorijski većim podatkovnim tipovima kao što je string ili niz bajtova za pohranjivanje slika ili neki drugi oblik multimedijskog sadržaja. Za zapisivanje takvih specifičnih velikih podataka u bazama se koristi ***BLOB*** za binarni tip podatka i ***CLOB*** za zapisivanje karaktera. JPA za zapisivanje takvih tipova podataka koristi anotaciju ***@Lob*** koja omogućava mapiranje upravo takvih tipova podataka. U kodu 32. prikazan je primjer upotrebe te anotacije za potrebe zapisivanje slike studenta.

Kada se kaže relacijska baza podataka, misli se na relacije koje postoje između pojedinih tablica unutar baze. JPA također podržava sve vrste međusobnih relacija koje su definirane na razini SQL baze, a za to koristi sljedeće anotacije:

- ***@ManyToMany*** – relacija više–na-više,
- ***@ManyToOne*** – relacija više-na-jedan,
- ***@OneToMany*** – relacija jedan-na-više,
- ***@OneToOne*** – relacija jedan-na-jedan.

U kodu 33. prikazana je primjena navedenih anotacija. One se nalaze u klasi koja definira entitetu, podatkovni model objekta kojim se definiraju atributi pojedine tablice.

```
@Entity
public class Student {

  @ManyToOne(fetch = FetchType.LAZY)
  private Studij;

  @OneToMany(mappedBy = "ocjene", cascade = CascadeType.ALL)
```

```
    private Set<Ocjena> popisOcjena;
    ...
}
```

Kod 33. Primjer korištenja relacijskih anotacija

U ovom primjeru ***@ManyToOne*** anotacija korištena je da se poveže tablica Student s tablicom Studij. Relacija ***više-na-jednu*** označava sljedeće: pojam više se odnosi na Studente, a jednu se odnosi na tablicu Studij (studijski program). Više studenata pripada jedno studijskom programu. Nije moguće da jedan student pripada više studijskim programima. Dakle, student se može upisati samo u jedan studijski program.

Druga ***@OneToMany*** anotacija odnosi se na relaciju između studenta i njegovih ocjena. Pojam jedan odnosi se na studenta, pojam više se odnosi na ocjene. Dakle, jedan student ima više ocjena, a da pri tome jedna te ista ocjena ne može biti dodijeljena na više studenata. Kao rezultat ove relacije dobiva se set podataka. Set podataka je kontejner podataka gdje su podaci jedinstveni. Znači da se ta ista ocjena pojavljuje samo jednom, a one mogu biti grupirane prema zadanom uvjetu. U konkretnom slučaju to može biti predmet ili kolegij kojemu pripadaju ocjene.

Relacija ***@OneToOne*** u ovom slučaju odnosila bi se na dodatne podatke o studentu, njegovi osobni podaci koji nemaju ništa zajedničko s ostalim studentima. Recimo, IKS-ica koja sadrži niz podataka koji su dodijeljeni isključivo odabranom studentu.

7. Poglavlje: CRUD operacije s relacijskom bazom

U ovom poglavlju naučit ćete:

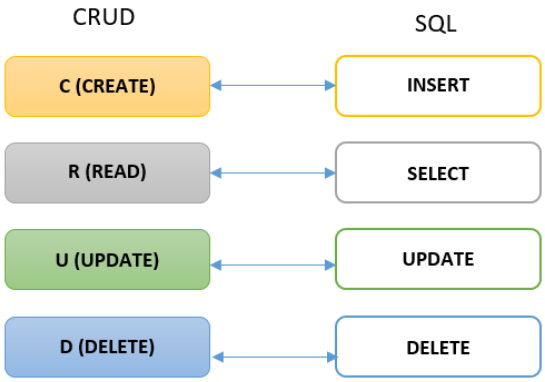
- ✓ Što su to CRUD operacije
- ✓ Definiranje repozitorija i sučelja za implementaciju CRUD operacije
- ✓ Struktura klasa za implementaciju CRUD operacija
- ✓ Primjer CRUD operacija za podatkovni model Student

7. CRUD operacije s relacijskim bazama i koncept SQL upita

7.1. Što su to CRUD operacije

Kada se spomene pojam ***CRUD*** , tada se taj pojam odnosi na četiri osnovne operacije za rad s podacima baze podatka kako je prikazano na slici 17.:

- **C**reate se odnosi na kreiranje tablica i kreiranje zapisa u tablici baze podataka,
- **R**ead se odnosi na čitanje podataka iz baze podataka,
- **U**pdate se odnosi na izmjenu podatka u bazi podataka,
- **D**elete se odnosi na brisanje zapisa iz baze podataka.



Slika 17. Povezanost CRUD operacija i SQL naredbi

Svaka od navedenih operacija ima razne varijacije, ovisno o tome što se želi postići s nekom od njih. Na koliko podataka i tablica se želi utjecati. U biti, te se operacije odnose na standardne SQL naredbe.

U kodu 34. prikazane su CRUD operacije implementirane u SQL naredbama.

```
CREATE
DROP TABLE IF EXISTS Student;
CREATE TABLE dbo.Student
(id   INT, ime VARCHAR(100) );

INSERT INTO < TableName > (column1,column2,….)
VALUES (value1,value2,….)

READ
SELECT * FROM <TableName>

UPDATE
UPDATE <TableName>
SET Column1=Value1, Column2=Value2,…
WHERE <Expression>

DELETE
DELETE FROM <TableName>
WHERE <Expression>
```

Kod 34. Primjer SQL naredbi koje predstavljaju CRUD naredbe

7.2. Kreiranje podatkovnog modela

Za pristup podacima u bazi podataka i za prilagodbu objekata koji će predstavljati zapise pojedine tablice, potrebno je za svaku tablicu u bazi podataka formirati entitetu ili podatkovni model. Podatkovni model sadrži popis atributa i njihovih podatkovnih tipova. Svaki atribut u podatkovnom modelu ekvivalentan je koloni u tablici koja se nalazi u bazi podataka. Tip podatka označava vrstu podatka koji će biti upisan u određenoj koloni tablice. Podatkovni model organiziran je u obliku klase kojoj je dodana anotacija ***@Entity***. U projektu u primjeru koda 35. prikazana je klasa podatkovnog modela za studenta. Prvi atribut u ovoj klasi definiran je kao indeks zapisa koji je ***AUTO NUMBER*** karaktera, što znači da se ***id_student*** generira automatski prilikom svakog upisa. Indeksirani atributi uvijek se stavljaju da budu ***Long*** podatkovnog tipa radi mogućnosti da se zapiše veći broj zapisa u tablicu. U ovom primjeru nije korištena anotacija kojom će se definirati naziv tablice. Međutim, Spring Boot sam generira naziv tablice koji je uvijek jednak nazivu klase. U ovom konkretnom slučaju, generirat će se tablica pod nazivom Student, ukoliko ne postoji u bazi podataka.

Nakon toga u klasi su definirana tri atributa za zapis podatka o imenu i prezimenu studenta te godinu studija. Nakon toga, u klasi dolaze get/set metode za pristup podacima objekta.

```
@Entity
public class Student {

	@Id
	@GeneratedValue(strategy=GenerationType.IDENTITY)
	private long id_student;

	@Column
	private String prezime;

	@Column
	private String ime;

	@Column
	private int godinaStudija;

	public long getId_student() {
		return id_student;
	}

	public void setId_student(long id_student) {
		this.id_student = id_student;
	}

  ...
}
```

Kod 35. Primjer podatkovnog modela

7.3. Kreiranje klasu repozitorija

Repozitorij (*engl. JPA Repository*) predstavlja klasu sučelja (*engl. Interface*) u kojoj se definiraju metode SQL pristupa podacima iz baze podataka. Sama klasa je izvedena iz ***JpaRepository*** klase u kojoj se definira podatkovni model koji će podržavati ovo sučelje. Uvijek se sučelje izrađuje za točno određeni podatkovni model, drugim riječima za točno određenu tablicu u bazi podataka. Na taj način dobiva se sistematizacija i bolja organiziranost klasa što je bitno za kasnije održavanje same aplikacije.

Kod 36. prikazuje repozitorij za podatkovni model Student. Potrebno je objasniti nekoliko anotacija koje su pri tom korištene. Oznaka ***interface*** označava da je ova klasa sučelje, a ***extends JpaRepository*** da se radi o repozitoriju baze podataka. Kao atributi ovog sučelja definiran je podatkovni model za koji je repozitorij namijenjen. Također, označava se i podatkovni tip ***Long*** po kojem su indeksirani podaci u toj tablici.

Sam repozitorij može ostati bez upisane naredbe. U tom slučaju razvojnom programeru su ponuđene standardne metode za pristup podacima. Ukoliko imamo potrebe za nekim svojim SQL metodama, potrebno ih je definirati u repozitoriju. U ovom primjeru navedene su dvije nove metode: ***findStudentsByName*** i ***findStudentsByGodinaStudija***. Prva metoda namijenjena je za selekciju podataka u tablici Student za one podatke onih studenata koji imaju traženo ime. Naime, u pozivu metode prosljeđuje se argument ime. On se uzima kod formiranja SQL naredbe. U izrazu ***value*** navodi se SQL naredba, a traženu vrijednost imena studenta zamjenjuje se za vrijednost ***?1*** . Argument ***nativeQuery*** označava da se radi u SQL upitu koji ide prema bazi podataka.

```
import hr.spring.fina.model.Student;

public interface StudentRepository extends JpaRepository<Student, Long> {

 @Query(
 value = "SELECT *  FROM student s WHERE s. ime = ?1",nativeQuery = true)
 Collection<Student> findStudentsByName(String ime);

 @Query(value = "SELECT * FROM student s WHERE s. godinaStudija = : godina“,nativeQuery = true)
 Collection<Student> findStudentsByGodinaStudija( @Param("godina") int godina);

...
}
```

Kod 36. Repozitorij za entitetu Student

U drugoj metodi za razliku od prve metode, korištena je ***@Param*** anotacija. Ova anotacija označava ulazni parametar ili atribut koji se treba definirati prilikom poziva ove metode. Navodi se naziv parametra te koji podatkovni tip se očekuje od podatka koji se šalje u metodu. Također, sama sintaksa naredbe je nešto različita u odnosu na prvu metodu. Umjesto ***?1*** koristi se ***dvotočka*** i naziv parametra. Oba načina su jednako vrijedna i jednako brzo se izvršavaju na razini aplikacije. U samoj klasi sučelja može se navesti neograničen broj metoda, najčešće složenih, koje nisu podržane od strane standardne

JpaRepository klase. Ovom klasom realizirana je izravna veza između entitete podatkovnog modela, tablice baze podataka te SQL upita prema podacima.

7.4. Kreiranje servis sučelja za pristup podacima

Podatkovno sučelje je sljedeći korak koji je potrebno realizirati za pristup podacima baze. Za razliku od prijašnjeg sučelja repozitorija, ovo sučelje namijenjeno je korištenju u REST kontrolerima u samoj aplikaciji. Dakle, sučelje repozitorija nije izravno vidljivo korisnicima u kontrolerima, nego pomoću servis sučelja pristupaju i pokreću se SQL naredbe. Na taj je način osigurana kontrola povezanosti i kompatibilnosti svih elemenata potrebnih za pristup podacima baze podataka. U praksi se koristi veći broj servisa i sučelja, najčešće svaki servis za jedan podatkovni model. U svakoj klasi servisa obavezno se implementiraju sve CRUD operacije te se dodaju neke specifične metode koje trebaju biti realizirane ovisno o zahtjevu same aplikacije.

Zbog jednostavnijeg testiranja servisa, servis se dijeli u dvije klase. Prva klasa je klasa sučelja u kojem se nalazi popis javnih metoda koje će se koristiti za pristup podacima. Te metode su jedine metode koje korisnik putem kontrolera može koristiti. Broj metoda nije ograničen te se proširuje prema potrebi rada i funkcionalnosti same aplikacije. Drugi dio je klasa u kojoj su navedene metode implementirane, tu klasu nazivamo servis. U toj su klasi detaljno razrađeni algoritmi rada svake metode te su navedene SQL naredbe repozitorija koje se pri tome koriste. U toj klasi potrebno je definirati sve metode iz prvog dijela sučelja. U slučaju da se neka metoda ne definira, Spring Boot okvir javlja pogrešku i nemogućnost pokretanja same aplikacije.

U kodu 37. prikazan je primjer servisa sučelja za pristup podacima studenata.

```
package hr.spring.fina.service;

import hr.spring.fina.model.Student;

public interface StudentService {

        Student createStudent(Student student);

        Student updateStudent(Student student);

        Iterable<Student> getStudentiIme(String ime);

        Iterable<Student> getStudentiGodina(int godinaStudija);

        void deleteStudent(long id);

}
```

Kod 37. Primjer servis sučelja za tablicu Student

U primjeru je navedeno pet (5) metoda za pristup podacima o studentima:

- ***createStudent*** je metoda za kreiranje novog zapisa o studentu. Kao argument koristi sam objekt studenta s podacima,

- ***updateStudent*** je metoda mijenjanje podataka o studentu koji se prosljeđuje u metodu,
- ***getStudentiIme*** je metoda koja kao rezultat vraća popis studenata čije ime je jednako traženom argumentu ime,
- ***getStudentiGodina*** je metoda koja kao rezultat vraća popis studenata tražene godine studija,
- ***deleteStudent*** je metoda za brisanje zapisa o studentu čiji id se podudara id_studenta u samoj tablici podataka u bazi.

```
package hr.spring.fina.service;

@Service
@Transactional
public class StudentServiceImpl implements StudentService {

 @Autowired
 private StudentRepository;

 @Override
 public Student createStudent(Student student) {
        return studentRepository.save(student);
 }

 @Override
 public Student updateStudent(Student student) throws ResourceNotFoundException {
        Optional<Student> productDb = this.studentRepository.findById(student.getId_student());

        if (productDb.isPresent()) {
                Student studentUpdate = productDb.get();
                studentUpdate.setId_student(student.getId_student());
                studentUpdate.setPrezime(student.getPrezime());
                studentUpdate.setIme(student.getIme());
                studentRepository.save(studentUpdate);
                return studentUpdate;
        } else {
          throw new ResourceNotFoundException("Zapis nije pronađen");
        }

 }

 @Override
 public Iterable<Student> getStudentiIme(String ime) {
   return this.studentRepository.findStudentsByName(ime);
 }
 ...
}
```

Kod 38. Implementacija servis sučelja Student

Sve te metode su samo navedene i označene da će biti kasnije detaljno definirane. Definiranje metoda odvija se u drugom dijelu sučelja kao je prikazano u kodu 38. Na početku koda stavljena je anotacija da se radi o klasi servisa i da je prilagođena Spring Boot. U samom zaglavlju klase navodi se da se radi o implementaciji servisa. To je egzaktno navedeno s naredbom ***implements StudentService***, što označava da se radi o implementaciji klase StudentService.

U daljnjem kodu upotrjebljena je ***@Autowired*** anotacija čija namjena je objašnjena u prijašnjem poglavlju. Servis se povezuje s pripadajućim repozitorijem čije SQL metode se koriste u samoj implementaciji.

Anotacija ***@Override*** označava da se u tom dijelu vrši implementacija same metode pod istim nazivom kako je to navedeno u prvom dijelu servisa. U toj klasi trebaju biti implementirane sve metode iz prvog dijela servisa.

Nakon metode za kreiranje zapisa Student u bazi, navedena je implementacija metode za uređivanje podataka ***updateStudent***. Prvo se čita podatak o traženom studentu pozivom SQL naredbe repozitorija ***findById***. Dobiveni podatak, ukoliko je pronađen, mijenja se na osnovi objekta Student koji je proslijeđen kao argument metode. Izmjene se upisuju u bazu naredbom ***save*** u repozitoriju. Slijedi implementacija metode ***getStudentiIme*** koja pronalazi studente u bazi koji imaju isto ime kao što je vrijednost argumenta ime u pozivu metode. Poziva se SQL naredba repozitorija findStudentByName koja pokreće SQL naredbu prema repozitoriju i bazi podataka. Metoda kao rezultat daje popis studenata koji se kasnije koristi u nekom od kontrolera.

Prema dobroj praksi razvoja aplikacije, svaka tablica u bazi ima poseban servis za pristup i uređivanje podataka. Time se osigurava preglednost programskog koda i programske logike koja je implementirana za potrebe aplikacije.

Ovako implementiran servis spreman je za uključivanje u REST kontroler koji će kontrolirati pristup podacima vanjskim korisnicima ili uređajima.

7.5. REST kontroler za pristupa podacima

REST kontroler, kako je u prijašnjem poglavlju navedeno, omogućuje udaljenim uređajima pristup podacima baze podataka. Za pristup podacima koriste se sve prije navedene klase, servisi i sučelja koja su objašnjena u prijašnjem poglavlju. U složenijim aplikacijama upotrebljava se više REST kontrolera, ovisno o njihovoj namjeni i podacima kojima se pristupa. Dobar pristup je da svaki REST kontroler pokriva jednu funkcionalnost aplikacije ili jedan modul koji je zadužen za određenu skupinu podataka. U kodu 39. prikazan je primjer REST kontrolera za pristup podacima o studentima.

Struktura REST kontrolera podijeljena je na dva dijela. Prvi dio kontrolera namijenjen je za komunikaciju i zaprimanje zahtjeva od strane neke web stranice. Anotacijom ***@RestController***, na početku klase, daje se informacija o tome da se radi upravo o ovoj vrsti kontrolera. Anotacijom ***@RequestMapping*** određuje se opća putanja pristupa ovom kontroleru. Pristup svakoj metodi u kontroleru je dodatno mapiran. Svaka putanja u mapi aplikacije treba biti jedinstvena, a može biti nadopunjena s varijablama ili objektima koji se šalju kod poziva. Pristup pojedinoj metodi kontrolera naziva se zahtjev (*engl. Request*). Zahtjev se šalje oblak poslužitelju, a prema putanji poslužitelj određuje o kojoj metodi u kojem kontroleru se zahtjev odnosi. Zaglavlje svake metode određuje vrstu HHTP zahtjeva te da li su pri tome potrebni nekakvi dodatni podaci. Ukoliko se radi o klasičnom HTTP

zahtjevu treba razlikovati Get, Post, Put i Delete metode. U ovom konkretnom slučaju, putanja je označena sa student. Ukoliko je domena oblak aplikacije recimo http://www.aplikacija.hr, tada bi mapirana putanja za pristup ovom REST kontroleru bila http://www.aplikacija.hr/student/. Pristup metodi ***getAllStudents*** putanja bi bila http://www.aplikacija.hr/student/ sviStudenti/.

```
@RestController
@RequestMapping("student")
public class StudentController {

 @Autowired
  private StudentService studentService;

 @GetMapping("/sviStudenti/")
  public ResponseEntity < Iterable < Student >> getAllStudents() {
     return ResponseEntity.ok().body(this.studentService.getAllStudents());
  }

 @GetMapping("/{id}")
  public ResponseEntity < Student > getStudentById(@PathVariable long id) {
     return ResponseEntity.ok().body(this.studentService.getStudent(id));
  }

  @PostMapping("/")
   public ResponseEntity < Student > createStudent(@RequestBody Student student) {
     return ResponseEntity.ok().body(this.studentService.createStudent(student));
   }

   @PutMapping("/")
   public ResponseEntity < Student > updateStudent(@RequestBody Student student) {
     return ResponseEntity.ok().body(this.studentService.updateStudent(student));
   }

   @DeleteMapping("/{id}")
   public HttpStatus deleteStudent(@PathVariable long id) {
          this.studentService.deleteStudent(id);
     return HttpStatus.OK;
   }

}
```

Kod 39. Primjer REST kontrolera

Na početku kontrolera nalazi se instrukcija koja povezuje ovaj kontroler sa studentski servis. Servis je povezan pomoću ***@Autowired*** anotacije. Definirani objekt ***studentService*** kasnije će se koristiti za pristup bazi podataka. Slijede za pristup podacima. Kod svake metode istaknuta je anotacija koja određuje način HTTP pristupa. Tako je za metodu getAllStudents, definiran GET pristup upotrebom ***@GetMapping*** anotacije. Kao rezultat ovog zahtjeva, metoda vraća popis svih studenata u obliku klase ***ResponseEntity < Iterable < Student >>***. ResponseEntity je standardna klasa i objekt koji se vraća kao rezultat rada kontrolera. U samoj metodi nalazi se samo jedna naredba. Naredba poziva izvršenje SQL upita koristeći servis i njegovu metodu ***getAllStudents***. Ukoliko je sve izvršeno bez pogreške, metoda vraća popis studenata i kod vrijednosti ***200*** izvršene naredbe. Broj 200 označava da je metoda ispravno izvršena.

Kod zahtjeva metode ***getStudentById*** šalje se i vrijednost id studenta. Tako da putanja zahtjeva izgleda http://www.aplikacija.hr/student/100. Rezultat ove metode je objekt Student s podacima traženog studenta koje su zapisane u JSON (*engl. JavaScript Object Notation*) formatu. Ovaj format postao je standard u komunikaciji između klijenta/web stranice i kontrolera.

Za pristup podacima putem mobilnog uređaja koristi se REST kontroler kojeg nazivamo ***API*** (*engl. Application Programming Interface*). Zbog različitog načina komunikacije, zaglavlje metoda su različita u odnosu na prijašnje metode. U zaglavlju se navodi vrsta HTTP pristupa te tip zaglavlja koje se koristi u komunikaciji. U API mogu se koristiti sve HTTP metode (GET,POST, PUT, DELETE). U kodu 31. prikazan je primjer metoda za pristup podacima studentima.

Standardna komunikacija Spring Boot oblak aplikacije s udaljenim uređajima je pretvaranje objekata u neki standard zapisa. To je najčešće JSON format pa se u samom zaglavlju navodi način prijenosa podataka. U konkretnom primjeru navodi se da zaglavlje podržava ***application/json*** oblik zapisivanja podataka.

GET API metoda

U primjeru koda 31. prikazana je metoda pod nazivom ***getSviStudentiArray*** koja kao rezultat vraća podatke o svim studentima. Rezultat metode je ***ArrayList*** objekt u kojem su zapisani objekti klase Student ali zapisani u JSON formatu. Putanja za zahtjev ovoj metodi je http://www.aplikacija.hr/student/api/getStudenti/. U samoj metodi koristi se servis za pristup podacima. Općenito, ***GET*** metode koriste se za pristup podacima te pomoću repozitorija izvršavaju se ***Select*** SQL upiti.

POST API metoda

Ova metoda koristi se za slanje podataka. Najčešće se putem udaljenog uređaja šalje objekt što je primjer u ovom kodu. Poziva se metoda ***upisStudenta*** koja u svojem zaglavlju zahtjeva očekuje deklarirani objekt klase Student ***@RequestBody Student dataStudent***. Za razliku od prijašnje metode, ova metoda vraća objekt ***ResponseData*** u obliku klase ***ResponseEntity*** koja je standardna klasa u HTTP komunikaciji. ResponseData je klasa koja je kreirana za potrebe oblak aplikacije. Klasa ima dva atributa: ***message*** za upis teksta poruke udaljenom uređaju te ***errorKod*** za oznaku uspješnosti izvršavanja metode. U metodi poziva se servis i metoda ***createStudent*** koja upisuje podatke o studentu u tablicu Student. U slučaju da je upis podataka uspješno izvršen, metoda vraća kod 200 što je standardna oznaka uspješno izvršenog HTTP zahtjeva. Ukoliko je došlo do pogreške u upisu, metoda generira kod greške 403 koja označava pogrešku u operaciji upisa. Uz oznaku koda dodaje se i tekst poruke koju udaljeni uređaj može prikazati korisniku.

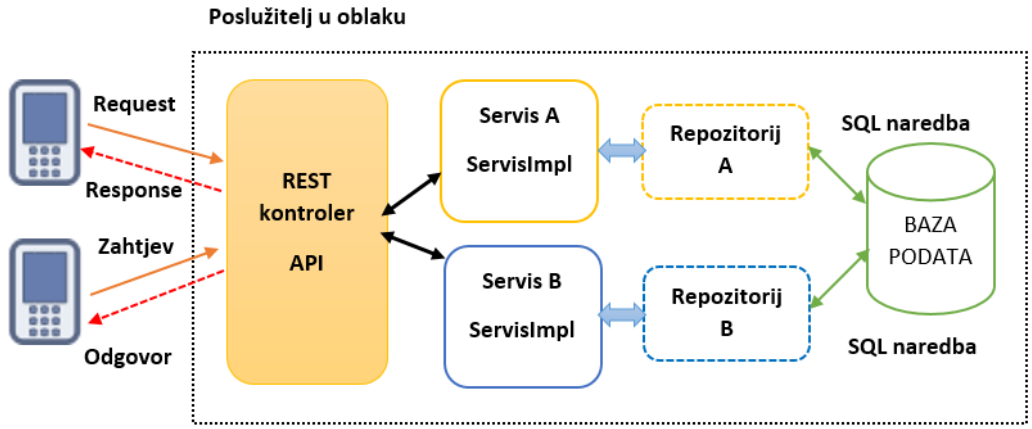
PUT API metoda

Metoda je predviđena za obnavljanje i mijenjanje podataka u bazi. Metoda ***obnoviStudenta*** također u svom zaglavlju zahtjeva očekuje JSON objekt klase Student. Metoda obnavlja podatke o studentu i kao i prijašnja metoda vraća povratnu informaciju klijentu na udaljenom računalu.

DELETE API metoda

Metoda za brisanje zapisa u bazi podataka. Zahtjevom se šalje objekt klase Student, čiji podaci se brišu. Metoda ***brisiStudenta*** pomoću servisa i implementirane metode ***deleteStudent*** briše podatak iz baze. Odabir ispravnog zapisa u tablici Student određuje se na temelju ***id_student*** oznake studenta. Kako je u podatkovnom modelu definirano, da je to indeks podataka u tablici te da je jedinstveni podatak, upravo id_student određuje zapis u tablici. U slučaju da podatak o tom studentu ne postoji, iz nekog razloga, u aplikaciji se neće ništa štetno desiti jer je u metodi za brisanje postavljen mehanizam provjere ovakve situacije. U kodu su prikazane osnovne CRUD metode koje se pozivaju putem API sučelja.

Na slici 18. prikazana je međusobna povezanost svih komponenti u Spring Boot aplikaciji za provođenje CRUD operacija. Zahtjev dolazi od strane udaljenog uređaja i prosljeđuje se putem HTTP upita u REST kontroler. U kontroleru se nalazi servis za pojedinu vrstu podataka. Implementirani servis povezan je s pripadajućim repozitorijem. U repozitoriju se nalaze implementirani SQL upiti koji se šalju prema bazi podataka. Komunikacija unutar svih komponenti je dvosmjerna. Kao rezultat cijelog procesa je odgovor koji dobiva korisnik na svoj uređaj. Broj korisnika nije ograničen. Jedino ograničenje je u strukturi oblak poslužitelja i kapaciteta poslova koje može obraditi u jedinici vremena. Ti parametri ovise o fizičkim svojstvima poslužitelja na koji je postavljena aplikacija.



Slika 18. Povezanost komponenti za izvršenje CRUD operacija putem REST kontrolera

```
@RestController
@RequestMapping("student")
public class StudentController {
```

```
 @Autowired
  private StudentService studentService;

   // API
  @RequestMapping(value = "/api/getStudenti/", method = RequestMethod.GET, headers =
"Accept=application/json")
 public @ResponseBody ArrayList<Student> getSviStudentiArray() {
   return (ArrayList<Student>)this.studentService.getAllStudents();
  }

 @RequestMapping(value = "/api/upisStudenta/", method = RequestMethod.POST)
public ResponseEntity<ResponseData> upisStudenta(@RequestBody Student dataStudent) throws
IOException {
          ResponseData RD = new ResponseData();
          try {
           this.studentService.createStudent(dataStudent);
           RD.setErrorCode(200);
           RD.setMessage("Uspješno upisan podatak");
           return ResponseEntity.status(200).body(RD);
          } catch (Exception err) {
           RD.setErrorCode(403);
           RD.setMessage("Pogreška kod upisa podataka");
           return ResponseEntity.status(403).body(RD);
          }
 }

 @RequestMapping(value = "/api/obnoviStudent", method = RequestMethod.PUT)
public ResponseEntity<ResponseData> obnoviStudenta(@RequestBody Student dataStudent) throws
IOException {
          ResponseData RD = new ResponseData();
          try {

           this.studentService.updateStudent(dataStudent);
           RD.setErrorCode(200);
           RD.setMessage("Uspješno obnovljen podatak");
           return ResponseEntity.status(200).body(RD);
          } catch (Exception err) {
           RD.setErrorCode(403);
           RD.setMessage("Pogreška prilikom obnavljanja podataka");
           return ResponseEntity.status(403).body(RD);
          }
  }

 @RequestMapping(value = "/api/obrisiStudenta", method = RequestMethod.DELETE)
 public ResponseEntity<ResponseData> brisiStudenta(@RequestBody Student dataStudent) throws
IOException {
                    ResponseData RD = new ResponseData();
                    try {

                              this.studentService.deleteStudent(dataStudent.getId_student());

                              RD.setErrorCode(200);
                              RD.setMessage("Podaci o studentu uspješno su obrisani");
                              return ResponseEntity.status(200).body(RD);
                    } catch (Exception err) {
                              RD.setErrorCode(403);
                              RD.setMessage("Pogreška prilikom brisanja podataka");
                              return ResponseEntity.status(403).body(RD);
                    }
          }
}
```

Kod 40. Primjer CRUD metoda u API sučelju

8. Poglavlje: Thymeleaf i MVC aplikacija

U ovom poglavlju naučit ćete:

- ✓ Koja je struktura MVC aplikacije
- ✓ Komponente MVC aplikacija i način njihove povezanosti
- ✓ Primjer CRUD operacija u MVC aplikaciji

8. Thymeleaf i MVC aplikacija

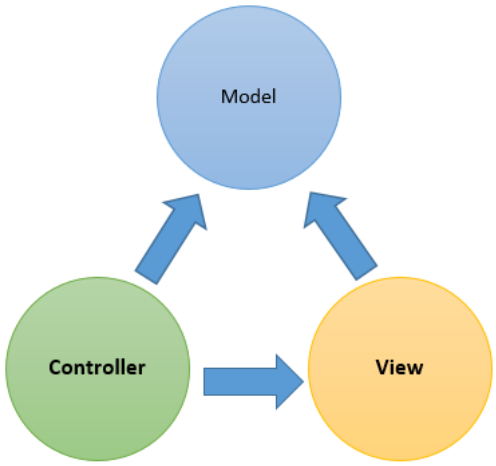
8.1. Pojam MVC aplikacije

MVC struktura aplikacija danas je postala svakodnevica. Ovaj pristup strukturiranja aplikacije moguće je vidjeti od standardnih aplikacija, mobilnih aplikacija kao i kod suvremenih web aplikacija. Sama MVC aplikacija sastoji se od tri komponente: M (Model), V(View) i C (Controller).

Model predstavlja razne podatkovne modele koji se koriste u aplikaciji za pristup podacima i rad s njima. Podaci su sastavni dio svake aplikacije, a u Spring Boot aplikaciji oni su definirani putem klase kao objekti, entitete ili bean objekti.

View predstavlja pogled ili prezentaciju podataka i raznih informacija koje pruža aplikacija. Kod Spring Boot aplikacije, view se odnosi na web stranice koje se prikazuju u web pregledniku. Za formiranje web stranica koriste se razne biblioteke kao što su Javascript, CSS, Thymeleaf, XML, CSV i druge. Sve to moguće je kombinirati u cjelinu s ciljem dobivanja što kvalitetnijeg prikaza. View komponente mogu zaprimati razne podatke te pomoću neke od tehnologija, prikazati njihove vrijednosti.

Controller predstavlja logiku aplikacije, određuje metode koje su zadužene za prikaz pojedinih web stranica. Određuje na koji način će uključivati podatke prilikom generiranja View komponenti. Na slici 19. prikazana je povezanost MVC komponenti. Istaknuti su smjerovi izmjene podataka i komunikacije. Iz strukture povezanosti, vidljivo je da je kontroler zadužen za formiranje View komponente, a Modelom podataka koriste Controller i View komponente aplikacije.



Slika 19. Povezanost komponenti u MVC aplikacijama

Spring Boot aplikacija može biti različitih karaktera i struktura. Aplikacija može biti građena od mikroservisa, koji nemaju potrebe za prikaz podataka. U tom slučaju aplikacija se odvija u pozadini cijele aplikacije, a web sučelje aplikacije kreirano je u nekom drugom programskom jeziku. Međutim, moguće je da cijela Spring Boot aplikacija bude realizirana kao web aplikacija sa svim njenim elementima. U tom slučaju radi se o standardnom MVC konceptu aplikacije. Ukoliko se radi o

standardnoj Spring Boot MVC aplikaciji, to je moguće naglasiti ***@EnableWebMvc*** anotacijom. U kodu 41. prikazana je primjena te anotacije. Anotacija je korištena u konstrukcijskoj klasi aplikacije.

```
@EnableWebMvc
@Configuration
public class WebConfig implements WebMvcConfigurer {

  @Override
  public void addViewControllers(ViewControllerRegistry registry) {
    registry.addViewController("/").setViewName("index");
  }

  @Bean
  public ViewResolver viewResolver() {
    InternalResourceViewResolver bean = new InternalResourceViewResolver();

    bean.setViewClass(JstlView.class);
    bean.setPrefix("/WEB-INF/view/");
    bean.setSuffix(".html");

    return bean;
  }
}
```

Kod 41. Primjer korištenja MVC anotacije

Sama klasa izvedena je iz bazne WebMvcConfigurer klase čime su naslijeđena određena svojstva. Na početku klase određuje se view kontroler i HTML predložak koji će se koristiti za početnu stranicu. Druga metoda u konstrukciji kreira bean objekt koji određuje karakteristike web stranica koje će biti prikazane. Određuje se pozicija web predložaka te njezin prefiks i sufiks ***".html"*** .

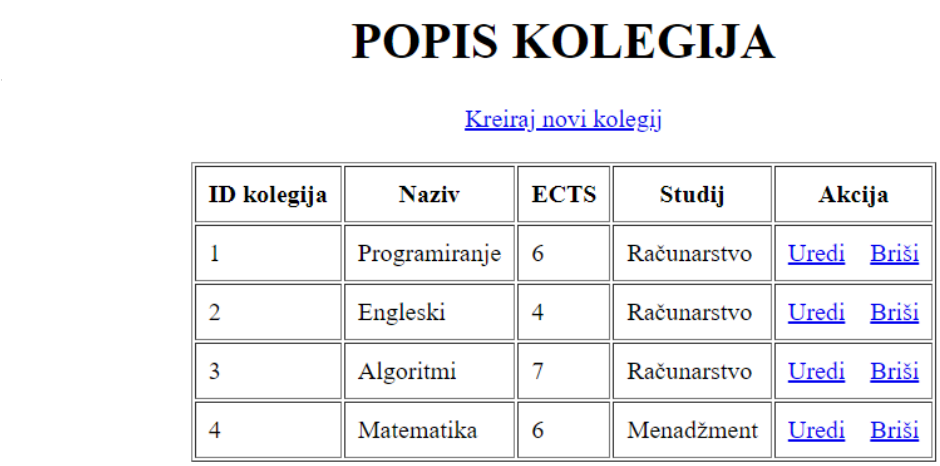
8.2. Spring Boot MVC aplikacija

Sam karakter MVC aplikacije određuje obavezne komponente koje trebaju biti implementirane u Spring Boot aplikaciji. Ukoliko se radi o aplikaciji koja samo prikazuje statične web stranice, bez potrebe za podacima iz baze podataka, tada je dovoljno web kontroler i predlošci koji će definirati izgled web stranica.

Međutim, ukoliko se radi o naprednoj aplikaciji koja koristi razne podatke iz baze podataka, aplikacija koja ima implementiranu sigurnosnu komponentu, tada se koriste sve one komponente koje su spomenute u prijašnjim poglavljima. Komponente kao što su servisi, Thymeleaf predlošci, REST kontroleri i još mnogo toga. U ovom poglavlju bit će prikaz primjer MVC aplikacije koja će uređivati podatke o kolegijima koje polažu studenti. Struktura aplikacije je sljedeća:

- ***View komponenta*** – sastoji se od tri predloška: ***kolegiji.html*** za prikaz podatak o kolegijima; ***uredi_kolegij.html*** za uređivanje podataka odabranog kolegija; ***novi_kolegij.html*** za upis podataka o novom kolegiju.

- ***Model komponenta*** – sastoji se od klase Kolegij.java; KolegijRepository.java; KolegijService.java; KolegijServiceImpl.java;
- ***Controller*** – sastoji se od klase za kontrolu prikaza web stranica WebController.java



Slika 20. Izgled web stranice aplikacije

Slika 20. prikazuje izgled početne stranice. Na stranici se na tablica s podacima o kolegijima. U koloni Akcija u tablici, nalaze se linkovi koji pokreću zahtjev za uređivanjem i brisanjem podataka odabranog kolegija. Kreiranje novog kolegija pokreće se pritiskom na link koji se nalazi iznad tablice. Pritiskom na bilo koji link, šalje se zahtjev prema web kontroleru i poziva se pojedina metoda.

8.3. Implementacija aplikacija

O karakteru aplikacije ovisi i način njene implementacije. Karakter može određivati razinu sigurnosti aplikacije i dodatne sigurnosne mjere koje će se provoditi prilikom pristupa kontroleru.

Podatkovni model Kolegij.java

Implementacija će započeti s klasom ***Kolegij.java*** koja je entiteta za tablicu baze podataka.

```
package hr.spring.fina.model;
import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;
import javax.persistence.Table;

@Entity
@Table(name = "Kolegij")
public class Kolegij {

	@Id
	@GeneratedValue(strategy = GenerationType.IDENTITY)
	private long id_koelgij;

	@Column
```

```
	private String naziv;

	@Column
	private int ECTS;

	@Column
	private String studij;

	public long getId_koelgij() {
		return id_koelgij;
	}
	...
}
```

Kod 42. Klasa podatkovnog modela Kolegij

U klasi je naveden naziv tablice koji će se koristiti u bazi podataka, istaknuti su atributi koji će biti kolone te tablice. Klasa ima sljedeće atribute: ***id_klasa*** kao indeks zapisa u tablici, ***naziv*** kolegija, ***ECTS*** za označavanje broja bodova koje ostvaruje student polaganjem pojedinog kolegija i ***studij*** naziv studija kojem pripada kolegij. Za svaki atribut u ovom entitetu kreirane su ***get/set*** metode za pristup podacima.

Repozitorij podatkovnog modela

Repozitorij za pristup podacima i izvršavanje SQL naredbi određena je klasa KolegijRepository.java. Kod 43. prikazuje implementaciju koda.

```
package hr.spring.fina.repository;

import org.springframework.data.jpa.repository.JpaRepository;
import hr.spring.fina.model.Kolegij;


public interface KolegijRepository extends JpaRepository<Kolegij, Long> {


}
```

Kod 43. Repozitorij podatkovne klase Kolegij

U zaglavlju repozitorija navedeno je da je klasa izvedena iz JpaRepository klase te da repozitorij pripada podatkovnom modelu Kolegij. U samom repozitoriju nema nikakvih dodatnih metoda SQL upita već se koriste metode koje su podržane od strane JPA biblioteke.

Servis podatkovnog modela

Klasa servisa određuje metode za pristup podacima te predstavlja sučelje pristupu repozitoriju. Servis predstavlja poslovnu razinu aplikacije. Kod 44. prikazuje metode koje su implementirane za potrebe ovog primjera. Implementirane su sve CRUD metode koje će se pozivati putem kontrolera. Navedene

su osnovne CRUD metode, a njihova implementacija je prikazana u nekoliko kodova. U kodu 45. prikazana je deklaracija same klase ***KolegijServiceImpl***. Na početku klase korištena je anotacija koja označava klasu kao servis koji je implementacija klase ***KolegijService***. Nakon zaglavlja klase nalazi se anotacija i uključenje repozitorija čiji kod je prikazan u kodu 43.

Uključivanjem repozitorija dolazi sama razrada CRUD metoda. Prva od metoda je metoda za kreiranje novog zapisa u tablici. Metoda poziva ***save*** metodu repozitorija koja je sastavni dio JPA i Hibernate biblioteke. Rezultat ove metode je novi objekt u kojem je JPA definirao indeks id_kriterija koji je označen kao AUTO NUMBER te se generira od strane JPA okruženja. Ovakav pristup olakšava pravilno dodavanje novih zapisa u tablicu i lakše indeksiranje istih zapisa.

```
package hr.spring.fina.service;

import hr.spring.fina.model.Kolegij;

public interface KolegijService {

	Kolegij createKolegij(Kolegij Kolegij);

	Kolegij updateKolegij(Kolegij Kolegij);

	Iterable<Kolegij> getAllKolegij();

	Kolegij getKolegij(long id_Kolegij);

	void deleteKolegij(long id);

}
```

Kod 44. Servis klasa za podatkovni model Kolegij

```
package hr.spring.fina.service;

import java.util.Optional;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;
import org.springframework.transaction.annotation.Transactional;

import hr.spring.fina.exceptions.ResourceNotFoundException;
import hr.spring.fina.model.Kolegij;
import hr.spring.fina.repository.KolegijRepository;



@Service
@Transactional
public class KolegijServiceImpl implements KolegijService {

	@Autowired
```

```
	private KolegijRepository KolegijRepository;

	@Override
	public Kolegij createKolegij(Kolegij Kolegij) {
		return KolegijRepository.save(Kolegij);
	}

	@Override
	public Kolegij updateKolegij(Kolegij dataKolegij) throws ResourceNotFoundException {
		Optional<Kolegij> productDb =
this.KolegijRepository.findById(dataKolegij.getId_kolegij());

		if (productDb.isPresent()) {
			Kolegij kolegijUpdate = productDb.get();
			kolegijUpdate.setNaziv(dataKolegij.getNaziv());
			kolegijUpdate.setECTS(dataKolegij.getECTS());
			kolegijUpdate.setStudij(dataKolegij.getStudij());
			KolegijRepository.save(kolegijUpdate);
			return kolegijUpdate;
		} else {
			throw new ResourceNotFoundException("Zapis nije pronađen : " +
dataKolegij.getId_koelgij());
		}
	}

	@Override
	public Iterable<Kolegij> getAllKolegij() {
		return this.KolegijRepository.findAll();
	}


	@Override
	public Kolegij getKolegij(long KolegijId) {

		if (KolegijId == 0)
			return new Kolegij();
		Optional<Kolegij> productDb = this.KolegijRepository.findById(KolegijId);

		if (productDb.isPresent()) {
			return productDb.get();
		} else {
			return new Kolegij();
		}
	}

	@Override
	public void deleteKolegij(long KolegijId) {
		Optional<Kolegij> productDb = this.KolegijRepository.findById(KolegijId);

		if (productDb.isPresent()) {
			this.KolegijRepository.delete(productDb.get());
		} else {
			throw new ResourceNotFoundException("Zapis nije pronađen.");
		}

	}
}
```

Kod 45. Implementacije servisa KolegijServiceImpl

Nakon toga dolazi metoda za mijenjanje podataka odabranog zapisa. Metoda ***updateKolegij*** na početku pristupa zapisu putem id_kriterija. Ukoliko postoji traženi zapis, provodi se upis novih podataka u objekt te upis promjena u bazu. Prilikom pristupa postojećem podatku, repozitorij kao rezultat vraća ***Optional<Kolegij>*** . Optional označava da se radi samo jedno podatku. Može se dogoditi da podatka nema u tablici, u tom slučaju vrijednost rezultata je ***null*** ili ne postoji. Ukoliko podatak postoji u tablici (kod ***: if (productDb.isPresent())***) započinje upis novih vrijednosti u objekt te zapis u tablicu. Izmjena podataka završava pozivom metode ***save*** repozitorija.

U klasi su definirane dvije ***Get*** metode. Prva metoda ***getAllKolegiji*** prikuplja sve zapise koji postoje u tablici, a kao rezultat vraća ***listu objekta*** klase Kolegij. Druga metoda getKolegij je usmjerena na pristup pojedinačnom zapisu u tablici. Metoda provjerava postoji li podatak. Ukoliko postoji, vraća njegovu instancu. U slučaju da ne postoji, metoda vraća novi nedefinirani objekt. Kod nedefiniranog objekta, njegov id ima vrijednost nula (0) što označava da ne postoji u tablici. Druga mogućnost, da metoda generira neku iznimku (*engl. Exception*) i na taj način signalizira da je došlo do određenih „problema“ u pristupu podatku u tablici. Koji način će se koristiti ovisi o pristupu i filozofiji rada same aplikacije.

Nakon get metoda dolazi metoda ***deleteKolegij*** , koja je predviđena za brisanje podatka u tablici baze podataka. Na početku metode čita se podatak na temelju indeksa ***id_kataloga***. Ukoliko podatak postoji, izvršava se brisanje zapis iz baze. Suprotno, metoda generira iznimku ***ResourceNotFoundException*** kao oznaku za nastalu situaciju.

Web kontroler

Web kontroler je zadužen za mapiranje web stranica i pripremu podataka. Koliko imamo web stranica, toliko imamo metoda za njihov pristup. U ovom konkretnom slučaju, trebamo metode koje će izvršiti HTTP GET i HTTP POST zahtjev. Kod uređivanja podataka može se koristiti POST ili PUT zahtjev. Kod 46. prikazuje web kontrolera s pripadajućim metodama.

Kontroler je mapiran s putanjom /kolegij Anotacijom ***@Autowired*** povezuje servis ***KolegijService*** kao glavno sučelje za pristup podacima. Slijede mapirane metode CRUD operacija. Prva metoda pocetakKolegiji predviđena je za prikupljanje podataka o svim zapisima tablice Kolegiji te slanje listu objekata klase Kolegij prema web stranici. Metoda listu podataka označava kao atribut pod nazivom ***listaKolegija***. Nakon prikupljanja podataka, određuje se HTML predložak koji će se koristiti za prikaz podataka. Za potrebe ovog primjera koristit će se predložak pod nazivom ***pocetnaKolegiji.html***.

Nakon toga deklarirana je metoda ***noviKolegij*** za kreiranje novog objekta klase Kolegij. Novi i neispunjen objekt prosljeđuje se u HTML predložak pod nazivom atributa kolegij. Za upis podataka o novom kolegiju koristi se HTML predložak ***novi_kolegij***.

Za cijeli proces kreiranja novog podatka koriste se dvije metode. Prva prikazuje web stranicu s neispunjenim podacima o novom kolegiju. Ova metoda koristi HTTP GET zahtjev. Druga metoda služi za zapisivanje podataka o novom kolegiju te koristi HTTP POST zahtjev koji se šalje od strane web

upisne forme. Potrebno je naglasiti da su mapa zahtjeva i putanje za ove dvije metode iste, samo je razlika u vrsti zahtjeva. Nakon upisa novog zapisa u tablicu, vrši se redirekcija ili usmjeravanje na početnu stranicu aplikacije.

Za uređivanje podataka, također se koriste dvije metode, nešto slično kao za potrebe kreiranja novog zapisa. Prva metoda, na temelju vrijednosti id_kolegija pristupa zapisu u tablici. Dobiveni zapis zapisuje se u objekt i prosljeđuje u model atributa ***kolegij***.

```
package hr.spring.fina.controllers;

@Controller
@RequestMapping("/kolegij")
public class KolegijController {

	@Autowired
	private KolegijService service;

	@RequestMapping("/")
	public String pocetakKolegiji(Model model) {
		ArrayList<Kolegij> popisKolegija = (ArrayList)service.getAllKolegij();
		model.addAttribute("listaKolegija", popisKolegija);

		return "pocetnaKolegiji";
	}

	@RequestMapping(value = "/novi", method = RequestMethod.GET)
	public String getNoviKolegij(Model model) {
		Kolegij kolegij = new Kolegij();
		model.addAttribute("kolegij", kolegij);

		return "novi_kolegij";
	}

	@RequestMapping(value = "/novi", method = RequestMethod.POST)
	public String postNoviKolegij(@ModelAttribute("Kolegij") Kolegij kolegij) {
		service.createKolegij(kolegij);
		return "redirect:/kolegij";
	}

	@RequestMapping("/uredi/{id}")
	public ModelAndView urediKolegij(@PathVariable(name = "id") int id) {
		ModelAndView mav = new ModelAndView("uredi_kolegij");
		Kolegij kolegij = service.getKolegij(id);
		mav.addObject("kolegij", kolegij);

		return mav;
	}

	@RequestMapping(value = "/uredi", method = RequestMethod.POST)
	public String snimiKolegij(@ModelAttribute("Kolegij") Kolegij kolegij) {
		service.updateKolegij(kolegij);
		return "redirect:/kolegij";
	}


	@RequestMapping("/brisi/{id}")
	public String brisanjeKolegija(@PathVariable(name = "id") int id) {
```

```
                service.deleteKolegij(id);
                return "redirect:/kolegij";
        }
}
```

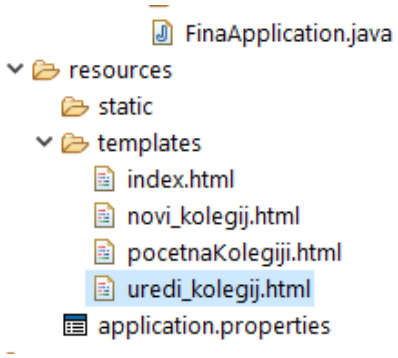
Kod 46. Web kontroler za kolegije

Kada se podaci u postojećem zapisu promijene i pritisne gumb, pokrene se HTTP POST zahtjev za upis izmjena u tablicu. Pozove se metoda ***snimiKolegij*** koja pomoću servis metode updateKolegij vrši upis izmijenjenih podataka u tablicu baze podataka. Nakon upisa izmjena, ponovno se ostvaruje redirekcija na početnu stranicu s popisom kolegija.

Za brisanje podataka koristi se metoda ***brisanjeKolegija*** koja koristi početni predložak. Metoda se pokreće pritiskom na gumb koji šalje id kolegija kojeg želimo obrisati. Metoda deleteKategorija briše trajno zapis iz baze te nakon toga ponovno usmjerava prikaz početnog predloška.

Web HTML predlošci

Svi predlošci nalaze se u mapi ***resources/templates*** u projektu aplikacije kako je prikazano na slici 21.



Slika 21. HTML predlošci za web stranice kolegija

HTML predložak početne stranice s popisom kolegija.

```
<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhtml"
    xmlns:th="http://www.thymeleaf.org">
<head>
<meta charset="utf-8"/>
<title>FINA</title>
</head>
<body>
<div align="center">
  <h1>POPIS KOLEGIJA</h1>
  <a href="/kolegij/novi">Kreiraj novi kolegij</a>
  <br/><br/>
  <table border="1" cellpadding="10">
    <thead>
      <tr>
        <th>ID kolegija</th>
        <th>Naziv</th>
        <th>ECTS</th>
        <th>Studij</th>
```

```
            <th>Akcija</th>
          </tr>
        </thead>
        <tbody>
          <tr th:each="kolegij : ${listKolegija}">
            <td th:text="${kolegij.id_kolegija}">Product ID</td>
            <td th:text="${kolegij.naziv}">Naziv</td>
            <td th:text="${kolegij.ECTS}">ects</td>
            <td th:text="${kolegij.studij}">studij</td>
            <td>
              <a th:href="@{'/kolegij/uredi/' + ${kolegij.id_kolegija}}">Uredi</a>
                
              <a th:href="@{'/kolegij/brisi/' + ${kolegij.id_kolegija}}">Briši</a>
            </td>
          </tr>
        </tbody>
    </table>
</div>
</body>
</html>
```

Kod 47. Web predložak pocetnaKolegiji.html

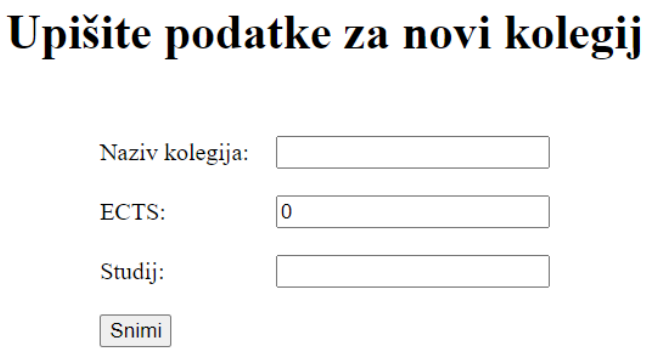
U svim predlošcima korištene su Thymeleaf naredbe koje su potrebne za rad s podacima koje šalje kontroler. Kod 47. prikazuje prvu web stranicu koja pomoću tablice prikazuje popis kolegija. Za dobivanje popisa kolegija upiše se putanja http://localhost:9040/kolegij/ , gdje 9040 predstavlja vrijednost porta na kojem se pokreće aplikacija. Na početnoj stranici, iznad tablice, nalazi se link za slanje HTTP zahtjeva za kreiranje novog zapisa. Ovaj zahtjev uzrokuje prikaz prikazuje se predložak novi_kolegij. Kod 48. prikazuje sadržaj tog predloška, a slika 22. izgled web forme za upis podataka.

Prikazana forma omogućuje upis podataka za naziv kolegija, vrijednost ECTS-a te naziv studijskog programa. Sam id_kolegija dodaje se automatski prilikom upisa. U zaglavlju forme stavljena je putanja akcije i putanje za metodu u kontroleru ***th:action="@{/kolegij/novi}"***. Gumb Snimi pokreće HTTP POST zahtjev prema web kontroleru. Kod zahtjeva se šalje ispunjeni objekt klase Kolegij kojeg zaprima metoda. U metodi se putem sučelja upisuju podaci objekta u tablicu baze podataka. Nakon obavljenog upisa ponovno se prikaže početna stranica s tablicom prikaza svih kolegija u bazi podataka.

```
<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:th="http://www.thymeleaf.org">
<head>
<meta charset="utf-8" />
<title>Kreiranje novog kolegija</title>
</head>
<body>
<div align="center">
      <h1>Upišite podatke za novi kolegij</h1>
      <br />
      <form action="#" th:action="@{/kolegij/novi}" th:object="${kolegij}" method="post">

            <table border="0" cellpadding="10">
                  <tr>
                        <td>Naziv kolegija:</td>
```

```
                        <td><input type="text" th:field="*{naziv}" /></td>
                    </tr>
                    <tr>
                        <td>ECTS:</td>
                        <td><input type="text" th:field="*{ECTS}" /></td>
                    </tr>
                    <tr>
                        <td>Studij:</td>
                        <td><input type="text" th:field="*{studij}" /></td>
                    </tr>

                    <tr>
                        <td colspan="2"><button type="submit">Snimi</button> </td>
                    </tr>
                </table>
            </form>
</div>
</body>
</html>
```

Kod 48. Web HTML predložak novi_kolegij.html



Slika 22. HTML predložak za upis novog kolegija

Ukoliko korisnik želi izmijeniti podatke u nekom od objektu Kriterij, u tablici pritisne gumb Uredi. Kod 49. prikazuje sadržaj predloška za uređivanje podataka, a slika 23. izgled zaslona forme za uređivanje podataka. U predlošku je navedena putanja te metode u kontroleru. Ovaj put to je putanja /kolegij/uredi/ i HTTP POST metoda.

```
<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhtml"
          xmlns:th="http://www.thymeleaf.org">
<head>
<meta charset="utf-8" />
<title>UREĐIVANJE PODATAKA</title>
</head>
<body>
 <div align="center">
          <h1>Kolegij</h1>
          <br />
          <form action="#" th:action="@{/kolegij/uredi}" th:object="${kolegij}" method="post">

           <table border="0" cellpadding="10">
             <tr>
                    <td>Kolegij ID:</td>
```

```
                <td>
                  <input type="text" th:field="*{id_kolegija}" readonly="readonly" />
                </td>
            </tr>
            <tr>
                <td>Naziv kolegija:</td>
                <td>
                  <input type="text" th:field="*{naziv}" />
                </td>
            </tr>
            <tr>
                <td>ECTS:</td>
                <td><input type="text" th:field="*{ECTS}" /></td>
            </tr>
            <tr>
                <td>Studij:</td>
                <td><input type="text" th:field="*{studij}" /></td>
            </tr>
            <tr>
                <td colspan="2"><button type="submit">Snimi</button> </td>
            </tr>
          </table>
    </form>
  </div>
</body>
</html>
```

Kod 49. Kod 48. Web predložak uredi_kolegij.html

9. Poglavlje: REST API u MVC aplikaciji

U ovom poglavlju naučit ćete:

- ✓ Način definiranja REST kontrolera
- ✓ Način komunikacije metode s udaljenim uređajem
- ✓ Nekoliko primjera REST metoda za slanje i primanje podataka

9. REST API u MVC aplikaciji

9.1. Pojam REST API sučelja

REST API (engl. Application Programming Interface) je sučelje koja omogućuje interakciju između web aplikacije koja se odvija na oblak poslužitelju i udaljenih uređaja ili neke web stranice. REST predstavlja standard transfera podataka koji se odvija prema utvrđenim protokolima. U oblak aplikacijama, API se koristi kao posrednik između korisnika ili klijenta i resursa ili web usluge koje pruža aplikacija. Također, API putem svojih metoda osigurava određenu sigurnost u pristupu podacima i resursima na razini poslužitelja te omogućuje provjeru autentičnosti korisnika čime se određuje razina pristupa pojedinim resursima.

REST (*engl. REpresentational State Transfer*) ne predstavlja protokol ili neki standard, već definira arhitekturu podatkovnog tipa koji se šalje prema servisu. Podatci koji se šalju imaju određeni standardni format zapisa. Putem HTTP protokola podaci mogu biti zapisani u različitim oblicima standarda: JSON, XLT, HTML, Python ili običan tekst. ***JSON*** je možda najpopularniji format koji se koristi jer je neovisan o programskom jeziku kojim je implementiran servis.

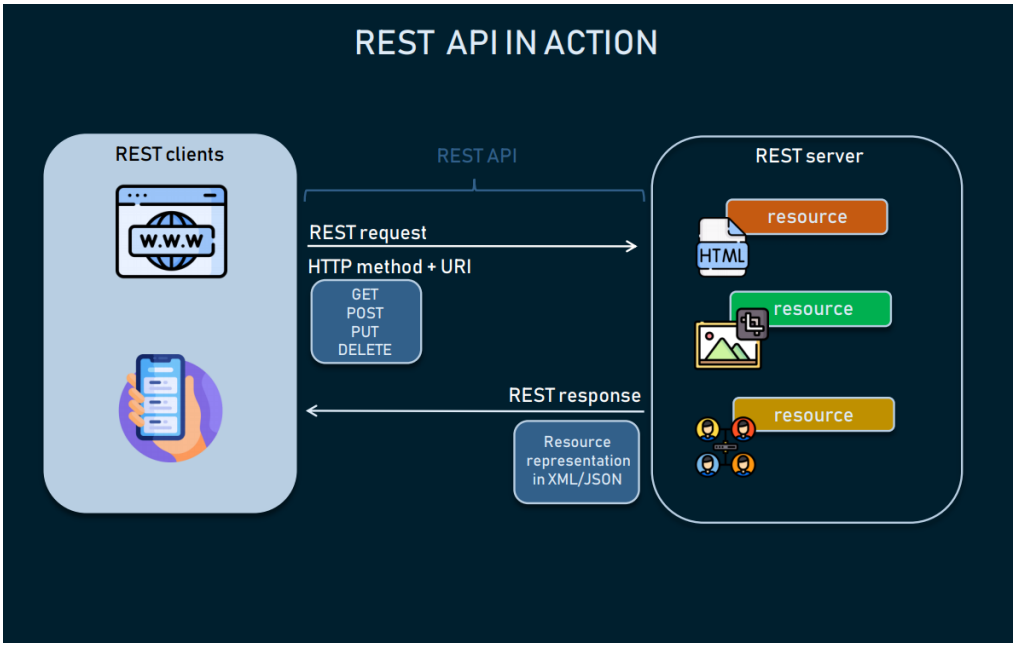
U ovom poglavlju bit će prikazan način definiranja metoda koje omogućavaju ovakvu arhitekturu distribucije podataka te način formiranja rezultata i objekata kao odgovor na zaprimljeni zahtjev. Kod svakog zahtjeva koji se šalje, definirani se sljedeći elementi:

- ***krajnja točka*** predstavlja putanju ili rutu kamo se šalje zahtjev. Kod Spring Boot API definira se REST kontroler putanja na jednak način kao i kod web kontrolera,
- ***metoda zahtjeva*** odnose se na to da li se podaci šalju, izmijenjuju ili se prikupljaju. Koriste se standardne HTTP metode: GET, POST, PUT, DELETE,
- ***zaglavlje zahtjeva*** osigurava informaciju klijentu i poslužitelju o načinu slanja podataka, što se očekuje kao rezultat i koje vrste podataka se prilikom zahtjeva šalju,
- ***podaci*** predstavljaju tijelo (engl. Body) zahtjeva. U njemu se nalazi opis podatka koji se šalje u obliku objekta ili standardne varijable.

Na slici 24. [5]prikazan je princip REST API servisa. Na slici je moguće uočiti dvije skupine komponenata. Prvu skupinu predstavljaju korisnici ili klijenti koji šalju zahtjeve prema servisu. Zahtjevi se mogu slati putem udaljenih uređaja, recimo mobilnih uređaja, ili standardnim web stranicama. REST zahtjev (*engl. REST Request*) odlazi prema poslužitelju, na domenu za koju je prijavljen. Pri tome se dodaje i ostatak putanje prema željenoj metodi servisa. HTTP protokol određuje i metodu pristupa servisu. REST poslužitelj i servis putem odabrane metode pristupa određenom resursu. Formira podatke prema podatkovnom standardu koji je definiran u zaglavlju zahtjeva. Na taj način, formirane podatke

[5] Izvor slike: https://www.altexsoft.com/blog/rest-api-design/

poslužitelj koristeći ***REST response*** šalje prema korisniku koji je uputio ovaj zahtjev. U tom odgovoru mogu biti podaci u obliku objekta, datoteke, liste objekata ili neki drugi multimedijski sadržaj.



Slika 24. Princip REST API servisa i načina komunikacije

9.2. Kreiranje REST API metoda

Za korištenje REST servisa potrebno je definirati ***REST kontroler***. Ta vrsta kontrolera definira se na jednak način kao i web kontroler samo se u zaglavlju treba koristiti ***@RestController*** anotacije. U nekim metodama koristi se ***@ResponseBody*** anotacija. Ona označava što se šalje u tijelu zahtjeva. To mogu biti objekti, datoteke, lista objekata ili neka druga kombinacija podataka. U kodu 50. prikazan je jednostavni primjer metode i upotrebe anotacije.

Za svaki kontroler potrebno je definirati putanju za pristup pojedinim metodama. Ukoliko korisnik pošalje zahtjev koji nije mapiran i dozvoljen HTTP protokolom, poslužitelj generira pogrešku u pristupu pod ***kodom 405***. U slučaju da je sve ispravno odrađeno, poslužitelj vraća korisniku koji je poslao zahtjev ***kod 200*** što označava ispravnost izvršavanja zahtjeva.

U kodu 50. nalazi se primjer koda jednog takvog kontrolera pod nazivom ***StudentController***. Njegova putanja je ***/studenti/api*** , na koju se onda dodaje sama mapa putanje pojedine metode koja se poziva od strane korisnika. U kodu je navedena metoda odrediStudentaById čiji je zadatak u tablici Studenata pronaći zapis o studentu kojemu se indeks poklapa s traženom vrijednosti varijable ***id_student***.

U mapi putanje poziva dodaje se numerička vrijednost indeksa zapisa koja je označena s ***@PathVariable*** anotacijom. Na ovaj način definira se da se vrijednost varijable nalazi u samoj putanji poziva metode, kao parametar koji se nalazi između ***{ ... }*** zagrada. U samom zaglavlju metode dodatno

je definiran podatkovni tip koji se očekuje od te varijable. U konkretnom primjeru, varijabla je ***long*** tip podataka.

Metode mogu, ali ne moraju vraćati rezultat njihovog rada. Ukoliko metoda vraća neki rezultat, vrsta i tip rezultata određuje se pomoću ***@ResponseBody*** anotacije. U ovom slučaju definirano je da rezultat koji metoda vraća bude ***objekt Student.*** Uobičajeno je da se uz metodu stavi i mehanizam za upravljanje iznimkama u slučaju da nešto krene kako nije planirano. U slučaju da metoda nema pristup bazi podataka ili iz nekog drugog razloga servis za pristup repozitoriju nije pravilno formiran. Tada na scenu dolazi upravo taj mehanizam, a korisnik s udaljenog mjesta dobiva povratnu informaciju o kodu greške. Ukoliko se radi o nekoj mobilnoj aplikaciji, korisnik dobiva prikaz informacije na zaslon uređaja na temelju čega zna kako reagirati na nepravilan rad aplikacije. Može biti da je sve ispravno s mobilnom aplikacijom, a da je greška u radu oblak poslužitelja. U oba slučaja, korisnik mobilne aplikacije treba dobiti povratnu informaciju.

```
@RestController
@RequestMapping(“/studenti/api”)
class StudentController {

  @GetMapping(“/{id_studenta }”)
  public @ResponseBody Student odrediStudentaById(@PathVariable long id_studenta) {
     return studentService.findById(id_studenta);
  }
}
```

Kod 50. Primjer korištenja anotacije @ResponseBody

U kodu 51. prikazan je primjer REST kontrolera za objavljivanje obavijesti. U složenim aplikacijama dobra je praksa da se REST kontroleri nalaze u zasebnoj mapi projekta, radi bolje organiziranosti koda same aplikacije. Što se tiče naziva klase, nepisana je praksa da je naziv kontrolera povezan s nazivom klase objekta ili tablice kojima se pristupa. Primjer, ako se radi o studentima, preporuka naziva bi bila ***StudentController***. Ako se radi o obavijestima naziv bi bio ***ObavijestController*** i tako dalje. Može se koristiti neka druga metodologija kreiranja naziva, ali bitno je da se klase pomoću naziva i lokacije u projektu pronalaze brzo te da su nazivi nedvosmisleni.

```
// 1. Klasa entiti Obavijest
package hr.spring.fina.model;

@Entity
@Table(name = "obavijest")
public class Obavijest {

        @Id
        @GeneratedValue(strategy = GenerationType.IDENTITY)
        private long id_obavijest;

        @Column
        private long id_fakultet; // oznaka fakulteta
```

```
        @Column(nullable = false)
        private String naslov;

        @Column(nullable = false)
        private String tekst;

        @Column
        private String slika;

        @Column
        private int id_studij; // oznaka studijskog programa

        @Column
        private String naziv_studija;

        @Column
        @Temporal(TemporalType.TIMESTAMP)
        @DateTimeFormat(pattern = "yyyy-MM-dd")
        private Date datum_objavljeno;

        @Column
        private String datum_objavljeno_tekst = "";

...

}
```

// 2. Repozitorij

```
package hr.spring.fina.repository;

public interface ObavijestRepository extends JpaRepository<Obavijest, Long> {

  @Query(
    value = "SELECT *  FROM obavijesti u WHERE u.id_fakultet = ?1 ORDER BY u.datum DESC",
                                    nativeQuery = true)
  Collection<Obavijest> findAllByFakultet(long id_fakultet);

  @Query(
   value = "SELECT *  FROM obavijesti u WHERE u.id_fakultet = ?1 AND u.id_studij = ?2 ORDER BY
u.datum DESC", nativeQuery = true)
  Collection<Obavijest> findAllByFakultetStudij(long id_fakultet, long id_studij);

}
```

// 3. Servis

```
package hr.spring.fina.service;

import hr.spring.fina.model.Obavijest;

public interface ObavijestiService {

        Obavijest createEntity(Obavijest entityData);

        Obavijest updateEntity(Obavijest entityData);

        Iterable<Obavijest> getAllEntityByFakultet(long id_fakultet);

        Iterable<Obavijest> getAllEntityByFakultetStudij(long id_fakultet, long id_studij);

        Obavijest getEntityById(long entityId);
```

```
	void deleteEntity(long entityId);

}

// 4. REST kontroler
package hr.spring.fina.restcontroller;

@RestController
@RequestMapping("obavijesti")
public class ObavijestiController {

@Autowired
private ObavijestiService obavijestiService;

// API
@RequestMapping(value = "/api/sveObavijestiFakultet/{id_fakultet}", method =
RequestMethod.GET, headers = "Accept=application/json")
public @ResponseBody ArrayList<Obavijest> getAllEntitiesFakultet(@PathVariable long id_fakultet)
{
	return (ArrayList<Obavijest>) this.obavijestiService.getAllEntityByFakultet(id_fakultet);
}

@RequestMapping(value = "/api/sveObavijestiFakultet/{id_fakultet}/{id_studij}", method =
RequestMethod.GET, headers = "Accept=application/json")
public @ResponseBody ArrayList<Obavijest> getAllEntitiesFakultetStudij(@PathVariable long
id_fakultet, @PathVariable long id_studij) {
	return (ArrayList<Obavijest>)this.obavijestiService.
			getAllEntityByFakultetStudij(id_fakultet, id_studij);
}

@RequestMapping(value = "/api/upisObavijesti", method = RequestMethod.POST)
public ResponseEntity<ResponseData> upisObavijesti(@RequestBody Obavijest dataObavijest)
throws IOException {
	ResponseData RD = new ResponseData();
	try {
		this.obavijestiService.createEntity(dataObavijest);
		RD.setErrorCode(200);
		RD.setMessage("Uspješno upisan podatak");
		return ResponseEntity.status(200).body(RD);
	} catch (Exception err) {
		RD.setErrorCode(403);
		RD.setMessage("Pogreška kod upisa podataka");
		return ResponseEntity.status(403).body(RD);
	}
}

@RequestMapping(value = "/api/obnoviObavijest", method = RequestMethod.PUT)
public ResponseEntity<ResponseData> obnoviObavijest(@RequestBody Obavijest dataObavijest)
throws IOException {
	ResponseData RD = new ResponseData();
	try {
		this.obavijestiService.updateEntity(dataObavijest);
		RD.setErrorCode(200);
		RD.setMessage("Uspješno obnovljen podatak");
		return ResponseEntity.status(200).body(RD);
	} catch (Exception err) {
		RD.setErrorCode(403);
		RD.setMessage("Pogreška prilikom obnavljanja podataka");
		return ResponseEntity.status(403).body(RD);
	}
}
```

```
@RequestMapping(value = "/api/obrisiObavijest", method = RequestMethod.DELETE)
public ResponseEntity<ResponseData> brisiObavijest(@RequestBody Obavijest dataObavijest)
throws IOException {
		ResponseData RD = new ResponseData();
		try {
			this.obavijestiService.deleteEntity(dataObavijest.getId_obavijest());
			RD.setErrorCode(200);
			RD.setMessage("Podaci o obavijesti uspješno su obrisani");
			return ResponseEntity.status(200).body(RD);
		} catch (Exception err) {
			RD.setErrorCode(403);
			RD.setMessage("Pogreška prilikom brisanja podatka.");
			return ResponseEntity.status(403).body(RD);
		}
}

}

public class ResponseData {
   long errorCode;
   String message;

   ...
}
```

Kod 51. Primjer REST kontrolera za pristup podacima repozitorija Obavijest

Na početku koda prikazana je klasa ***entitete Obavijest***. Klasa sadrži određeni broj atributa. Svaka obavijest određena je za točno određeni fakultet te studijski program odabranog fakulteta. Klasa osim atributa sadrži ***get/set*** metode koje nisu prikazanu u primjeru koda.

Slijedi kod repozitorija koji sadrži dvije dodatne metode. Prva metoda izvršava Select SQL upit koji pristupa podacima za traženi fakultet. Zapisi u tablici raspoređuju se s id_fakultet vrijednosti. Rezultat je popis objekata klase Obavijest. Druga dodana metoda u repozitoriju je metoda koja pretražuje zapise koji odgovaraju traženom studijskom programu, traženog fakulteta. Rezultat SQL upita, također je popis objekata nastalih iz zapisa tablice Obavijest.

Za pristup podacima baze podataka, udaljeni korisnici koriste sučelje servisa ***ObavijestiService***. U servisu su definirane osnovne CRUD metode sa svim karakteristikama koje su navedene u prijašnjem poglavlju.

Posljednji dio koda sadrži REST kontroler. Putanja za pristup kontroleru i metodama je ***/obavijesti*** što je određeno anotacijom na početku samog kontrolera. Prva metoda u kontroleru je ***sveObavijestiFakultet*** koja je namijenjena za pristup podacima o obavijestima određenog fakulteta. Putanja za pristup metodi je ***/obavijesti/api/sveObavijestiFakultet/{id_fakultet}*** . U toj putanji nalazi se varijabla {id_fakultet} koja je definirana u samom zaglavlju metode. Varijabla je definirana pomoću ***@PathVariable*** anotacije. Uz anotaciju je određen i podatkovni tip varijable koji je u ovom primjeru tipa ***long***. Osim definiranja varijable u zaglavlju, određen je i način transfera podataka. U konkretnom kodu stavljena je naredba ***headers = "Accept=application/json"*** koja označava da je JSON format

zapisa podataka i prijenosa podataka objekata. U samom kodu metode nalazi se naredba koja poziva metoda ***getAllEntityByFakultet*** servis te prihvaća zapise iz baze za određeni fakultet. Rezultat se pretvara u ArrayList listu podataka te se prosljeđuje u ***@ResponseBody*** metode. Korisnik koji je uputio ovaj ***GET*** zahtjev kao rezultat dobiva popis objekata formiranih u JSON zapisu.

Slijedi vrlo slična metoda, metoda koja pribavlja podatke o obavijestima, ali za točno određeni studijski program određenog fakulteta. U putanji pristupa, navedene su dvije varijable.

Metoda ***upisObavijesti*** predviđen je za upis nove obavijesti na zahtjev udaljenog korisnika aplikacije. Recimo da je zahtjev upućen od strane mobilnog uređaja autoriziranog korisnika aplikacije. Putanja do metode je ***/obavijesti/api/ upisObavijesti .***, Zahtjevom se šalje objekt Obavijesti. podaci koje korisnik želi zapisati u bazu podataka. Podatkovni tip objekta i naziv objekta je ***@RequestBody Obavijest dataObavijest*** . U tijelu metode, poziva se sučelje i metoda ***createEntity*** servisa za kreiranje novog zapisa u bazi. Rezultat metode je odgovor na zahtjev u obliku ***ResponseData*** objekta. Ovo je korisnička klasa koja zahtjevu daje povratnu informaciju. Klasa se sastoji od dva atributa, ***errorCode*** za upis koda rezultata i ***message*** za mogućnost dodatnog upisa poruke korisniku. Sama metoda mogla je biti realizirana standardnim HTTP klasama za evidentiranja rezultata, ali na ovaj način daje se mogućnost da server sam definira oblik povratne informacije korisniku. U samoj metodi, radi sigurnosti, moguće je dodati određeni kod za validaciju podataka prije samog upisa u bazu.

9.3. Upravljanje iznimkama

Tijekom rada aplikacije i izvršavanja raznih zahtjeva, može doći do određenih pogrešaka u radu ili nekih nepravilnosti. Takva situacija nije poželjna u aplikaciji, ali se događa. Zbog toga je u aplikaciji i njenim metodama potrebno implementirati razne mehanizme za upravljanje iznimkama (*engl. Error Handling*).

U prvim verzijama Spring Boot aplikacija za MVC aplikacije korištene su dvije osnovne anotacije, ***@HandlerExceptionResolver*** i ***@ExceptionHandler***. U najnovijoj verziji uvedena je nova bazna ***@ResponseStatusException*** anotacija. Anotacije omogućuju da aplikacija vraća rezultat iznimke i na taj način indicira status izvođenja zahtjeva.

@ExceptionHandler anotacija za implementaciju upravljanja iznimkama realizira se na razini pojedinačnog kontrolera i nije predviđena za globalnu razinu cijele aplikacije. Naravno, ukoliko se anotacija stavi u svaki kontroler, ona postaje globalna. Kod 52. prikazuje primjenu te anotacije.

```
public class ObavijestController{

   //...
   @ExceptionHandler({ CustomException1.class, CustomException2.class })
   public void handleException() {
      //
   }
}
```

Kod 52. Primjena @ExceptionHandler anotacije

@HandlerExceptionResolver anotacija omogućuje rješenje na razini aplikacije. Anotacija omogućuje definiranje jedinstvenog mehanizma upravljanja iznimkama. U toj anotaciji implementirane su dvije klase za upravljanje iznimkama: ***ExceptionHandlerExceptionResolver*** i ***DefaultHandlerExceptionResolver***.

DefaultHandlerExceptionResolver klasa prevodi iznimke koje su nastale u Spring MVC aplikacijama te generiraju specifični kod pogreške. U tablici 1. prikazanje pregled kodova pogreške i upravljači iznimkama.

Ukoliko postoji potreba za kreiranjem svog vlastitog mehanizma za upravljanje iznimkama, možemo koristiti ResponseStatusExceptionResolver. U zasebnoj klasi koja je izvedena iz neke Exception klase, definiraju se metode koje će se pokrenuti. U kodu 53. prikazana je klasa koja se koristi za upravljanje iznimkama u slučaju da je došlo do pogreške u pristupu nekom resursu. Status pogreške HTTP.Status.NOT_FOUND.

Naziv iznimke	Kod
BindException	400 (Bad Request)
ConversionNotSupportedException	500 (Internal Server Error)
HttpMediaTypeNotAcceptableException	406 (Not Acceptable)
HttpMediaTypeNotSupportedException	415 (Unsupported Media Type)
HttpMessageNotReadableException	400 (Bad Request)
HttpMessageNotWritableException	500 (Internal Server Error)
HttpRequestMethodNotSupportedException	405 (Method Not Allowed)
MethodArgumentNotValidException	400 (Bad Request)
MissingServletRequestParameterException	400 (Bad Request)
MissingServletRequestPartException	400 (Bad Request)
NoSuchRequestHandlingMethodException	404 (Not Found)
TypeMismatchException	400 (Bad Request)

Tablica 1. Pregled kodova iznimka

```
@ResponseStatus(value = HttpStatus.NOT_FOUND)
public class MyResourceNotFoundException extends RuntimeException {
   public MyResourceNotFoundException() {
      super();
   }
   public MyResourceNotFoundException(String message, Throwable cause) {
      super(message, cause);
   }
   public MyResourceNotFoundException(String message) {
      super(message);
   }
   public MyResourceNotFoundException(Throwable cause) {
      super(cause);
   }
}
```

Kod 53. Primjer kreiranja vlastitog mehanizma upravljanja iznimkama

Na početku klase postavlja se ***@ResponseStatus*** anotacija u kojoj se navodi i koji kod greške podržava kreirana klasa. U ovom primjeru to je određeno s konstantom ***NOT_FOUND***.

10. Poglavlje: Upravljanje datotekama

U ovom poglavlju naučit ćete:

- ✓ Način organizacije datoteka u oblak okruženju
- ✓ Način konfiguriranja klase za pristup statičkim resursima
- ✓ Kreiranje servisa za pristup datotekama
- ✓ Implementirati web stranicu s pristupom datotekama

10. Procedure pohranjivanja datoteka

U svakoj web aplikaciji za prikaz informacija i podataka koriste se razni resursi. Pri tome se misli na razne fotografije, datoteke, dokumente. Također, u složenim aplikacijama često postoji potreba za slanje raznih dokumenata na poslužitelj. Potreba za zamjenom sadržaja s novim ili uklanjanje dokumenata i fotografija s poslužitelja. Sve te operacije iziskuju korištenje sigurnosnih procedura da ne bi došlo do neautoriziranog pristupa resursima ili štetnog uklanjanja dio resursa koji su zaštićeni i bitni za rad same aplikacije.

Koriste se razni načini kontrole pristupa resursima i definiranje načina njihovog korištenja. Prvi i osnovni način je da u konfiguracijskoj klasi, izvedenoj iz ***WebMvcConfigurer*** klase, odrede putanje koje su dozvoljene za pristup resursima. Drugi način je da se kreira poseban servis i sučelje za pristup resursima. Najčešće se to odnosi na datoteke različitih formata.

10.1. Statički resursi

Putanje do statičkih resursa najjednostavnije se definiraju pomoću application.properties datoteke. U njoj se navodi putanja i mapa gdje se nalaze resursi. Prilikom definiranja moguće je označiti razne mape gdje se statički resursi nalaze. U kodu 54. prikazan je sadržaj ***application.properties*** datoteke.

```
spring.web.resources.static-locations=classpath:/files/,classpath:/static-files

storage.location=./uploads
```

Kod 54. Primjer definiranja putanje za statičke resurse

U ovom kratkom primjeru, određene su lokacije statičnih resursa koje mogu biti u samom projektu ili van njega. Ukoliko se radi o resursima koji su u samom projektu, kao što je logo, razne slike ili određeni dokumenti, tada je takve resurse moguće samo čitati, bez mogućnosti izmjene. To je zato što su ti dokumenti zapakirani u WAR dokument aplikacije. U slučaju da imamo potrebe dodavanja dokumenata, tada je potrebno definirati mapu izvan projekta.

Znatno bolje rješenje je da se kreira konfiguracijska klasa u kojoj se definiraju putanje raznih resursa. U kodu 55. prikazan je primjer jedne takve klase. Na početku deklaracije klase nalazi se ***@Configuration*** anotacija kojom se deklarira klasa da je dio konfiguracije aplikacije. Sama klasa implementira ***WebMvcConfigurer*** klasu predviđenu za konfiguriranje MVC aplikacije.

U samoj klasi nalazi se metoda ***addResourceHandlers*** koja dinamički u aplikaciju dodaje informacije o lokacijama resursa. Prikazana su dva načina definiranja resursa. Prvi je da se definira putanja resursa koja je u ovom primjeru izvan aplikacije, negdje na samom poslužitelju. Određuje se bazna putanja kojoj se kasnije dodaju dodatne putanje mapa. File.separator se koristi za određivanje karaktera koji označava putanju mape ili podmape. Na taj način aplikacija postaje neovisna od operativnog sustava koji je instaliran na poslužitelju. To je zato što postoje određene razlike u označavanju te da se izbjegne potreba za različitim verzijama aplikacije.

Naredbom ***file:putanja*** određuje se konačna lokacije resursa. Nakon određivanja konačne putanje lokacije poziva se metoda ***registry.addResouceHandler*** kojom se u aplikaciju dodaje informacija o određenom resursu. U toj naredbi može se specificirati naziv koji će se koristiti za pristup resursu. To je izuzetno dobar način definiranja jer se pomoću njega sakriva od korisnika stvarna putanja resursa. Korisnik koristi za pristup resursu taj ***naziv resursa + naziv_dokumenta***. Korisnik ne može pristupiti izravno resursu već preko zaštićenog naziva koji je određen prilikom registracije resursa. U ovom primjeru koristi se naziv ***„/ImgEx/“*** , a iza tog naziva krije se stvarna putanja koja je određena u varijabli ***myExternalFilePath***.

U drugom primjeru, kod registracije resursa koristi se izravna putanja. Kod takvog načina registracije u HTML kodu web stranice potrebno je napisati kompletnu putanju. Prvi primjer daje fleksibilnost prilikom izmjena u samoj aplikaciji i eventualnog premještanja resursa na neku drugu lokaciju. Izvorni kod web stranice ostaje nepromijenjen što daje mogućnost lakšeg održavanja aplikacije.

```
@Configuration
public class StaticResourceConfiguration implements WebMvcConfigurer {

 @Override
  public void addResourceHandlers(ResourceHandlerRegistry registry) {
    String pathGL ="/opt/tomcat/webapps/";
    String mapaSlike = pathGL+ File.separator+ "uploads"+ File.separator+"images"+ File.separator;
    String myExternalFilePath = "file:" + mapaSlike; // slike

     registry.addResourceHandler("/ImgEx/**").addResourceLocations(myExternalFilePath);  // slike

     registry
       .addResourceHandler("/files/**")
       .addResourceLocations("file:/opt/files/");
  }

}
```

Kod 55. Primjer konfiguracijske klase za definiranje statičkih resursa

10.2. Servis i Sučelje za pristup resursima

Kada se pristupa resursima unutar kontrolera ili servisa, potrebno je definirati servis i sučelje za pristup. Unutar klase postavljaju se metode koje osiguravaju da se proces pristupa odvija ispravno i pod kontrolom. Posebno se to odnosi na dodavanje novih resursa gdje je potrebno prije zapisivanja nekog resursa ili dokumenta provesti provjeru o veličini datoteke, tipu datoteke i slično. Takve provjere su potrebne ukoliko korisnik šalje zabranjen format datoteke. Format koji bi mogao sadržavati određenu sigurnosnu prijetnju. Kod 56. prikazuje klasu sučelja za upravljanje resursima.

Na početku primjera koda nalazi se sučelje servisa ***StorageServices*** koji je predviđen za upravljanje statičnim resursima. Prve tri metode predviđene su za zapisivanje resursa u predviđenu mapu. Nakon toga nalazi se metoda koja određuje putanju do određenog resursa na poslužitelju. Metoda kao rezultat vraća objekt klase ***Path***. Slijedi metoda koja vraća traženi resurs, datoteku na osnovi naziva

dokumenta i atributa mode kojim se određuje mapa resursa. Posljednja metoda izvršava brisanje dokumenta.

Potrebno je sučelje za rad s resursima. U primjeru klase, nalazi se varijabla ***rootLocation*** koja je klase Path i pokazuje lokaciju web Spring Boot aplikacije na poslužitelju. Sve metode store na početku validiraju ulazne atribute. U slučaju određenih nepravilnosti svaka metoda generira iznimku koja se prosljeđuje u kontroler koji je koristio ovo sučelje. Ukoliko je sve ispravno naredbom ***Files.copy*** pomoću toka podataka, sadržaj datoteke zapisuje se u mapu na koju pokazuje varijabla ***rootLocation*** .

U metodu za brisanje statičnih resursa, prvo se provjerava postoji li resurs u mapi. U slučaju da nema traženog resursa, generira se iznimka. Suprotno iznimci, metoda naredbom ***getFile.delete*** metoda briše traženi resurs. Rezultat ove aktivnosti, metoda vraća ***boolean*** vrijednost kao oznaku uspješnosti izvršavanja zahtjeva.

```
public interface StorageService {

   String store(MultipartFile file);

   String store(MultipartFile file, int  mode, String nazivFile);

   String storeFile(ByteArrayOutputStream file, int  mode, String nazivFile);

   Path getPath(int mode);

   Resource loadAsResource(String filename, int mode);

   boolean deleteFile(String filename, int mode);

}

@Service
public class FileSystemStorageService implements StorageService {

 private final Path rootLocation;

 @Autowired
 public FileSystemStorageService(StorageProperties properties) {
  this.rootLocation = Paths.get(properties.getLocation());
 }

 @Override
 public String store(MultipartFile file) {
  String filename = StringUtils.cleanPath(file.getOriginalFilename());
  try {
            if (file.isEmpty()) {
              throw new StorageException("Failed to store empty file " + filename);
            }
            if (filename.contains("..")) {
              // This is a security check
              throw new StorageException(
                        "Nije moguće snimiti datoteku izvan zadanog direktorija " + filename);
            }
            try (InputStream inputStream = file.getInputStream()) {
              Files.copy(inputStream, this.rootLocation.resolve(filename),
```

```
                                        StandardCopyOption.REPLACE_EXISTING);
            }
  } catch (IOException e) {
            throw new StorageException("Failed to store file " + filename, e);
  }

  return filename;
 }
 ...

 @Override
 public Resource loadAsResource(String filename, int mode) {
  try {
    Path file = load(filename,mode);
    Resource resource = new UrlResource(file.toUri());
    if (resource.exists() || resource.isReadable()) {
            return resource;
    } else {
            throw new FileNotFoundException("Could not read file: " + filename);
    }
  } catch (MalformedURLException e) {
            throw new FileNotFoundException("Could not read file: " + filename, e);
  }

 }

 @Override
 public boolean deleteFile(String filename, int mode) {
    try {
    Path file = load(filename,mode);
    Resource resource = new UrlResource(file.toUri());
    if (resource.exists() ) {
         if (resource.getFile().delete()) {
            return true;
         } else
            return false;
         }
    else {
        return false;
    }
    } catch (Exception e) {
       return false;
    }

  }
...
}
```

Kod 56. Primjer klase sučelja za upravljanje statičkim resursima (datotekama)

Ovako pripremljeno sučelje i klasa servisa može se koristiti u web ili REST kontroleru. Kod 57. prikazuje primjer korištenja sučelja i servisa u REST kontroleru. Na početku kontrolera pomoću anotacije u klasu se umeće servis ***StorageService*** koji je prikazan u prijašnjem kodu. Sama metoda ima nekoliko parametara. Metoda podržava HTTP POST metodu transfera podataka. U tijelu metode naveden je parametar koji se šalje. On je deklariran kao ***MultipartFile*** , što označava da se u metodu može poslati jedan ili više datoteka. Zajedno s datotekama, u zahtjevu se šalje i objekt klase Obavijest zapisan u JSON formatu. Kasnije u metodi, taj objekt se pomoću ***Gson klase*** pretvara iz JSON formata u

podatkovni model Obavijest koji je definiran u samoj aplikaciji. Nakon određenih provjera, formira se naziv resursa te se naredbom ***store***, datoteka zapisuje na poslužitelj u mapu koja je određena putanjom resursa.

```
package hr.hrcity.restcontroller;

@RestController
@RequestMapping("obavijesti")
public class ObavijestiController {

 @Autowired
 private StorageService storageService;

 @RequestMapping(value = "/upload", method = RequestMethod.POST)
 public ResponseEntity<String> uploadFile(HttpSession request, @RequestParam(value =
"formDataJson")
                Object data,  @RequestParam("fileData") MultipartFile file) throws IOException {
          try {
           Gson gson = new Gson();
           Obavijesti entityData = gson.fromJson(data.toString(), Obavijesti.class);
           ...
           if (!file.isEmpty()) {
                    // zapis na disk u mapu
                    String filename = StringUtils.cleanPath(file.getOriginalFilename());
                    String ext = FilenameUtils.getExtension(filename).toUpperCase();
                    // naziv dokumenta
                    String name = "info_"+ naziv_datoteke +"."+ext;
                    name = storageService.store(file, 2, name); // zapis u mapu dokumenata
           }
           return ResponseEntity.ok().body("Ispravan upis podatka");
          } catch (Exception err) {
           return ResponseEntity.status(602).body("Pogreška prilikom upisa podataka");
          }
 }

}
```

Kod 57. Primjer REST kontrolera za upis statičnih resursa

U slučaju da je potrebno urediti podatak vezano za određeni resurs, tada je najjednostavnije da se stari resurs (datoteka) obriše, a nakon toga nova datoteka zapiše na lokaciju predviđenu za statični resurs.

11. Poglavlje: Spring Boot mikroservis

U ovom poglavlju naučit ćete:

- ✓ Što je mikroservis
- ✓ Koji su preduvjeti za korištenje mikroservisa
- ✓ Način kreiranja mikroservisa
- ✓ Implementirati mikroservis u aplikaciju

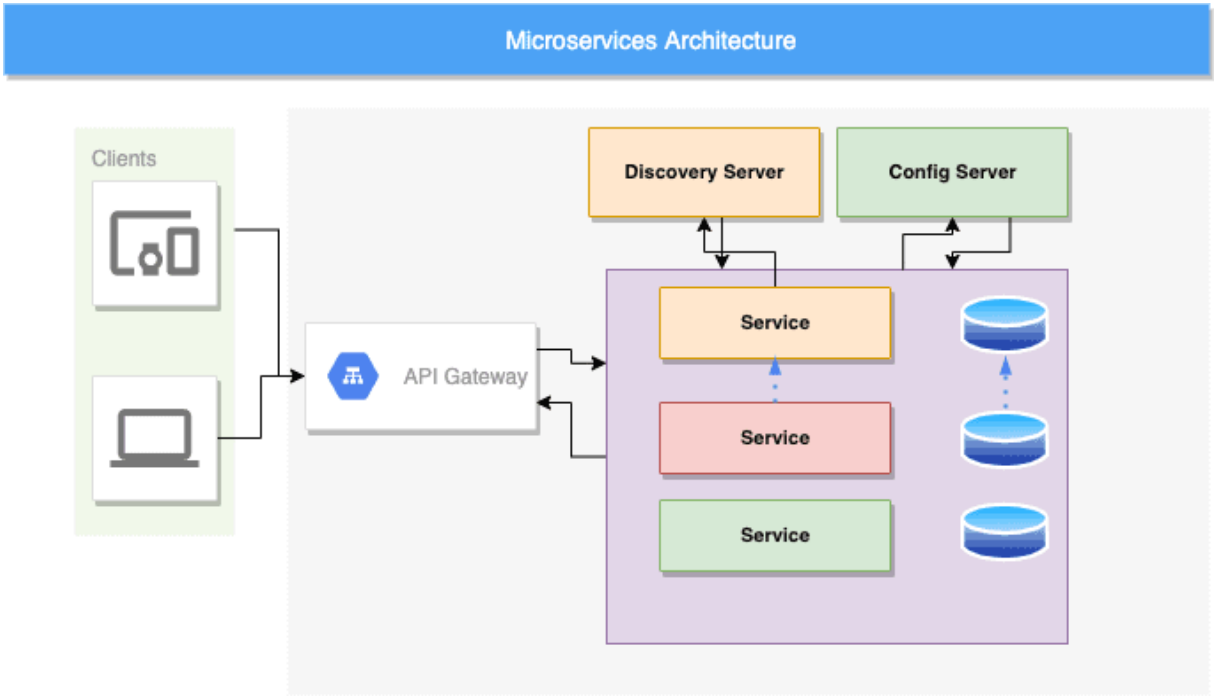
11. Spring Boot mikroservis

11.1. Kreiranje mikroservisa

Mikroservis (*engl. Micro Service*) je arhitektura koja omogućuje programerima da samostalno razvijaju i implementiraju usluge. Svaka pokrenuta usluga ima svoj vlastiti proces i time se postiže lagani model za podršku poslovnih aplikacija. Mikroservisi znače mnogo manjih usluga ili samostalnih aplikacija spremnih za pokretanje. Za razliku od monolitne arhitekture, mikroservis je kombinacija većeg broja mikroservisa koji međusobno komuniciraju i razmjenjuju podatke i na taj način formiraju aplikaciju.

Mikroservisi razbijaju veliku aplikaciju na različite manje samostalne dijelove. U slučaju da se pojavi bilo kakav problem, lako ga je locirati. U uklanjanju tog problema nema utjecaja na rad ostalih mikroservisa. Time se izbjegava zaustavljanje cijele aplikacije zbog uklanjanja manjeg problema.

Svaki mikro servis ima svoju bazu podataka. Klijentski API nemaju izravan pristup uslugama već mogu komunicirati samo preko API sučelja i servise. Svaku takvu mikro uslugu potrebno je registrirati na poslužitelju. Za tu potrebu koristi se poseban poslužitelj, ***poslužitelj za otkrivanje*** (*engl. discovery server*). Osnovna zadaća poslužitelja je otkrivanje informacija o svim mikro uslugama dostupnima u sustavu. Uz njega se koristi i ***konfiguracijski poslužitelj*** koji sadrži sve konfiguracije za mikroservise.



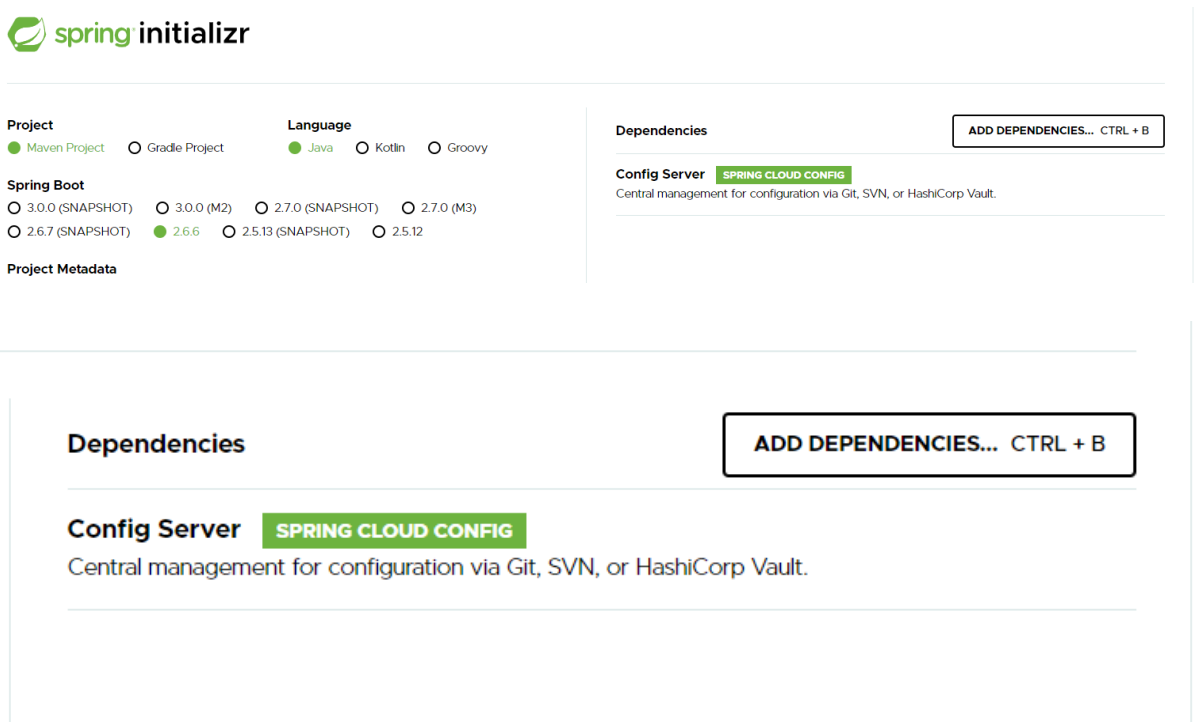
Slika 25. Arhitektura oblak sustava za pokretanje mikroserrvisa[6]

Na slici 25. prikazana je arhitektura oblak poslužitelja na kojem se pokreću mikroservisi. Može se vidjeti da korisnici sustava nemaju izravni pristup resursima te da je njihov pristup isključivo preko REST API servisa. Dakle, kod mikroservisa nema standardne web stranice, već se radi o servisima koji najčešće

[6] Izvor: https://www.javadevjournal.com/spring-boot/microservices-with-spring-boot/

imaju namjenu da pristupaju dijelu podataka, resursa te izvršavaju određene zahtjeve usmjerene na podatke.

Na slici se korisnici dolaze do mikroservisa putem REST API servisa (API Geteway). Na temelju poslanog zahtjeva od strane korisnika, poslužitelj otkrivanja određuje o kojem mikroservisu se radi te ga pokreće. ***Konfiguracijski poslužitelj*** je mjesto gdje se pohranjuju i održavaju svi konfiguracijski parametri svih mikroservisa i samog sustava. Poslužitelj se može ugraditi u Spring Boot aplikaciju pomoću ***@EnableConfigServer*** anotacije. Konfiguracijski poslužitelj može se dodati aplikaciji kod same inicijalizacije Spring Boot projekta kako je prikazano na slici 26. Odabirom ovisnosti odabere se ***Spring Cloud Config*** što će rezultirati davanjem ovisnosti u Maven konfiguracijsku datoteku projekta buduće Spring Boot aplikacije.



Slika 26. Konfiguracijski poslužitelj kod inicijalizacije

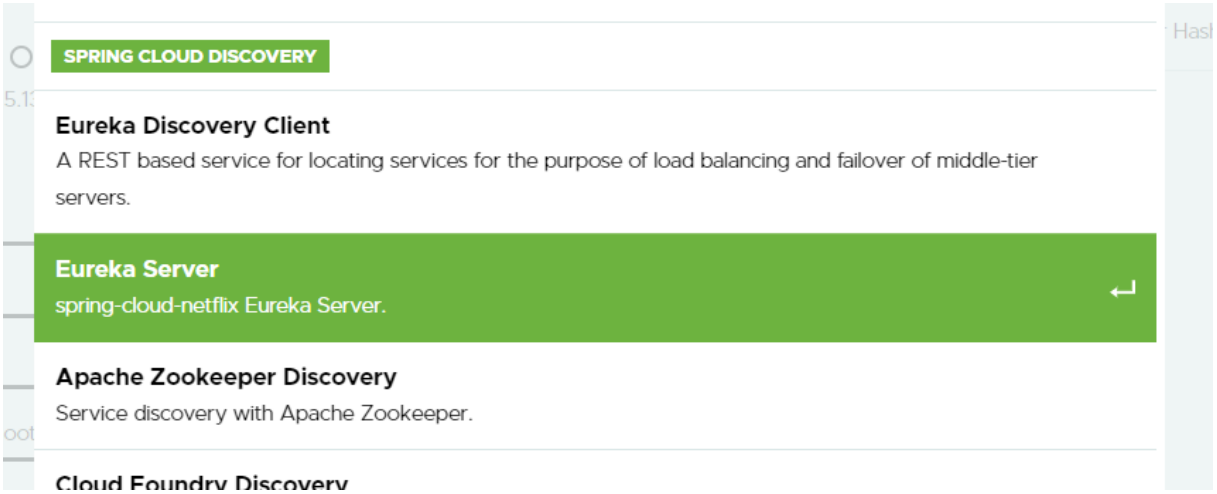
U samoj glavnoj aplikaciji potrebno je dodati anotaciju kako je prikazano u kodu 58.

```
package hr.spring.fina;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.cloud.config.server.EnableConfigServer;
import org.springframework.cloud.netflix.eureka.server.EnableEurekaServer;

@EnableConfigServer
@EnableEurekaServer
@SpringBootApplication
public class FinaApplication {

	public static void main(String[] args) {
		SpringApplication.run(FinaApplication.class, args);
	}
```

```
}
```

Kod 58. Uključivanje konfiguracijskog poslužitelja u aplikaciju

Osim konfiguracijskog poslužitelja potrebno je u projekt uključiti i poslužitelja za otkrivanje. Ovaj poslužitelj dodaje se na način da se označi Eureka server u ponuđenim opcijama prilikom inicijalizacije. Na slici 27. prikazana je ponuđena opcija za uključivanje poslužitelja. Ovo uključivanje u Maven datoteci dodaje se uključivanje ovisnosti, prikazano u kodu 59. U samoj aplikaciji potrebno je dodati ***@EnableEurekaServer*** anotaciju koja će uključiti poslužitelj pronalaženja u samu aplikaciju.



Slika 27. Uključivanje poslužitelja otkrivanja mikroservisa

```
<dependency>
   <groupId>org.springframework.cloud</groupId>
   <artifactId>spring-cloud-starter-netflix-eureka-server</artifactId>
</dependency>
```

Kod 59. Maven kod za dodavanje Eureka poslužitelja u projekt

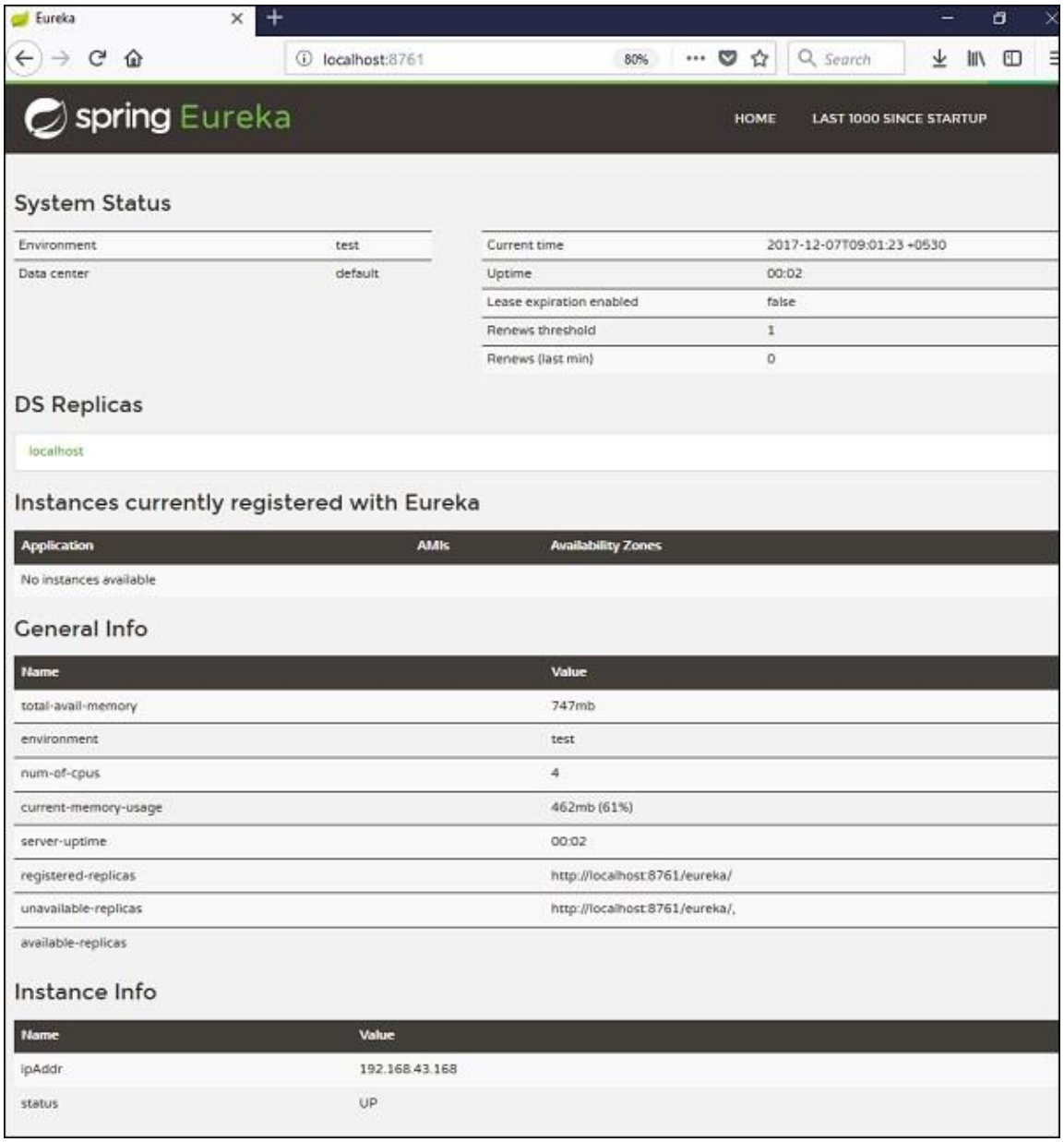
Da bi se na nekom poslužitelju mogli izvršavati mikroservisi, potrebno je postaviti aplikaciju koja će se izvršavati paralelno s mikroservisima i koja će obavljati ulogu Eureka poslužitelja. To je u biti standardna Spring Boot aplikacija, samo ima svojstvo poslužitelja. U samoj aplikaciji potrebno je odrediti na kojem portu će biti aktivan Eureka server. Te postavke se definiraju u ***application.properties*** datoteci kako je prikazano u primjeru u kodu 60.

```
server.port = 8761
eureka.client.registerWithEureka = false
eureka.client.fetchRegistry = false
```

Kod 60. Definiranje porta za pristup Eureka poslužitelju

U toj datoteci potrebno je obavezno dodati dvije naredbe koje označavaju da se radi o aplikaciji koja je poslužitelj, a ne klijent (*engl. client*). To je potrebno jer Eureka poslužitelj može biti i klijent. Oznaka klijenta stavlja se u samoj aplikaciji mikroservisa. Kada se pokrene Eureka poslužitelj za lokalno

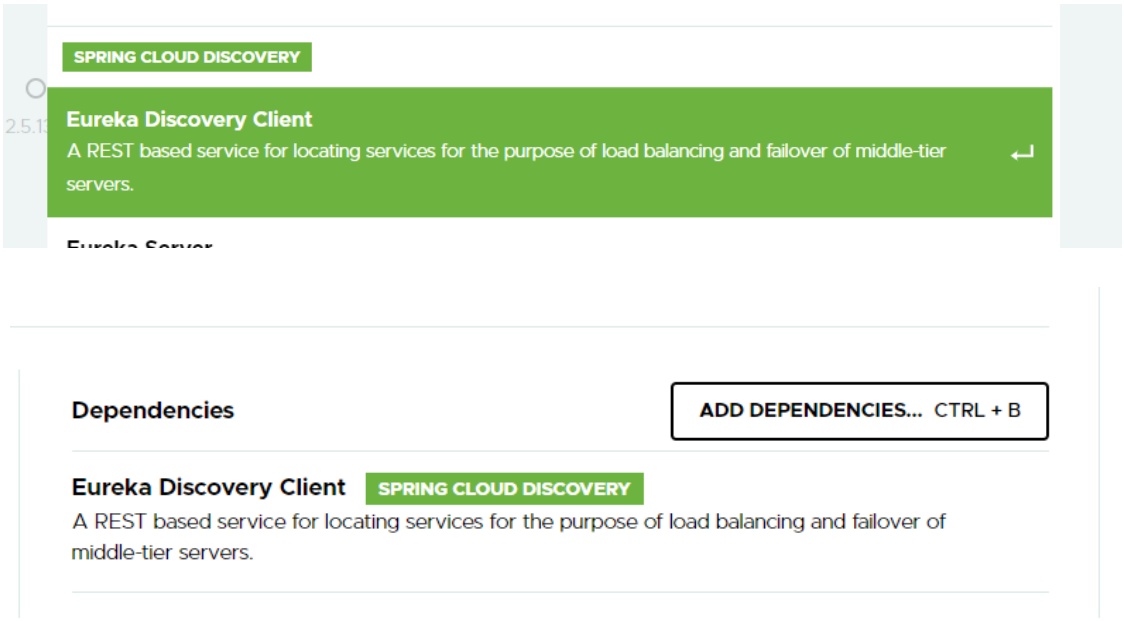
testiranje na zaslonu web preglednika prikazuje se početna stranica poslužitelja kako je prikazano na slici 28.



Slika 28. Eureka poslužitelj početna web stranica

Na slici je vidljivo da je poslužitelj pokrenut na ***portu 8761*** kako je definirano u datoteci svojstava. Na stranici se nalaze informacije o samom poslužitelju. Kada je pokrenut i koliko vremena radi te koliko se aplikacija, mikroservisa trenutno izvodi na njemu. U ovom primjeru trenutno nema niti jednog pokrenutog mikroservisa što se vidi u ***tablici Application***.

U odnosu na Eureka poslužitelja, Eureka klijent (*engl. Eureka client*) predstavljaju svi mikroservisi koji će se pokretati na Eureka poslužitelju. Iz tog razloga u Spring Boot mikroservisu dodaje se ovisnost kako je prikazano na slici 29.



Slika 29. Definiranje Eureka klijent ovisnost u projektu

Za razliku od aplikacije poslužitelja, klijent aplikacija koristi nešto drugačiju anotaciju. Na početku aplikacije stavlja se ***@EnableDiscoveryClient*** anotacija koja definira aplikaciju kao mikroservis. Također, u samoj klasi aplikacije definira se bean objekt servisa koji se poziva u mikroservisu.

```
package hr.spring.fina;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.cloud.config.server.EnableConfigServer;
import org.springframework.cloud.netflix.eureka.server.EnableEurekaServer;

@EnableDiscoveryClient
@SpringBootApplication
public class FinaApplication {

   public static void main(String[] args) {
        SpringApplication.run(FinaApplication.class, args);
   }

   @LoadBalanced    // obavezno koristiti anotaciju
   @Bean
   RestTemplate restTemplate() {
      return new RestTemplate();
   }

}
```

Kod 61. Definiranje bean objekta u mikroservisu

Treba napomenuti da je dobra praksa da mikroservis uvijek koristi jedan servis. Za pristupanje REST servisu potrebno je u mikroservisu dodati REST predložak , ***TestTemplete***. Kreiranjem bean objekta aplikaciji daje sva svojstva potrebnih za rad jednog mikroservisa. U kodu 61. prikazano je definiranje bean objekt RestTemplate. U samom REST API servisu potrebno je uključiti RestTemplate bean objekt

kako je navedeno u primjeru koda 62. Na početku klase definira se ***RestTemplete*** pomoću ***@Autowired*** anotacije. Ostale metode koriste se kao u standardnoj web Spring Boot aplikacije.

```
package hr.spring.fina;

@RestController
public class HelloResource {

   @Autowired
   private RestTemplate restTemplate;

   @RequestMapping(
         value = "/rest/student/popis",
         method = RequestMethod.GET,
         produces = MediaType.APPLICATION_JSON_VALUE
   )
   public ResponseEntity<String> popisGet() {
      .. programske naredbe
   }
...

}
```

Kod 62. REST API za mikroservis

Za svaki mikroservis moguće je odrediti njegov naziv i port na kojem će se mikroservis izvoditi. To se zapisuje u standardnu ***application.properties*** datoteku aplikacije. Primjer deklaracije prikazan je u kodu 63. U kodu na početku je naveden port rada mikroservisa te njegovo ime koje će se prikazati na Eureka poslužitelju. Nakon toga definirana je HTTP putanja za pristup REST servisu, a nakon toga podaci vezani za povezivanje s MySQL bazom.

```
server.port=8380
spring.application.name=studenti-server
server.servlet.context-path=/stuenti-api

spring.datasource.driver-class-name=com.mysql.jdbc.Driver
spring.datasource.url=jdbc:mysql://localhost:3306/studenti?useSSL=false&createDatabaseIfNotExist=true
spring.datasource.username=[username]
spring.datasource.password=[password]

spring.jpa.hibernate.ddl-auto=update
spring.jpa.database-platform=org.hibernate.dialect.MySQL57Dialect
spring.jpa.generate-ddl=true
spring.jpa.show-sql=true
```

Kod 63. Primjer konfiguracijske datoteke mikroservisa

12. Poglavlje: Sigurnost web aplikacije

U ovom poglavlju naučit ćete:

- ✓ Zbog čega je važna sigurnost
- ✓ Na koji način uključiti sigurnosnu ovisnost
- ✓ Definiranje pristupnih rola korisnika
- ✓ Implementacija klasa potrebnih za autorizaciju korisnika

12. Sigurnost web aplikacije

12.1. Osnovni pojmovi

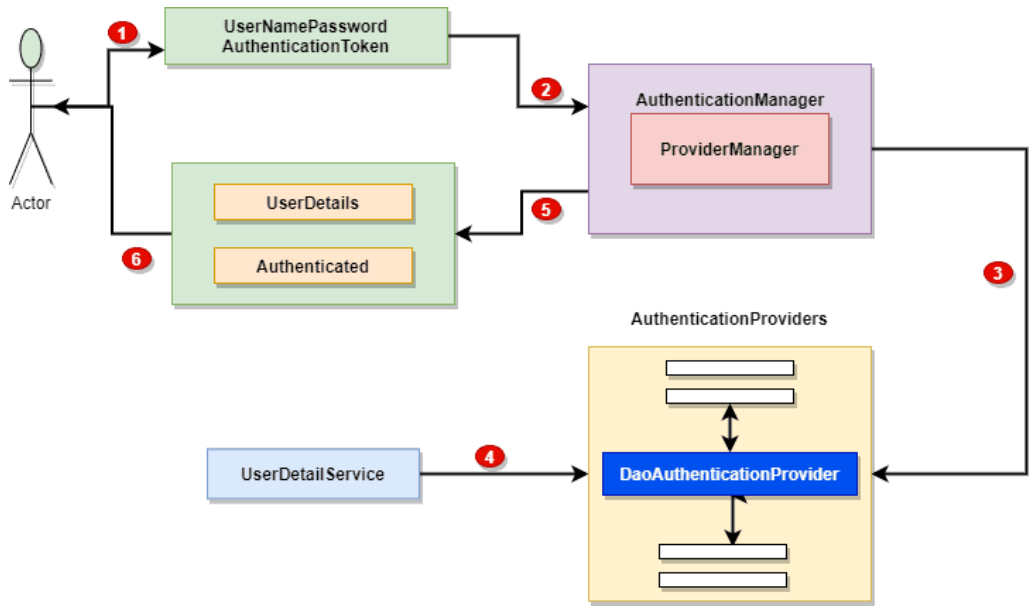
Razvojem interneta i raznih usluga koje se nude putem njega, pojavio se problem sigurnosti. Sigurnost vezana za zaštitu podataka, za siguran transfer podataka, sigurnost prilikom financijskih transakcija pa do sigurnosti u pristupu pojedinim dijelovima aplikacija. Svjedoci smo svakodnevnih napada na web stranice i aplikacije raznih poduzeća i pružatelja internetskih usluga. Zbog toga se nametnula potreba za ugradnju raznih sigurnosnih mehanizama u web i oblak aplikacija.

Razina sigurnosti i njena kompleksnost ovisi o vrijednosti podataka i sadržaja koji se žele zaštititi. Spring Boot također nudi sigurnosni sustav i arhitekturu kojom je moguće zaštititi aplikacije. Kada se promatra pojam sigurnosti potrebno je objasniti nekoliko pojmova koji su direktno vezani uz pojam sigurnosti.

Autentifikacija (*engl. Authentication*) korisnika je proces provjere autentičnosti korisnika, što znači da aplikacija provjerava da li je korisnik onaj za koga se predstavlja ili ne. To se uobičajeno provodi provjerom korisničkog imena i lozinke. Autentifikacija se provodi pomoću pristupne forme u koju korisnik upisuje svoje pristupne podatke. Podaci se šalju u kontroler koji pomoću određenog algoritma provjerava da li u sustavu aplikacije postoji korisnik s tim pristupnim podacima.

U slučaju da ne postoji korisnik aplikacije s tim podacima, aplikacija ponavlja upis podatka ili može poduzeti određene sigurnosne radnje kao što je blokiranje pristupa tom dijelu aplikacije, blokiranje korisnika s tom IP adresom ili neku sličnu aktivnost. Međutim, ukoliko su pristupni podaci ispravni, aplikacija propušta korisnika da nastavi s njenim korištenjem.

Na slici 30. prikazan je proces autentifikacije korisnika.



Slika 30. Proces autentifikacije korisnika

Na slici 30. u točki 1, korisnik upisuje svoje pristupne podatke. Ti se podaci šalju u kontroler koji predstavlja autentifikacijski menadžer koji je zadužen za provedbu cijelog procesa autentifikacije. Menadžer, u točki 3 šalje upit u bazu gdje se nalaze podaci o korisnicima. Na temelju upita, menadžer dobiva podatke o korisniku, točka 4. Na temelju dobivenih podataka u točki 5, menadžer odlučuje o sljedećoj akciji i o tome obavještava korisnika (*engl. Actor*).

Nakon uspješne autentifikacije dolazi drugi sigurnosni korak, a to je autorizacija (*engl. Authorization*). ***Autorizacija*** je proces u kojem aplikacija utvrđuje u koje dijelove aplikacija korisnik smije pristupiti, a u koje mu je zabranjen pristup. Provjerava se „rola“ korisnika. Rola određuje što korisnik može raditi u aplikaciji, a što ne smije. Ukoliko korisnik ima rolu „Administratora“ ima neograničena ovlaštenja. U slučaju da ima neku drugu rolu u aplikaciji, mogu mu biti uskraćene određene aktivnosti koje ima administrator aplikacije. Na osnovi role korisnika, aplikacija filtrira korisnika te mu određuje na koje web stranice može pristupiti.

Prednosti korištenja Spring Boot sigurnosnog mehanizma:

- Servlet API integracija,
- proširiva podrška za autentifikaciju i autorizaciju,
- zaštita od napada kao što su fiksiranje sesije, ukidanje klikova,
- podrška za Spring Boot MVC integracija,
- sposobnost zaštite aplikacije od napada grube sile,
- prenosivost,
- zaštita od CSRF (*engl. Cross Site Request Forgery*) napada,
- podrška za Java konfiguraciju.

12.2. Konfiguriranje sigurnosnih mehanizama

Za korištenje sigurnosnih mehanizama u Spring Boot aplikaciji potrebno je u aplikaciju ugraditi sigurnosnu ovisnost, potrebne biblioteke. U Maven ***pom.xml*** datoteku upisuje se ovisnost koja prilagođava vašu aplikaciju u sigurnu aplikaciju. U kodu 64. prikazane su naredbe koje se upisuju u datoteku ukoliko sigurnost nije uključena u samom početku inicijalizacije projekta. Može se uočiti da su stavljene dvije ovisnosti. Prva ovisnost se odnosi na samu sigurnost, a druga ovisnost omogućava testiranje aplikacije vezano za sigurnost.

Nakon uključivanja ovisnosti i nadogradnje aplikacije, potrebno je izvršiti konfiguriranje sigurnosti u samoj aplikaciji. Potrebno je odrediti način provođenja autentifikacije i odrediti korisničke role u aplikaciji. Postoji nekoliko strategija konfiguriranja sigurnosti u samoj aplikaciji, a odabir pristupa ovisi o karakteru aplikacije, vrti korisnika te do koje razine sigurnosti postoji potreba kontrole.

Može se koristiti posebna klasa za konfiguriranje sigurnosni koja je izvedena iz ***WebSecurityConfigurerAdapter*** klase. Može se konfiguracija sigurnosti navesti u samoj glavnoj klasi

aplikacije što je i najjednostavniji pristup. Međutim, u svim pristupima potrebno je u aplikaciji staviti ***@EnableWebSecurity*** anotaciju, koja osigurava ispravnost definiranja sigurnosne strukture i definiranje svih potrebnih klasa. Kod 65. prikazuje primjer sigurnosnog adaptira.

```
<dependency>
 <groupId>org.springframework.boot</groupId>
 <artifactId>spring-boot-starter-security</artifactId>
</dependency>
<dependency>
 <groupId>org.springframework.security</groupId>
 <artifactId>spring-security-test</artifactId>
 <scope>test</scope>
</dependency>
```

Kod 64. Sigurnosna ovisnost u Maven datoteci projekta

Na početku klase stavlja se anotacija koja definira klasu kao konfiguracijsku komponentu, a druga anotacija da se omogućuje implementacija sigurnosti u aplikaciji. U klasi je potrebno obavezno implementirati metodu ***configure*** gdje se navode razine autorizacije te kako će aplikacija reagirati ukoliko je autentifikacija ispravna ili ne.

```
@Configuration
@EnableWebSecurity
public class SecurityJavaConfig extends WebSecurityConfigurerAdapter {

 @Override
 protected void configure(HttpSecurity http) throws Exception {
          http.authorizeRequests()
                    .antMatchers("/homePage").access("hasRole('ROLE_USER')")
                    .and()
                    .formLogin().loginPage("/loginPage")
                    .defaultSuccessUrl("/homePage")
                    .failureUrl("/loginPage?error")
                    .usernameParameter("username").passwordParameter("password")
                              .and()
                    .logout().logoutSuccessUrl("/loginPage?logout");

  }

  @Autowired
  public void configureGlobal(AuthenticationManagerBuilder auth) throws Exception {
     auth
          .inMemoryAuthentication()
          .withUser("user").password("{noop}pass").roles("USER");
  } // ...

}
```

Kod 65. Sigurnosni adapter aplikacije

U ovom primjeru koda, u metodi configure određuje se sigurnost u nekoliko instrukcija. Instrukcija ***antMatchers*** se koristi za konfiguriranje URL staza s kojih bi sigurnost aplikacije Spring Boot trebala dopustiti zahtjeve na temelju korisničke role. Metoda ***antmatchers()*** je preopterećena, metoda koja prima i metode HTTP zahtjeva i specifične URL-ove kao svoje argumente. Određuje se do koje razine

domene određena rola korisnika ima pravo pristupa. Na ovu instrukciju nadodaju se metode koje točno specificiraju pristup korisnika, tako da možemo koristiti sljedeće metode:

- ***hasAnyRole()*** metoda omogućuje bilo kojim korisnikom čija je uloga uključena u konfigurirane uloge stvorene u aplikaciji,
- ***hasRole()*** metoda određuje pojedinačnu rolu korisnika za određenu skupinu URL poddomena,
- hasAuthority() metoda omogućuje pristup svakom korisniku kojemu su dodijeljena određena ovlaštenja,
- ***hasAnyAuthority()*** metoda omogućuje pristup korisnicima kojima su dodijeljena autorizacije u konfiguracijskom dijelu aplikacije,
- ***anonymous()*** metoda omogućuje pristup ne autoriziranim korisnicima, pristup je javni bez uključenih sigurnosnih mjera. Ta metoda koristi se za dijelove aplikacije gdje su web stranice javne i ne zahtijevaju od korisnika prijavu,
- ***authenticated()*** metoda omogućuje pristup bilo kojem provjerenom korisniku.

U primjeru konfiguracije instrukcijom ***formLogin()*** određuje se web stranica gdje se nalazi prijavna forma za pristup aplikaciji. Također, ukoliko je prijava korisnika uspješno obavljena, instrukcijom ***defaultSuccessUrl()*** određena je početna stranica web aplikacije koja će se prikazati korisniku nakon uspješne prijave. U slučaju pogrešne prijave, instrukcijom ***failureUrl()*** prikazuje se stranica ***loginPage?error*** gdje error vrijednost označava vrstu pogreške. Na kraju, instrukcijom ***logout()*** određena je web stranica koja će se prikazati korisniku nakon njegove odjave iz aplikacije.

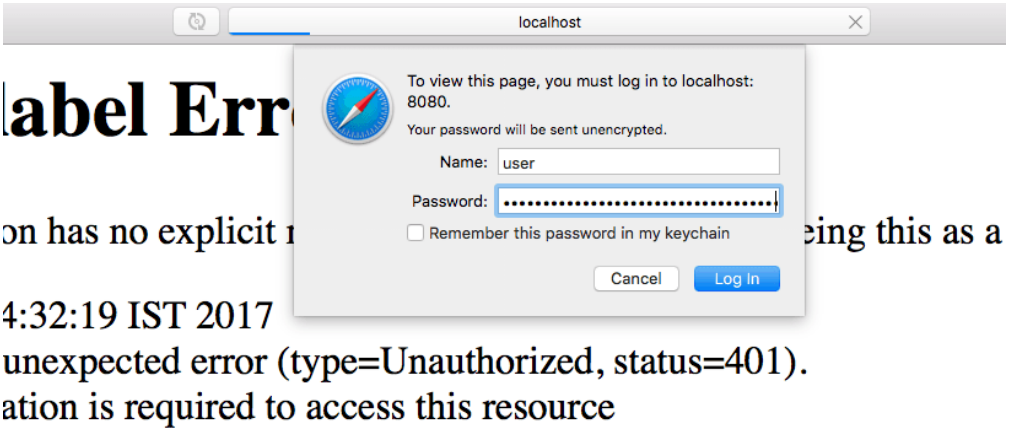
U metodi ***configureGlobal*** definirana je globalna rola korisnika i pristupni podaci. Ovaj način se koristi u slučaju niže razine sigurnosti te kada korisnici imaju samo jednu rolu i svi korisnici imaju iste pristupne podatke.

12.3. Implementacija sigurne web aplikacije

U ovom dijelu poglavlja detaljnije će se prikazati klasa za implementaciju sigurnosti. Izgled jednostavne web stranice s prijavnom formom. Ukoliko se uključi sigurnost web aplikacije, a ne implementiramo vlastitu prijavnu formu i web stranicu, Spring Boot će korisniku prikazati inicijalnu prijavnu formu kako je prikazano na slici 31. Kako je to prijavna forma koja je na engleskom jeziku i nema neki poseban dizajn, najbolje je da sami formiramo svoju prijavnu stranicu i cijeli proces autentifikacije korisnika.

Temelj cijele procedure provjere korisnika nalazi se u klasi koja je izvedena iz ***WebSecurityConfigurerAdapter*** klase. Ukoliko ne želimo koristiti sigurnosnu konfiguraciju koja je zadana, Spring Boot okruženjem potrebno je da sami kreiramo konfiguracijsku klasu. Isključenje auto konfiguracije sigurnosti može se postići anotacijom kako je prikazano u kodu 66. Naredba exclude definira koja klasa će biti isključena prilikom prevođenja aplikacije. To isto možemo postići ukoliko u konfiguracijsku datoteku application.properties dodamo sljedeću naredbu:

spring.autoconfigure.exclude=org.springframework.boot.autoconfigure.security.SecurityAutoConfiguration



Slika 31. Prijava korisnika u Spring Boot aplikaciju

```
@SpringBootApplication(exclude = { SecurityAutoConfiguration.class })
public class SpringBootSecurityApplication {

   public static void main(String[] args) {
      SpringApplication.run(SpringBootSecurityApplication.class, args);
   }

   @Bean
   public PasswordEncoder passwordEncoder(){
      return new BCryptPasswordEncoder(15);
   }
}
```

Kod 66. Isključivanje auto konfiguracije sigurnosti aplikacije

Ukoliko smo u našoj aplikaciji onemogućili automatsku konfiguraciju, potrebno je da kreiramo vlastitu. U konfiguracijsku datoteku može se dodati naredba ***spring.security.user.password=password*** gdje je lozinka upravo vrijednost koja je upisana za atribute password. Radi sigurnosti, često su lozinke korisnika kodirane raznim algoritmima. Tako u ovom primjeru, kreiran je bean objekt koji vrši kriptiranje lozinke čime se povećava sigurnost same aplikacije.

U kodu 67. prikazan je adapter za definiranje vlastite sigurnosne strukture. U metodi configure određene su tri role korisnika: INTER, ADMIN i SUPERVISOR i svakoj od njih je dodijeljena putanja pristupa. Ukoliko želimo, možemo specificirati i HTTP zahtjeva koji pojedina rola može pokrenuti. Tako je stavljeno da DELETE, brisanje može izvršiti korisnik koji ima rolu ADMIN. Iza te metode nalazi se metoda koja kreira korisničke podatke. U praksi se podaci o korisniku preuzimaju iz baze podataka na temelju pristupnih podataka. U ovom primjeru stavljeno je samo radi informacije da ta mogućnost postoji.

```
package hr.spring.fina.security;
```

```
@Configuration
@EnableWebSecurity
public class AppSecurityConfig extends WebSecurityConfigurerAdapter {

   @Autowired
   private final PasswordEncoder passwordEncoder;

   public AppSecurityConfig(PasswordEncoder passwordEncoder) {
      this.passwordEncoder = passwordEncoder;
   }

   @Override
   protected void configure(HttpSecurity http) throws Exception {
      http.sessionManagement().sessionCreationPolicy(SessionCreationPolicy.STATELESS);
      http
            .csrf().disable()
            .authorizeRequests()
            .antMatchers(HttpMethod.DELETE, "/api/v1/products/{productId}").hasRole(ADMIN.name())
            .antMatchers(HttpMethod.PUT, "/api/v1/products/{productId}").hasRole(ADMIN.name())
            .antMatchers("/api/v1/products/add").hasAnyRole(ADMIN.name(), SUPERVISOR.name())
            .antMatchers("/api/v1/products").hasAnyRole(ADMIN.name(), SUPERVISOR.name(),
INTERN.name())
            .anyRequest()
            .authenticated()
            .and()
            .httpBasic();
   }

   @Bean
   @Override
   protected UserDetailsService userDetailsService() {
      UserDetails ana = User.builder()
            .username("ana")
            .password(passwordEncoder.encode("password"))
            .roles(INTERN.name())
            .build();

      UserDetails tina = User.builder()
            .username("tina")
            .password(passwordEncoder.encode("password"))
            .roles(SUPERVISOR.name())
            .build();

      UserDetails bruno = User.builder()
            .username("bruno")
            .password(passwordEncoder.encode("password"))
            .roles(ADMIN.name())
            .build();
      InMemoryUserDetailsManager userDetailsManager = new InMemoryUserDetailsManager(ana,
tina,
      bruno);

      return userDetailsManager;
   }
}
```

Kod 67. Klasa za konfiguriranje sigurnosti web aplikacije

U konfiguracijskoj metodi moguće je detaljno definirati svaku rolu korisnika pojedinačno. U primjeru 68. prikazan je taj slučaj konfiguracije.

```
@Configuration
```

```
@EnableWebSecurity
public class SecurityConfig extends WebSecurityConfigurerAdapter {
 // Autentifikacija : Korisnik --> Roles
 protected void configure(AuthenticationManagerBuilder auth)throws Exception {
        auth.inMemoryAuthentication().passwordEncoder
        (org.springframework.security.crypto.password.NoOpPasswordEncoder.getInstance())
        .withUser("user1").password("secret1")
        .roles("USER").and().withUser("admin1").password("secret1")
        .roles("USER", "ADMIN");
 }

 // Autorizacija : Role -> Access
 protected void configure(HttpSecurity http) throws Exception {
        http.httpBasic().and().authorizeRequests().antMatchers("/students/**")
                .hasRole("USER").antMatchers("/**").hasRole("ADMIN").and()
                .csrf().disable().headers().frameOptions().disable();
        }
}
```

Kod 68. Konfiguriranje autentifikacije i autorizacije korisnika

Tijekom prijave korisnika, potrebno je predvidjeti mehanizam koji će reagirati ukoliko prijava nije ispravna. U slučaju da korisnik pokuša pristupiti zaštićenom dijelu aplikacije aktivirat će se stranica koja će informirati korisnika o nedozvoljenoj aktivnosti kako je prikazano u kodu 65. Međutim, moguće je kreirati posebnu klasu za upravljanje iznimkama ukoliko su pristupni podaci pogrešni. U kodu 69. prikazana je klasa za upravljanje iznimkama. U samoj klasi formira se poruka s pripadajućim kodom pogreške.

```
@Component
public class RestAccessDeniedHandler implements AccessDeniedHandler {

   @Override
   public void handle(HttpServletRequest httpServletRequest, HttpServletResponse
httpServletResponse, AccessDeniedException e) throws IOException, ServletException {
      ApiResponse response = new ApiResponse(403, "Nedozvoljeni pristup");
      response.setMessage("Nedozvoljeni pristup ");
      OutputStream out = httpServletResponse.getOutputStream();
      ObjectMapper mapper = new ObjectMapper();
      mapper.writeValue(out, response);
      out.flush();
   }
}
```

Kod 69. Klasa za upravljanje iznimkama sigurnosnog pristupa

13. Poglavlje: Testiranje web aplikacije

U ovom poglavlju naučit ćete:

- ✓ Osnovne metode testiranja
- ✓ Pokretanje testiranja u razvojnoj okolini
- ✓ Tumačenje rezultat testiranja
- ✓ Način kreiranja testnih metoda

13. Testiranje web aplikacije

13.1. Osnovni pojmovi

Testiranje je izrazito važna faza u razvoju aplikacije. Postoje razni načini i metode testiranja aplikacije. Pri tom se koriste razni alati koji na određeni način automatiziraju cijeli proces testiranja. U Spring Boot aplikacijama izrađuju se posebne testne klase. Prilikom testiranja, fokus testiranja je na manjim funkcionalnostima aplikacije. Ne testira se cijela aplikacija u jednoj metodi, već se piše veći broj testnih metoda gdje je svaka od metoda usmjerena na provjeru neke ciljane funkcionalnosti u aplikaciji. Ovi testovi su izrazito važni, posebno prilikom nadogradnje aplikacije. To je zbog toga što se bilo kakva mala izmjena u aplikaciji može negativno reflektirati na ispravnost rada pojedinih dijelova aplikacije.

Kada se napišu testne klase s više testnih metoda, pokreće se testiranje. Ukoliko se aplikacija ispravo izvršava, sve testne metode trebaju dati rezultat ***TRUE***, što označava da je dobiven očekivani rezultat u metodi. U slučaju da bilo koja metoda kao rezultat testiranja vrati rezultat ***FALSE***, aplikacija treba ići na dodatnu analizu i ne može ići u produkciju.

Kod razvoja Spring Boot projekata, testne klase i samo testiranje odvojeno je od izvornog koda aplikacije. U projektu sve testne klase nalaze se u mapi ***src/test*** unutar projekta. Kod testnih klasa stavlja se ***@SpringBootTest*** anotacija koja upravo sugerira prevoditelju da su te klase namijenjene isključivo testiranju, a ne radu same aplikacije. Ispred svake testne metode stavlja se ***@Test*** anotacija. Unutar jedne teste klase možemo imati veći broj testni metoda, a u svakoj metodi može se testirati više različitih funkcionalnosti. Na primjer, testna klasa za repozitorij i klasu student može u jednoj te istoj metodi testirati sve CRUD metode. U primjeru koda 70. prikazana je struktura standardne testne klase.

```
import org.junit.jupiter.api.Test;
import org.springframework.boot.test.context.SpringBootTest;

@SpringBootTest
class DemoApplicationTest {

  @Test
  void contextLoads() {
  }

}
```

Kod 70. Primjer strukture testne klase

Za potrebe testiranja mogu se koristiti razni alati i biblioteke. Za Spring Boot aplikaciju na određeni način nametnuli su se sljedeći alati:

- ***JUnit 5*** je najpopularnije okruženje za testiranje koje se koristi za sve aplikacije pisane Java programskim jezikom,
- ***Mockito*** je također okvir koji omogućuje testiranje svih komponenti Spring Boot aplikacije,

- ***MockMVC*** je Spring modul koji omogućuje testiranje integriteta aplikacije.

13.2. JUnit testiranje

JUnit 5 testiranje je najnovija verzija za testiranje. Za njeno uključivanje potrebno je isključiti nižu verziju JUnit 4 iz projekta koja je inicijalno ugrađena u Spring Boot projektu. JUnit 4 je dio ovisnosti ***spring-boot-starter-test*** koja se inicijalno dodaje u projekt. Zbog toga je u Maven pom.xml datoteku potrebno dodati ***junit-jupiter-engine*** ovisnost i isključiti JUnit 4. Primjer toga prikazan je u kodu 71.

```
<dependency>
 <groupId>org.springframework.boot</groupId>
 <artifactId>spring-boot-starter-test</artifactId>
 <scope>test</scope>
 <exclusions>
  <exclusion>
   <groupId>junit</groupId>
   <artifactId>junit</artifactId>
  </exclusion>
 </exclusions>
</dependency>

<dependency>
  <groupId>org.junit.jupiter</groupId>
  <artifactId>junit-jupiter-engine</artifactId>
  <version>5.8.1</version>
  <scope>test</scope>
</dependency>
```

Kod 71. Uključivanje JUnit 5 ovisnosti u projekt radi testiranja

Prilikom testiranja JUnit 5 okruženje korisnicima nudi sljedeće anotacije za razne modele testiranja:

- ***@TestFactory*** – kreiranje testnog objekta za potrebe dinamičnog testiranja,
- ***@DisplayName*** – definira se korisničko ime za testnu klasu ili metodu,
- ***@Nested*** – označava se da je testna klasa ugniježdena te da nije klasična statična klasa,
- ***@Tag*** – deklaracija taga za filtriranje testova,
- ***@ExtendWith*** – registrira se korisnička ekstenzija podataka za testiranje,
- ***@BeforeEach*** – anotacija koja deklarira metodu koja će se izvršavati prije izvršavanja bilo koje testne metode. Najčešće se u toj metodi vrši inicijalizacija podataka i slične predradnje testiranja,
- ***@AfterEach*** – anotacija kojom se deklarira metoda koja će se izvršiti nakon završetka testnih metoda,
- ***@BeforeAll*** – anotacija deklarira metodu koja će se izvršiti prije bilo koje testne metode,
- @***AfterAll*** – anotacija deklarira metodu koja će se izvršiti nakon završetka bilo koje testne metode,
- ***@Disable*** – anotacija onemogućuje određenu testnu klasu ili metodu.

U kodu 72. prikazani su primjeri upotreba prije navedenih anotacija.

```
@BeforeAll
static void setup() {
   log.info("@BeforeAll – izvršava se samo jednom prije testiranja svih metoda. ");
}

@BeforeEach
void init() {
   log.info("@BeforeEach – izvršava se prije svake testne metode u klasi gdje je anotacija
primijenjena.");
}

@DisplayName("Pojedinačo testiranje")
@Test
void testSingleSuccessTest() {
   log.info("Uspješno");
}

@Test
@Disabled("Nije još implementirana")
void testShowSomething() {
}

@AfterEach
void tearDown() {
   log.info("@AfterEach – izvršena nakon svake test metode.");
}

@AfterAll
static void done() {
   log.info("@AfterAll – izvršava se nakon završetka svih testnih metoda.");
}
```

Kod 72. Primjer korištenja testni anotacija

Prilikom testiranja koriste se posebne naredbe koje su implementirane u JUnit 5 okruženju. Jedna od njih je ***Assertions*** ili tvrdnje. Ovo je skupina naredbi kojima se ispituju tvrdnje o istinitosti određenih relacija koje mogu biti istinite ili lažne. Naredbom je moguće provesti višestruko testiranje te podržava lambda sintaksu naredbi. U kodu 73. prikazana je primjena ove naredbe.

U kodu se koristi naredba ***assertTrue*** koja provjerava jesu li određene tvrdnje istinite. Tako se u ovom primjeru ispituje vrijednosti brojeva. U daljnjem primjeru koda ispituje se poziva li se pravilno web stranica pozivom metode u kontrolera. Koristi se naredba ***assertEquals*** koja provjerava je li string kojeg metoda vraća jednak traženoj vrijednosti.

```
@Test
void lambdaExpressions() {
   List numbers = Arrays.asList(1, 2, 3);
   assertTrue(numbers.stream()
     .mapToInt(i -> i)
     .sum() > 5, () -> "Suma bi trebala biti veća od 5");
}

@Test
 void groupAssertions() {
```

```
    int[] brojevi = {0, 1, 2, 3, 4};
    assertAll("brojevi",
        () -> assertEquals(brojevi [0], 1),
        () -> assertEquals(brojevi [3], 3),
        () -> assertEquals(brojevi [4], 1)
    );
}

// test Rest servisa
@RestController
public class WebController {

   @RequestMapping("/")
   public String pocetak() {
      return "Dobar dan!";
   }
}

@Test
public void testHomeController() {
      WebController webController = new WebController();
      String result = webController.pocetak();
      assertEquals(result, "Dobar dan!");
}
```

Kod 73. Primjer korištenja Assertions JUnit naredbe

Testovi koji su podržani prikazani su u tablici 1.

Naredba	Objašnjenje
assertEquals	Provjerava da li su dva elementa jednaka
assertTrue	Provjerava da li je tvrdnja ili relacija između elemenata istinit
assertFalse	Provjerava da li je tvrdnja ili relacija između elemenata lažan
assertNotNull	Provjerava da li objekt ili podataka **NEMA** null vrijednost
assertNull	Provjerava da li objekt ili podataka **IMA** null vrijednost
assertSame	Provjerava da li dvije reference pokazuju na isti objekt u memoriji
assertNotSame	Provjerava da li dvije reference pokazuju na različiti objekt u memoriji
assertArrayEquals	Provjeravaju da li su dva polja podataka jednaka

Tablica 1. Naredbe za testiranje

13.3. Mockito testiranje

Prijašnji način testiranja je statičko testiranje jer se provjeravaju statični objekti ili rezultat neke aktivnosti. Ukoliko postoji potreba za dinamičkim testiranjem, za korištenjem ovisnog umetanja pojedinih servisa i sličnih komponenti aplikacije, tada se koristi Mockito ili MockMVC testiranje. Ukoliko izvorni kod aplikacije kojeg želimo testirati koristi ovisnost, tada se ovaj problem rješava upravo s ovim testnim okruženjem. Mockito se uglavnom koristi u kombinaciji s JUnit testnim okruženjem. U kodu 74. prikazan je primjer upotrebe Mockito testiranja. U samoj testnoj klasi ***@InjectMocks*** anotacijom se uključuje određena komponenta, kontroler ili servis koji će se koristiti za testiranje. Na početku se nalazi kod REST kontrolera koji pristupa podacima korisnika. ima jedna metoda koja iz repozitorija pristupa podatku korisnika prema traženom id ključu.

Testna metoda ***testGetUserById*** na početku kreira objekt User i dodjeljuje mu id vrijednost. Nakon toga provjerava nalazi li se u repozitoriju korisnik s tom id vrijednosti. Sljedeće se testira pristup

podacima upotrebom kontrolera i njegove get metode. Na kraju se koristi JUnit naredba i provjerava se ima li dobiveni objekt User traženu id vrijednost. Kod ovog testiranja ne testira se rad baze podataka, nego servis i repozitorij klase, njihova povezanost i ispravnost rada metoda.

```
@RestController
@RequestMapping("api/v1/")
public class UserController {

   @Autowired
   private UserRepository userRepository;

   @RequestMapping(value = "user/{id}", method = RequestMethod.GET)
   public User get(@PathVariable Long id) {
      return userRepository.findOne(id);
   }
}

// Testna klasa

public class MockitoControllerTest {

   @InjectMocks
   private UserController userController;

   @Mock
   private UserRepository userRepository;

   @Before
   public void init() {
      MockitoAnnotations.initMocks(this);
   }

   @Test
   public void testGetUserById() {
      User u = new User();
      u.setId(1l);
      when(userRepository.findOne(1l)).thenReturn(u);

      User user = userController.get(1L);

      verify(userRepository).findOne(1l);

      assertEquals(1l, user.getId().longValue());
   }
}
```

Kod 74. Primjer Mockito testne klase

13.4. MockMVC testiranje

Prijašnja dva okruženja za testiranje fokusirana su na testiranje ispravnosti rada pojedine komponente u aplikaciji. Međutim, često je potrebno testirati i krajnje točke aplikacije, do onoga čemu korisnik aplikacije može pristupiti, određenu razinu same MVC aplikacije. Uz MockMVC, Spring Boot pruža izvrstan alat za testiranje Spring Boot aplikacija.

Postavlja se pitanje što sve možemo testirati? Odgovor je sve što se odnosi na Spring Boot MVC aplikaciju. Provjeru krajnjih točaka @Controller i @RestController te sve komponente u aplikaciji. Za ovaj pristup testiranja potrebno je uključiti standardno ovisne biblioteke kako je prikazano u primjeru koda 75.

```
<dependency>
 <groupId>org.springframework.boot<groupId>
 <artifactId>spring-boot-starter-web</artifactId>
</dependency>

<dependency>
 <groupId>org.springframework.boot</groupId>
 <artifactId>spring-boot-starter-test</artifactId>
 <scope>test</scope>
</dependency>

<dependency>
 <groupId>org.springframework.security</groupId>
 <artifactId>spring-security-test</artifactId>
 <scope>test</scope>
</dependency>
```

Kod 75. Ovisnost potrebna za testiranje aplikacije pomoću MockMVC okruženjem

Kod testiranja postoje dvije mogućnosti da se kreira MockMVC instanca za testiranje. Prva je da se koristi standardni pristup upotrebom anotacije, a da se Spring Boot okolina pobrine da se pravilno kreira testna klasa. Pri tome se koristi ***@WebMvcTest*** anotacija. Drugi pristup je da se izradi posebna testna klasa i kojoj će se kreirati MockMVC instanca. U sljedećem primjeru prikazano je testiranje pomoću anotacije. Korišteni su podaci o Studentu čiji su podatkovni model, repozitorij i servis prezentirani u prijašnjim primjerima koda.

Na početku testne klase stavlja se ***@WebMvcTest*** anotacija gdje se navodi koji kontroler se testira. U primjeru je naveden StudentController kontroler. Nakon toga kreira se MockMVC instanca te MockBean instanca za servis koji će se koristiti. Prva testna metoda provjerava metodu getAllStudents u kontroleru koja vraća popis svih studenta. Nakon poziva servisa u metodi se provjeravaju podaci o studentu koji je upisan kao testni podatak. Drugi test odnosi se na testiranje situacije da u bazi nema traženog studenta. U metodi se aktivira iznimka s pripadajućom porukom. Treće testiranje je fokusirano na prikaz web stranice i korištenje podatkovnog modela u Thymeleaf skriptnom jeziku. U web kontroleru se dodaje podatkovni model s nazivom atributa Student. U toj metodi provjerava se također, je li korišten pravi predložak te je li HTTP zahtjev ispravno obrađen od strane kontrolera.

```
@WebMvcTest(StudentController.class)
class StudentControllerTest {

 @Autowired
 private MockMvc mockMvc;

 @MockBean
 private StudentService studentService;

 @Test
 public void getAllStudenti() throws Exception {
  when(studentService. getAllStudents())
    .thenReturn(List.of(new Student("Ana", "Novak")));

  this.mockMvc
    .perform(MockMvcRequestBuilders.get("/api/studenti"))
    .andExpect(MockMvcResultMatchers.status().isOk())
    .andExpect(MockMvcResultMatchers.jsonPath("$.size()").value(1))
    .andExpect(MockMvcResultMatchers.jsonPath("$[0].ime").value("Ana"))
    .andExpect(MockMvcResultMatchers.jsonPath("$[0].prezime").value("Novak"));
 }

 @Test
 public void shouldReturn404WhenStudentIsNotFound() throws Exception {
  when(studentService. getStudentImePrezime("Ana", "Novak"))
    .thenThrow(new StudentNotFoundException("Ana nije pronađena"));

  this.mockMvc
    .perform(get("/api/student/Ana/Novak"))
    .andExpect(status().isNotFound());
 }

@Test
 public void testThymeleaf() throws Exception {
  when(studentService. getStudentImePrezime()).thenReturn(new Student("Ana", "Novak"));

  this.mockMvc
    .perform(get("/popisStudenata"))
    .andExpect(status().isOk())
    .andExpect(view().name("popisStudenata"))
    .andExpect(model().attribute("student", "Ana"));
 }
 ...
}
```

Kod 76. Testna metoda upotrebom MockMVC alatom

14. Poglavlje: Postavljanje web aplikacije u oblak okruženje

U ovom poglavlju naučit ćete:

- ✓ Osnovni principi postavljanja aplikacije u oblak okruženje
- ✓ Način upravljanja produkcijama aplikacije
- ✓ Način kreiranja kontejnera aplikacija i njihova produkcija

14. Postavljanje web aplikacije u oblak okruženje

14.1. Osnovni principi

Da bi se Spring Boot web aplikacija mogla javno koristiti potrebno je aplikaciju postaviti na javni oblak poslužitelj. Proces objavljivanja aplikacije (*engl. Deployment*) naziva se produkcija aplikacije. Svaka produkcija definirana je verzijom aplikacije, datumom objave te kratkim opisom što je izmijenjeno u novoj verziji aplikacije. Za produkciju aplikacije potrebno je aplikaciju pravilno pripremiti da se može izvoditi u okolini oblak poslužitelja.

Za objavu produkcije koriste se dva osnovna pristupa:

- objava produkcije aplikacije na aplikacijskom poslužitelju koji je postavljen na oblak poslužitelju,
- objava produkcije aplikacije putem kontejnerizacije i pakiranja aplikacije u strukturu kontejnera.

Kod prvog pristupa za pokretanje aplikacije koristi se aplikacijski poslužitelj. Uglavnom su aplikacijski poslužitelji povezani s programskim jezikom i tehnologijom koja je korištena u razvoju oblak aplikacije. Tako da odabir tehnologije na neki način nameće odabir aplikacijskog poslužitelja. Na tržištu postoji niz aplikacijskih poslužitelja kao što su WebSphere, WebLogic, JBoss i Apache Tomcat. Određeni poslužitelji su besplatni, a neki se plaćaju. Za Spring Boot aplikacije uglavnom se koristi Apache Tomcat aplikacijski poslužitelj otvorenog koda.

Drugi pristup pokretanja aplikacije, izrađuje se datoteka koju nazivamo kontejner. Kontejner sadrži web aplikaciju, sve njene resurse te cijeli operativni sustav potreban za pokretanje aplikacije. Svi sadržaji su zapakirani u strukturu koja omogućuje virtualno izvođenje aplikacije na oblak poslužitelju. Za ovakav pristup, potrebno je pripremiti oblak poslužitelja na poseban način. Cijeli proces izrade kontejnera izvodi se posebnim alatima. Ovaj pristup postao je vrlo popularan jer pojednostavljuje prenosivost aplikacije na širok spektar različitih oblak poslužitelja. Također, oslobađa razvojnog inženjera poznavanja strukture i načina rada oblak poslužitelja pa se može fokusirati na samu funkcionalnost aplikacije.

14.2. Aplikacijski poslužitelj – produkcije aplikacije

Za potrebe ovog kolegija koristit će se Apache Tomcat aplikacijski poslužitelj. U strukturu aplikacije treba staviti ovisnost kao na primjeru koda 77. Za izvršavanje aplikacije na aplikacijskom serveru, potrebno je aplikaciju i sve potrebne resurse zapakirati u datoteku prihvatljivog formata. Za Spring Boot aplikacije to je format datoteke JAR ili WAR. Preporučuje se da to bude WAR format. Svaki IDE alat nudi mogućnost generiranja takvog formata datoteke. Ovaj format također se navodi u pom.xml datoteci projekta kako je prikazano u kodu 77. gdje se koristi tag ***packaging***.

```
<packaging>war</packaging>

<parent>
   <groupId>org.springframework.boot</groupId>
   <artifactId>spring-boot-starter-parent</artifactId>
   <version>2.4.0</version>
   <relativePath/>
</parent>
<dependencies>
   <dependency>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-starter-web</artifactId>
   </dependency>

 <dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-tomcat</artifactId>
  <scope>provided</scope>
 </dependency>

</dependencies>

<build>
 <finalName>ROOT</finalName>
   <plugins>
         <plugin>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-maven-plugin</artifactId>
         </plugin>
   </plugins>
 </build>
```

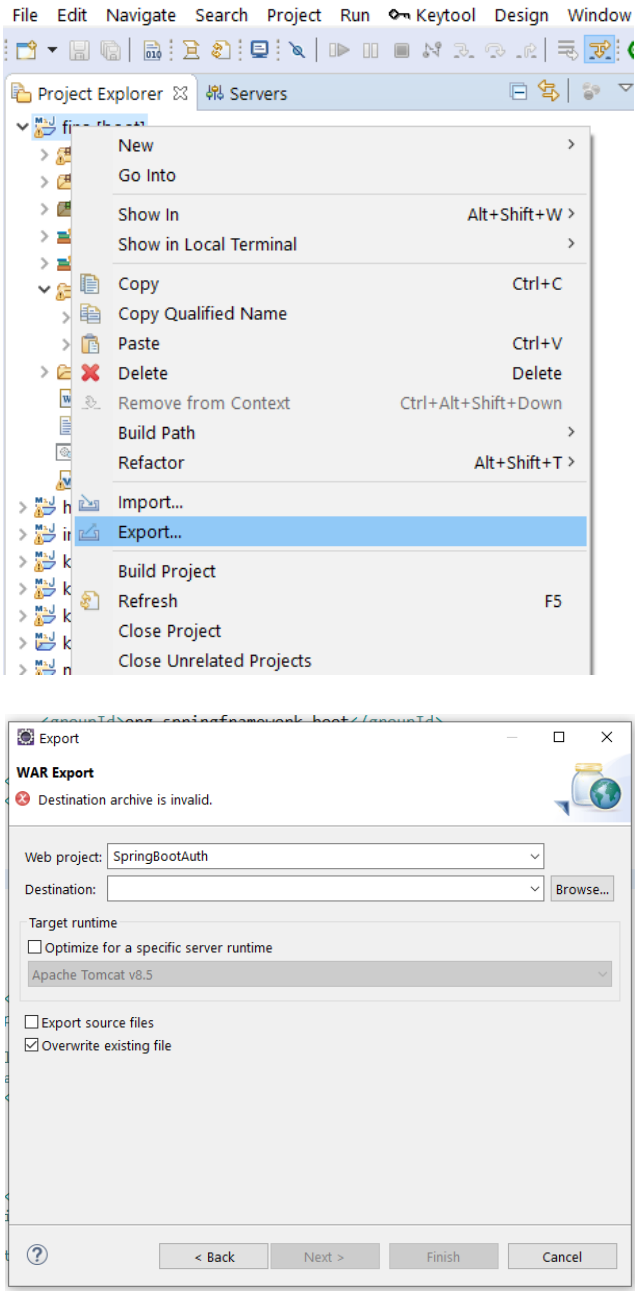
Kod 77. Primjer pom.xml datoteke za oblak aplikaciju

Također, potrebno je dodati ovisnost upravo za Tomcat aplikacijski poslužitelj. Osim toga, u datoteci se stavlja naziv zapakirane datoteke i aplikacije koja će biti postavljena na poslužitelj u oblaku. U ovom je primjeru stavljeno da će finalni naziv biti ROOT što znači da aplikacija neće imati svoje ime, već će se prilagoditi nazivu internet domene koja će se koristiti na internetu. Ukoliko na aplikacijskom poslužitelju imamo objavljeno više različitih aplikacija, tada svaka aplikacija ima svoje jedinstveno ime po kojem će se pozivati oblak aplikacija.

Kada je aplikacija spremna za produkciju, u IDE aplikaciji treba odabrati opciju za generiranje WAR datoteke. U Eclipse IDE odabire se opcija za izvoz (*engl. Export*) aplikacije i projekta. Na slici 32. prikazan je odabir navedene opcije. Nakon odabira, potrebno je upisati podatke buduće generirane datoteke. Pri tome treba pripaziti na naziv generirane datoteke jer će naziv definirati domenu pristupa aplikaciji u oblak okruženju.

Kada je WAR datoteka generirana, potrebno je pristupiti aplikacijskom poslužitelju koji se nalazi na oblak platformi. Tomcat poslužitelj ima svoj vlastitu kontrolnu ploču (***engl. Dashboard***) koja je namijenjena za upravljanje aplikacijama. Za pristup kontrolnoj ploči potrebno je imati

administratorske pristupne podatke, a ploči se pristupa u poddomeni ***manager***. Izgled kontrolne ploče prikazan je na slici 33.

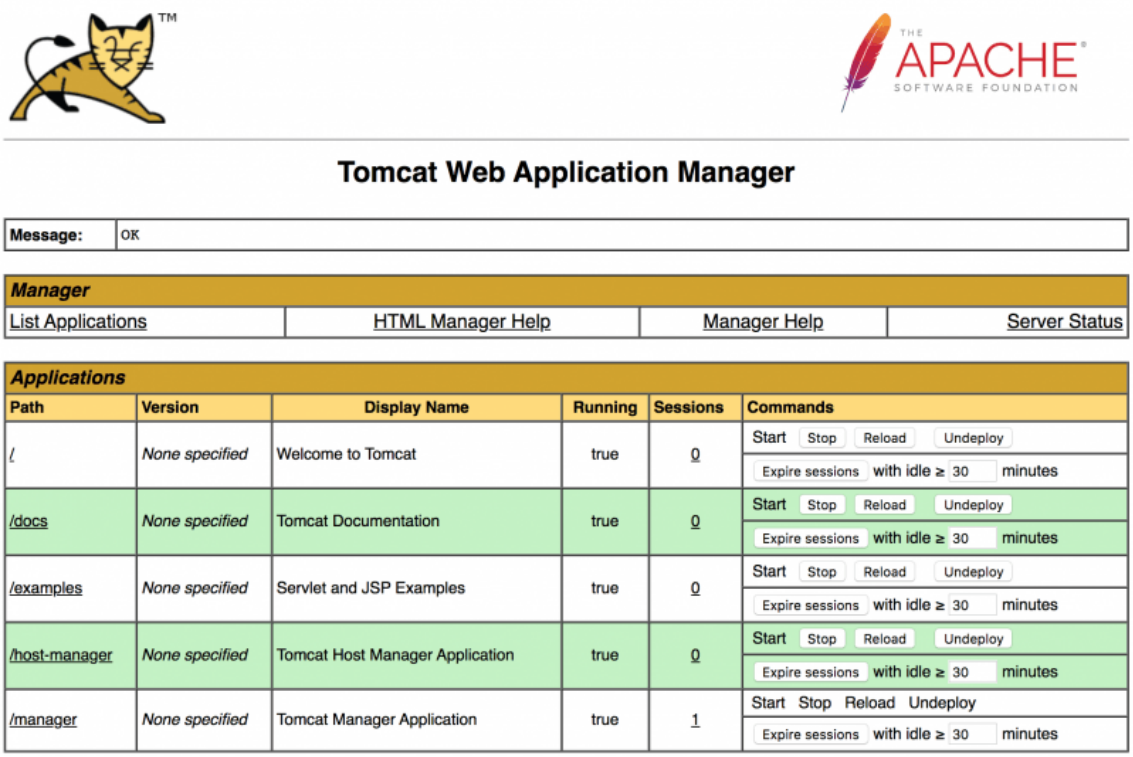


Slika 32. Izvoz aplikacije u WAR format

Application Manager upravlja i uređuje aplikacije na ovom poslužitelju. Centralni dio kontrolne ploče je tablica s popisom aplikacija koje su postavljane na poslužitelj. Tablica ima sljedeće kolone:

- ***Path*** kolona prikazuje putanje ili internetsku domenu pristupa aplikaciji,
- ***Version*** kolona prikazuje informacije o verziji aplikacije. Na slici 33., za sve aplikacije prikazana je informacija da verzija nije specificirana. Ukoliko želimo da naša aplikacija ima oznaku verzije, tu informaciju je potrebno upisati prilikom generiranja WAR datoteke,
- ***Display Name*** kolona prikazuje nazive aplikacija koje su stavljene na poslužitelj. Svaka aplikacija treba imati jedinstveno ime,

- ***Running*** kolona prikazuje izvodi li se pojedina aplikacija ili ne. Ukoliko smo pokrenuli određenu aplikaciju, u koloni za tu aplikaciju pojavljuje se vrijednost TRUE,
- ***Sessions*** kolona nam daje informacije koristi li trenutno netko aplikaciju. Broj označava broj korisnika instanci aplikacije,
- ***Commands*** kolona namijenjena je za upravljanje s aplikacijama. U koloni se nalaze četiri gumba kojima se može promijeniti status aplikacije.



Tomcat Web Application Manager

Message:	OK

Manager			
List Applications	HTML Manager Help	Manager Help	Server Status

Applications					
Path	Version	Display Name	Running	Sessions	Commands
/	None specified	Welcome to Tomcat	true	0	Start Stop Reload Undeploy Expire sessions with idle ≥ 30 minutes
/docs	None specified	Tomcat Documentation	true	0	Start Stop Reload Undeploy Expire sessions with idle ≥ 30 minutes
/examples	None specified	Servlet and JSP Examples	true	0	Start Stop Reload Undeploy Expire sessions with idle ≥ 30 minutes
/host-manager	None specified	Tomcat Host Manager Application	true	0	Start Stop Reload Undeploy Expire sessions with idle ≥ 30 minutes
/manager	None specified	Tomcat Manager Application	true	1	Start Stop Reload Undeploy Expire sessions with idle ≥ 30 minutes

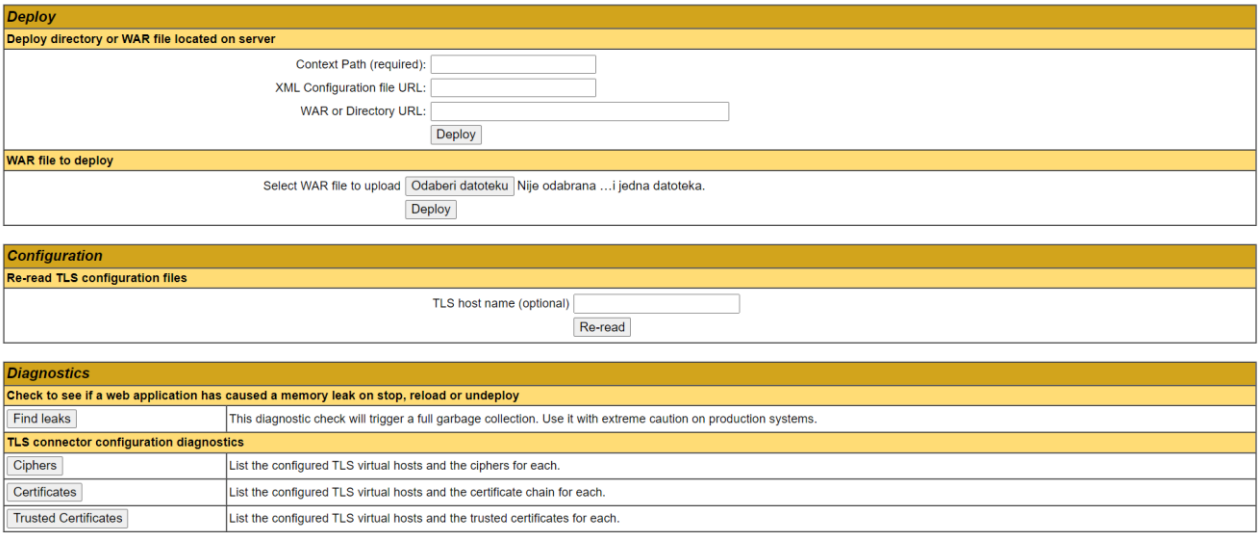
Slika 33. Kontrolna ploča Apache Tomcat aplikacijskog poslužitelja

U koloni su ponuđene sljedeće komande:

- ***Start*** – pokreće se izvršavanje aplikacije,
- ***Stop*** – odabrana se aplikacija zaustavlja u izvođenju,
- ***Reload*** – odabrana se aplikacija zaustavlja, te se pokreće inicijalizacija aplikacije. Nakon inicijalizacije, aplikacije se automatski pokreće i prelazi u status Start,
- ***Undeploy*** –koristi se ukoliko želimo obrisati aplikaciju s poslužitelja. Aplikacija se trajno briše iz oblak okoline.

Kada se želi postaviti nova aplikacija na aplikacijski poslužitelj, potrebno je pritisnuti gumb „Odaberi datoteku“ u polju za prijenos WAR datoteke (*engl. WAR file to deploy*). Na slici 34. prikazan je dio kontrolne ploče koji je namijenjen upravo za prijenos aplikacija u aplikacijski poslužitelj. Nakon odabira WAR datoteke, pritisne se gumb „Deploy“ i započinje proces prijenosa datoteke. Nakon uspješnog prijenosa datoteke, menadžer pokušava aplikaciju staviti u status Start. Provjeravaju se svi uvjeti koji

trebaju biti ispunjeni za ispravno pokretanje nove aplikacije. Ukoliko neki od uvjeta nije ispunjen, aplikacija neće biti automatski pokrenuta, već će biti u stanju Stop. U tom slučaju, menadžer korisniku prikaže dodatnu informaciju o problemu na koji je aplikacijski poslužitelj naišao. Detalje o nastaloj situaciji moguće je dobiti iz log datoteke koja se nalazi na oblak poslužitelju gdje je Tomcat instaliran.



Slika 34. Tomcat, objava aplikacije

Na samom Tomcat poslužitelju može se instalirati veći broj aplikacija, a sam broj ovisi o karakteristikama oblak poslužitelja. Konkretno, o veličini memorije koju poslužitelj može koristiti.

14.3. Kontejnerizacija – produkcija aplikacije

Pod pojmom kontejnerizacija podrazumijeva se alternativa produkcije aplikacije, koristeći virtualnu okolinu rada aplikacije. Aplikacija je začahurena u strukturu koju nazivamo kontejner. Okolina sadrži aplikaciju zajedno sa svim resursima i operacijskim sustavom. Precizniji termin bi bio virtualizacija na razini operacijskog sustava, dok je kolokvijalni termin kontejnerizacija preuzet iz discipline logistike jer brodski kontejner u stvarnosti oponaša ono što kontejner na razini operacijskog sustava čini virtualno. Poput virtualnih strojeva kontejneri omogućuju pakiranje aplikacija skupa s pripadajućim potpornim datotekama o kojima ovise, pružajući tako izolirane okoline u kojima se aplikacije pokreću.

Kontejneri pružaju mehanizam pakiranja aplikacije u okolinu koja nije ovisna o vrsti poslužitelja na kojem se izvodi. To omogućuje prenosivost aplikacije te oslobađanja potrebe da se na svakom poslužitelju stvara potrebna okolina za izvođenje aplikacije.

Ovakav pristup objave produkcije aplikacije ima određenih prednosti:

- dosljedna okolina izvođenja aplikacije što znači da je na svim poslužiteljima okolina potpuno identična i upravo takva kakva je u razvojnoj okolini. Na taj se način osigurava veća produktivnost i kraće vrijeme objave nove verzije aplikacije,
- prenosivost aplikacije na različite oblak platforme,

- izolacija aplikacije od ostalih aplikacija. Aplikacija dobiva svoj virtualni memorijski prostor i sve resurse potrebne za rad. Aplikaciji je onemogućen negativni utjecaj na rad ostalih aplikacija koje su postavljene na istom oblak poslužitelju.

Za kontejnerizaciju i pripremu aplikacije trenutno se koriste razni alati koji su često integrirani u samom razvojnom alatu u kojem se razvija web aplikacija. Za kreiranje kontejnera potrebno je instalirati Docker menadžera kojim će se izrađivati kontejneri aplikacija za oblak poslužitelja. Aplikacija se preuzima sa službenih stranica Dockera[7]. Na slici 35. prikazana je stranica za preuzimanje Docker aplikacije.

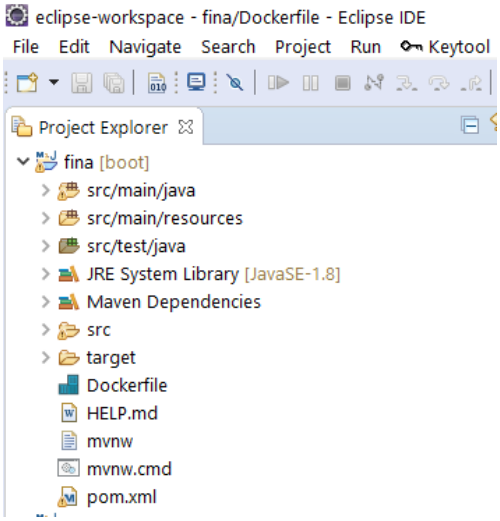
Slika 35. Doker službena stranica

Nakon instalacije Docker desktop aplikacije potrebno je u Spring Boot projekt dodati docker format datoteku pod nazivom ***Dockerfile*** kojom se definiraju organizacijska struktura u budućem kontejneru slike web aplikacije. Na slici 36. prikazana je pozicija datoteke u Spring Boot projektu.

U ovu datoteku potrebno je upisati određen set instrukcija koje će definirati razine/mape u strukturi kontejnera. Kod 78. prikazuje primjer sadržaja docker datoteke. Upisuje se ciljana Java JDK koji koristi web aplikacija, oznaku je li ciljana arhiva JAR ili WAR, naziv JAR aplikacije.

Ukoliko aplikacija koristi Maven strukturu, potrebno je pokrenuti naredbu ***docker build -t springio/gs-spring-boot-docker*** za kreiranje kontejnera / slike aplikacije. Pokretanjem

[7] Docker - https://hub.docker.com/

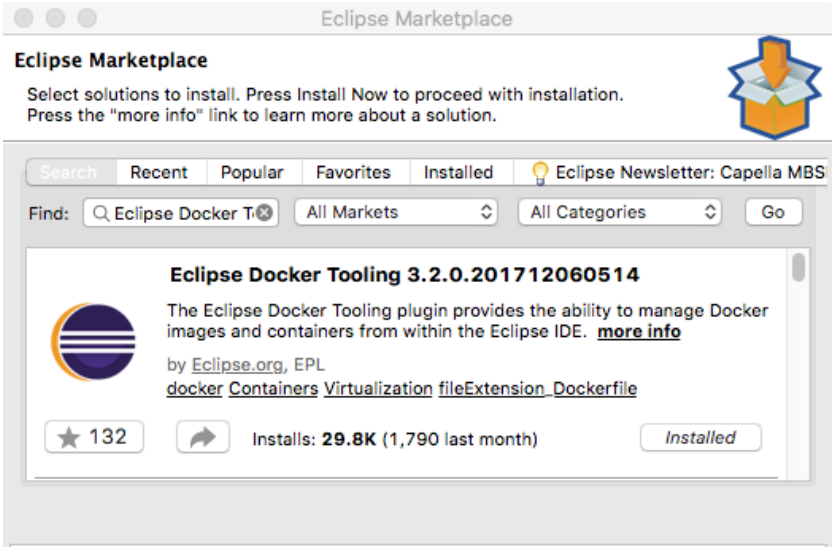


Slika 36. Dockerfile u Spring Boot projektu

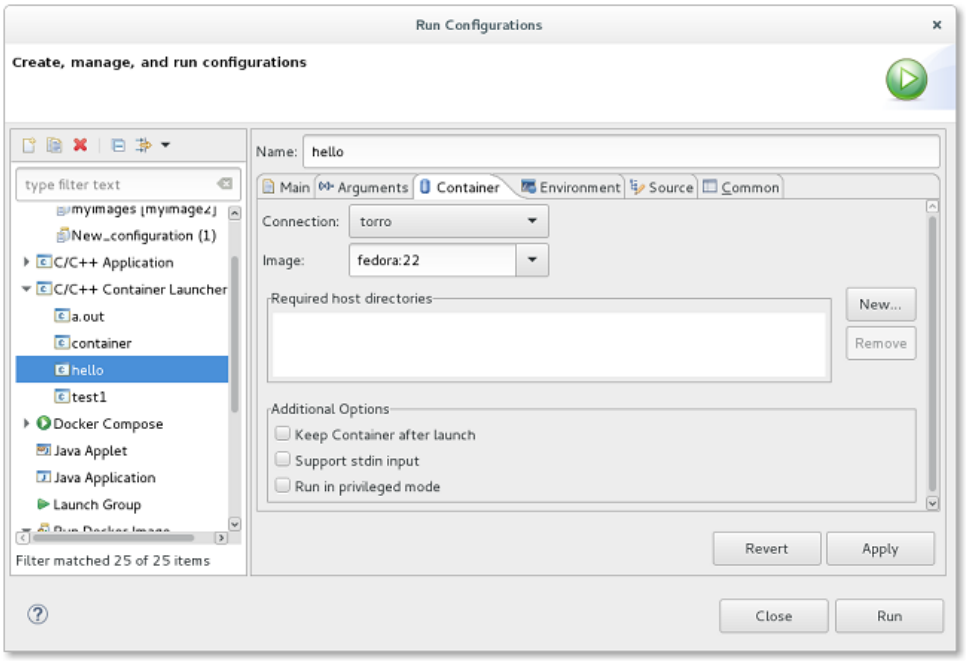
```
FROM openjdk:8-jdk-alpine
ARG JAR_FILE=target/*.jar
COPY ${JAR_FILE} app.jar
ENTRYPOINT ["java","-jar","/app.jar"]
```

Kod 78. Dockerfile kontrakcijska datoteka

Za Eclipse IDE može se koristiti dodatak ***Eclipse Docker Tooling*** koji se može preuzeti na Eclipse Marketplace. Izgled alata prikazan je na slici 37. Nakon instalacije alata i njegovog pokretanja, korisnik može odabira vrstu aplikacije, vrstu oblak poslužitelja te razne parametre za generiranje kontejnera. Na slici 38. prikazan je izgled prozora Docker alata. U datoteci

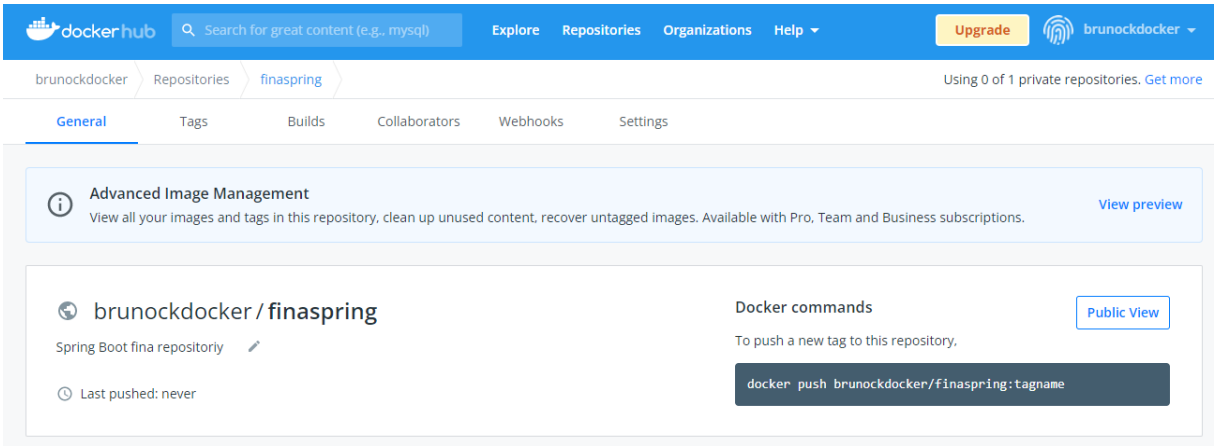


Slika 37. Eclipse IDE alat za kontejnerizaciju aplikacija



Slika 38. Docker alat za kreiranje kontejnera

Nakon generira kontejner Spring Boot aplikacije, kontejner je potrebno postaviti u Docker oblak okolini na oblak poslužitelju. Za postavljanje kontejnera postoje razne oblak platforme koje na razne načine automatiziraju cijeli proces. Ukoliko za testiranje želimo koristiti ***DockerHub*** okruženje, potrebno je kreirati repozitorij u kojem će biti stavljen kontejner. Nakon kreiranja repozitorija, potrebno je staviti kontejner. To se pokreće s naredbom doker push koja je sastavni dio platforme. Sve je prikazano na slici 39. Nakon pravilnog učitavanja kontejnera, aplikacija se može početi koristiti.



Slika 39. Dockehub platforma za testiranje kontejnera