Uzorci dizajna u programskom okviru Spring

Cvitković, Matej

SVEUČILIŠTE U ZAGREBU

FAKULTET ORGANIZACIJE I INFORMATIKE

V A R A Ž D I N

Matej Cvitković

UZORCI DIZAJNA U PROGRAMSKOM OKVIRU SPRING

DIPLOMSKI RAD

Varaždin, 2018.

SVEUČILIŠTE U ZAGREBU

FAKULTET ORGANIZACIJE I INFORMATIKE

V A R A Ž D I N

Matej Cvitković

Matični broj: 45291/16-R

Studij: Informacijsko i programsko inženjerstvo

UZORCI DIZAJNA U PROGRAMSKOM OKVIRU SPRING

DIPLOMSKI RAD

Mentor:

Prof. dr. sc. Dragutin Kermek

Varaždin, lipanj 2018.

Matej Cvitković

Izjava o izvornosti

Izjavljujem da je moj diplomski rad izvorni rezultat mojeg rada te da se u izradi istoga nisam koristio drugim izvorima osim onima koji su u njemu navedeni. Za izradu rada su korištene etički prikladne i prihvatljive metode i tehnike rada.

Autor/Autorica potvrdio/potvrdila prihvaćanjem odredbi u sustavu FOI-radovi

Sažetak

U ovome radu biti će objašnjen pojam uzorak dizajna. Navest će se najbitniji uzorci dizajna te će isti biti klasificirani u pripadajuće skupine uzoraka. Svaki od navedenih uzoraka biti će detaljno objašnjen i popraćen slikama kako njihova struktura treba izgledati.

Zatim će biti detaljno obrađen pojam Spring programskog okvira, odnosno biti će objašnjen nastanak samog programskog okvira, njegovo korištenje i eventualna povezanost s uzorcima dizajna koja će kasnije biti prikazana u vidu funkcionalne aplikacije.

Za kraj kako bi se prethodno navedeni pojmovi stavili u zajednički kontekst biti će napravljena web aplikacija pomoću Spring programskog okvira koristeći uzorke dizajna, koji bi trebali olakšati shvaćanje i održavanje programskog rješenja. Na temelju iskustva dobivenog iz prethodno kreiranog programskog proizvoda prikazati će se mišljenje i zaključak o Spring programskom okviru te na koji način i u kolikoj mjeri su korišteni uzorci utjecali na kvalitetu programskog rješenja.

Ključne riječi: Java, web aplikacije, Spring, programski okviri, uzorci dizajna

Sadržaj

1.Uvod

Spring programski okvir (eng. *Spring framework*) nastao je 2003. godine kao odgovor na složenost ranih Java 2 Enterprise Edition (nadalje u tekstu J2EE) specifikacija. J2EE je skup specifikacija koji proširuju Java Standard Edition (nadalje u tekstu JSE) sa specifikacijama poput distribuiranog računalstva i web servisa (eng. *Distributed computing and web services*). Dok neki smatraju da su Java Enterprise Edition (nadalje u tekstu JEE) i Spring konkurencija, činjenica je da je Spring zapravo komplementaran s JEE. Springov programski model ne obuhvaća specifikaciju JEE platforme, već se integrira s pomno odabranim pojedinačnim specifikacijama iz JEE spektra. Neki od korištenih specifikacija su Servlet API, WebSocket API, JSON Binding API, Bean Validation itd. Što znači da Spring programski okvir zapravo predstavlja lagano rješenje i potencijalno mjesto za razvoj aplikacija spremih za tvrtke (eng. *enterprise-ready applications*). Također bitno je napomenuti kako je on modularan i omogućava korištenje dijelova koji su potrebni za rad aplikacije, bez da postoji potreba za uzimanjem ostalih nepotrebnih specifikacija iz J2EE spektra (Johnson, 2005)

Uzorci dizajna (eng. *design patterns*) generalno su smatrani kao ponavljajuće rješenje na često ponavljajući problem u dizajnu softvera. To je zapravo opis ili još bolje rečeno predložak rješenja nekog problema koje može biti korišteno u raznim situacijama. Oni olakšavaju i ubrzavaju razvojni proces koristeći provjerene i testirane paradigme razvoja. Korištenje istih može dovesti do sprječavanja velikih problema i poboljšati čitljivost programskog koda ne samo za razvojne inženjere, već i za arhitekte koji su upoznati s uzorcima dizajna. Uzorci su prvi put spomenuti 1977. godine kada je Christopher Alexander uveo ideju uzoraka tj. primjera uspješnih rješenja za probleme. Trebalo je proći 10 godina kako bi se ta ideja prenijela u kontekst objektno-orijentiranih jezika od strane Ward Cunninghama i Kent Becka. 1994. godina predstavlja vjerojatno i najbitniju godinu što se tiče uzoraka dizajna. Može se slobodno nazvati prekretnicom što se tiče njihove primjene i popularizacije u svijetu programiranja. Iste godine izlazi najbitnija knjiga vezana uz uzorke dizajna pod nazivom „Uzorci dizajna: Elementi ponovo iskoristivog objektno-orijentiranog softvera“ (eng. *Design Patterns: Elements of Reusable Object-Oriented Software*) ili popularni naziv je GoF (eng. *Gang of Four*) aludirajući na četiri autora koji su napisali knjigu. Ona će također biti korištena za potrebe pisanja ovoga rada i većina primjera će biti proizvedena uz pomoć nje.

Kako bi se uzorci mogli uočiti i primijeniti primjeri najbolje prakse prilikom rada i dizajniranja aplikacija koje će biti razvijene pomoću Spring programskog okvira, potrebno je

povezati Spring programski okvir i uzorke dizajna, što je ujedno i cilj i motivacija pisanja ovog rada.

Poglavlje Uzorci dizajna objašnjava zašto nastaju uzorci dizajna, kako se oni dijele, zašto se koriste te kako.

Poglavlje Spring programski okvir objašnjava zašto nastaje Spring programski okvir, zatim koju filozofiju razvoja prati i od kojih modula se sastoji. Svaki modul je detaljno objašnjen i podijeljen na pod module. Nakon toga, objasniti će se osnovne tehnologije koje čine Spring posebnim te za kraj prikazati će se koje verzije i vrste Spring programskih okvira postoje u trenutku pisanja rada.

Poglavlje Usporedba Java programskih okvira objašnjava koji programski okviri za Javu postoje te koji su najpopularniji. Zatim vrši se analiza 5 najpopularnijih programskih okvira.

Poglavlje Primjena uzoraka dizajna na arhitekturnoj razini Springa objašnjava koje uzorke Spring koristi kako bi olakšao posao razvojnim inženjerima. Zatim prikazan je primjer i autorov osobni pogled na arhitekturu koja se često primjenjuje u Spring aplikacijama.

Poglavlje Prototip i razvoj aplikacije objašnjava koje tehnologije će se koristiti za izradu aplikacije, zatim razrađuje se ideja aplikacije pomoću tijeka rada i određuje se dizajn baze podataka. Također u kratkim crtama objašnjava se proces razvoja aplikacije i njezina konačna arhitektura.

Poglavlje Prikaz aplikacije sadrži slike koje prikazuju sve glavne funkcionalnosti aplikacije.

2. Uzorci dizajna

Uzorci dizajna identificiraju ime i zajedničke apstraktne teme u objektno-orijentiranom dizajnu. Oni shvaćaju namjeru korištenja dizajna uz pomoć identificiranja objekata, njihovih veza i distribucije njihovih odgovornosti. Uzorci dizajna igraju mnoge uloge u objektno-orijentiranom razvojnom procesu, pružaju zajednički rječnik za dizajn, smanjuju složenost imenujući i definirajući apstrakcije. Također oni čine osnovu iskustva za izgradnju ponovo iskoristivog softwarea i djeluju kao građevni blokovi čijim korištenjem se mogu izgraditi složeniji dizajni aplikacija. Mogu biti smatrani ponovo iskoristivim mikro arhitekturama koji doprinose cjelokupnoj arhitekturi sustava (Gamma, Helm, Johnson i Vlissides, 1997, str. 15).

2.1 Općenito

Uzorak dizajna sastoji se od četiri ključna elementa (Gamma, Helm, Johnson i Vlissides, 1997, str. 17):

1. Naziv uzorka

On predstavlja naziv koji će se koristiti za opisivanje dizajnerskog problema, njegovog rješenja u jednoj ili dvije riječi. Imenovanje uzoraka povećava rječnik dizajna. Korištenje naziva omogućava lakše razmišljanje i komuniciranje o uzorcima, zapravo pronalazak dobrog imena predstavlja jedan od najtežih dijelova razvoja kataloga koji sadrži uzorke dizajna.

2. Problem koji rješava

Problem opisuje u kojim situacijama i kada je potrebno primijeniti odgovarajući uzorak dizajna. Objašnjava sami problem i njegov kontekst. Može opisati strukture klasa ili objekata koje su karakteristični za nefleksibilni dizajn. Također ponekad će problem uključivati popis uvjeta koji moraju biti ispunjeni prije nego što ima smisla primijeniti uzorak.

3. Rješenje problema

Rješenje opisuje elemente koji čine dizajn, njihove veze, odgovornosti i suradnje. Ono ne opisuje točno definiran dizajn ili implementaciju iz razloga što uzorci zapravo predstavljaju predloške koji su upotrebljivi u više različitih situacija. Stoga uzorci pružaju apstraktni opis dizajnerskog problema i način na koji bi trebao izgledati opći raspored elementa koji taj problem rješavaju.

4. Posljedice rješenja

Posljedice su rezultati i kompromisi koji nastaju primjenom uzoraka. Iako su posljedice često nenaglašene prilikom opisa dizajnerske odluke, one su kritične za procjenu dizajnerskih posljedica i za razumijevanje troškova i prednosti primjene uzoraka. Kako je ponovna iskoristivost čest faktor u objektno-orijentiranom dizajnu posljedice moraju uključivati njihov utjecaj na fleksibilnost sustava, proširivost i prenosivost.

Naziv uzorka dizajna, sažima i identificira ključne aspekte zajedničke strukture dizajna koji ga čine korisnima za izradu ponovo iskoristivog objektno-orijentiranog dizajna. Oblik dizajna identificira razrede i primjere koji sudjeluju, njihove uloge i suradnje te raspodjelu odgovornosti. Svaki uzorak usredotočuje se na određeni objektno-orijentirani dizajn ili problem. Ono opisuje kada se primjenjuje, bez obzira na to može li se primijeniti u pogledu drugih ograničenja u dizajnu, kao i posljedice i kompromise njegove uporabe. Svaki uzorak je podijeljen u odjeljke prema predlošku koji se koristi za opisivanje istih. Predložak daje jedinstvenu strukturu informacijama, što olakšava učenje, uspoređivanje i korištenje uzoraka dizajna. Ukoliko neki novi uzorak dizajna bude otkriven on bi trebao pratiti sljedeći predložak koji je prikazan u obliku tablice.

Tablica 1. Prikaz opisa uzoraka Dizajna

Naziv i klasifikacija	Ime uzorka predstavlja srž problema koji neki uzorak rješava. Koristeći samo naziv razvojni inženjer morao bi biti u stanju prepoznati o kojem se strukturnom problemu radi i na koji način ga treba riješiti. Dobar naziv je jako bitna stavka prilikom kreiranja uzorka, zato što će isti ući u rječnik uzoraka, po tome taj naziv bi trebao sažeto objasniti o kojem problemu se radi i na koji način ga riješiti.
Namjera	Kratka izjava koja odgovara na sljedeća pitanja: Što radi dizajn? Koji je razlog njegovog postojanja (eng. *rationale*) i namjera? Koji određeni problem dizajna ili problem rješava?
Poznao kao	Ostala poznata imena za uzorak, ako ih ima.
Motivacija	Scenarij koji ilustrira dizajnerski problem i način rješavanja tog problema postavljanjem određene strukture klasa i objekata

Primjenjivost	Koje su situacije u kojima se može primijeniti uzorak dizajna? Kako izgledaju primjeri loših dizajna čiji problemi se mogu riješiti pravilnom upotrebom uzoraka? Kako se mogu prepoznati te situacije?
Struktura	Grafički prikaz klasa u uzorcima, za prikaz koriste se razni dijagrami.
Sudionici	Klase i / ili objeki koji sudjeluju u uzorku dizajna i njihove odgovornosti.
Suradnje	Način na koji sudionici surađuju međusobno da bi izvršili svoje odgovornosti.
Posljedice	Kako uzorak podržava svoje ciljeve? Koji su kompromisi i rezultati korištenja uzorka?
Implementacija	Kojih zamki, naputaka ili tehnika moramo biti svjesni prilikom provođenja uzorka? Postoje li jezični problemi?
Primjer koda	Dijelovi programskog koda koji ilustriraju način na koji se pojedini uzorak može implementirati.
Poznate uporabe	Poznati primjeri primjene uzoraka u stvarnim sustavima.
Povezani uzorci	Koji su uzorci obrasci blisko povezani s tim uzorkom? Koje su važne razlike? S kojim drugim uzorcima trebao bi se koristiti ovaj?

(Izvor: Gamma, Helm, Johnson I Vlissides, 1997, str. 20-21)

Također prilikom opisa svih uzoraka u kasnijim poglavljima djelomično će biti korišten pristup iz tablice 2. za njihovo opisivanje i objašnjavanje.

2.2 Katalog uzoraka dizajna

Uzorci dizajna mogu varirati u njihovoj zrnatosti i razini apstrakcije. Budući da postoji mnogo uzoraka dizajna, potrebno je bilo uvesti određeni način organiziranja istih. Određeno je kako će se uzorci grupirati na temelju obitelji srodnih uzoraka. Ovakav način klasifikacije pomaže u bržem saznavanju uzoraka u katalogu i pomaže na putu za otkrivanje i klasificiranje novih uzoraka. Prema GoF-u u trenutku pisanja u katalogu nalazila su se 23 uzorka. Navedeni uzorci dijelili su se na temelju svrhe i opsega.

Kada se gleda aspekt svrhe uzorci se dijele na tri kategorije uzorci za kreiranje, strukturni i za ponašanje. Uzorci vezani za kreiranje bave se procesom stvaranja objekata,

strukturni bave se sastavljanjem i organiziranjem klasa i objekata, a uzorci za ponašanje bave se na koji način objekti i klase međusobno djeluju i dijele odgovornosti.

Opseg se pak odnosi primjenjuje li se određeni uzorak na objektnoj razini ili klasnoj. Klasni uzorci svoje veze stvaraju kroz nasljeđivanje i zbog toga su statične odnosno fiksirane za vrijeme kompiliranja. Objektni uzorci bave se vezama između objekata, koje su dinamične i mogu se mijenjati za vrijeme kompiliranja. Bitno je napomenuti da skoro svi uzorci u nekoj mjeri koriste nasljeđivanje, ali samo oni koji su usredotočeni na veze među klasama su prozvani klasnim uzorcima.

Tablica 2. Klasifikacija uzoraka

		Svrha		
		Uzorci za kreiranje	Strukturni uzorci	Uzorci za ponašanje
Opseg	Klasni	Factory Method	Adapter	Interpreter Template Method
	Objektni	Abstract Factory Builder pattern Prototype Singleton	Adapter Bridge Composite Decorator Facade Proxy	Chain of responsability Command Iterator Mediator Memento Flyweight Observer State Strategy Visitor

(Izvor: Gamma, Helm, Johnson i Vlissides, 1997, str. 23)

Kasnije u ovome poglavlju detaljno će se objasniti svaki od gore navedenih uzoraka uz neke dodatne uzorke koji su popularni u vrijeme pisanja ovoga rada.

2.3 Kako uzorci dizajna rješavaju probleme

Prethodno je rečeno kako su uzorci dizajna ponavljajuće rješenje na često ponavljajući problem u dizajnu softvera. Stoga će se na temelju zaključaka donesenih u knjizi GoF, iznijeti načini na koje uzorci doprinose rješavanju problema i u kojim situacijama oni mogu biti od pomoći (Gamma, Helm, Johnson i Vlissides, 1997, str. 25-37).

2.3.1 Pronaći odgovarajuće objekte

Objektno-orijentirano programiranje temelji se na objektima i jedan od najtežih zadataka u objektno-orijentiranom programiranju je razložiti (eng. *decompose*) sustav na njegove komponente tj. objekte. Navedeni zadatak predstavlja problem zbog svoje složenosti, ona se manifestira iz razloga što prilikom razlaganja potrebno je u obzir uzeti velik broj faktora. Neki od njih su učahurivanje, zrnatost, zavisnosti, fleksibilnost, radni učinak, ponovna iskoristivost itd. Zapravo razvoj kvalitetnih objektno-orijentiranih programa bio bi složen postupak, ukoliko bi on uključivao samo nabrojane faktore, ali postoji još veliki broj faktora koji mogu utjecati na sveukupnu kvalitetu i težinu izvedbe nekog rješenja. Još treba napomenuti da nisu svi faktori zasebni, zapravo realna je situacija da izvršenje jednog faktora može negativno utjecati na niz drugih faktora.

Uzorci dizajna pomažu u rješavanju ovoga problema na način omogućuju pojednostavljeno uočavanje apstrakcija i objekata koji mogu pomoći u rješavanju navedenih problema te oni zapravo predstavljaju ključnu ulogu prilikom gradnje fleksibilnih dizajna.

2.3.2 Utvrđivanje zrnatosti objekta

Ovisno o veličini aplikacije objekti mogu varirati u veličini i u broju. Cilj je napraviti objekte koji nisu preopširni, bolje je podijeliti odgovornosti kroz cijelu aplikaciju kako bi kasnije bilo lakše nadograđivati, testirati i održavati.

Uzorci dizajna ovom problemu izlaze u susret koristeći pojedine uzorke za kreiranje i ponašanje. Oni omogućavaju da se koristi veliki broj objekata s minimalnom zrnatosti, zatim predstavljaju načine pomoću kojih se složeni objekti mogu razložiti na više jednostavnijih objekata čije onda odgovornosti su bitno smanjene i to olakšava prethodno navedene ciljeve.

2.3.3 Odrediti sučelje objekta

Sučelja su ključna stavka u objektno-orijentiranom programiranju i objekti bi trebali biti vidljivi kroz svoja sučelja. Nema drugog načina da se saznaju informacije o objektu ili da mu se pošalje zahtjev za neki zadatak osim da ga se pozove preko njegovog sučelja. Svaki objekt može napraviti svoju i jedinstvenu implementaciju metoda koje mu nalaže njegovo sučelje.

Uzorci olakšavaju dizajniranje sučelja na način da identificiraju njihove ključne elemente i koje vrste podataka će se slati kroz sučelja, također pojedini uzorci mogli bi dati uvid što staviti u sučelje i što bi trebalo ukloniti iz njega, a da nije potrebno. Osim navedenog

pomažu i pri definiranju veza među sučeljima tj. često zahtijevaju da određene klase imaju slična sučelja ili stavljaju određenu vrstu ograničenja na sučelja drugih klasa.

2.3.4 Odrediti implementaciju objekta

Ovo poglavlje najviše je vezano uz problem nasljeđivanja. Koju vrstu nasljeđivanja primijeniti u kojoj situaciji, koje su prednosti nasljeđivanja korištenjem apstraktnih klasi, a koje su prednosti nasljeđivanja putem sučelja. Klasno nasljeđivanje predstavlja mehanizam proširivanja funkcionalnosti aplikacija ponovnim iskorištavanjem funkcionalnosti klasa roditelja. Omogućuje stvaranje novih vrsta objekata i to praktično besplatno, zato što nasljeđuje većinu funkcionalnosti od postojećih klasa.

Pravilno korištenje sučelja omogućuje da pod klase koriste tj. nadjačavaju metode koje se nalaze u popisu sučelja i zbog toga bilo koja klasa koja implementira to sučelje može odgovoriti na zahtjev. Samim time klijent ne mora znati koji specifični objekti su korišteni za realizaciju, sve dok se objekti pridržavaju sučelja koje klijenti koriste. Isto tako klijenti ne moraju biti upoznati s klasama koje čine taj objekt. Moraju biti upoznati samo s apstraktnim klasama koje čine korišteno sučelje. Time se uveliko smanjuju implementacijske zavisnosti između podsustava i to vodi do sljedećeg principa ponovno iskoristivog objektno-orijentiranog dizajna koji glasi *Programiraj ka sučelju, a ne prema implementaciji (eng. Program to an interface, not an implementation)*.

2.3.5 Uključivanje mehanizma ponovnog korištenja

Dvije najčešće tehnike korištene za ponovnu iskoristivost u objektno-orijentiranim sustavima su nasljeđivanje klasa i sastavljanje objekata. Obje tehnike imaju svoje prednosti i mane, npr. nasljeđivanje je statično definirano i nema mogućnost promjena tijekom kompiliranja, ali isto tako ono omogućuje lakšu izmjenu postojeće implementacije. Za razliku od nasljeđivanja, sastavljanje objekta je dinamično za vrijeme izvođenja, jer je nasljeđivanje definirano za vrijeme kompiliranja.

Često stručnjaci tvrde da nasljeđivanje ruši princip učahurivanja, zato što implementacija pod klasa može postati previše vezana za implementaciju klasa roditelja i time svaka promjena u roditeljskoj klasi prisiljava na promjenu njezinih pod klasa. Stoga se dolazi do drugog principa u dizajnu ponovo iskoristivih objektno-orijentiranih sustava koji glasi *Biti više naklon korištenju sastavljanju objekata u odnosu na klasno nasljeđivanje (eng. Favor object composition over class inheritance)*.

2.3.6 Delegacija

Delegacija je način stvaranja objekata koji je jednako moćan za ponovnu uporabu kao i nasljeđivanje. U ovom procesu dva objekta su uključena u rješavanju zahtjeva. Prvi objekt prima zahtjev i delegira operacija ka svom delegatu. Glavna prednost delegiranja je što olakšava sastavljanje ponašanja za vrijeme izvođenja i promjenu načina na koji su sastavljeni. Delegiranje može biti dobar dizajnerski izbor, ali samo u slučaju kada pojednostavljuje rješenje umjesto da ga komplicira. Razlog tome je što korištenje dinamičkog i vrlo parametiziranog softvera otežava njegovo razumijevanje uključujući i neke nedostatke vezane za vrijeme izvršavanja.

2.3.7 Nasljeđivanje naprotiv parametiziranih tipova

Još jedan pristup za ponovnu iskoristivost komponenti je kroz korištenje parametiziranih tipova. Ova tehnika omogućuje definiranje tipa bez direktnog vezanja na tip koji će biti korišten. Pošto je ovaj rad temeljen na Spring programskog okviru, a on je temeljen na Javi, parametizirani tipovi u Javi su poznati pod nazivom generički tipovi. Generički tipovi proslijeđeni su kao parametri na mjestima gdje su planirani za korištenje. Stoga on predstavlja treći način za sastavljanje ponašanja u objektno-orijentiranim sustavima. Bilo kojih od navedenih je ispravan, ali treba odabrati ovisno o prilikama i potrebama sustava.

2.3.8 Dizajniranje za promjene

Ključ za maksimiziranje ponovne iskoristivosti leži u predviđanju novih zahtjeva i izmjena postojećih te u oblikovanju sustava kako bi se mogao razvijati u skladu s tim. Da bi sustav bio otporan na takve promjene, prilikom dizajniranja mora se uzeti u obzir da će se on morati mijenjati tijekom svog vijeka trajanja. Dizajn koji ne uzima u obzir rizike velikih redizajna u budućnosti, mogao bi uključivati redefiniranje i ponovnu implementaciju klasa, izmjenu klijenata i ponovno testiranje. Redizajn utječe na mnoge dijelove softverskog sustava, a neočekivane promjene su uvijek skupe novčano i vremenski. Uzorci dizajna pomažu na način da osiguravaju da se sustav može promijeniti na određene načine. Svaki od uzoraka omogućuje da se neki aspekt strukture sustava razlikuje od ostalih aspekata, čime se stvara sustav koji je otporniji na promjene.

2.4 Uzorci za kreiranje

Navedeni uzorci usko su vezani uz instantaciju klasa. Oni mogu biti još podijeljeni na dvije pod klase kao što je vidljivo u tablici 3, koja prikazuje detaljnu klasifikaciju uzoraka. Uzorci za kreiranje mogu se podijeliti na klasno i objektno vezane kreacijske uzorke. Klasno kreacijski uzorci najčešće koriste nasljeđivanje, ali bitna stavka da je korišteno nasljeđivanje efikasno izvedeno u instancijskom procesu, dok objektni kreacijski uzorci koriste delegiranje (Shvets,

2013, str.17). Uzorci za kreiranje postaju važni jer kako se sustavi razvijaju sve češće postaju ovisniji o sastavu i povezanosti objekata, nego o klasnom nasljeđivanju. Kako se to događa, naglasak se odmiče od čvrstog zapisivanja ustaljenih ponašanja prema definiranju skupa manjih temeljnih ponašanja čijim kombiniranjem i suradnjom može se sastaviti jedno ili više složenijih ponašanja(Gamma, Helm, Johnson i Vlissides, 1997, str. 72).

Iz toga se da zaključiti da samo stvaranje objekata s posebnim ponašanjima neće riješiti problem, nego dugoročno gledano taj kod će biti jako težak za održavanje i testiranje. U sljedećim uzorcima postoje dva česta ponašanja. Prvo je da učahuruju svo znanje o konkretnim klasama koje sustav koristi, a drugo je da skrivaju na koji način instance tih klasa su kreirane i sastavljane. Također bitno je napomenuti da uzorci za kreiranje mogu često biti suparnici, ali također mogu biti i komplementarni s drugim uzorcima, stoga prilikom opisa svakog od uzoraka biti će navedeno koji uzorci međusobno dobro surađuju i oni koje bi trebalo izbjegavati u zajedničkom korištenju.

2.5 Strukturni uzorci

Strukturni uzorci dizajna bave se načinom na koji bi klase i objekti morali biti sastavljeni i povezani kako bi tvorili veće strukture. Strukturni uzorci dijele se na klasne i objektne. Klasni pojednostavljuju strukturu tako da identificiraju odnose i uglavnom koriste nasljeđivanje za sastavljanje sučelja ili implementacija. Objektni za razliku od klasnih umjesto sastavljanja sučelja ili implementacija, opisuju načine sastavljanja objekata kako bi se ostvarila nova funkcionalnost. Ta novo dodana fleksibilnost iz objektnog sastavljanja proizlazi iz sposobnosti mijenjanja sastava u vremenu izvođenja, što je nemoguće sa statičkim klasnim sastavljanjem (Gamma, Helm, Johnson i Vlissides, 1997, str. 118).

2.6 Uzorci za ponašanje

Uzorci za ponašanje bave se algoritmima i dodjelom odgovornosti između objekata. Osim što opisuju klasne i objektne uzorke dodatno opisuju uzorke koji se koriste u njihovoj komunikaciji. Ovi uzorci karakteriziraju složeni kontrolni tok koji je teško pratiti u vremenu izvođenja. Postoje klasni i objektni uzorci.

Klasni uzorci ponašanja koriste se nasljedstvom za distribuciju ponašanja između klasa, a objektni uzorci koriste sastavljanje objekata umjesto nasljeđivanja (Gamma, Helm, Johnson i Vlissides, 1997, str. 188).

2.7 Java EE uzorci dizajna

Dizajn aplikacija može biti neizmjerno pojednostavljen primjenom Java EE uzoraka dizajna. Oni su uspostavljeni na temeljnim uzorcima dizajna, koji su opisani u već spomenutoj i popularnoj knjizi poznatoj pod akronimom GOF. Katalog Java EE uzoraka osim samih temeljnih načela objektnog dizajna uzima u obzir strategije za ispunjavanje izazova udaljenih dostupnih distribuiranih objekata. On pruža vremenski testirane smjernice i najbolje prakse za interakciju objekata u svakom sloju Java EE aplikacije, te se on baš kao i sama platforma razvija tijekom vremena. Stoga se uzorci dijele u tri skupine: uzorke u prezentacijskom sloju, uzorke u sloju poslovne logike i uzorke u integracijskom sloju (Kayal, 2008, str. 13-15).

2.7.1 Uzorci u prezentacijskom sloju

- *View Helper*: Odvaja pogled (eng. *view, nadalje u tekstu view*) od poslovne logike u J2EE aplikacijama.
- *Front Controller*: On pruža jedinstvenu točku akcije za obradu svih dolaznih zahtjeva u J2EE web aplikaciji, zatim prosljeđuje zahtjeve određenom upravljaču (eng. *controller,* nadalje u tekstu controller) kako bi moglo pristupiti modelu i viewu za prezentaciju sadržaja korisniku.
- *Application Controller*: Zahtjev koji je stvarno upravljan od strane *Application Controllera* i on djeluje kao pomoćni *Front Controller*. Odgovoran je za koordinaciju s poslovnim modelima i view komponentama.
- *Dispatcher View*: Odnosi se samo na prikaz i izvršava se bez poslovne logike da pripremi odgovor (eng. *response*) na sljedeći view.
- *Intercepting filters*: U J2EE web aplikaciji moguće je konfigurirati više presretača (eng. *interceptors*) za prethodnu i naknadnu obradu zahtjeva korisnika, kao što su praćenje i revidiranje (eng. *tracking and auditing*) korisničkih zahtjeva.

2.7.2 Uzorci u sloju poslovne logike

- *Business Delegate*: Djeluje kao most između *Application Controllera* i poslovne logike.
- *Application Service*: Pruža poslovnu logiku da implementira model kao jednostavne Java objekte za prezentacijski sloj.

2.7.3 Uzorci u integracijskom sloju

- *Data Access Object*: Implementiran je za pristup poslovnim podacima i odvaja logiku pristupa podacima od poslovne logike koja se nalazi u aplikaciji poduzeća.
- *Web Service Broker*: Učahuruje logiku za pristup resursima vanjske aplikacije, a zatim ona biva izložena kao web servis.

(Kayal , 2008, str. 1-20)

2.8 Anti uzorci dizajna

Anti uzorci dizajna (eng. AntiPatterns) poput svojih suparnika uzoraka dizajna definiraju rječnik za zajedničke neispravne procese i implementacije unutar organizacija. Rječnik na višoj razini pojednostavljuje komunikaciju između razvojnih inženjera i omogućuje sažet opis pojmova na višoj razini, koji opisuju najčešće rješenje problema koje generira izrazito negativne posljedice. Oni mogu biti rezultat arhitekta i/ili razvojnog inženjera koji ne znaju bolje, nemaju dovoljno znanja ili iskustva u rješavanju određene vrste problema ili primjene dobrog uzorka u pogrešnom kontekstu (Sourcemaking, 2018).

Oni prezentiraju detaljan plan za rješavanje temeljnih uzroka i omogućuju implementaciju učinkovitih rješenja. Kako bi to postigli učinkovito opisuje mjere koje se mogu poduzeti na nekoliko razina kako bi se poboljšao razvoj aplikacija, projektiranje softverskih sustava i učinkovito upravljanje softverskim projektima.

Dijele se u tri skupine anti uzorci u razvoju softvarea (eng. *Software Development AntiPatterns*) , anti uzorci u arhitekturi softverskih aplikacija (eng. Software Architecture AntiPatterns) i anti uzorci prilikom upravljanja projektima razvoja softverskih rješenja (eng. *Software Project Management AntiPatterns*) (Sourcemaking, 2018).

3. Spring programski okvir

S povećanjem veličine i složenosti implementacije informacijskih sustava potrebno je koristiti logičku konstrukciju ili arhitekturu za definiranje i kontrolu sučelja i integraciju svih komponenti sustava (J. A. Zachman,1987). Povedeni za prethodno definiranim potrebama nastaju programski okviri, a Spring je jedan od njih. Spring programski okvir je modularan tj. podržava sve funkcionalnosti iz JSE, također je moguće birati module koji su potrebni za rad aplikacije iz JEE paketa.

3.1 Općenito

Iako Java platforma pruža bogatstvo funkcionalnosti razvoja aplikacija, nedostaje joj način za organizirati osnovne građevinske blokove u koherentnu cjelinu, ostavljajući taj problematičan zadatak arhitektima i razvojnim inženjerima. Stoga su arhitekti i razvojni inženjeri bili su primorani sami razvijati sustave koristeći uzorke poput *Factory method*, *Abstract Factory*, *Builder patterna*, *Decoratora* i *Service Locatora* kako bi kvalitetno povezali kreirane klase i instancirane objekte (Rajup, 2017, str. 37). Ipak kao što je prije napomenuto uzorci dizajna su zapravo imenovane najbolje prakse, ali oni ne bi trebali predstavljati rješenje problema Java platforme (Spring docs, 2018 b). Oni trebaju biti implementirani od strane razvojnih inženjera u njihovim aplikacijama, ne kako bi diktirali arhitekturu aplikacije/programskog okvira nego da učine aplikacije čitljivijima, učinkovitijima itd. Stoga Spring programski okvir ima svoja rješenja za prethodno objašnjeni problem, a jedno od njih je inverzija kontrole (eng. *inversion of Control*, nadalje u tekstu IoC). IoC je pružio formalizirano sredstvo sastavljanja raznih komponenata spremnih za upotrebu u aplikaciju. Također poznat pod nazivom *Dependency Injection*. To je uzorak dizajna u kojem objekti definiraju svoje zavisnosti tj. druge objekte s kojima rade (koristeći konstruktorske argumente, argumente za *Factory method* ili svojstva koja su postavljena na objekt nakon njegovog kreiranja). Spremnik zatim ubrizgava te zavisnosti kada kreira zrno. Prethodno objašnjeni proces je zapravo obrnut, stoga ima naziv *Inversion of Control*. Spring objedinjuje formalne dizajnerske uzorke kao objekte prvoga razreda koji mogu biti integrirani u svaku aplikaciju (Fowler, 2004).

Veliki broj poduzeća koristi Spring programski okvir kako bi kreirali robusne i održive aplikacije. Trenutno je objavljeno 5 verzija, radna verzija u trenutku pisanja ovog rada je 5.0.3 i ona radi samo sa verzijama Jave 7, 8 i 9, dok za prijašnje verzije se preporuča korištenje verzije 4.3. (Spring docs, 2018 a) .

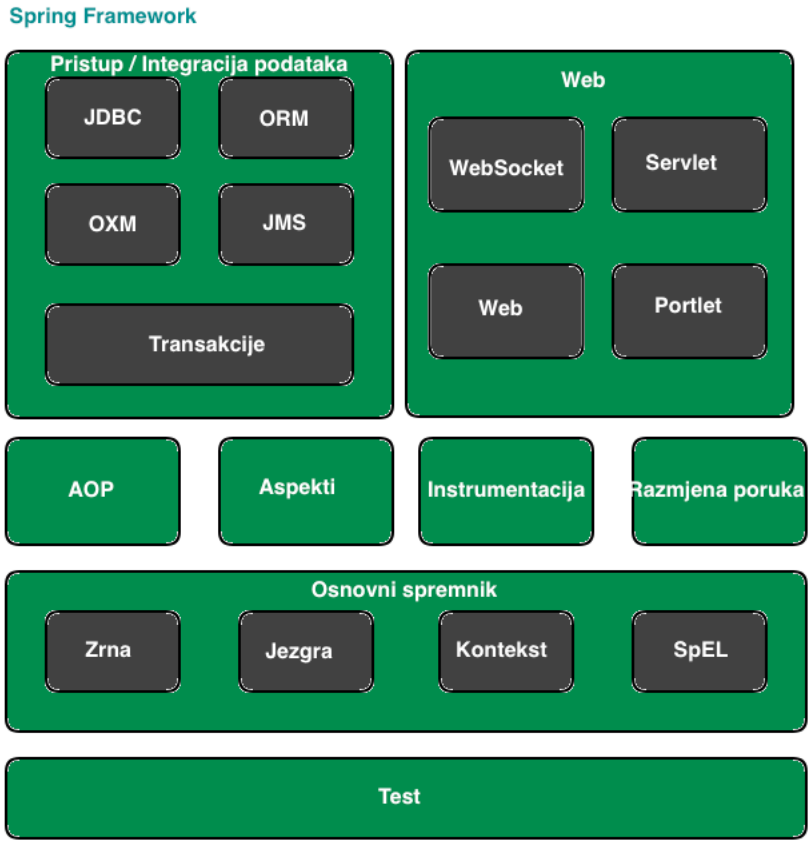
Prilikom učenja programskog okvira osim samog shvaćanja kako okvir funkcionira jako je bitna filozofija dizajna koju prati taj okvir. Filozofija koju slijedi Spring programski okvir je:

- Pružanje izbora na svim razinama. Odgađanje odluka koje imaju utjecaj na dizajn što je kasnije moguće.
- Promiče i podržava fleksibilnost, ne naginje prema određenom pogledu (eng. *opinionated*) kako bi funkcionalnosti trebale biti izvedene. Podržava širok spektar aplikacijskih potreba sa više različitih perspektiva.
- Održavanje čvrste kompatibilnosti gledajući na prošle verzije. Evolucija Spring programskog okvira je pažljivo upravljana i podržava pažljivo odabrani spektar Java Development Kit (nadalje u tekstu JDK) verzija i vanjskih biblioteka kako bi se olakšalo održavanje aplikacija i biblioteka koje ovise o Spring programskom okviru.
- Vođenje brige o dizajnu API-ja (eng. *application programming interface*) tj. razvojni tim odvaja puno vremena i resursa kako bi svaki API bio intuitivan i održavan kroz velik broj verzija.
- Postavljanje visokih standarda za kvalitetu koda. Stavlja snažan naglasak na značajan, aktualan i točan Javadoc. Jedan je od rijetkih projekata koji se može pohvaliti čistom strukturom koda bez kružnih zavisnosti između paketa.

(Spring docs, 2018 a)

3.2 Moduli

Nakon što je objašnjeno ono što Spring programski okvir koristi kao temeljna načela u svome daljnjem razvoju objasniti će se kako je Spring zapravo zamišljen. Sastoji od velikog broja mogućnosti organiziranih u 20 modula te oni su grupirani u nekoliko skupina. Skupine su osnovni spremik, pristup/integracija podataka, web, aspektno orijentirano programiranje, instrumentacija, razmjena poruka i test. Na slici je vidljivo kako su prethodno navedeni moduli podijeljeni i koje osnovne funkcionalnosti oni sadrže. (Spring docs, 2018 b)



Slika 1. Moduli Spring programskog okvira (Spring docs, 2016)

3.2.1 Osnovni spremnik

Osnovni spremnik sastoji se od modula *spring-core*, *spring-beans*, *spring-context*, *spring-context-support* i *spring-expression*. (Spring docs, 2016)

Spring-core pruža temeljne dijelove okvira uključujući i *Dependency Injection*, *spring-beans* sadrži *BeanFactory* što je zapravo sofisticirana implementacija *Factory Method* uzorka te ona uklanja potrebu za programskim *Singletonima* i omogućuje razdvajanje postavki i specifikacija zavisnosti od poslovne logike. (Spring docs, 2016)

Spring-context modul se temelji na čvrstoj bazi koju pružaju moduli jezgre i zrna. Sučelje *ApplicationContext* je ishodišna točka modula, a on je zapravo sredstvo za pristup objektima u okvirnom stilu koji je sličan Java Naming and Directory Interfaceu (nadalje u tekstu JNDI) registru. Nasljeđuje svoje funkcionalnosti od modula zrna i dodaje podršku za internacionalizaciju, širenje događaja, učitavanje resursa i transparento stvaranje konteksta (Spring docs, 2016).

Spring-context-support pruža podršku za integraciju čestih vanjskih biblioteka u Spring aplikacijski kontekst za predmemoriranje, razmjena mailova, upravljanje rasporedom i motore predložaka (Spring docs, 2016).

Spring-expression ima snažni jezik izraza (eng. *expression language*) koji podržava upite i manipuliranje grafikonom objekata pri izvođenju. Jezik podržava dohvaćanje i postavljanje vrijednosti, pozivanje metoda, pristupanje sadržaju polja, kolekcija i indekserima, logičke i aritmetičke operatore, imenovane varijable i dohvaćanje objekata pomoću imena iz IoC spremnika (Spring docs, 2016).

3.2.2 AOP i Instrumentacija

Spring-aop pruža AOP savezno usklađenu aspektno orijentiranu programsku implementaciju koja omogućuje definiranje presretača metoda za čisto odvajanje koda koji implementira funkcionalnost koja bi trebala biti odvojena.

Spring-instrument pruža podršku instrumentacije klasa i implementacije učitavača klasa (eng. *classloader*) koji se može koristiti u nekim aplikacijskim poslužiteljima. Spring-instrument-tomcat modul sadrži instrumentacijski agent za poslužitelja Tomcat (Spring docs, 2016).

3.2.3 Razmjena poruka

Spring-messaging je glavna stavka ove skupine i on sadrži ključne apstrakcije poput *Message*, *MessageChannel*, *MessageHandler* i druge koji služe kao osnova za aplikacije temeljene na razmijeni poruka. Također, uključuje skup anotacija za mapiranje poruka za metode, slične su kao Spring MVC anotacije (Spring docs, 2016).

3.2.4 Pristup/integracija podataka

Ovaj sloj sastoji se od Java Database Connectivity (nadalje u tekstu JDBC), objektno-relacijsko mapiranje (eng. *object-relational mapping*, nadalje u tekstu ORM), objektno XML mapiranje (eng. *object XML Mapping*, nadalje u tekstu OXM), Java Message Service (nadalje u tekstu JMS) i transakcijskog modula.

Spring-jdbc pruža JDBC apstrakcijski sloj koji uklanja potrebu za napornim JDCB kodiranjem i raščlanjivanjem (eng. *parsing*) kodova grešaka za specifične pružatelje baze podataka (Spring docs, 2016). Biti će prikazano naknadno primjerom koji prikazuje na koji način Spring olakšava njegovo korištenje.

Spring-tx podržava upravljanje programskim i deklarativnim transakcijama za klase koje implementiraju posebna sučelja i za sve POJO (eng. *Plain Old Java Object*) klase (Spring docs, 2016).

Spring-orm pruža integracijske slojeve za popularne ORM API-je uključujući JPA, JDO i Hibernate. Koristeći ovaj modul može se koristiti svaki od nabrojanih, bilo sam, bilo u kombinaciji sa drugim ORM-ovima (Spring docs, 2016).

Spring-oxm pruža apstrakcijski sloj koji podržava implementacije Objekt/XML mapiranja poput JAXB, Castora, XMLBeans, XStream itd (Spring docs, 2016).

Spring-jms sadrži mogućnosti za stvaranje i obrađivanje poruka. Također, ovaj modul podržava integraciju sa spring-messaging modulom (Spring docs, 2016).

3.2.5 Web

Web sloj sastoji se od modula *spring-web*, *spring-webmvc*, *spring-websocket*, *spring-webmvc-portlet*. *Spring-web* pruža osnovne web orijentirane funkcionalnosti poput učitavanja multipart datoteka i inicijalizaciju IoC spremnika koristeći web orijentirani aplikacijski kontekst i slušatelje Servleta. Također sadrži HTTP klijent (Spring docs, 2016).

Spring-webmvc sadrži Springov model-view-controller(MVC) uzorak i implementaciju REST web servisa za web aplikacije. Spring MVC pruža čisto odvajanje između domena modela, web formi i on se integrira sa svim drugim mogućnostima Spring programskog okvira (Spring docs, 2016).

Spring-webmvc-portlet pruža MVC implementaciju koja će biti korištena u Portlet okružju i ona preslikava funkicionalnosti spring-webmvc modula koji je zapravo temeljen na Servletima (Spring docs, 2016).

3.2.6 Test

Ovaj sloj sastoji se od *spring-test* modula koji podržava jedinične i integracijske testove Spring komponenti. Također, omogućava učitavanje *ApplicationContexta* za korištenje u

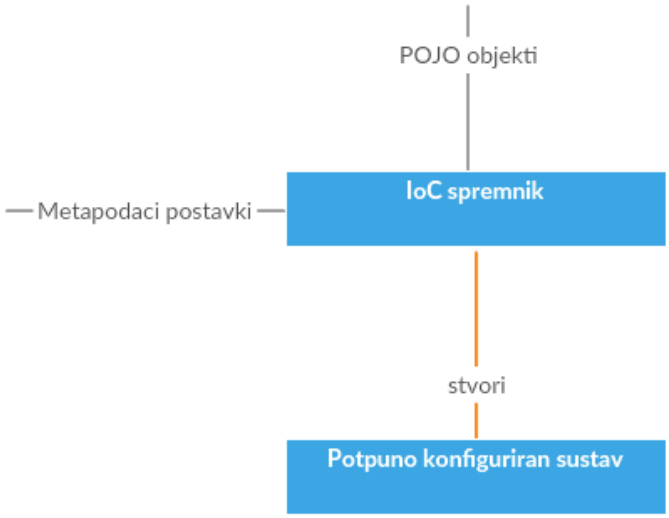
testovima i omogućuje kreiranje objekata koji oponašaju druge objekte (eng. *mock objects*) (Spring docs, 2016).

3.3 Osnovne tehnologije

Nakon što su definirani i objašnjeni moduli od kojih se Spring programski okvir sastoji, sljedeću bitnu stavku predstavljaju osnovne tehnologije koje sačinjavaju Spring programski okvir. Spring je poseban zbog svog IoC spremnika, načina korištenja zrna, načina korištenja zavisnosti itd.

3.3.1 IoC spremnik

IoC spremnik je prethodno objašnjen, ali ponoviti će se ukratko da on je odgovoran za instantaciju, postavljanje i sastavljanje zrna. Sada će biti objašnjen način na koji on to zapravo izvodi. Sljedeća slika prikazuje proces izgradnje zrna.



Slika 2. IoC spremnik (Spring docs, 2016)

Spremnik prvo prima POJO objekte i zatim učitava meta podatke (eng. *metadata*) koji se nalaze u obliku XML datoteka, anotacija ili Java programskoga koda. Nakon što se povežu POJO objekti sa meta podacima stvara i inicijalizira se *ApplicationContext*. Aplikacija je u potpunosti konfigurirana i spremna za izvođenje s završetkom procesa stvaranja *ApplicationContexta* (Spring docs, 2018 b).

Prethodno je napomenuto da zrna mogu biti kreirana kroz XML datoteke putem kojih se referencira zrno za točno određeni sloj u kojem se želi kreirati npr. sloj Data Access Object (nadalje u tekstu DAO) ima svoju XML datoteku gdje se definiraju zrna samo za taj sloj, onda ovisno o zavisnostima s drugim slojevima to zrno se može putem *Dependency Injectiona* koristiti u drugom sloju, ako taj sloj sadrži zavisnost za DAO sloj. Drugi navedeni način je putem anotacija, ali tada u XML datotekama mora biti definirano da se vrši *component-scan*, kako bi znao gdje tražiti podatke za kreiranje zrna (Walls, 2014, str. 38-40).

Kada se kreiraju zrna zapravo se kreira recept za kreiranje stvarnih instanci tog objekta utemeljenog na definiciji tog zrna. Ideja ovoga pristupa je da definicija zrna zapravo predstavlja recept po kojem se onda mogu kreirati mnoge instance isključivo koristeći tim receptom. To omogućava ne samo kontrolu raznih ovisnosti i konfiguracijskih vrijednosti koje su uključene u taj stvoreni objekt iz određene definicije zrna, nego omogućava da i opseg (eng. *scope*) zrna bude kontroliran na isti način. Takav pristup je snažan i fleksibilan zato što omogućuje biranje opsega zrna kod kreiranja kroz konfiguracija umjesto da se mora ručno kroz kod. Spring programski okvir podržava 7 opsega, od kojih 5 su dostupni ukoliko se koristi *ApplicationContext*. Također je bitno napomenuti da je moguće kreirati vlastite opsege zrna (Deinum, Serneels, Yates, Ladd i Vanfleteren, 2012).

Tablica 3. Prikaz opsega zrna

Opseg	Opis
Singleton	On je zadan. Jedna definicija zrna za jednu instancu objekta u IoC spremniku.
Prototype	Opisuje jedinu definiciju zrna koja se koristi na bilo koji broj instanci objekta.
Request	Opisuje jedinu definiciju zrna vezanu uz životni ciklus jednog HTTP zahtjeva. Svaki HTTP zahtjev ima svoju instancu.
Session	Opisuje jedinu definiciju zrna vezanu uz životni ciklus jedne HTTP sesije.
globalSession	Opisuje jedinu definiciju zrna vezanu uz životni ciklus globalne HTTP sesije. Koristi se uglavnom samo u *Portlet* kontekstu.
Application	Opisuje jedinu definiciju zrna vezanu uz životni ciklus *ServletContexta*.
Websocket	Opisuje jedinu definiciju zrna vezanu uz životni ciklus *WebSocketa*.

(Izvor: Spring docs, 2018 b)

Request, session, globalSession, application i websocket dostupni su samo kroz web-svjestan (eng. *web-aware*) ApplicationContext.

3.3.2 Resursi

Standardna Java klasa(java.net.URL) i standardni rukovatelji za različite URL prefikse jednostavno nisu adekvatni za dohvaćanje resursa niske razine. Spring predstavlja rješenje za ovaj problem u obliku sučelja pod nazivom Resource (Spring docs, 2018 b). Implementacija sučelja nalazi se ispod:

```
public interface Resource extends InputStreamSource {

    boolean exists();

    boolean isOpen();

    URL getURL() throws IOException;

    File getFile() throws IOException;

    Resource createRelative(String relativePath) throws IOException;

    String getFilename();

    String getDescription();

}
```

Ovaj skup metoda koristi se za upravljanje resursima tj. dohvaćanje, provjeravanje stanja, otvaranje resursa, dohvaćanje informacija o resursu te omogućavaju dohvaćanje stvarnog URL-a ili datoteke. Navedena apstrakcija opsežno je korištena unutar samog Springa kao tip argumenta u mnogim potpisima metoda kada je neki resurs potreban. Osim korištenja unutar Springa apstrakcija je korisna i prilikom korištenja u vlastitom kodu za pristupanje resursima.Iiako navedeni kod veže taj mali dio koda uz sami Spring iz razloga što on ovisi o malom setu pomoćnih klasa koje su koriste samo unutar Spring programskog okvira u okviru apstrakcije *Resource*. Postoji nekoliko implementacija *Resource* apstrakcije koje dolaze zapakirani unutar paketa Spring. Neke od implementacija su:

1. *ClassPathResource*

Ova klasa predstavlja resurs koji bi trebao biti dohvaćen iz putanje klasa (eng. *classpath*). Putanja klasa je mjesto na kojem će Java virtualni stroj pronaći kompilirane klase. *ClassPathResource* koristi učitavač klasa dretvinog konteksta (eng. *thread context class loader*), određeni učitavač klasa ili određenu klasu za učitavanje resursa. Koristi se kada je resurs podržan od strane *java.io.File* klase i nalazi se unutar datotečnog sustava, ali ne i za one koji se nalaze unutar Java ARchive datoteke (nadalje u tekstu JAR). Da bi taj problem bio riješen razne implementacije *Resource* sučelja uvijek podržavaju java.net.URL klasu.

2. *FileSystemResource*

Ovo je implementacija za upravljanje java.io.File resurse. Podržava ovo rješenje za datoteke i za URL.

3. *ServletContextResource*

Ovo je implementacija za resurse *ServletContexta*, tj. za tumačenje relativnih puteva unutar odgovarajućeg korijenskog direktorija web aplikacije. Uvijek podržava pristup putem streama i pristup putem URL-a, ali dopušta *java.io.File* pristup samo kada se arhiva web aplikacija proširuje i resurs je fizički na datotečnom sustavu. Bez obzira na to je li proširen, je li na datotečnom sustavu, pristupa mu li se izravno iz JAR-a ili negdje drugdje poput baze podataka.

4. InputStreamResource

Implementacija za određeni *InputStream*. Preporučeno je da se koristi samo u slučaju da je jedini izbor. Ovo je zapravo opisnik za već otvorene resurse.

5. *ByteArrayResource*

Implementacija za dani niz bajtova, zapravo kreira *ByteArrayInputStream* i koristan je za učitavanje sadržaja iz bilo kojeg niza bajtova, bez potrebe da se koristi jednokratni *InputStreamResource*.

(Spring docs, 2018 b)

3.3.3 Validacija, povezivanje podataka i pretvorba tipova

Validacija, povezivanje podataka i pretvorba tipova (eng. *Validation, Data Binding and Type Conversation*). Postoje prednosti i mane vezane za procjenu valjanosti kao poslovnu logiku, a Spring programski okvir nudi dizajn za provjeru valjanosti i povezivanje podataka koji ne isključuje niti jedno od njih. Konkretno, provjera valjanosti ne bi trebala biti vezana uz mrežni sloj, ona bi trebala biti jednostavna za lokalizaciju i trebalo bi biti moguće priključiti bilo koji dostupni Validator. Stoga unutar Spring programskog okvira Validator se može koristiti u bilo kojem sloju aplikacije. Sučelje Validator u svome radu koristi *Errors* objekte, tako da prilikom procjene valjanosti može izvještavati o neuspjesima *Errors* objektima. Navedeni postupak izvodi se unutar metode *validate*(Object o, Errors e). Prilikom korištenja *Validatora* jako je korisno poznavanje *ValidationUtils* klase (Spring 2018, b).

Povezivanje podataka je korisno za omogućavanje da korisnički unos bude dinamički vezan s modelom domene aplikacije. Spring programski okvir pruža *DataBinder* i *Validator*. Oni zajedno čine validacijski paket koji se prvenstveno koristi unutar MVC okvira, ali nije ograničen izričito na njega. Možda i najbitnija klasa je sučelje *BeanWrapper* i njegova implementacija *BeanWrapperImpl*, iako potreba za njegovim direktnim korištenjem je rijetka. Ona pruža mogućosti dohvaćanja i formatiranja vrijednosti koje se nalaze u postavkama.

BeanWrapper izvodi navedene akcije na način da zapakira zrno, kako bi mogao na tom zrnu izvoditi akcije (Spring 2018, b).

Spring programski okvir pruža paket core.convert koji pruža sustav za opći tip pretvorbe. Sustav pruža Service Provider Interface (nadalje u tekstu SPI) koji se koristi za definiranje proširenja i prilagodbi uglavnom na razini programskih okvira, te on omogućava implementaciju logike pretvorbe tipova (Spring docs, 2018 b). Kreiranje vlastitih pretvarača je vrlo jednostavno. Potrebno je implementirati sljedeće sučelje:

```
package org.springframework.core.convert.converter;

public interface Converter<S, T> {

    T convert(S source);

}
```

Parametar S predstavlja izvor, a parametar T predstavlja cilj, to znači da parametar S se predstavlja u parametar T prilikom korištenja ovog pretvarača. Osim ovog generičkog pretvarača koji se koristi za kreiranje vlastitih postoji nekoliko unaprijed definiranih implementacija u paketu *core.convert.support*. Neki od njih su *StringToInteger*, *StringToEnum* itd. Osim prethodno navedenog načina postoji još i *ConversationService* koji se nalazi u paketu *core.convert.support*. On je zamišljen kao objekt bez stanja koji je dizajniran da bude instanciran na samom pokretanju aplikacije i zatim bude dijeljen od strane više dretvi. U Spring aplikacijama najčešće se može vidjeti da je *ConverterService* instanca postavljena po jednom Spring spremniku, zatim Spring će je uočiti i koristiti svaki puta kada unutar okvira je potrebno neki oblik pretvaranja. Također moguće je i ručno ubrizgavanje u vlastita zrna (Spring docs, 2018 b).

3.3.4 Aspektno orijentirano programiranje

Aspektno orijentirano programiranje sjajno nadopunjuje objektno-orijentirano programiranje pružajući drugi način razmišljanja o strukturi samoga programa. Glavna stavka modularnosti u objektno-orijentiranom programiranju je klasa, dok u aspektno orijentiranom programiranju glavna stavka modularnosti je aspekt. Aspekti omogućuju modularizaciju problema kao upravljanje transakcija koji se šire preko većeg broja tipova i objekata. Stoga AOP se može smatrati kao jedna od ključnih komponenti samoga Spring programskog okvira zato što jako dobro nadopunjuje IoC spremnik kako bi pružio vrlo sposobno među posredno (eng. *middleware*) rješenje. Spring prati AOP terminologiju, kako bi se smanjila razina zbunjenosti (Walls, 2014, str. 97-105).

Pristup prilikom razvoja AOP u Springu razlikuje se od pristupa razvoju drugih AOP okvira. Glavni razlog tomu nalazi se u cilju postavljenom prilikom razvoja Spring programskog okvira. Cilj je pružiti blisku integraciju AOP implementacija i IoC spremnika, dok drugi okviri uglavnom ciljaju kako bi napravili što potpuniju AOP implementaciju. Aspekti u Springu su uglavnom postavljeni korištenjem sintakse za standardno definiranje zrna i to zapravo predstavlja ključnu razliku u odnosu na druge AOP implementacije. Isto tako, postoje neke stvari koji su jako teške za napraviti ili jednostavno nisu učinkovite koristeći Springovu AOP implementaciju. Stvari poput definiranja sitno zrnatih objekata i u tom slučaju preporuča se korištenje *AspectJ* okvira. Prije je napomenuto kako je jedan od glavnih temelja Springa mogućnost samostalnog odabira modula i zavisnosti. Stoga Spring pruža mogućnost izbora između anotacijski vođenog AspectJ okvira ili Springovog XML pristupa. Moguće je koristiti *AspectJ* putem anotacija ili XML datoteka, isto tako moguće je koristiti i Springov AOP okvir koristeći *AspectJ* anotacijski stil ili putem XML datoteka. (Spring docs, 2018 b)

3.4 Verzije i vrste Spring programskog okvira

Spring je nastao 2003. godine i od tada do trenutka pisanja ovoga rada objavljeno je 5 velikih verzija Springa. Svaka verzija uglavnom dolazi i prati novu verziju Jave i donosi velik broj poboljšanja. Bilo u obliku poboljšanja učinkovitosti, brzine ili dodavanja novih funkcionalnosti (Spring, 2004).

3.4.1 Verzije Spring programskog okvira

Spring 2.0 je donio brojna poboljšanja poput dodavanja novih opsega zrna. Spring 1.0 imao je samo Singleton i Prototype, a u 2.0 verziji njihova implementacija je promijenjena na način da na strani korisnika ne bi trebalo biti nikakvih promjena u načinu korištenja. Također dodana je mogućnost implementacije vlastitih opsega, dodana je implementacija Portlet okvira koji je konceptualno sličan Spring MVC okviru. Što se tiče AOP olakšana je njegova konfiguracija i dodana je podrška za AspectJ anotacije (Spring docs, 2006).

Spring 3.0 je temeljen na Javi 5, ali u potpunosti podržava Javu 6. Bitna stavka ove verzije je dodavanje Spring ekspresijskog jezika i brojna poboljšanja unutar IoC spremnika. U ovoj verziji dodana je podrška za sustav pretvorbe tipova, također dodano je asinkrono MVC obrađivanje na Servletu 3.0 i omogućeno je učitavanje *WebApplicationContexta* u testovima (Spring docs, 2009).

Spring 4.0 temeljen je na Javi 7, ali u potpunosti podržava verzije 6 i 8, s time da je Java 6 izbačena iz podrške nakon verzije 4.3. Bitnija poboljšanja su dodavanje modula spring-websocket i spring-messaging. Također napravljena su brojna poboljšanja u modulu *spring-test* s naglaskom na čišćenje postojećeg koda (eng. *legacy code*). Predstavljena su brojna poboljšanja u IoC spremniku, web modulu itd. Dodan je Groovy Bean Definition DSL koji predstavlja novi način za definiranje vanjskih zrna (zapravo predstavlja jako sličan koncept kao korištenje XML-a za definiranje zrna, ali dopušta sažetiju sintaksu) (Spring docs, 2013).

Spring 5.0 je zadnja verzija koje je izašla i ona je temeljena na Javi 7 i 8 te podržava Javu 9. Osim poboljšanja u osnovnom spremniku i brojnih poboljšanja u Spring Web MVC okviru dodan je novi okvir pod nazivom Spring WebFlux. Navedeni okvir predstavlja alternativu za Spring MVC i izgrađen na reaktivnoj podlozi. Jedna od novo dodanih stavki je podrška za programski jezik Kotlin i podrška za HTTP/2 (Spring docs, 2016). Također jedna od bitnih stavki da je cijeli Spring okvir prepisan u potpunosti da koristi najbolje značajke Jave 8 (Github, 2018).

3.4.2 Vrste Spring web okvira

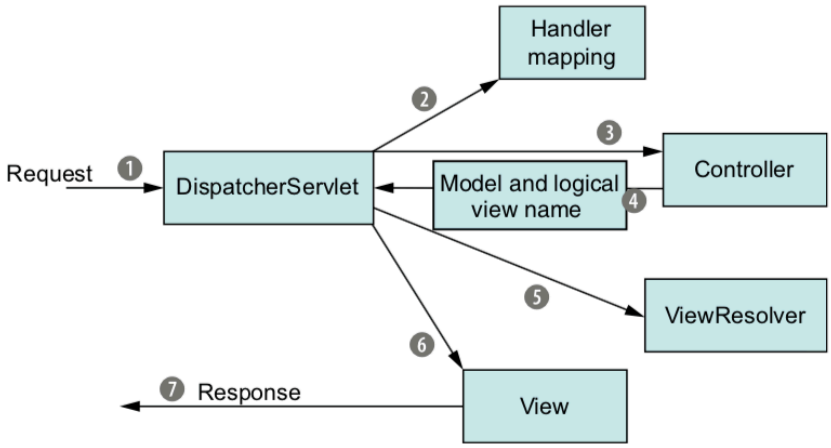
Nakon zadnje verzije Springa postoje tri vrste Spring web okvira za razvoj aplikacija. Originalna verzija web okvira korištenog u Spring programskom okviru zove se Spring Web MVC ili skraćeno Spring MVC, druga verzija nastala je u Springa 5.0 verziji i poznata je pod nazivom Spring WebFlux. Razlika između navedenih okvira je što je Spring MVC temeljen na Servletima, a Spring WebFlux je temeljen na reaktivnoj podlozi. Zadnja verzija Spring web okvira je Spring Boot, koja je vrlo slična Spring MVC okviru, ali pojednostavljena za korištenje i početno postavljanje. (Spring docs, 2016).

3.4.2.1 Spring MVC

Spring MVC postoji od samih početaka razvoja Spring programskog okvira, naziv je dobio na temelju *spring-webmvc* modula u kojem se nalazi. Razvijena je na Servlet APIu i dizajniran je oko uzorka *Front controller* gdje se u centru svega nalazi *Servlet*. Točnije rečeno u slučaju Springa *DispatcherServlet*, on pruža zajednički algoritam za obradu zahtjeva dok se stvarni zadaci obavljaju u podesivim komponentama. Ovaj model je fleksibilan i podržava različite radne tokove (eng. *diverse workflows*). *Front controller* je česti uzorak koji se koristi u web aplikacijama gdje je jedan *Servlet* korišten za upravljanje odgovornostima zahtjeva koji se prosljeđuju drugim komponentama aplikacije. (Spring docs, 2018 a)

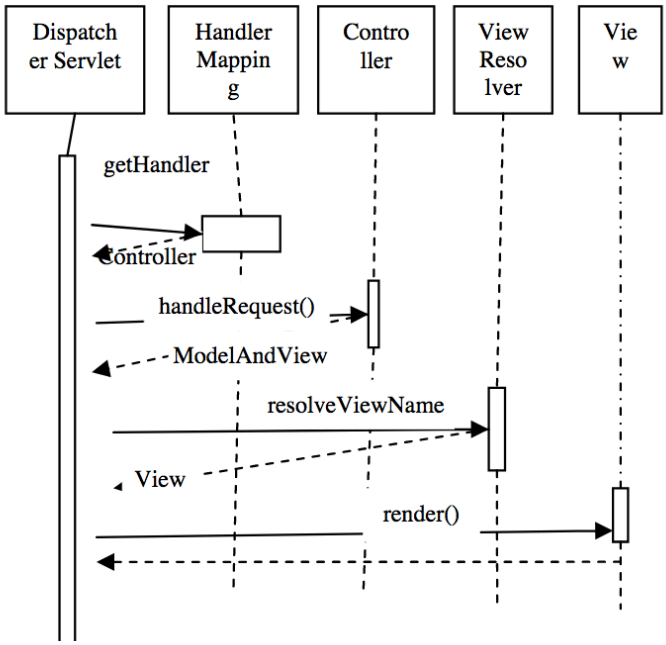
Spring MVC se sastoji od tri komponente koje surađuju. Prva komponenta je Model, druga komponenta controller, dok treća komponenta naziva se view. *Controller* je komponenta koja služi za obrađivanje zahtjeva tj. za upravljanje navigacijskom logikom i ona djeluje sa slojem servisa u kojem se nalazi poslovna logika. U aplikacijama je predviđeno postojanje nekoliko *controllera*, zbog toga *DispatcherServlet* treba pomoć oko odlučivanja kojem controlleru treba proslijediti zahtjev. Za navedeni problem koristi mapiranje rukovatelja koji na temelju URL adrese (koja se nalazi unutar zahtjeva) određuje kojem *controlleru* treba proslijediti zahtjev. Kada je *controller* odabran *DispatcherServlet* prosljeđuje zahtjev odabranom *controlleru*, a on bi nakon toga trebao obraditi podatke (dobro dizajnirani *controlleri* bi trebali imati malo ili nemati uopće poslovnu logiku, ona bi zapravo trebala biti obavljena unutar jednog ili više servisa). Logika obavljena putem *controllera* često rezultira informacijama koje bi trebale biti vraćene nazad i prikazane korisniku. Te dobivene informacije nazivaju se modelom i one korisniku ne mogu biti prikazane korisniku u svom izvornom obliku nego bi trebale biti formirane na način da budu čitke i jasne. Tipično, u Javi se koriste Java Server Pages (nadalje u teksu jsp) datoteke i one zapravo predstavljaju *view* putem kojeg je prikazan sadržaj korisniku. Model zapravo reprezentira i upravlja podacima, a *view* se koristi za prikaz istih (Walls, 2014, str. 132-162).

Jedna od posljednjih stvari koje controller mora obaviti je pripremiti podatke dobivene iz modela i dohvatiti naziv viewa na kojem bi trebao biti prikazan sadržaj. Kada je to odradio šalje podatke i naziv viewa nazad DispatcherServletu. Controller nije vezan na određeni view, taj naziv viewa koji je vraćen DispatcherServletu ne identificira direktno određenu datoteku na kojoj bi sadržaj trebao biti prikazan. On zapravo prenosi samo logičko ime koje će biti korišteno prilikom traženja odgovarajućeg viewa za prikaz. Za odabir viewa DispatcherServlet koristi odlučitelj pogleda (eng. *view resolver*) za mapiranje naziva viewa na specifičnu implementaciju. Nakon što je DispatcherServlet saznao na kojem će viewu biti prikazan rezultat njegov posao je skoro gotov. Preostalo mu je još proslijediti podatke dobivene iz modela do viewa, koji će zatim iskoristiti dobivene podatke za prikaz rezultata korisniku. Ovaj kratki opis sastoji se zapravo od 7 koraka koji su vidljivi na slici (Walls, 2014, str. 132-162). Ovakva implementacija omogućuje biranje između tehnologija prikaza i na taj način nije usko vezana na niti jednu.



Slika 3. Tijek zahtjeva u Spring MVC okviru (Walls, 2014, str.132)

Na sljedećoj slici moguće je vidjeti isti proces, ali on je prikazan koristeći dijagram slijeda koji omogućuje jasno vidljiv redoslijed i način izvođenja koji je prethodno opisan.



Slika 4. Dijagram slijeda za tijek zahtjeva u Spring MVC okviru (Gupta, 2014)

Ovaj dijagram se sastoji od Spring MVC komponenti. To su osnovne komponente koje se koriste prilikom procesiranja zahtjeva u Spring programskom okviru. Sve spomenute komponente su složene i sastoje se od velikog broja raznih adaptera, ali je to preopširan pojam za objasniti u ovome radu.

Stoga za zaključak *DispatcherServlet* predstavlja ključnu komponentu u Spring MVC okviru, on u svom radu koristi velik broj različitih komponenti. Glavne su *HandlerMapping* i *HandlerAdapter* koji predstavljaju ključne komponente prilikom mapiranja i obrađivanja zahtjeva. MVC uzorak u Springu je poprilično logičan odnosno controller procesuira zahtjeve, popunjava model koristeći servise i odabire koji view treba prikazati (Walls, 2014, str.132).

3.4.2.2 Spring WebFlux

Suvremene IT poslovne potrebe su se značajno promijenile u odnosu na prethodnih nekoliko godina. Količina podataka koji se generiraju iz različitih izvora, kao što su web stranice društvenih medija, IoT uređaja i slično je ogromna. Tradicionalni modeli obrade podataka više neće biti prikladni za obradu tolikih količina podataka. Iako postoji bolja hardverska podrška ovih dana, mnogi postojeći API-ji su sinkronizirani i blokirajući, čime postaju uska grla za bolju propusnost. To je razlog nastanka reaktivnog programiranja koje promovira asinkroni, ne blokirajući i pristup vođen događajima prilikom obrade podataka. Reaktivno programiranje u trenutku pisanja dobiva zamah i mnogi programski jezici i okviri kreću s pružanjem reaktivnih okvira i biblioteka. Stoga Spring kao vodeći okvir za razvoja web aplikacija u Javi, morao je napraviti taj korak naprijed i isto to je učinio izlaskom verzije 5.0 (Blackheath i Jones, 2016).

Spring 5.0 predstavlja Spring WebFlux web okvir koji je potpuno asinkron i ne blokirajuć, namijenjen za korištenje u modelu izvedbe događaja u obliku petlje (eng. *event-loop execution model*). Nalazi se unutar modula spring-webflux koji pruža podršku za stvaranje reaktivnih poslužiteljskih aplikacija kao i reaktivnih klijentskih aplikacija koristeći REST servise, HTML preglednike i komunikaciju putem WebSocketa. Podržava anotacije iz Spring MVC okvira sa jednom bitnom razlikom. *HandlerMapping* i *HandlerAdapter* koje on koristi su ne blokirajući i koriste *ServletRequest* i *ServletResponse*, za razliku od Spring MVC koji koristi *HttpServletRequest* i *HttpServletResponse* (Reddy, 2017, str. 159-160).

Spring WebFlux koristi *Reactor* biblioteku za reaktivno programiranje. Biblioteka koja omogućuje Mono i Flux API koji mogu raditi na sekvencama podataka od 0..1 i 0..N kroz bogat skup operatora usklađenih sa *ReactiveX* rječnikom operatora. *Reactor* je biblioteka reaktivnih tokova i zato svi njegovi operatori podržavaju ne blokirajući pristup. Bitna stavka kod *Reactora* je što njegov razvoj teče u bliskoj suradnji sa Springom. WebFlux koristi *Reactor* kao ključnu zavisnost, ali on je interoperabilan sa drugim reaktivnim bibliotekama preko reaktivnih tokova (Reddy, 2017, str. 160-179).

Reaktivni tokovi su inicijativa za pružanje standarda za asinkronu obradu tokova sa ne blokirajućim stražnjim pritiskom (eng. *non-blocking back pressure*). Sastavnice reaktivnih tokova su sučelja *Publisher* i *Subscriber*.

Izdavač je pružatelj neograničenog broja slijednih elementa, koji se objavljuju prema zahtjevima primljenih od pretplatnika.

```
public interface Publisher<T> {
    public void subscribe(Subscriber<? Super T> s);
}
```

Pretplatnik preplaćuje se izdavačima za povratne pozive. Izdavači ne šalju automatski podatke pretplatnicima, nego samo u slučaju da se pretplatnici preplate i zatraže podatke.

```
public interface Subscriber<T> {
    public void onSubscribe(Subscription s);
    public void onNext(T t);
    public void onError(Throwable t);
    public void onComplete();
}
```

Nakon što su objašnjeni osnovni koncepti pojasniti će se i kako su oni izvedeni u *Reactoru*. U Spring WebFluxu, odnosno u *Reactoru* postoje dva sučelja koji zapravo reprezentiraju izdavača i pretplatnika, a to su Flux i Mono. Flux<T> koji predstavlja standardno sučelje Publisher<T> koje je zapravo asinkroni niz od 0 do N emitiranih elemenata, a Mono predstavlja specijalizirano sučelje Publisher<T> koje emitira najviše jedan element i onda opcionalno završava sa signalima onComplete ili onError. Mono se također može koristiti za predstavljanje bez vrijednosnih asinkronih procesa koji vraćaju Mono<Void>. U sljedećem isječku nalazi se kratki primjer njihove uporabe (Spring docs, 2017).

```
Mono<String> mono = Mono.just(„Spring“);
Mono<String> mono = Mono.empty();

Flux<String> flux = Flux.just(„Spring“, „Design patterns“, „Reactor“);
Flux<String> flux = Flux.fromArray(new String[]{„Spring“, „Reactor“});
Flux<String> flux = Flux.fromIterable(Arrays.asList(„Spring“,“Reactor“));

Flux.log().subscribe();
```

Zadnja linija koda u isječku predstavlja prijavu i pretplatu na pretplatnika kako bi mogli primati podatke.

WebFlux kao programski okvir zapravo je jako sličan Spring MVC okviru. Kao i MVC radi preko uzorka Front controllera. *DispatcherServlet* koristi istu logiku za procesuiranje zahtjeva, iste anotacije se koriste za logiku korištenja controllera, iste tehnologije koristi kao Spring MVC za prikaz pogleda. Rezultat ovog načina rada je mogućnost istovremenog postojanja oba okvira u jednoj aplikaciji. Jedina bitna razlika u odnosu na MVC okvir je zbog

dizajna Servlet API-ja. Servlet API originalno je dizajniran za slijednu obradu zahtjeva. Asinkrona obrada zahtjeva dodana je u Servletu 3.0 i ona omogućuje aplikacijama da izađu iz Filter-Servlet lanca, ali time ostavljaju odgovor otvorenim čime se narušava princip dizajna ovog modela koji glasi da bi trebala biti jedna dretva po zahtjevu (Spring docs, 2017).

Spring WebFlux uopće nije svjestan Servlet API, ali on mu nije ni potreban pošto je WebFlux po dizajnu asinkron. Obrađuje svaki zahtjev u fazama umjesto da čini jedan prolaz kroz pozivni stog u jednoj dretvi. To zapravo znači da je asinkrono rukovanje ugrađeno u sve okvirne ugovore i stoga je u potpunosti podržan u svim fazama obrade zahtjeva (Spring docs, 2017).

Zapravo oba Spring web okvira podržavaju asinkrone i reaktivne tipove za povratak vrijednosti iz metoda controllera. Spring MVC podržava i strujanje uključujući reaktivni stražnji pritisak, međutim pojedinačni zapisi za odgovor ostaju blokirajući (izvode se u zasebnoj dretvi) i to predstavlja ključnu razliku u odnosu na WebFlux koji je ne blokirajući. Još jedna temelja razlika između okvira je što Spring MVC ne podržava asinkrone ili reaktivne tipove među argumente u metodama controllera i nema eksplicitnu podršku za asinkrone i reaktivne tipove kao atribute modela, a te dvije navedene stavke Spring WebFlux podržava (Spring docs, 2017).

3.4.2.3 Spring Boot

Spring Boot olakšava stvaranje samostalnih aplikacija koje se temelje na Spring proizvodima. Spring razvojni inženjeri prilikom razvoja Spring Boota uzeli su subjektivni pogled na Spring okvir i biblioteke treće strane. Stoga su uključili biblioteke i module koje su smatrali bitnima i omogućili nikad lakše pokretanje Spring aplikacija. Spring Boot aplikacije zahtijevaju vrlo malo konfiguracije u odnosu na tradicionalni Spring programski okvir i aplikacija se može pokrenuti doslovno u par minuta (Spring.io, 2018).

Spring programski okvir zbog načina razvoja i načina na koji je zamišljen ima jedan problem. Problem je što je potrebna velika količina konfiguracije za vrlo jednostavne aplikacije. Recimo na primjeru „*Hello world*“ aplikacije potrebno je jako puno konfiguracije kako bi se ispisala jednostavna „*Hello world*“ poruka. Zbog toga je nastao *Spring Boot* okvir koji predstavlja rješenje za navedeni problem. Zamišljen je kao okvir koji će riješiti problem početnog podešavanja i automatski umjesto korisnika odabrati početne komponente koje je kasnije moguće nadjačati, ukoliko je to potrebno (Spring docs, 2018 c).

Zavisnosti *spring-boot-starter-** je skup zavisnosti koje su automatski uključene u Spring Boot projekt. Taj paket sadrži standardne biblioteke koje se koriste u razvoju Spring aplikacija, biblioteke poput *spring-webmvc*, *jackson-json*, *validation-api* i *tomcat*. Navedena zavisnost osim dodavanja navedenih biblioteka sama postavlja standardno korištena zrna poput *DispatcherServlet*, *ResourceHandler* itd. (Reddy, 2017, str. 19).

Isto tako omogućuje nikad lakše pokretanje aplikacije na način da se pokrene klasa koja koji ima metodu main() i anotirana je anotacijom @SpringApplication. Dovoljno je pokrenuti main() metodu da bi se aplikacija pokrenula i moguće joj je onda pristupiti na http://localhost:8080/. U sljedećem isječku koda prikazana je klasa za pokretanje:

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

To je moguće isključivo zato što unutar zavisnosti *spring-boot-starter-web* se automatski nalazi spring-boot-starter-tomcat koji prilikom pokretanja automatski pokreće *Tomcat* poslužitelj kao ugrađeni spremnik, tako da korisnici ne moraju samostalno postaviti vanjsku instancu *Tomcat* poslužitelja. Ukoliko bi npr. umjesto *Tomcat* poslužitelja htjeli koristiti Jetty poslužitelj potrebno je isključiti spring-boot-starter-tomcat iz spring-boot-starter-* zavisnosti i uključiti spring-boot-starter-yetty (Reddy, 2017, str. 19).

Spring Boot okvir ne predstavlja razliku u odnosu na Spring MVC i Spring WebFlux nego omogućuje jednostavnije pokretanje i razvijanje aplikacija na način da uklanja zadatke vezane uz konfiguraciju sa razvojnih inženjera i odrađuje automatski koristeći predefinirane recepte za instalaciju zavisnosti koje su obavezne za rad Spring aplikacija. Pomoću Spring Boota moguće je stvoriti Spring MVC aplikacije, Spring WebFlux aplikacije i aplikacije koje kombiniraju oba okvira zato što su oni poprilično kompatibilni kao što je objašnjeno u prethodnim poglavljima.

Tablica 4. Verzije Springa

	Spring MVC	**Spring WebFlux**
Temelj	Servlet API	Adapteri reaktivnih tokova
Spremnici	Tomcat, Netty, Servlet spremnik	Tomcat, Netty, Servlet 3.1+ spremnik
Repozitoriji	JDBC, JPA, NoSQL	Mongo, Cassandra, Redis, Couchbase
Blokiranje	Blokirajući	Ne blokirajući
Sinkronost	Sinkron / asinkron	asinkron

(Izvor: Spring.io, 2018)

U tablici iznad prikazane su dvije verzija Springa koje se oslanjaju na različite aspekte. U tablici ne nalazi se Spring Boot, zato što može biti podešen da koristi bilo MVC ili WebFlux stoga nema smisla uspoređivati ga s njima. Prethodno je napomenuto kako je glavna razlika između MVC i WebFluxa zapravo u obradi zahtjeva, MVC ih obrađuje slijedno i ukoliko dođe do preduge obrade jednog od njih dolazi do usporavanja u obradi zahtjeva, a kod Webfluxa nije moguće zbog načina na koji je dizajniran.

4. Usporedba Java programskih okvira

Java programski okviri posljednjih godina nalaze se među naj rasprostranjenijim okvirima za izradu web aplikacija za poduzeća u zadnjih par godina. Definitivno koncept MVC-a je jedan od razloga tolike primjene, zato što on pruža brojne prednosti poput standardiziranja aplikacijske strukture, smanjuje potrebno vrijeme i trud prilikom razvoja. Međutim, nakon godina evolucije, izumljeni su brojni Java web okviri s različitim fokusima, razvojnim inženjerima postaje sve teže odabrati odgovarajući okvir za svoje web aplikacije. Neki od najpopularnijih Java programskih okvira vidljivi su sljedećoj tablici. Rezultati popularnosti Java Web okvira, dobiveni su kombinacijom javnih podatka iz aplikacija *StackOverflow*, *LinkedIn*, *GitHub* i *Google Search*. Analizirane stavke daju približan poredak popularnosti na temelju učestalosti korištenja i spominjanja okvira na popularnim uslugama (ZeroTurnaround, 2018).

Tablica 5.Popularnost Java programskih okvira

#	Programski okvir	Popularnost (%)
1.	Spring MVC	28,82
2.	JSF	15,20
3.	Spring Boot	13,35
4.	GWT	7,74
5.	Grails	6,35
6.	Struts	5,40
7.	Dropwizard	4,90
8.	Play	3,26
9.	JHipster	2,49
10.	Vaadin	2,15

(Izvor: ZeroTurnaround, 2018)

Prema rezultatima ankete vidljivo je da je Spring neprikosnoveni vladar među Java programskim okvirima. U top 10 programskih okvira nalaze se 2 Spring programska okvira i njihova ukupna korištenost iznosi 42,17%. Skoro pola anketiranih ljudi koji razvijaju aplikacije za poduzeća koristeći Javu koriste Spring kako bi isporučili traženi proizvod. Također treba napomenuti kako taj postotak raste iz godine u godinu. Spring Boot dobiva sve više na značaju, a zašto ga dobiva biti će objašnjeno u ovome poglavlju. Osim Spring Boota, Dropwizarda i JHipstera zastupljenost ostatka programskih okvira je u nekoj vrsti pada kroz 2017. godinu. U ovome radu analizirati će se prednosti i nedostatci 5 programskih okvira koji su trenutno najkorišteniji. Analiza će biti izvršena na temelju nekoliko parametara (Koblentz, 2018).

4.1 Spring MVC

1. Mogućnost brzog razvoja aplikacija

Spring programski okvir definitivno nije prikladan odabir ukoliko je potrebno brzo i čisto generiranje kostura aplikacija. On je opsežan i težak za shvatiti početnicima te iako postoji opcija preuzimanja *Petclinic* paketa koji uklanja sve nepotrebnosti i dalje predstavlja problem početnicima prilikom postavljanja projekata. Stoga Spring programski okvir nije dobar izbor za brzo razvijanje aplikacija, zbog velike količine vremena i truda koje je potrebno uložiti za postavljanje projekta.

2. Jednostavnost korištenja

Spring je opsežan i zbog toga nije ga lako odmah primijeniti, nego je potreban određen period privikavanja. Kako bi se iskoristile njegove prednosti, potrebno je znati kako Spring radi kao cjelina. Springova primjena leži u izgradnji ozbiljnih aplikacija sa čvrstim temeljima, bogatim korisničkim sučeljima i API-jem, što ga čini potpuno nepotrebnim za sve jednostavnije aplikacije.

3. Dokumentacija

Činjenica da su najkorišteniji Java programski okvir definitivno ima svoje prednosti. Postoje ogromne količine podataka o primjeni i radu Spring okvira da bi se moglo napraviti čitavo poglavlje koje bi bilo posvećeno prikazu i analizi dostupnih resursa. Web stranica koja pripada Spring programskom okviru sama po sebi pruža brojne vodiče bilo u obliku video sadržaja ili u pisanom obliku. Korisna je za početnike, ali može poslužiti i naprednim korisnicima. Također na svojoj stranici postoji blog koji se sastoji od sadržaja pisanih od strane zajednice i od članova razvojnog tima koji radi na razvoju Springa.

4. Skalabilnost

Kod Spring MVC aplikacije podrazumijeva se da će u jednom trenu doći do skaliranja aplikacije, isključivo zato što primjena Springa leži u razvoju velikih i opsežnih aplikacija koje će se koristiti širom svijeta. Sadrži neophodne komponente za paralelnu obradu i rješenja poput *EhCachea* koji se mogu lako prilagoditi za skaliranje memorije. Spring Batch omogućava izgradnju aplikacija s više dretvi, aplikacija za particiju i skupnu obradu (eng. *multi-threaded, partition applications and do bulk processing*). Spring aplikacije mogu se modularizirati i različiti moduli mogu biti postavljeni na različitim okruženjima (produkcija, testno okruženje ili kontinuirana integracija.

5. Korisničko iskustvo

Spring MVC ima vrlo bogat skup mogućnosti za razvoj i održavanje koda na strani poslužitelja, ali unatoč svojoj opsežnosti još uvijek ne pruža nikakav bogat okvir za izgradnju lijepih i jednostavnih sučelja. Bez obzira na to moguće je upotrijebili bilo koji drugi okvir namijenjen za izgradnju korisničkog sučelja (npr. React, AngularJS).

4.2 JSF

1. Mogućnost brzog razvoja aplikacija

JSF nije sjajan izbor za brz razvoj aplikacija. Mogućnost generiranja koda nije ugrađena i razvoj prototipa aplikacije zahtijeva jednaku konfiguraciju kao i prava aplikacija. To zapravo nije pogreška JSF-a, jer on se oslanja na Java EE specifikaciju. JSF ima nekoliko korisnih *Maven* ovisnosti koji pružaju dobru polaznu točku za osnovne aplikacije. Najkorisnija stvar za produktivnost samog JSF programskog okvira su čarobnjaci dostupni u većini razvojnih okolina koji generiraju većinu koda i konfiguracije.

2. Jednostavnost korištenja

JSF nastoji pružiti okvir jednostavan za korištenje i za stvaranje ponovljivih web komponenti u Java aplikacijama bez velike krivulje učenja. JSF je vrlo jednostavan za pokretanje i često ne zahtijeva dodatna preuzimanja ili konfiguraciju jer potreban kod u nalazi se u paketu u bilo kojem aplikacijskom poslužitelju kompatibilnom s Java EE. Može ponekad biti pomalo zbunjujući, isto tako uključujući i Java EE, čije nevjerojatno korisne značajke se mogu činiti tupim ili su zamagljene lošom dokumentacijom.

3. Dokumentacija

JSF je za različit u odnosu na sve obrađene okvire iz jednog glavnog razloga, a to je da je u potpunosti podržan i ima referentnu implementaciju od strane Oraclea. JSF ima vrhunsku zajednicu s Oracleom na čelu, koji plaćaju zaposlenicima za pisanje dokumentacije i stvaranje jednostavnih primjera. Iako JCP (eng. *Java Community Process*) određuje Java EE specifikaciju, Oracle ima velik utjecaj u određivanju i podržavanju značajki prisutnih u specifikaciji. Dokumentacija za JSF često je specifična za određena prilagođena okruženja, a odlaskom sa preporučene konfiguracije često dovodi do frustracije za Java razvojne inženjere, isto tako nažalost većina Oracleove dokumentacije za JSF izgrađena je oko Oracleove referentne implementacije vezane uz Java EE.

4. Skalabilnost

JSF aplikacije mogu biti učinkovite i brze, ali poboljšanja u performansama dolaze iz grupiranja Java EE aplikacijskih poslužitelja. JSF sam kao takav ne pruža eksplicitnu podršku za asinkrone pozive i oslanja se na Java EE specifikaciju za pružanje @Asynchronous i @Schedule anotacija na poslovnu logiku EJB (eng. *Enterprise Java Beans*) modula.

5. Korisničko iskustvo

JSF je odličan za razvojne timove koji žele stvoriti vrhunska korisnička sučelja bez da postanu JavaScript stručnjaci. U katalogu se nalaze mnoge odlične komponente, a postoje i sjajni dodatci JSF-u kako bi se omogućile komponente još više kvalitete. IceFaces i PrimeFaces su dva primjera sjajnih dodataka JSF katalogu.

4.3 Spring Boot

1. Mogućnost brzog razvoja aplikacija

Spring programski okvir pruža fleksibilnost za konfiguriranje zrna na više načina koristeći XML, anotacije i JavaConfig. S brojem mogućnosti također se povećava i složenost. Samim time postavljanje Spring aplikacija postaje dosadan pristup koji je sklon pogreškama. Zbog toga, napravljen je Spring Boot programski okvir i on radi točno ono što je potrebno, a to je da će automatski postaviti projekt. Istovremeno omogućuje poništavanje zadanih postavki ukoliko je to potrebno. Zbog toga u roku par sati moguće je napraviti jednostavnu aplikaciju za stvaranje, brisanje i ažuriranje (eng. *create, update, delete*, nadalje CRUD) podataka, dok sa Spring MVC okvirom to jednostavno nije moguće. Kreiranje projekta jednostavno je za napraviti koristeći Spring Initalizr dodatak gdje je potrebno odabrati zavisnosti koje će se koristiti u aplikaciju, naravno naknadno je moguće te zavisnosti mijenjati.

2. Jednostavnost korištenja

Prilikom kreiranja projekta Spring Boot ima *Spring Application* klasu koja omogućava jednostavno pokretanje aplikacije, ukoliko se aplikacija ne uspije pokrenuti analitičari neuspjeha (eng. *failure analyzers*) nude konkretne radnje za rješavanje problema. Spomenuti način rješava jedan od najvećih problema Spring MVC okvira i on omogućuje brzi razvoj aplikacija. Sve prethodno postavljene postavke mogu se prilagođavati potrebama projekta, tako da Spring Boot sadrži skoro pa jednake mogućnosti kao Spring MVC programski okvir uz jednostavnije korištenje.

3. Dokumentacija

Spring Boot dokumentacija jednako je kvalitetna kao i za Spring MVC. Dokumentacija je pisana na isti način, koriste se slični primjeri, također ima prikaze i primjere kako i na koji način integrirati razne biblioteke u Spring Boot.

4. Skalabilnost

Skalabilnost Spring Boota definitivno nije na razini Spring MVC okvira, ali dobrim dizajnom baze podataka, pravilnim izborom baze podataka, korištenjem sinkronih i asinkronih poziva te primjenom mikro servisa skalabilnost Spring Boota je definitivno zadovoljavajuća za projekte srednje veličine i posjećenosti.

5. Korisničko iskustvo

Situacija u Spring Bootu jednaka je kao i kod Spring MVC okvira, jedina razlika je što prilikom rada sa Spring Bootom on podupire korištenje nove tehnologije Thymeleaf, dok većina Spring MVC projekata i dalje koristi JSP tehnologiju. Thymleaf je temeljen na HTML-u, tako da i bez uključene zavisnosti za Thymeleaf stranica izgleda normalno, dok u slučaju JSP-a izgled stranice je narušen.

4.4 GWT

1. Mogućnost brzog razvoja aplikacija

Postoji mnogo prethodno pripremljenih značajki (eng. *widgeta,* nadalje widget) za brzu uporabu, ali sve što je moguće učiniti s JavaScriptom i DOM-om web preglednika može se izvršiti pomoću GWT-a. Dizajneri mogu upotrijebiti ugrađeni dizajnerski način rada GWT-a koji sadrži jednostavno sučelje za povlačenje i spuštanje s automatskim generiranjem koda.

2. Jednostavnost korištenja

Bez obzira koristi li ga razvojni inženjer ili dizajner GWT ima dobru i prilično jednostavnu strukturu. Struktura komponenti koju koriste razvojni inženjeri je jednostavna za korištenje i mogućnosti widgeta su brojne. Kod je jednostavan za čitanje, izmjenu i ponovnu uporabu. Kod je nevjerojatno sličan JavaScriptu.

3. Dokumentacija

GWT ima veliku službenu dokumentaciju. Većina novih web stranica (još su u beti) GWT projekta su priručnici i dokumentacija koja detaljno navodi sve od osnova korištenih prilikom programiranja do korištenja složenih widgeta. Postoji mnogo mjesta sa vodičima, pitanjima i primjerima o načinu uparivanju GWT-a s drugim okvirima, te na stranici je dostupno preuzimanje većeg broja primjera projekta i prezentacija.

4. Skalabilnost

Ako je GWT stvoren zbog isključivo jedne stvari, onda je stvoren za skalabilnost. Kod korišten u GWT-u izgleda slično mnogim drugim standardnim API-jima, ali s asinkronim pozivima. Uključeni kompajler je ono čime se GWT ističe. Kompajler koji pretvara Javu u JavaScript skriva razlike u pregledniku pa se razvojni inženjeri mogu usredotočiti na stvarni kod bez brige o prilagodbi na različite preglednike i verzije preglednika. Također vrši analizu koda i uklanja nedostupni kod. Dakle bez obzira na veličinu aplikacije GWT će analizirati i sažeti vraćeni JavaScript kako bi bio što učinkovitiji bez obzira na preglednik koji se upotrebljava.

5. Korisničko iskustvo

GWT je vrlo svestran kada se radi o prilagodbi i kreiranju lijepog izgleda web aplikacija. Budući da je okvir sastavljen od komponenti, moguće je promijeniti izgled aplikacije u bilo kojem trenutku bez da se pokvari prethodno napisana logika. Ovaj način razdvajanja pomaže da aplikacije rade brže, troše manje memorije i olakšavaju prilagođavanje prilikom uređivanja i testiranja napisanog koda. GWT omogućuje povezivanje stilskih listova s aplikacijama kako bi se omogućilo dodatno prilagođavanje. GWT teme povezane su s aplikacijom u cjelini i trenutno postoje tri teme: standardna, Chrome i Dark. Također, moguće je stvoriti stilske listove gdje se samostalno mogu odrediti specifični stilovi koji će se koristiti u aplikaciji.

4.5 Grails

1. Mogućnost brzog razvoja aplikacija

Ovo je područje u kojem se Grails programski okvir stvarno ističe. Postavljanje je vrlo brzo i uz pomoć generiranja koda, moguće je uštediti jako puno vremena. Načelo konvencije preko načela konfiguracije uklanja skoro sve probleme vezane uz konfiguraciju. Grails dolazi s mehanizmom za ponovno učitavanje promjena, ali ta mogućnost ima neka ograničenja. Jedno od bitnih ograničenja je što to omogućava samo za Groovy klase, ali ukoliko se koristi Java i dalje postoji opcija korištenja JRebela(aplikacija koja se plaća i ona omogućava dinamičko učitavanje promjena u Java klasama tijekom vremena izvođenja). Ukoliko je potrebno implementirati malu do srednju aplikaciju čije funkcionalnosti zahtijevaju CRUD, Grails se može nametnuti kao kvalitetan izbor.

2. Jednostavnost korištenja

Grails je dizajniran da bude brz razvojni okvir i brzina je izravna posljedica njegove jednostavnosti korištenja. Već je rečeno kako se zalaže za pristup konvenciju preko načela konfiguracije i proširivost je vrlo jednostavna kada se koriste dodatci, kojih ima mnogo. Jedan od nedostataka je primoranost učenja jezika Groovy, ali kako je on vrlo sličan Javi to ne bi trebao biti problem.

3. Dokumentacija

Trenutno postoji ogromna količina informacija o Grails programskom okviru, odjeljak za dokumentaciju je zapravo wiki stranica, koja može biti izmijenjena od strane bilo kojeg prijavljenog korisnika na Grails službenoj stranici. Dokumentacija sadrži službeni priručnik, vodiče, primjerak jednostavne aplikacije i još mnogo toga. Također moguće je pretražiti 42 tisuće tema na popisu za slanje e-pošte.

4. Skalabilnost

Grails je apstrakcija okvira Spring i Hibernate. Kako su oba navedena okvira skalabilna, rezultat je da je i Grails skalabilan unatoč dodatnim sloju zbog navedene apstrakcije. Navedeni sloj ne bi trebao utjecati na skalabilnost, jer je on napisan u Java bajt kodu.

5. Korisničko iskustvo

Grails sadrži veliku količinu integracija s raznim bibliotekama poput jQuerya, Twitter Bootstrapa te raznih Ajax komponenti. Na taj način definitivno olakšava stvari, npr. *datepicker* komponentu se ne bi moralo kreirati ručno, već ju je moguće iskoristiti pomoću neke od biblioteka i pritom urediti izgled pomoću CSS-a (eng. *cascading style sheets*). Automatski generirani kostur projekta ima neke osnovne predloške za prikaz i CSS za početak, ali rad na nekim složenijim stvarima ostaje u rukama razvojnih inženjera.

4.6 Analiza i usporedba Java programskih okvira

Za kraj kako bi se sažela prethodno obavljena analiza, njezini rezultati biti će prikazati u obliku tablice. Svaki okvir biti će ocijenjen u rangu od 1 do 5 za svaki analizirani parametar. Sljedeća tablica reflektira rezultate autorovog osobnog pogleda na uspoređivane okvire. Svaki od parametara autor je analizirao i na temelju toga došao je do sljedećih zaključaka koji su vidljivi u obliku tablice 7.

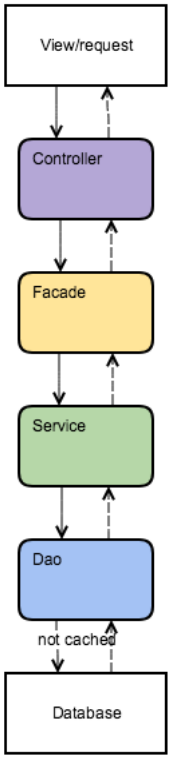
Tablica 6. Usporedba Java okvira

	Spring MVC	**JSF**	**Spring Boot**	**GWT**	**Grails**
Mogućnost brzog razvoja aplikacija	2.5	3	4	4	5
Jednostavnost korištenja	3	4	4.5	4	4.5
Dokumentacija	5	4.5	5	4.5	5
Skalabilnost	4	4	3	4.5	4
Korisničko iskustvo	2	4.5	2.5	5	4
Ukupno	**16.5**	**20**	**18.5**	**22**	**22.5**

Kada se pogledaju rezultati analize, oni se ne slažu s brojkama koje indiciraju na korištenost pojedinih okvira. Jer definitivno najkorišteniji okvir Spring ima najlošiji rezultat u ovoj analizi, ali to ne treba uzeti kao konačno mjerilo kvalitete nekog okvira jer opseg ove analize nije dovoljno velik. Ona se vrši isključivo na 5 parametara i zbog toga ne može prikazati punu sliku. Kako bi ova analiza postala značajna, potrebno je ispitati koji je razlog dominacije Springa, koje su njegove prednosti nad ostalim okvirima i koje značajke čine najbitnije razlike koje se ogledaju na razlog njegove popularnosti. Ova analiza ponajviše je usmjerena početnicima zato što se uzima u obzir karakteristike poput jednostavnosti korištenja i mogućnosti brzog razvoja aplikacija. Navedene karakteristike ne predstavljaju bitnu stavku u profesionalnom razvoju aplikacija za poduzeća, jer uglavnom se razvijaju velike aplikacije, čiji razvoj i održavanje se ogleda u godinama. Grails se istaknuo kao pobjednik ove analize, on je okvir koji se temelji na Spring MVC okviru, ali za razliku od njega istovremeno omogućuje brzo i jednostavno izrađivanje aplikacija. Promiče se kao definitivni izbor za bilo kojeg razvojnoj inženjera koji treba u kratkom vremenskom roku napraviti oku ugodnu i kvalitetnu aplikaciju.

5. Primjena uzoraka dizajna na arhitekturnoj i internoj razini Springa

Prethodno je rečeno da se Spring temeljni na MVC arhitekturi koja je zapravo vrlo učestala u svijetu razvoja web aplikacija. Samim time svaka Spring aplikacija bi trebala imat barem 3 sloja, a to su *Controller*, *Model* i *View*. Razvojni inženjeri shvatili su kako na taj način previše poslovne logike se nalazi u samom *Controlleru* i zbog toga je dodan novi sloj pod nazivom *Service* i njegova uloga je da sadrži poslovnu logiku. Također može se primijeniti i *Facade* uzorak koji će se primjenjivati u situacijama kada je potrebno pretvarati DTO objekte(oni se primjenjuju za slanje *viewima*, da se ne izlažu prevelike količine podataka) u same *Modele*(oni predstavljaju objekte koji su spremljeni u bazi podataka). U slučaju da tog sloja nema navedena radnja morala bi se odvijati u *Controlleru* ili u *Service* sloju, ali u oba slučaja to nije idealan izbor. Samim time dolazi do miješanja odgovornosti u samom dizajnu.



Slika 5. Tijek zahtjeva

Osim *Facade* uzorka može se primijeniti još i *DAO* sloj (eng. *data access object*). DAO sloj se nalazi između slojeva *Service* i baze podataka. Ovaj sloj najpraktičniji je u situacijama kada je potrebno ili isplativo korištenje pred memoriranja zahtjeva (eng. *caching*), kako bi se izbjeglo nepotrebno dohvaćanje podatka iz baze podataka, ukoliko su prethodno već dohvaćeni i znamo da nisu mijenjani. Na kraju primjenom svih slojeva dolazi se do primjera složene arhitekture koja se temelji na odvajanju odgovornosti. Primjenom ovog pristupa ne bi

trebalo doći do ponavljanja koda, nego promovirati će se njegova ponovna upotreba korištenjem sučelja. Zbog toga ovaj pristup promovira jednostavnije skaliranje, održavanje i testiranje programskog koda. Navedeni primjer arhitekture biti će korišten u aplikaciji koja će biti kreirana za potrebe ovoga rada.

Nakon što je pojašnjeno kako izgleda osobni pogled autora na kvalitetni arhitekturni dizajn jedne Spring aplikacije, može se prijeći na koji način Spring primjenjuje uzorke, kako bi olakšao posao razvojnim inženjerima.
Spring koristi sljedeće strategije kako bi Java razvoj ostao jednostavan i testiran:

- koristi snagu POJO uzorka za lagani i minimalno invazivni razvoj poslovnih aplikacija
- koristi snagu obrasca ubrizgavanja ovisnosti (eng. *Dependency Injection*) za labav spoj i čini sustav usmjerenim radu pomoću sučelja.
- upotrebljava moć *Decorator* i *Proxy* uzorka dizajna za deklarativno programiranje kroz aspekte i uobičajene konvencije.
- koristi *Template method* uzorak za uklanjanje koda koji se višestruko ponavlja (eng. *boilerplate code*) uz pomoć aspekata i predložaka.

To su neki od osnovnih načina putem kojih Spring olakšava razvoj, održavanje, testiranje koda. Naravno osim prethodno navedenih, postoji još uzoraka koji pronalaze svoju primjenu u Spring programskom okviru. Oni će biti nabrojani i opisani kroz sljedećih par paragrafa s uključenim primjerima kako i na koji način su korišteni.

1. *Factory method*

Spring koristi uzorak *Factory method* za provedbu Spring spremnika koristeći *BeanFactory* i *ApplicationContext* sučelja. Springov spremnik radi na temelju *Factory method* uzorka kako bi pripremio Spring zrna za aplikaciju i također na taj način upravlja životnim ciklusom svakog zrna. *BeanFactory* i *ApplicationContext* su sučelja koja pripadaju Factory *method* uzorku i Spring sadrži mnogo klasa koje ih implementiraju. Metoda *getBean*() je metoda koja vam daje potrebna zrna. Ispod je prikazano navedeno sučelje.

```
public interface BeanFactory {
     String FACTORY_BEAN_PREFIX = "&";
     Object getBean (String var1) throws BeansException;
     <T> T getBean (String var1, Class <T> var2) throws BeansException;
```

```
     <T> T getBean (Class <T> var1) throws BeansException;

     Object getBean (String var1, Object ... var2) throws BeansException;

     <T> T getBean (Class <T> var1, Object ... var2) throws BeansException;

     boolean containsBean (String var1);

     boolean isSingleton (String var1) throws
NoSuchBeanDefinitionException;

boolean isPrototype (String var1) throws NoSuchBeanDefinitionException;

     boolean isTypeMatch (String var1, Class <?> var2) throws
NoSuchBeanDefinitionException;

     Class <?> GetType (String var1) throws NoSuchBeanDefinitionException;

     String [] getAliases (String var1);
}
```

2. Abstract Factory

U Spring programskom okviru *FactoryBean* sučelje temelji se na *Abstract Factory* uzorku. Spring pruža mnogo implementacija navedenog sučelja, neka od njih su *ProxyFactoryBean*, *JndiFactoryBean*, *LocalSessionFactoryBean* itd. *FactoryBean* ima svoju primjenu u situacijama gdje je potrebno sastaviti složene objekte s mnogo zavisnosti, također ima svoju primjenu kada je konstrukcija objekta promjenjiva i recimo da ovisi o konfiguraciji. Ispod je vidljivo sučelje *FactoryBean*. Metoda *getObject()* vraća zrno prilikom pozivanja.

```
public interface FactoryBean <T> {
     T getObject () throws Exception;

     Class <?> GetObjectType ();

     boolean isSingleton ();
}
```

3. Singleton

Prethodno je objašnjeno kakvi opsezi zrna postoje i poznato je da je *Singleton* zadani opseg prilikom kreiranja zrna. Spring ne koristi *Singleton* uzorak na način kojim je on primijenjen u Javi, nego zapravo kreira jednu instancu zrna po spremniku i po tipu zrna. Što znači da ukoliko se kreira jedno zrno za jednu specifičnu klasu u jednom Spring spremniku, onda Spring spremnik stvara jednu instancu te klase koja je definirana definicijom pripadajućeg zrna.

4. Builder pattern

Neke od primjena *Builder patterna* u Spring programskom okviru su *EmbeddedDatabaseBuilder*, *AuthentificationManagerBuilder*, *UriComponentsBuilder*, *BeanDefinitionBuilder*, *MockMvcWebClientBuilder*. Ispod je vidljiva primjena *Builder patterna* u *EmbeddedDatabaseBuilder,* te je jasno da primjenjuje verziju s ugniježđenom pod klasom.

```
public class EmbeddedDatabaseBuilder {
    private final EmbeddedDatabaseFactory databaseFactory;
    private final ResourceDatabasePopulator databasePopulator;
    private final ResourceLoader resourceLoader;

    public EmbeddedDatabaseBuilder () {
        this (new DefaultResourceLoader ());
    }

    public EmbeddedDatabaseBuilder (ResourceLoader resourceLoader) {
        this.databaseFactory = new EmbeddedDatabaseFactory ();
        this.databasePopulator = new ResourceDatabasePopulator ();
        this.databaseFactory.setDatabasePopulator (this.databasePopulator);
        this.resourceLoader = resourceLoader;
    }

    public EmbeddedDatabaseBuilder setName (String databaseName) {
        this.databaseFactory.setDatabaseName (DataBaseName);
        return this;
    }

    public EmbeddedDatabaseBuilder setType (EmbeddedDatabaseType
databaseType) {
        this.databaseFactory.setDatabaseType (databaseType);
        return this;
    }

    public EmbeddedDatabaseBuilder setDataSourceFactory (DataSourceFactory
dataSourceFactory) {
        Assert.notNull (dataSourceFactory, "DataSourceFactory is
required");
        this.databaseFactory.setDataSourceFactory (dataSourceFactory);
        return this;
    }

    public EmbeddedDatabase build () {
        return this.databaseFactory.getDatabase ();
    }
}
```

5. Adapter

Neke od primjena *Adapter* uzorka u Spring programskom okviru su *JpaVendorAdapter*, *HibernateJpaVendorAdapter*, *HandlerInterceptorAdapter*, *SpringContextResourceAdapter* itd.

HandlerInterceptorAdapter je apstraktna Adapter klasa za *HandlerInterceptor* sučelje i ono se primjenjuje za postavljanje implementacije za akcije koje se izvršavaju prije i poslije rada *Interceptora*.

```
public abstract class HandlerInterceptorAdapter implements
HandlerInterceptor {
    public HandlerInterceptorAdapter() {
    }

    public boolean preHandle(HttpServletRequest request,
HttpServletResponse response, Object handler) throws Exception {
        return true;
    }

    public void postHandle(HttpServletRequest request, HttpServletResponse
response, Object handler, ModelAndView modelAndView) throws Exception {
    }

    public void afterCompletion(HttpServletRequest request,
HttpServletResponse response, Object handler, Exception ex) throws
Exception {
    }
```

6. *Decorator*

Spring primjenjuje *Decorator* uzorak prilikom izgradnje funkcionalnosti poput transakcija, sinkronizacije predmemoriranja i zadataka vezanih uz sigurnost. Uglavnom primjenjuje Decorator uzorak uz pomoć CGLib proxy biblioteke. CGLib proxy djeluje tako da generira pod klase ciljne klase za vrijeme izvođenja. Neke od primjena su *BeanDefinitionDecorator* i *WebSocketHandlerDecorator*. *BeanDefinitionDecorator* je vidljiv u sljedećem isječku koda i on se primjenjuje u situacijama kada je potrebno ukrasiti definiciju zrna primjenom prilagođenih atributa.

```
public interface BeanDefinitionDecorator {
    BeanDefinitionHolder decorate(Node var1, BeanDefinitionHolder var2,
ParserContext var3);
}
```

7. *Chain of responsibility*

Spring Security je implementiran koristeći uzorak *Chain of responsibility*. On omogućava implementaciju autentifikacije i autorizacije u Spring aplikacijama koristeći lance sigurnosnih filtera. Zbog načina implementacije, navedeni okvir je jako podesiv i mogu se dodati prilagođeni filteri na već postojeće Spring filtere kako bi se lakše zadovoljile potrebe aplikacije.

Ispod je vidljiv jedan primjer primjene *Chain of Responsibility* uzorka u Spring Security postavkama.

```
httpSecurity
                .authorizeRequests()
                .antMatchers("/").permitAll()
                .antMatchers(LOGIN).permitAll()
                .antMatchers("/registration").permitAll()
                .antMatchers("/user/**").hasAuthority("ROLE_USER")
                .antMatchers("/admin/**").hasAuthority("ROLE_ADMIN")
                .anyRequest()
                .authenticated()
                .and()
                .formLogin()
                .loginPage(LOGIN)
                .failureUrl(LOGIN + "?error=true")
                .successHandler(handler)
                .usernameParameter("email")
                .passwordParameter("password")
                .and()
                .logout()
                .logoutRequestMatcher(new AntPathRequestMatcher("/logout"))
                .logoutSuccessUrl(LOGIN)
                .and()
                .exceptionHandling()
                .and()
                .exceptionHandling()
                .accessDeniedPage("/access-denied")
```

8. Observer

U Spring programskom okviru pronađena je primjena uzorka *Observer* u implementaciji upravljanja događajima od *ApplicationContexta*. Kako bi omogućio upravljanje događajima Spring pruža klasu ApplicationEvent i sučelje ApplicationListener. Svako zrno u Spring aplikaciji koje implementira sučelje ApplicationListener prima obavijesti svaki puta kada je ApplicationEvent objavljen. To znači da u ovom specifičnom slučaju ApplicationListener je *Observer*, a klasa koja objavljuje događaje je *Subject*.

```
public interface ApplicationListener <E extends ApplicationEvent> extends
EventListener {
    void onApplicationEvent(E var1);
}
```

9. Template method

Template method uzorak je jedan od najčešće primijenjenih uzoraka u Spring programskom okviru. On omogućava uklanjanje nepotrebnog ponavljajućeg koda na način da

kreira kostur algoritma i onda unutar njega se pozivaju samo dijelovi koda koji sadrže poslovnu logiku i koji bi u svakome slučaju trebali biti jedinstveni. Najčešća primjena ovog uzorka je u komunikaciji s bazama podataka. Neke od primjena su *JdbcTemplate*, *JmsTemplate*, *RestTemplate*, *WebServiceTemplate* itd. Navedene klase imaju jako puno koda, stoga one ovdje neće biti prikazane nego biti će prikazana primjena samog *Template method* uzorka kako bi bilo vidljivo koliko se njegovom primjenom izbjegava pisanje nepotrebnoga programskog koda. Ispod je prikaz primjer korištenja *JdbcTemplatea*.

```
public Account getAccountById(long id) {
      return jdbcTemplate.queryForObject(
        "select id, name, amoount" +
        "from account where id=?",
         new RowMapper<Account>() {
           public Account mapRow(ResultSet rs,
            int rowNum) throws SQLException {
              account = new Account();
              account.setId(rs.getLong("id"));
              account.setName(rs.getString("name"));
              account.setAmount(rs.getString("amount"));
              return account;
            }
         },
      id);
    }
```

Kao što je vidljivo nema nepotrebnih stvaranja *Connectiona*, *PreparedStatmenta* i *ResultSetova*, nego je to odrađeno unutar kostura Template methoda, dok nama je jedino potrebno napraviti upit i dohvatiti željene podatke.

6. Prototip i razvoj aplikacije

Aplikacija koja će biti napravljena za potrebe ovog rada naziva se *SoleX*. *SoleX* je akronim za *Solution Explorer* i svrha ove web aplikacije je predstaviti učinkovit alat za poduzeća kojima je potrebna neka vrsta vanjskih suradnika. Aplikacija je prvenstveno usredotočena na poduzeća u IT sektoru i to onima koje trebaju pomoć oko razvoja aplikacija. Cilj sustava je omogućiti pojedincima koji imaju višak slobodnog vremena jednostavnu zaradu, a poduzećima (koje su u škripcu sa rokovima ili poduzećima na nekom potencijalnom projektu, male magnitude, nedostaje određena stručnost) kratkotrajnu pomoć. Dva su glavna agenta u sustavu, razvojni inženjer i poduzeće. Poduzeće može izdati objavu da će raditi projekt, potom može dodavati "potrebna radna mjesta" na projekt za određeni vremenski period i pritom jasno zadati potrebne kompetencije za rad. Razvojni inženjeri mogu pretražiti poslove prema vlastitim ili zadanim kompetencijama i prijaviti se na odabrano radno mjesto. Komunikacija između agenata će biti ostvarena korištenjem email klijenta.

6.1 Baza podataka

Slijedi opis razvoja ERA modela koji se koristi u aplikaciji. Aplikacija zbog svoje prirode zahtjeva bazu podataka, stoga je napravljen model koji prikazuje entitete i njihove međusobne odnose koji prikazuju strukturu podataka s kojom će se aplikacija baviti. Model je rađen u alatu MySQL Workbench.

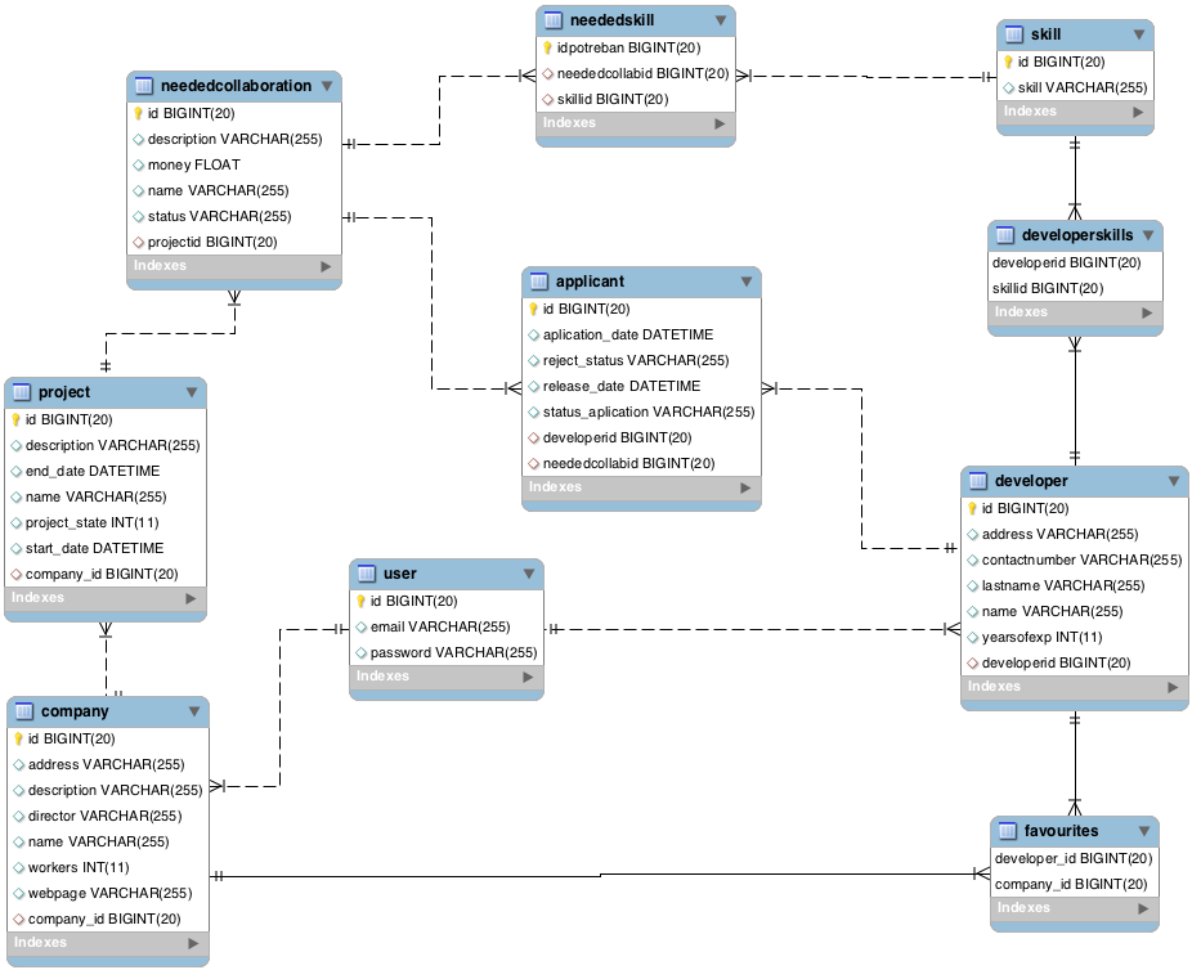
Prethodno je objašnjeno da se središtu aplikacije nalaze dvije uloge, poduzeće i razvojni inženjer, svaki od njih ima svoj primarni cilj. Poduzeće stvara projekte na kojima im treba suradnja odnosno pomoć, a razvojni inženjer traži postojeće projekte i aplicira se za njih. Stoga se uz poduzeće usko vežu projekti, a uz razvojne inžinjere suradnje (eng. *needed collaborations*). Obje uloge imaju svoju pripadajuće entitete koji su povezanu na entitet *User,* koji sadrži osnovne podatke potrebne za prijavu u aplikaciju, dok pripadajući entiteti *Company* i *Developer* sadrže podatke usko vezane i specifične za navedene uloge.

Razvojni inženjer sudjeluje u nekoliko suradnji s drugim entitetima, postoje veze sa vještinama, s poduzećem i s potrebnim suradnjama. Svaka od veza izvedena je u obliku više-naprema-više i poznato je da je nusprodukt takvih veza novi entitet. U ovome slučaju nove tablice su *developerskills, applicant, favourites*. Entitet *Developerskills* predstavlja vještine koje jedinstveni razvojni inženjer posjeduje, *favourites* se referencira na mogućnost da poduzeća imaju određene razvojne inženjere za favorite te *applicant* sadrži podatke o apliciranju razvojnog inženjera na neku potrebnu suradnju.

Entitet *company* koji predstavlja poduzeće ima vezu više-naprema-više s *developer*, zatim ima vezu jedan-naprema-više u odnosu na project, što znači da jedno poduzeće može imati više projekata.

Osim suradnji u koje su uključene uloge razvojnog inženjera i poduzeća, također postoje suradnje među entitetima *project* i *neededcollaboration*. Ona predstavlja da jedan projekt može imati više potrebnih suradnji i *neededcollaboration* također ima vezu s entitetom *skill* gdje nastaje novi entitet *neededskill* koji predstavlja potrebne vještine za određenu potrebnu suradnju.

Ispod je prikazan ERA model pripadajuće prethodno opisane baze podataka.



Slika 6. ERA model

Legenda modela: sličica žutog ključa predstavlja primarni ključ. Vanjski ključ je predstavljen sa rombom tamnocrvene boje. Atributi su predstavljeni svijetloplavim rombovima.

Ukoliko atribut može biti NULL (pravilo vrijedi i za vanjske ključeve) onda je romb prazan, a inače je ispunjen.

6.2 Mockup

Mockup je zamišljen više kao tijek rada (eng. *workflow*) pomoću kojeg će naknadno biti prikazane mogućnosti unutar aplikacije jednom kada budu dizajnirane i funkcionalne.

Detaljan opis nacrta za tijek rada:

1. Početna stranica - stranica koja sadrži linkove za prijavu, registraciju i prikazuje vizualni identitet aplikacije
2. Prijava - stranica za prijavu (jednaka za obje uloge unutar sustava)
3. Registracija - stranica za registraciju (jednaka za obje uloge, potrebno je samo odabrati kojoj ulozi se pripada).
4. Glavni izbornik – nalazi se na svim stranicama gdje je prisutna sesija, odnosno gdje je korisnik prijavljen. Nalazi se s lijeve strane i proširuje se tako što se pređe mišem preko nje. Svaka uloga ima svoje mogućnosti u glavnom izborniku.

SLUČAJ PODUZEĆE:

5. Početna stranica poduzeća - prikazuje se nakon uspješne prijave. Od sadržaja u sebi sastoji tablicu koja prikazuje podatke vezana uz apliciranja na njihovo kreirane suradnje. U tablici pomoću gumba može odobriti ili odbiti aplikaciju. Prilikom odgovora na aplikaciju razvojnom inženjeru se šalje email poruka s više informacija.
6. Moji projekti – sadrži tablicu gdje se prikazuju projekti koji su trenutno u tijeku. Svaki projekt u tablici ima gumb koji vodi na stranicu detalji projekta. Također ispod tablice nalazi se gumb pomoću kojeg se otvara modal u kojem je moguće kreirati novi projekt.
7. Detalji projekta –sadrži formu koju nije moguće uređivati. Ona služi za prikaz detalja o projektu. Osim forme sadrži tablicu suradnji koje postoje na tom projektu i moguće je pogledati detalji tih suradnji.
8. Detalji suradnji - sadrži formu koju nije moguće uređivati i ona sadrži detalje o samoj suradnji.
9. Suradnje – prikazuje trenutno slobodne suradnje u obliku tablice. Svaka suradnja može se ažurirati pritiskom na gumb. Ispod tablice nalazi se gumb koji otvara modal u kojem je moguće dodati novu suradnju.
10. Ažuriranje suradnji – sadrži formu putem koje je moguće mijenjati postojeće podatke o suradnji.
11. Dodijeljene suradnje – sadrži popis već dodijeljenih suradnji.

12. Favoriti – sadrži popis favorita. Favoriti se dodaju pomoću modala koji se otvara klikom na gumb. Favoriti se brišu pritiskom na gumb *Delete* koji se nalazi u tablici. Svakom favoritu možemo posjetiti profil klikom na gumb *Details*.
13. Profil razvojnog inženjera – sadrži osnovne podatke o razvojnom inženjeru i gumb putem kojeg se otvara zadani email klijent i upisuje njegova email adresa.
14. Ažuriranje podataka poduzeća – sadrži formu za izmjenu postojećih podataka o poduzeću.

SLUČAJ RAZVOJNI INŽENJER

15. Početna stranica razvojnog inženjera – prikazuje tablicu koja prikazuje suradnje koje odgovaraju vještinama razvojnog inženjera. Ispod tablice može mijenjati postotak podudarnosti s njegovim vještinama, u slučaju da ne može pronaći projekt koji točno odgovaranja njegovom znanju. Ukoliko ga ponuda zanima može odabrati detalje neke suradnje.
16. Detalji suradnje – prikazuje formu koju nije moguće uređivati i ona sadrži podatke o odabranoj suradnji. Ispod može se aplicirati na suradnju ili ukoliko je već apliciran može povući svoju prijavu. Prilikom apliciranja, povlačenja poduzeću se šalje email poruka o statusu aplikacije.
17. Moji projekti – sadrži prethodno odrađene projekte. Moguće je vidjeti detalje projekta(isto kao stranica 7.)
18. Suradnje – pregled prethodno odrađenih suradnji. Moguće je vidjeti njihove detalje i profil poduzeća.
19. Detalji suradnji – jednako kao 8.
20. Profil poduzeća – jednako kao 13. , samo u ovom slučaju se prikazuju podaci o poduzeću.
21. Moja apliciranja – sadrži popis prethodnih suradnji, ukoliko je apliciranje još aktivno moguće ga je povući.
22. Ažuriranje podataka razvojnog inženjera – prikazuje formu pomoću koje je moguće mijenjati postojeće podatke o razvojnom inženjeru.

6.3 Izbor tehnologija

Izbor tehnologija je uvijek ključan zadatak prije početka same implementacije. Potrebno je izabrati tehnologije u nekoliko područja, a da one odgovaraju primjeni i veličini same aplikacije. Potrebno je npr. odabrati web programski okvir za razvoj aplikacije. Pošto je tema ovog rada „Uzorci dizajna u Spring programskom okviru“, jedino ima smisla da izbor web okvira je Spring i stoga aplikacija će biti izrađena *Spring Boot* programskom okviru. Glavni razlog je što omogućava olakšano stvaranje i postavljanje projekta, ali da i dalje sadrži sve glavne Spring MVC funkcionalnosti. Za prikaz podataka korisniku *Spring Boot* pruža mogućnost između standardnog jspa i *Thymeleafa*, ali pošto je *Thymeleaf* modernija verzija i više nalikuje

HTML-u. U ovom slučaju odabran je *Thymeleaf* kako bi se istražilo što novo donosi u odnosu na jsp te koje su njegove eventualne prednosti ili mane.

Zatim je odabran alat za upravljanje i razumijevanje softverskog projekta (eng. *software project management and comprehension tool*) koji će olakšati upravljanje zavisnostima koje će se eventualno primjenjivati u projektu. Izbor je bio između *Ant* i *Maven* alata te je odlučeno da će se primjenjivati *Maven,* iz razloga što ima jako kvalitetan i bogat repozitorij s bibliotekama koje su planirane za korištenje.

Kako bi aplikacija mogla komunicirati s prethodno odabranom bazom podataka potrebno je odabrati okvir za komunikaciju s bazom podataka. Odlučeno je da će se koristiti *Hibernate* u kombinaciji sa *Spring Datom*, navedeni alat samostalno kreira tablice unutar baze podataka uz pomoć mapiranja veza unutar Java koda, ali također moguće je koristiti i postojeću bazu uz odgovarajuće mapiranje u konfiguraciji aplikacije. Kako bi okvir funkcionirao s prethodno postavljenom bazom, potrebno je pomoću konfiguracije postaviti podatke o lokaciji baze podataka i podacima za prijavu.

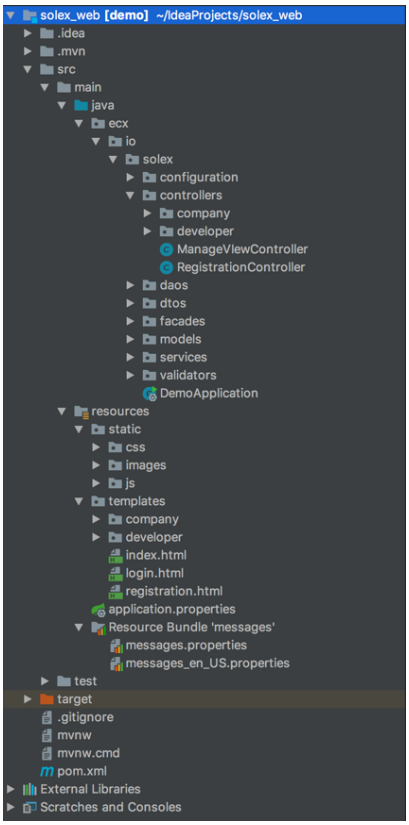
Kako bi stranica bila dinamična i omogućavala jednostavniju interakciju s korisnikom potrebno je odabrati tehnologiju ili programski okvir za pisanje *Javascript* programskog koda. Koristiti će se jQuery zbog svoje jednostavnosti, velike količine primjera, dokumentacije i velike zajednice koja ga koristi svakodnevno u rješavanju problema. Osim jQuery biblioteke korištena je i njegova *Datatables* biblioteka, koja omogućava ljepši prikaz tablica i omogućuje jednostavnu implementaciju straničenja, pretraživanja i filtriranja nad tablicom. Osim navedenih biblioteka biti će korišten *font-awesome* biblioteka koja sadrži razne ikone koje će olakšati kreiranje izbornika i kao posljednja biblioteka vezana uz uređivanje prikaza aplikacije koristiti će se *Bootstrap,* koja omogućava brzo i jednostavno uređivanje dizajna stranice na način da je ono automatski prilagođeno za sve tipove uređaja.

Za kraj preostalo je odabrati sustav za verzioniranje koda na kojem će se nalaziti projekt. Izbor je bio jednostavan, trenutno se u skoro svim programerskim poduzećima koristi *Git*, stoga je odlučeno za navedenu opciju i točnije rečeno klijent *BitBucket*. Repozitorij se nalazi na sljedećoj adresi:
https://bitbucket.org/cvitka/solex_web/src/dev/

6.4 Razvoj aplikacije

U ovome poglavlju dosad je objašnjena arhitektura koja će biti primijenjena u aplikaciji SoleX, objašnjenje su tehnologije koje će biti primijenjene te za kraj napravljen je tijek rada pomoću kojeg je moguće zamisliti na koji način će se aplikacija koristiti. Nakon što su navedeni koraci bili ispunjeni moguće je krenuti s razvojem aplikacije. Početak razvoja krenuo je sa funkcionalnostima prijave i registracije, gdje najveći problem predstavlja ispravno postavljanje *Spring Security* funkcionalnosti, već je poznato da se on radi pomoću *Chain of Responsibilities* uzorka. Osim *Spring Security* funkcionalnosti korišteni su *Spring Validaton* i *Spring i18n* za validaciju podataka korištenih u registraciji te za ispis ispravnih grešaka korisnicima.

Nakon završetka funkcionalnosti prijave i registracije, bilo je potrebno odrediti koji oblik dizajna će biti primijenjen i kreirati osnovni vizualni identitet koji će se primjenjivati kroz čitavu aplikaciju. Prilikom kreiranja dizajna korištena je kombinacija Bootstrapa, CSSa, HTMLa koji u sebi sadrži Thymeleaf, jQuery i font-awesome biblioteka. JQuery koristi se u situacijama gdje se izvršavaju RESTful pozivi kojima se tablice popunjavaju sadržajem, font-awesome ikone primijenjene su unutar glavnog izbornika i gumba.



Slika 7. Struktura aplikacije

S završetkom kreiranja osnovnog dizajna bilo je moguće početi s razvojem osnovnih funkcionalnosti vezanih uz uloge poduzeća i razvojnog inženjera. Razvoj je tekao na način

tako da povezane funkcionalnosti između uloga se rade istovremeno kako bi se olakšalo testiranje aplikacije.

S desne strane nalazi se slika koja prikazuje kako aplikacijska struktura izgleda unutar razvojne okoline IntelliJ IDEA. Već je prethodno napomenuto da će biti korištena nadopunjena MVC arhitektura i njezinu primjenu možemo vidjeti na slici. Programski kod dijeli se na pakete configuration, controllers, daos, dtos, facades, models, services i validators. Unutar paketa configuration nalazi se osnovna Spring Security konfiguracija i Spring i18n konfiguracija. Controlleri nalaze se unutar istoimenog paketa gdje su podijeljeni prema ulogama radi lakšeg snalaženja. Također unutar njih nalazi se i api paket kako bi odvojio RestControllere od standardnih.

Zatim postoji paket dtos koji odvaja DTO (eng. *Data Transfer Object*) objekte i oni se koriste za prikaz podataka korisniku, facades predstavlja jedan od uzoraka koji se koriste unutar aplikacije. On enkapsulira pretvaranje dobivenih podataka iz controllera u Modele ili DTO objekte te zatim prosljeđuje novo stvorene objekte sloju *services* gdje se nad njima izvršava određena poslovna logika ili prosljeđivanje sloju daos. Velika većina DTO objekata instancira se uz pomoć *Builder Patterna* kako bi se odvojili obavezni parametri od neobaveznih i kako bi objekti bili nepromjenjivi (eng. *immutable*).

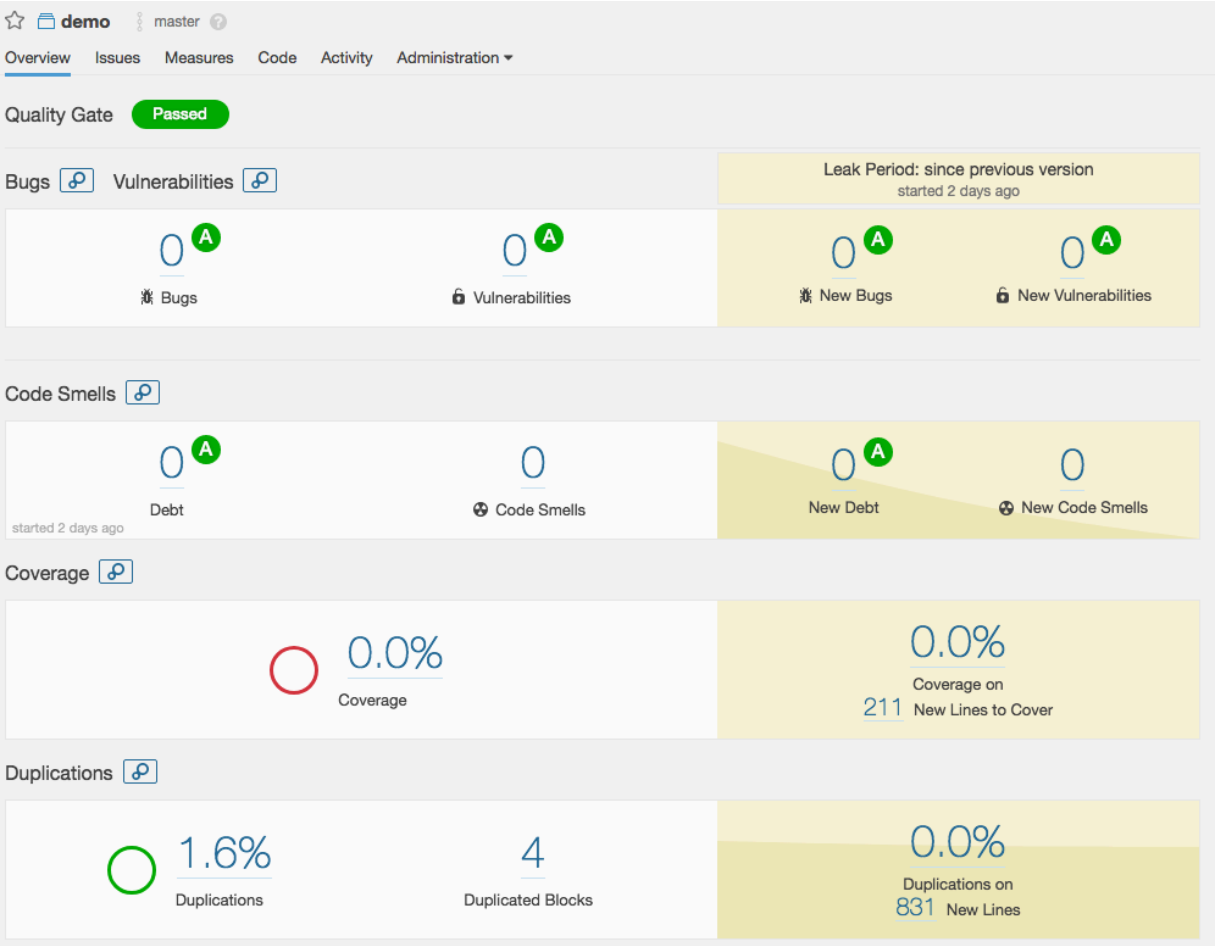
Neke DTO klase sadržavaju primjenu koja je jako slična *Prototype* uzorku, ali kako je on zastario i više nije preporučen nije korišten u ovome radu. Umjesto njega korištena je Mapper biblioteka koja uz pomoć refleksije kopira atribute objekta. Uz Mapper korišteno je kopiranje vrijednosti preko konstruktora umjesto kloniranja putem *Protoypea*. Kopiranje uz pomoć konstruktora ima nekoliko prednosti u odnosu na *Prototype* uzorak, neke od njih su mogućnost kopiranja različitih reprezentacija od originala, mogućnost kopiranja vrijednosti raznih popisa poput ArrayList ili LinkedList (Bloch, 2002). Za iteriranje popisa koriste se najčešće u duhu Java 8 tokovi podataka, ali gdje oni nisu bili korišteni koristio se uzorak *Iterator*.

Daos sadrži *Spring Data Repository* sučelja koja su predodređena za isključivo jedan model. Pomoću njih vrši se komunikacija s bazom podataka i akcije poput spremanja, brisanja i ažuriranja. Modeli su nalaze unutar istoimenog paketa i oni reflektiraju entitete i njihove atribute koji se nalaze u bazi podataka. Spring Dana Repository omogućava pisanje jednostavnih upita uz pomoć svojih generičkih sučelja skoro bez imalo truda, ali postojale su situacije gdje taj način nije zadovoljavao potrebe aplikacije i u tim slučajevima pisani su ručni

upiti. Preostao je paket validators koji sadrži Validator koji se koristi za validaciju prijave i registracije korisnika.

Spring svoje statičke datoteka drži unutar istoimenog paketa te tamo se nalaze css datoteke, Javascript datoteke i slike koje se koriste unutar aplikacije. U paketu *templates* nalaze se HTML datoteke koje se koriste za renderiranje stranica te imamo još paket *messages*. U njemu moguće je dodati datoteku svojstava za svaki jezik koji će aplikacija podržavati.

Cijelo vrijeme tijekom razvoja aplikacije korišteni su dodatci pod nazivom *SonarQube* i *SonarLint* koji služe za inspekciju koda. Oni pružaju mogućnost analize programskog koda i prikaz raznih upozorenja, grešaka u kodu, ponavljanja koda i slično. Na sljedećoj slici moguće je pronaći zadnje Sonar analize nakon što je aplikacija bila završena.



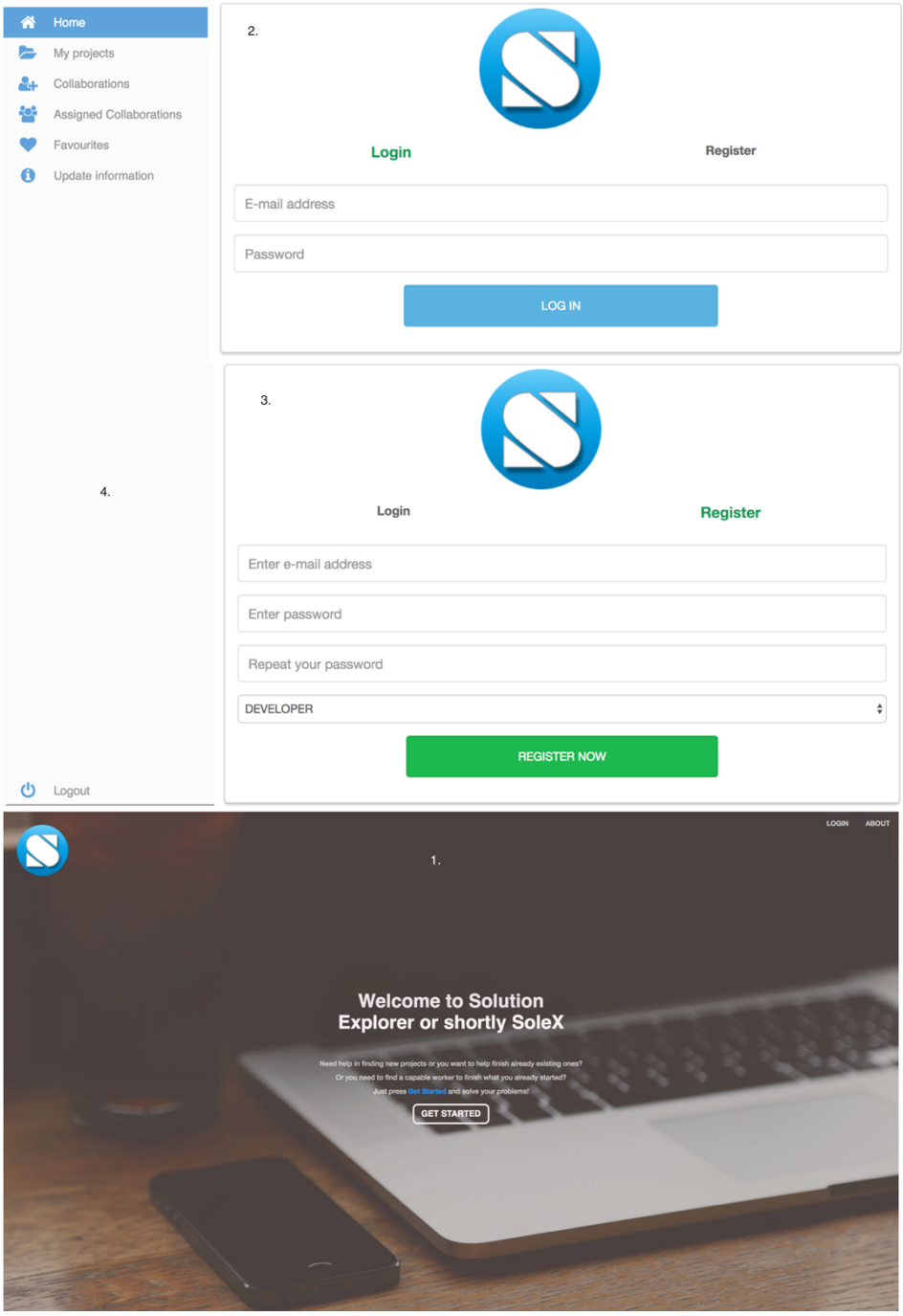
Slika 8. Rezultati SonarQube analize

Vidimo prema rezultatima da aplikacija pronalazi provedenu analizu koda. Jasno je vidljivo na slici da su ocjene u skoro svim područjima A što je dokaz kvalitetno napravljene

aplikacije. Jedini nedostatak je pokrivenost koda, a ona se postiže pisanjem jediničnih i integracijskih testova. Ona nije postignuta zbog malog opsega aplikacije i neučinkovitosti pisanja testova u sličnim situacijama (više vremena bi bilo utrošeno na pisanje testova, nego što bi oni pomogli u uočavanju i otklanjanju mogućih regresija prilikom razvoja).

7. Prikaz aplikacije

U ovome poglavlju biti će prikazane slike konačnog izgleda aplikacija prema tijeku rada. Slike će biti podijeljene u tri grupe, zajedničke funkcionalnosti, funkcionalnosti poduzeća i funkcionalnosti razvojnog inženjera. Svaka mogućnost biti će naznačena prema rednom broju u tijeku rada. Na svakoj od mogućnosti osim prijave, registracije i početne stranice nalazi se izbornik, ali su izrezani kako bi se uštedilo na prostoru. Slika 10. prikazuje početnu stranicu, prijavu, registraciju i glavni izbornik.



Slika 9. Zajedničke mogućnosti

Prethodno prikazane mogućnosti također će biti objašnjene uz pripadajuće isječke programskog koda. Ispod je prikazana implementacija *RegistrationControllera*. Postoji metoda *registrationSubmit* koja na putanji */registration* prihvaća poslanu formu za registraciju o zatim uz pomoć sloja Facade sprema korisnika ukoliko su podaci valjani.

```
@Controller
public class RegistrationController
{
    private static final Logger LOGGER =
LoggerFactory.getLogger(RegistrationController.class);
    private static final String ERROR = "error.save.user";
    private static final String LOGIN = "login";
    private static final String REGISTRATION = "registration";
    private final UserFacade userFacade;
    private final CompanyFacade companyFacade;
    private final DeveloperFacade developerFacade;

    private final RegistrationValidator registrationValidator;

    @Autowired
    public RegistrationController(
            UserFacade userFacade,
            CompanyFacade companyFacade,
            DeveloperFacade developerFacade,
            RegistrationValidator registrationValidator
    )
    {
        this.userFacade = userFacade;
        this.companyFacade = companyFacade;
        this.developerFacade = developerFacade;
        this.registrationValidator = registrationValidator;
    }

    @PostMapping("/registration")
    public String registrationSubmit(
            @Valid
            @ModelAttribute("user") LoginRegistrationForm
loginRegistrationForm,
            BindingResult bindingResult
    )
    {
        registrationValidator.validate(loginRegistrationForm,
bindingResult);

        if (bindingResult.hasErrors())
        {
            return RegistrationController.REGISTRATION;
        }
        else
        {
            try
            {
                UserModel savedUser =
userFacade.saveUser(loginRegistrationForm);
                if (loginRegistrationForm.getType().equals("1"))
                {
                    developerFacade.saveDeveloper(loginRegistrationForm,
savedUser);
```

```
                }
                else
                {
                    companyFacade.saveCompany(loginRegistrationForm,
savedUser);
                }
                return RegistrationController.LOGIN;
            }
            catch (Exception e)
            {
                bindingResult.rejectValue("email",
RegistrationController.ERROR);

RegistrationController.LOGGER.error(RegistrationController.ERROR, e);
            }

        }
        return RegistrationController.REGISTRATION;
    }
}
```

Vidljivo je da se unutar registracije koriste mnoge zavisnosti, jedan od njih je i *RegistrationValidator* koji služi za provjeru ispravnosti podataka. Također ispod je prikaz kako izgleda njegova implementacija. On zapravo validira jesu svi podaci na formi uneseni i jesu li unesi u ispravnom formatu.

```
@Component
public class RegistrationValidator implements Validator
{
    private static final String EMAIL_REGEX = "\\b[A-Za-z0-9._%+-]+@[A-Za-
z0-9.-]+\\.[A-Za-z]{2,4}\\b";

    private final UserFacade userFacade;

    @Autowired
    public RegistrationValidator(UserFacade userFacade)
    {
        this.userFacade = userFacade;
    }

    @Override
    public boolean supports(Class<?> aClass)
    {
        return LoginRegistrationForm.class.equals(aClass);
    }

    @Override
    public void validate(Object o, Errors errors)
    {
        LoginRegistrationForm user = (LoginRegistrationForm) o;

        if (userFacade.getUserByEmail(user.getEmail()) == null)
        {
            validateEmail(user, errors);
            validatePassword(user, errors);
```

```
            validateMatchingPassword(user, errors);
        }
        else
        {
            errors.rejectValue("email", "validation.mail.exists");
        }
    }

    private void validateEmail(
            LoginRegistrationForm user,
            Errors errors
    )
    {
        if (StringUtils.isEmpty(user.getEmail()) ||
!validateEmailAddress(user.getEmail()))
        {
            errors.rejectValue("email", "validation.email.validity");
        }
    }

    private void validatePassword(
            LoginRegistrationForm user,
            Errors errors
    )
    {
        if (StringUtils.isEmpty(user.getPassword()) ||
StringUtils.isEmpty(user.getPasswordRepeat()))
        {
            errors.rejectValue("password", "validation.password.validity");
        }
    }

    private void validateMatchingPassword(
            LoginRegistrationForm user,
            Errors errors
    )
    {
        if (!user.getPassword().equals(user.getPasswordRepeat()))
        {
            errors.rejectValue("passwordRepeat",
"validation.password.matching");
        }
    }

    private boolean validateEmailAddress(final String email)
    {
        final Pattern pattern =
Pattern.compile(RegistrationValidator.EMAIL_REGEX);
        final Matcher matcher = pattern.matcher(email);
        return matcher.matches();
    }

}
```

Prethodno prikazani programski isječci događaju se ispod površine, odnosno korisnik ih ne može vidjeti. Stoga biti će prikazana HTML stranice za prikaz sadržaja registracije. Html stranica koristi kao dodatak koristi tehnologiju Thymeleaf.

```
<!DOCTYPE html>
<html xmlns:th="http://www.thymeleaf.org">
<head>
    <title>Registration page</title>
    <link rel="stylesheet" type="text/css"
href="webjars/bootstrap/3.3.7/css/bootstrap.min.css"/>
    <link type="text/css" href="css/login.css" rel="stylesheet"/>
    <link rel="icon" type="image/png" href="images/logo1-mdpi.png"/>
</head>
<body>

<div id="body" class="col-md-6 col-md-offset-3">
    <div class="panel panel-login">
        <img src="images/logo1-xxhdpi.png" class="img-responsive"
             style="margin-left: 38%; height:auto;margin-top: 5px;margin-
bottom: 5px;" alt="SoleX Logo"/>
        <div class="panel-heading">
            <div class="row">
                <div class="col-xs-6">
                    <a href="/login" id="login-form-link">Login</a>
                </div>
                <div class="col-xs-6">
                    <a href="#" class="active" id="register-form
        link">Register</a>
                </div>
            </div>
        </div>

        <div class="panel-body">
            <div class="row">


                <div class="col-lg-12">
                    <form th:action="@{/registration}" th:object="${user}"
        id="register-form" method="POST" role="form">

                        <div class="alert alert-danger"
        th:if="${#fields.hasErrors('*')}">
                            <p th:each="err : ${#fields.errors('*')}"
        th:text="${err}"></p>
                        </div>
                        <div class="form-group">
                            <input type="text" name="username"
        th:field="*{email}" id="reg_username" tabindex="1"
                                   class="form-control"
                                   required="required"
                                   placeholder="Enter e-mail address"
        value=""/>
                        </div>
                        <div class="form-group">
                            <input type="password" th:field="*{password}"
        name="password" id="reg_password" tabindex="2"
                                   required="required"
                                   class="form-control" placeholder="Enter
        password"/>
                        </div>
                        <div class="form-group">
                            <input type="password"
th:field="*{passwordRepeat}" name="password"
                                   id="reg_password_repeat" tabindex="2"
required="required"
```

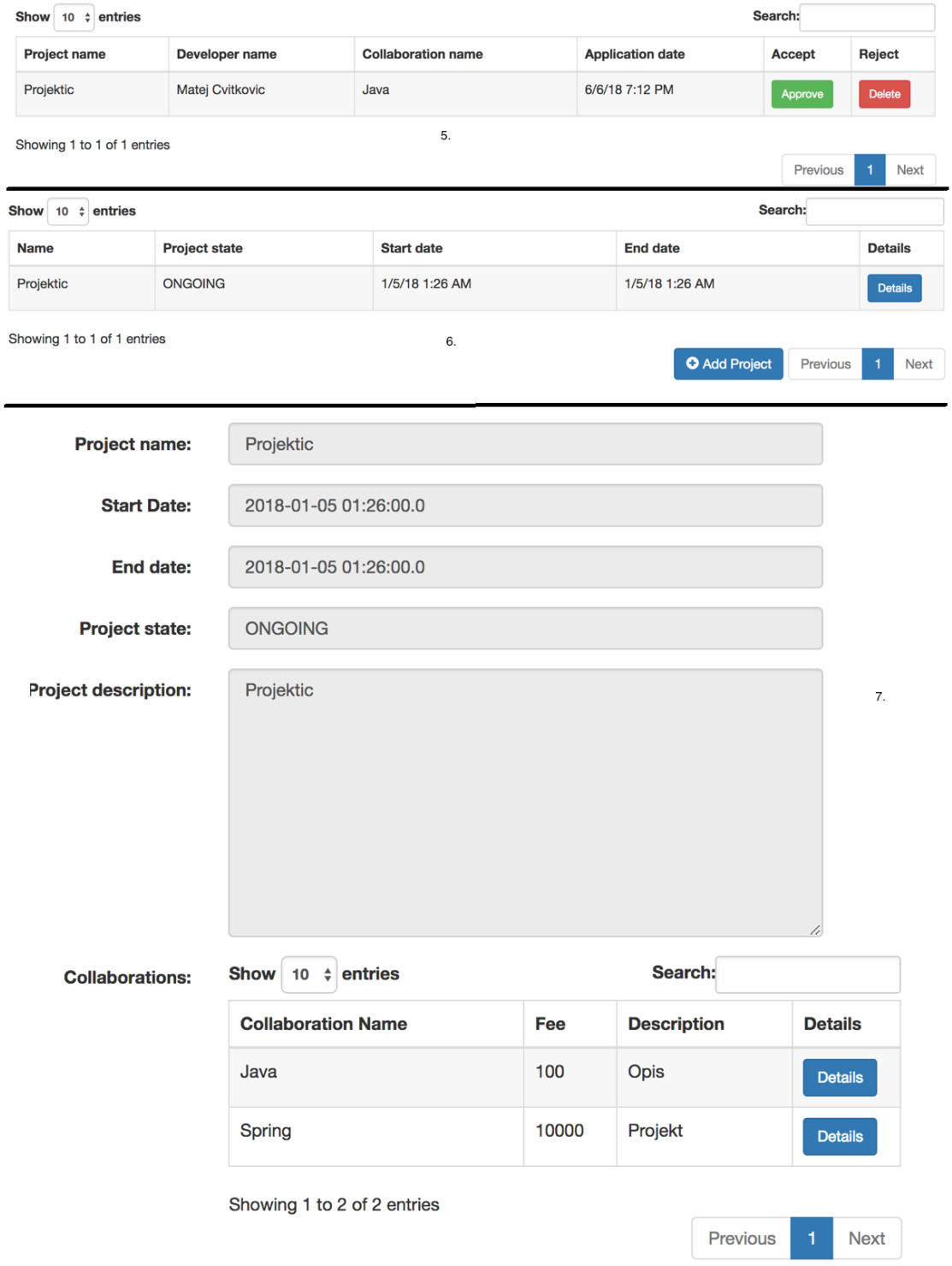
```
                                    class="form-control" placeholder="Repeat
your password"/>
                        </div>
                        <div class="form-group">
                            <select id="choice" class="form-control"
th:field="*{type}" required="required">
                                <option value="" disabled="disabled"
selected="selected">Choose role</option>
                                <option value="1">DEVELOPER</option>
                                <option value="2">COMPANY</option>
                            </select>
                        </div>
                        <div class="form-group">
                            <div class="row">
                                <div class="col-sm-6 col-sm-offset-3">
                                    <input type="submit" name="register-
submit" id="register-submit"
                                           tabindex="4" class="form-control
btn btn-register"
                                           value="Register Now"/>
                                </div>
                            </div>
                        </div>
                    </form>
                </div>
            </div>
        </div>
    </div>
</div>
<script
src="https://ajax.googleapis.com/ajax/libs/jquery/3.2.1/jquery.min.js"></sc
ript>
<script type="text/javascript"
src="webjars/bootstrap/3.3.7/js/bootstrap.min.js"></script>
</body>
</html>
```

Nakon uspješne registracije korisnika, isti se može prijaviti u aplikaciju. Kako u aplikaciji postoje dvije uloge, bilo je potrebno napraviti vlastitu logiku za mogućnost prijave svake od uloga. To je izvedeno implementacijom ispod prikazanog servisa.

```
@Service
public class CustomUserDetailsService implements UserDetailsService
{
    private final UserDAO userDAO;
    private final DeveloperDAO developerDAO;
    private final CompanyDAO companyDAO;

    @Autowired
    public CustomUserDetailsService(
            UserDAO userDAO,
            DeveloperDAO developerDAO,
            CompanyDAO companyDAO)
    {
        this.userDAO = userDAO;
        this.developerDAO = developerDAO;
        this.companyDAO = companyDAO;
```

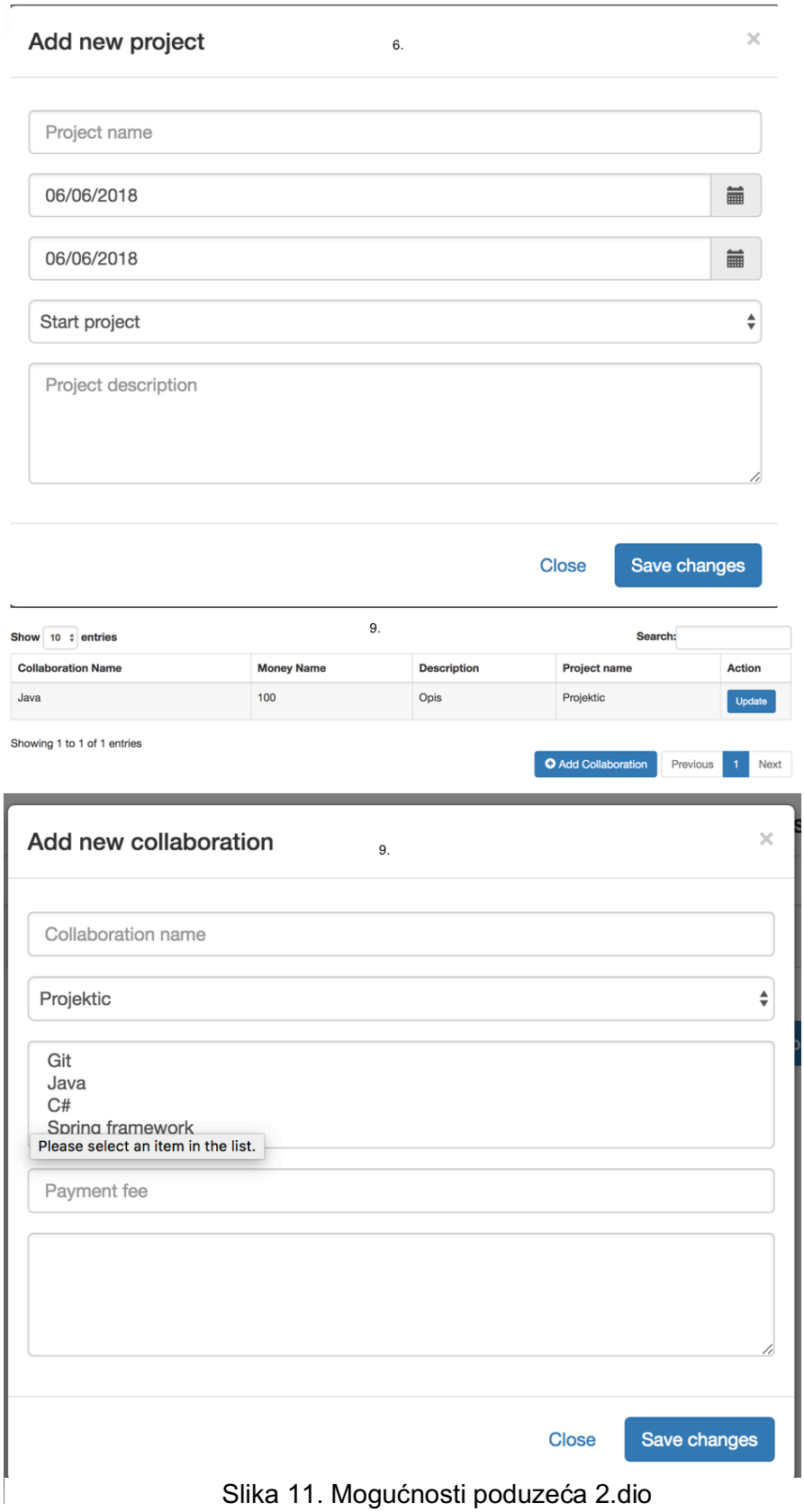
```
    }

    @Override
    public UserDetails loadUserByUsername(String s)
    {
        UserModel user = userDAO.findByEmail(s);
        ArrayList<String> userRoles = new ArrayList<>();
        if (null == user)
        {
            throw new UsernameNotFoundException("No user present with
username: " + s);
        }
        else
        {
            DeveloperModel dev = developerDAO.findByUser(user);
            if (dev == null)
            {
                CompanyModel comp = companyDAO.findByUser(user);
                if (comp == null)
                {
                    throw new UsernameNotFoundException("No user present
with username: " + s);
                }
                else
                {
                    userRoles.add("COMPANY");
                    return new CustomUserDetails(user, userRoles);
                }
            }
            userRoles.add("DEVELOPER");
            return new CustomUserDetails(user, userRoles);
        }
    }
}
```

Slika 10. sastoji se od dijelova 5, 6 i 7 iz tijeka rada. Oni prikazuju početnu stranicu uloge poduzeća gdje može reagirati na prijave, zatim mogućnost 6 prikazuje popis trenutnih projekata i od tamo može se pristupiti detaljima pojedinog projekta što je zapravo mogućnost 7 na slici.



Slika 10. Mogućnosti poduzeća 1.dio

Slika 11. sastoji se od dijelova 6 i 9 iz tijeka rada. Prikazane su mogućnosti dodavanje projekata koja je prikazana uz pomoć modala, zatim prikazana je mogućnost pregleda suradnji i njihovo dodavanje također uz pomoć modala.

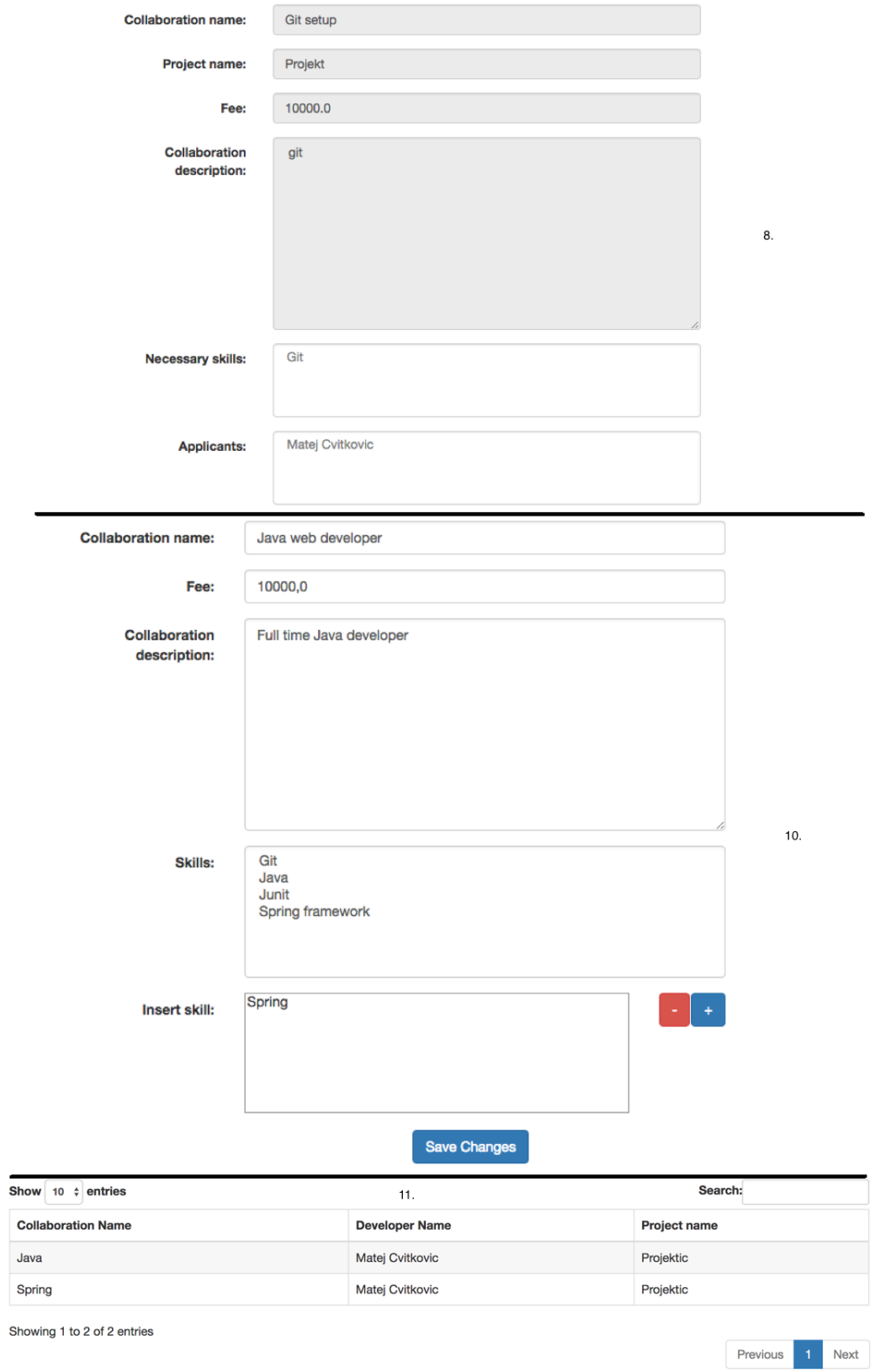


Slika 11. Mogućnosti poduzeća 2.dio

Na nekoliko mogućnosti koristi se funkcionalnost modala. On se koristio u situacijama gdje pojednostavljuje korištenje aplikacije, odnosno kako se ne bi morala otvarati isključivo nova stranica za dodavanje neke nove stavke. Ispod se nalazi isječak koda koji prikazuje način upotrebe modala u SoleX aplikaciji.

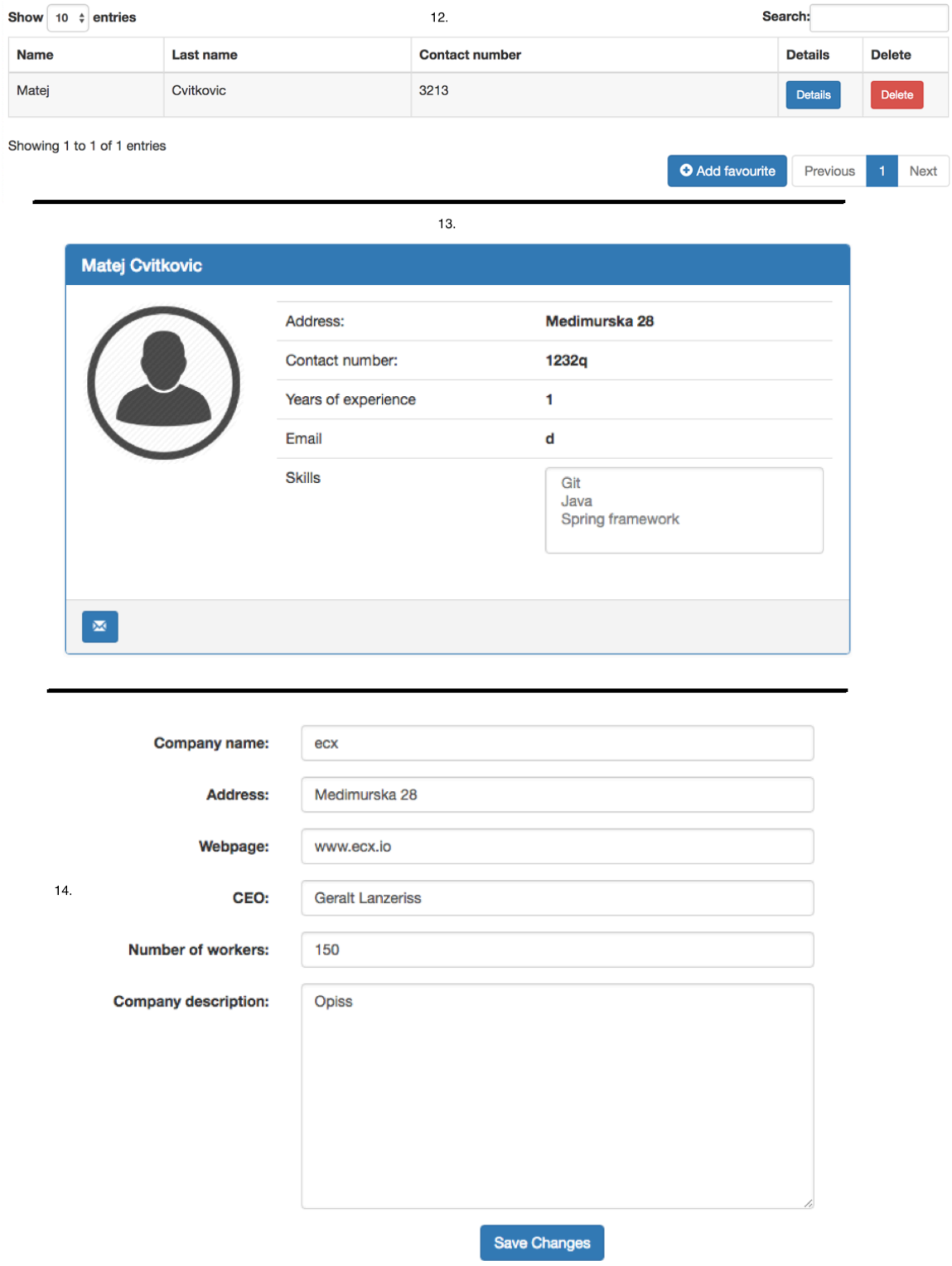
```
<div class="modal fade" tabindex="-1" role="dialog" id="modal-project"
        name="modal-project">
         <div class="modal-dialog" role="document">
             <div class="modal-content">
                 <div class="modal-header">
                     <button type="button" class="close" data-
        dismiss="modal" aria-label="Close"><span
                                 aria-hidden="true">&times;</span>
                     </button>
                     <h4 class="modal-title">Edit search</h4>
                 </div>
                 <div class="modal-body">
                     <form id="project-form"  method="POST"
                           role="form" style="display: block;">
                         <div class="hidden" id="error">
                             <div class="alert alert-danger" >
                                 <p id="errorcek">You must choose skill!</p>
                             </div>
                         </div>
                         <div id="slider" >
                             <div style="margin-bottom: 4px">
                                 <output for="rangevalue" id="labela"
        style="float: left">Percentage </output>
                                 <output id="rangevalue" style="font-size:
        15px ;float: right">10</output>
                             </div>
                             <input class="bar" type="range" id="rangeinput"
        value="10" onchange="rangevalue.value=value" style="margin-bottom:
        3px"/>

                         </div>
                         <div class="form-group">
                             <output for="strucnosti" id="lab"
        style="margin-bottom: 4px">Skills</output>
                             <select class="form-control" id="strucnosti"
        name="strucnosti" multiple="multiple">
                             </select>
                         </div>
                     </form>
                 </div>
                 <div class="modal-footer">
                     <a href="#" id="close" class="btn" data-
        dismiss="modal">Close</a>
                     <button id="submit" class="btn btn-primary">Search
        projects</button>
                 </div>
             </div><!-- /.modal-content -->
         </div><!-- /.modal-dialog -->
</div><!-- /.modal -->
```

Slika 12. prikazuje mogućnosti 8, 10 i 11 iz tijeka rada. 8 predstavlja pregled pojedine suradnje, 10 prikazuje ažuriranje suradnje i 11 prikazuje pregled prihvaćenih suradnji.



Slika 12. Mogućnosti poduzeća 3.dio

Slika 13. prikazuje mogućnosti 12, 13 i 14 iz tijeka rada. To su mogućnosti pregleda favorita gdje je moguće posjetiti mogućnost 13 tj. pregled profila favorita. Osim prethodne dvije funkcionalnosti u slici 13 nalazi se i ažuriranje profila poduzeća.



Slika 13. Mogućnosti poduzeća 4.dio

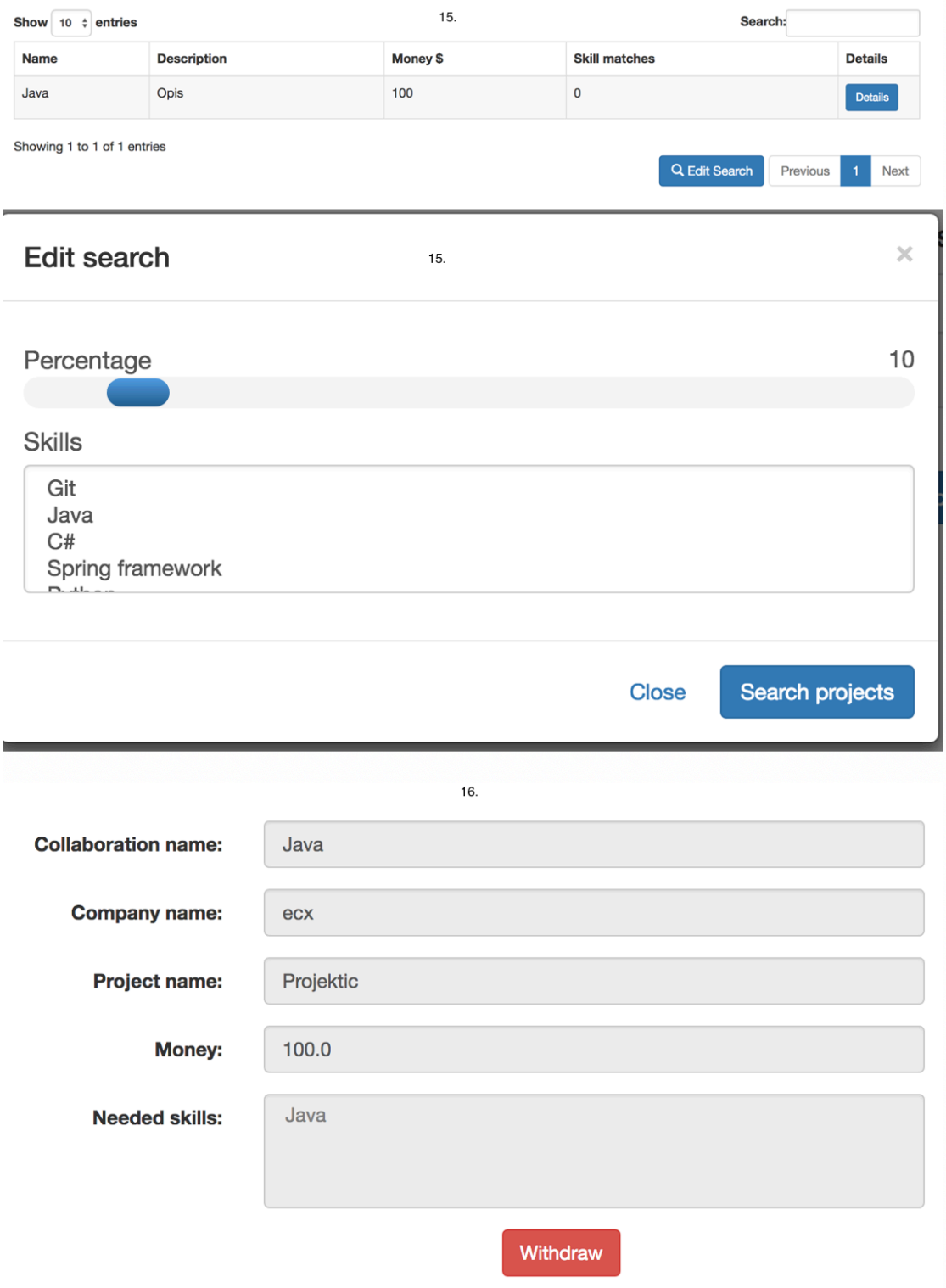
Za prikaz favorita potreban je samo djelić podataka koje sadrži *Developer* model. Stoga je napravljen DTO objekt koji korisniku vraća isključivo podatke koji su njemu potrebni i njega zanimaju. Navedeno sažimanje podataka i kreiranje novog objekta odvija se u Facade sloju i ispod je prikazan jedan od primjera gdje se koristi navedeni način rada. Za prebacivanje podataka iz Modela u DTO objekt korištena je mapper biblioteka koja uz pomoć refleksije kopira vrijednosti koje imaju iste nazive atributa u spomenutim klasama.

```
@Override
    public List<DeveloperDTO> getAllFavourites(CompanyModel company)
    {
        List<DeveloperModel> developersList =
        developerService.getFavorite(company.getId());
        ModelMapper mapper = new ModelMapper();

        return developersList
                .stream()
                .map(developer -> mapper.map(developer,DeveloperDTO.class))
                .collect(Collectors.toList());
    }
```

Na sljedećoj slici prikazana je mogućnost ažuriranja. Kako navedena mogućnost nije automatski napravljena unutar *Hibernate* okvira, ispod je prikazan kratki isječak koda koji za model *Company* vrši akciju ažuriranja u bazi podataka.

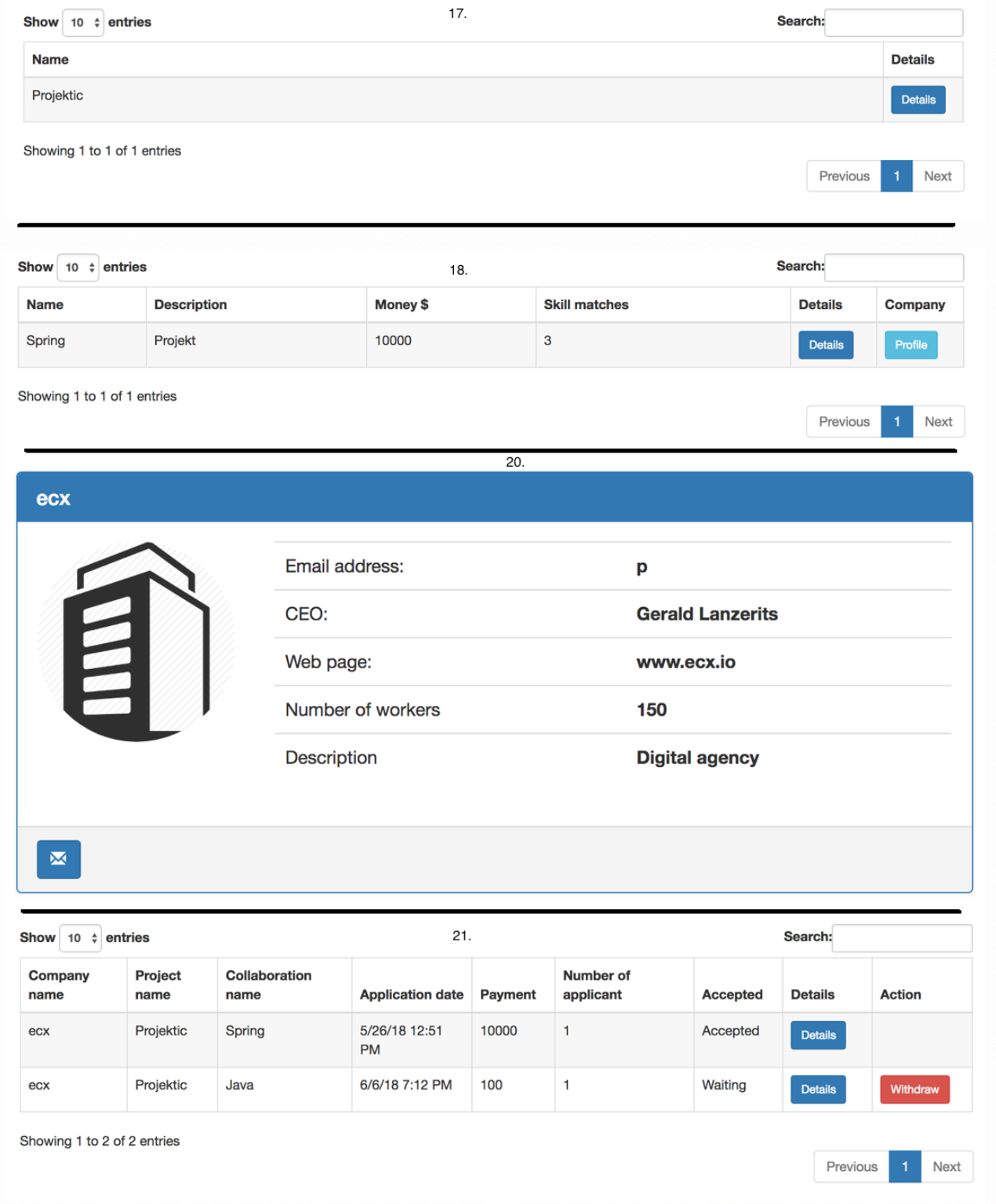
```
   @Repository
public interface CompanyDAO extends JpaRepository<CompanyModel, Long>
{
    CompanyModel findByUser(UserModel user);

    @Modifying
    @Query("UPDATE CompanyModel c SET c.address = :address, c.description =
:description, c.director = :director, c.name = :name, c.numOfWorkers =
:workers, c.webPage = :webpage WHERE c.user = :user")
    void updateCompany(
            @Param("address")
                    String address,
            @Param("description")
                    String description,
            @Param("director")
                    String director,
            @Param("name")
                    String name,
            @Param("workers")
                    int worker,
            @Param("webpage")
                    String webpage,
            @Param("user")
                    UserModel user);
}
```

Slika 14. prikazuje mogućnosti 15 i 16 iz tijeka rada. Mogućnost 15 je početna stranica razvojnog inženjera gdje su mu prikazane suradnje na koje se može prijaviti i također ima mogućnost filtriranja istih prema određenim parametrima. Mogućnost 16 prikazuje stranicu gdje razvojni inženjer može vidjeti njezine detalje i istovremeno se prijaviti.



Slika 14. Mogućnosti razvojnog inženjera 1.dio

Slika 15. prikazuje mogućnosti 17, 18, 20, 21 iz tijeka rada. Većina nabrojanih mogućnosti su uglavnom pregledi raznih podatka poput projekata i potrebnih suradnji, osim njih prikazana je mogućnost 20 koja je pregled profil poduzeća.



Slika 15. Mogućnosti razvojnog inženjera 2.dio

Mnoge od prethodno nabrojanih mogućnosti prikazuju podatke u obliku tablica. Prethodno je napomenuto da se za prikaz tablica koristi jQuery dodatak Datatables, stoga ispod je prikazan isječak koda koji demonstrira njegovo korištenje.

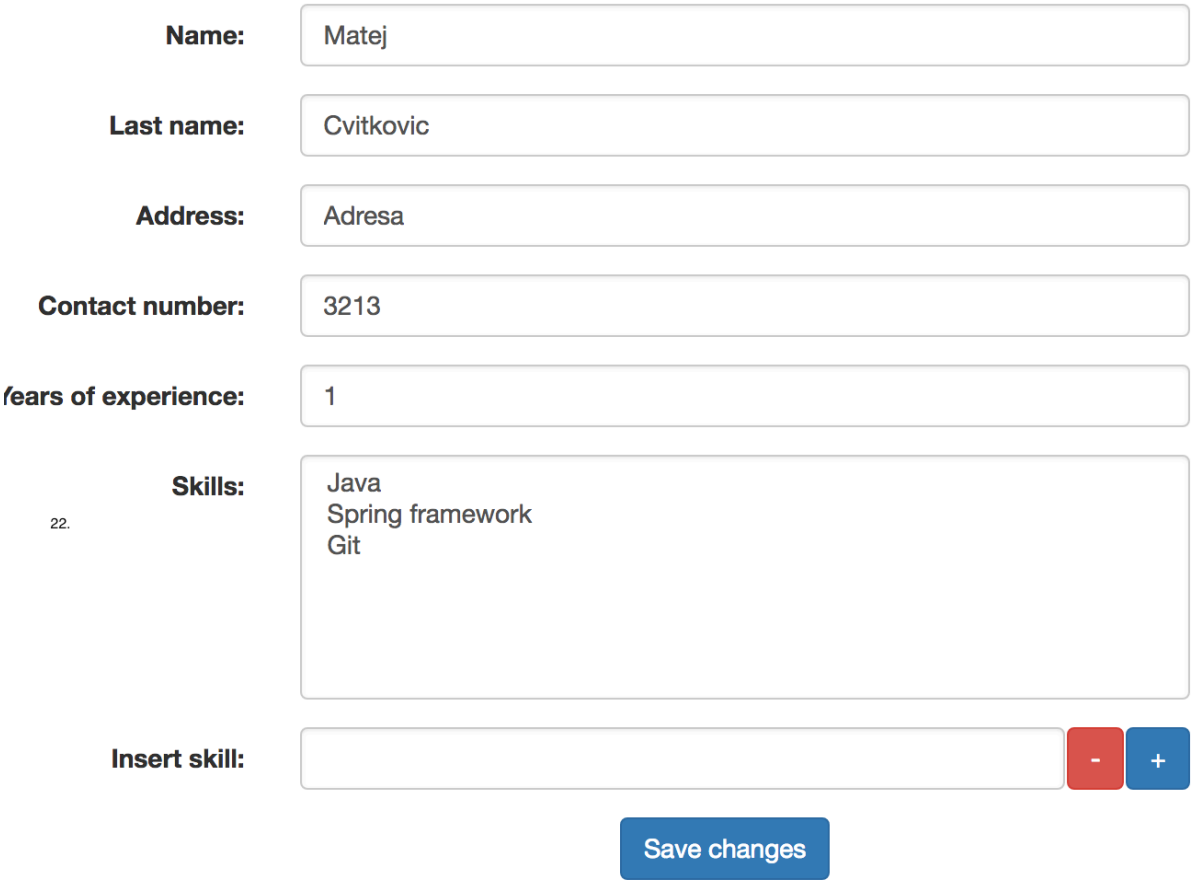
```
var table = $('#applicationTable').DataTable({
    "dom": 'flrtpi',
    "sAjaxSource": "/api/company_applications_list",
    "sAjaxDataProp": "",
    "order": [[0, "asc"]],
    "aoColumns": [
        {"mData": "projectName"},
        {"mData": "developerName"},
        {"mData": "collaborationName"},
        {"mData": "applicationDate"},
        {
            "mData": "applicationId",
            "bSortable": false,
            "mRender": function (applicationId) {
                return '<button id="approve_application" type="button"
class="btn btn-success btn-sm" onclick=approve_application(' +
applicationId + ")>" + 'Approve' + '</button>';
            },
            "sWidth": "10%"
        },
        {
            "mData": "applicationId",
            "bSortable": false,
            "mRender": function (applicationId) {
                return '<button id="delete_application" type="button"
class="btn btn-danger btn-sm" onclick=delete_application(' + applicationId
+ ")>" + 'Delete' + '</button>';
            },
            "sWidth": "10%"
        }
    ]
});
```

Kako bi navedeni dodatak mogao prikazati podatke, mora postojati RestController koji će mu te podatke ispostaviti. Ispod je prikazan isječak koda kako izgleda implementacija rest servisa za putanju */api/company_applications_list*.

```
@RestController
public class CompanyApplicationsRestController
{
    private static final String EMAIL_SUBJECT_SUCCESS =
"email.subject.success";
    private static final String EMAIL_CONTENT_SUCCESS =
"email.content.success";

    private static final String EMAIL_SUBJECT_FAILURE =
"email.subject.failure";
    private static final String EMAIL_CONTENT_FAILURE =
"email.content.failure";
```

```
    private final ApplicationFacade applicationFacade;

    private final CompanyFacade companyFacade;

    private final CollaborationFacade collaborationFacade;

    private final EmailFacade emailFacade;

    @Autowired
    public CompanyApplicationsRestController(
            ApplicationFacade applicationFacade,
            CompanyFacade companyFacade,
            CollaborationFacade collaborationFacade,
            EmailFacade emailFacade)
    {
        this.applicationFacade = applicationFacade;
        this.companyFacade = companyFacade;
        this.collaborationFacade = collaborationFacade;
        this.emailFacade = emailFacade;
    }

    @RequestMapping(path = "api/company_applications_list", method =
RequestMethod.GET)
    public List<CompanyApplicantsDTO> getCompanyApplications()
    {
        return applicationFacade.getCompanyApplications();
    }


}
```

Preostalo je prikazati zadnju mogućnost u aplikaciji pod brojem 22 u tijeku rada, to je mogućnost ažuriranja profila razvojnog inženjera i ona je vidljiva na slici 16.



Slika 16. Mogućnosti razvojnog inženjera 3.dio

Prethodne slike prikazuju sve mogućnosti aplikacije i kako one izgledaju. Svaki dio slike na sebi ima redni broj, pomoću kojeg je u tijeku rada moguće vidjeti koja je to funkcionalnost. Slike su spojene i iz njih je uklonjen glavni izbornik kako bi se uštedilo na prostoru.

8. Zaključak

Tema obuhvaćena u ovome radu je Uzorci dizajna u Spring programskom okviru. Spring je trenutno je najpopularniji Java web programski okvir. Primjenom kvalitetnih programerskih paradigmi i kvalitetnom primjenom odabranih uzoraka dizajna omogućava olakšani i ubrzani razvoj web aplikacija. Podržava spektar raznih aplikacija iz Spring grupacije koje olakšavaju razvoj složenih aplikacija. Neke od tih aplikacija s kojima je omogućena jednostavna integracija su Spring Data, Spring Security, Spring Social, Spring Mobile itd. Zbog navedenih stavki Spring predstavlja svestran i koristan programski okvir, koji nije bez razloga najpopularniji Java web programski okvir u trenutku pisanja ovoga rada. Jedan od nedostataka Springa je definitivno nedostatak kvalitetnog okvira koji omogućuje brzo i jednostavno stvaranje oku ugodnih korisničkih sučelja.

U programskom inženjerstvu uzorci dizajna su opće, ponovo iskoristivo rješenje za problem koji se najčešće pojavljuje u određenom kontekstu u programskom dizajnu. To nije gotov dizajn koji se može pretvoriti izravno u izvorni ili strojni kod. Oni su opis ili predložak za rješavanje problema koji se ponavljaju u mnogim različitim situacijama. Zapravo oni predstavljaju najbolje formalizirane prakse koje razvojni inženjer može primijeniti za rješavanje uobičajenih problema pri projektiranju aplikacije ili sustava (Gamma, Helm, Johnson i Vlissides, 2000). Preteča uzoraka dizajna je knjiga poznata pod nazivom Gang of Four (GOF) i nakon njezinog izlaska svijet je uočio kako popis kvalitetno opisanih i dokumentiranih programskih paradigmi može pomoći u razvoju i održavanju programskih aplikacija ili sustava. Stoga unutar Springa uzorci dizajna su korišteni promišljeno i kvalitetno kroz cijeli Spring programski okvir kako bi učinili njegov razvoj i korištenje jednostavnijim. Neki od uzoraka koji se koriste unutar Spring okvira su *Adapter*, *Template method*, *Builder pattern*, *Front Controller* te *Abstract Factory*. Treba napomenuti da su ovo samo neki od najučestalijih primjera, ali da postoji velik broj uzoraka koji nisu spomenuti, ali su korišteni unutar Springa. Bilo da bi olakšali posao razvojnim inženjerima ili ih Spring koristi samostalno, bez da se razvojni inženjeri svjesni njegove primjene.

Osim što se uzorci koriste unutar Springa, razvojni inženjeri otkrili su nekoliko kvalitetnih praksi koje se koriste prilikom razvoja složenih aplikacija. Ukoliko se zanemari standardna Spring MVC arhitektura, ona može biti dodatno unaprijeđena primjenom Facade uzorka, DAO slojem, proxyem , Bridge implementacijom itd., ali treba napomenuti kako ni uzorke ne treba koristiti samo da bi bili korišteni. Prilikom dizajniranja potrebno je uočiti i primijeniti isključivo uzorke koji će stvarno poboljšati strukturu aplikacije, olakšati održavanje i daljnji razvoj aplikacija.

Ovim radom su detaljno teorijski i praktično opisani pojmovi Spring programskog okvira i uzorci dizajna. Obrađeni su svi uzorci iz knjige GOF, uz JEE uzorke i neke od anti uzoraka. Kako osim Spring programskog okvira na tržištu postoji i velika konkurencija, napravljena je kratka analiza 5 najpopularnijih Java programskih okvira u trenutku pisanja. Analiza je raščlanjena na par faktora kako bi se lakše procijenili prednosti i nedostatci obuhvaćenih Java programskih okvira.

Za kraj kako bi se praktično primijenile stavke koje su prethodno teorijski opisane kreirana je Spring Boot aplikacija. Njezina arhitektura je također detaljno objašnjena, korišteni su neki od uzoraka koje je autor smatrao korisnima i koji su mu olakšali razvoj same aplikacije. Neki od korištenih uzoraka su *Facade*, *Iterator*, *Composite*, *Bridge*, *Builder pattern, Dependency Injection.* Osim arhitekture prikazan je i konačan izgled same aplikacije.

Programski kod aplikacije dostupan je na sljedećoj poveznici: https://bitbucket.org/cvitka/solex_web/src/dev/

Popis literature

Stephen Blackheath, Anthony Jones (2016) *Functional Reactive Programming.* New York: Manning Publications Co.

Marcus Biel *(2017) Shallow vs. Deep Copy in Java.* Preuzeto 15. travnja 2018. s: https://dzone.com/articles/java-copy-shallow-vs-deep-in-which-you-will-swim

Josh Bloch (2002) *Copy Constructor versus*. Preuzeto 13. lipnja 2018 s: https://www.artima.com/intv/bloch13.html

Sebastian Daschner (2017) *Architecting Modern Java EE Applications.* Birmingham: Packt Publishing.

Marten Deinum, Koen Serneels, Colin Yates, Seth Ladd, Christophe Vanfleteren *(2012) Pro Spring MVC: with Web Flow.* New York: Apress

Martin Fowler (2004) *Inversion of Control Containers and the Dependency Injection pattern*. Preuzeto 08. veljače 2018 s: https://martinfowler.com/articles/injection.html

Erich Gamma, Richard Helm, Ralph Johnson, John M. Vlissides (1997) *Design Patterns: Elements of Reusable Object-Oriented Software.* Toronto: Addison Wesley

Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides *(2000) Design patterns: Abstraction and Reuse of Object-Oriented Design* obavljen na European Conference on Object-Oriented Programming.
ECOOP 1993: ECOOP' 93 — Object-Oriented Programming *7th European Conference Kaiserslautern*, svezak 707, pp 406-431.
Preuzeto 26. veljače 2018. s: https://link.springer.com/chapter/10.1007/3-540-47910-4_21

Github *(2018) Spring Framework Versions*
Preuzeto 20. veljače 2018. s: https://github.com/spring-projects/spring-framework/wiki/Spring-Framework-Versions

Praveen Gupta (2010) *Spring Web MVC Framework for rapid open source J2EE application development: a case study*.
Preuzeto 22. veljače 2018. s: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.168.2988&rep=rep1&type=pdf

Rod Johnson (2005) *J2EE development frameworks*
Preuzeto 08.veljače 2018. s: http://ieeexplore.ieee.org/abstractx/document/1381270/

Dhrubojyoti Kayal (2008) *Java EE Spring Patterns*. New York: Apress

Adam Koblentz (2018) *The Ultimate Java Web Frameworks Comparison: Spring MVC, Grails, Vaadin, GWT, Wicket, Play, Struts and JSF*
Preuzeto 20. svibnja 2018 s: https://zeroturnaround.com/rebellabs/the-curious-coders-java-web-frameworks-comparison-spring-mvc-grails-vaadin-gwt-wicket-play-struts-and-jsf/

Dinesh Rajput (2017) *Spring 5 Design Patterns.* Birmingham: Packt Publishing.

K. Siva Prasad Reddy *(2017) Beginning Spring Boot 2.* New York: Apress.

Sourcemaking (2018) *Antipatterns*
Preuzeto 20. lipnja. 2018. s: https://sourcemaking.com/antipatterns

Alexander Shvets (2013) *Design Patterns Explained Simply.* Sourcemaking

Spring docs *(2018, a) Spring Framework Overview*
Preuzeto 06. veljače 2018. s: https://docs.spring.io/spring/docs/current/spring-framework-reference/overview.html

Spring docs (2018, b) Spring Core Technologies
Preuzeto 06. veljače 2018. s: https://docs.spring.io/spring/docs/current/spring-framework-reference/core.html#spring-core

Spring docs (2004) *Spring 1.0 Documentation*
Preuzeto 06. veljače 2018. s: https://docs.spring.io/spring-framework/docs/1.0.0/

Spring docs (2006) *Spring 2.0 Documentation*
Preuzeto 06. veljače 2018. s:https://docs.spring.io/spring-framework/docs/2.0.0/reference/

Spring docs (2009) *Spring 3.0 Documentation*
Preuzeto 06. veljače 2018. s: https://docs.spring.io/spring-framework/docs/3.0.0.RELEASE/spring-framework-reference/htmlsingle/

Spring docs (2013) *Spring 4.0 Documentation*
Preuzeto 06. veljače 2018. s: https://docs.spring.io/spring-framework/docs/4.0.0.RELEASE/spring-framework-reference/

Spring docs (2016) *Spring Framework Reference Documentation*
Preuzeto 06. veljače 2018. s: https://docs.spring.io/autorepo/docs/spring-framework/4.3.11.RELEASE/spring-framework-reference/pdf/spring-framework-reference.pdf

Spring docs (2017) *Spring 5.0 Documentation*
Preuzeto 06. veljače 2018. s: https://docs.spring.io/spring-framework/docs/5.0.0.RELEASE/spring-framework-reference/

Spring.io (2018) *Spring Boot*
Preuzeto 24. veljače 2018 s: https://projects.spring.io/spring-boot/

Spring docs (2018, c) *Spring Boot Reference Guide*
Preuzeto 24. veljače 2018 s: https://docs.spring.io/spring-boot/docs/2.0.0.RC2/reference/htmlsingle/

Craig Walls (2014) *Spring in Action, Fourth Edition.* New York: Manning Publications Co.

J. A. Zachman (1987) *A framework for information systems architecture* objavljen na IBM Systems Journal.
IBM Systems Journal(1987), svezak 26, stavka 3, pp 276-292.
Preuzeto 08. veljače 2018. s: http://ieeexplore.ieee.org/abstract/document/5387671/

ZeroTurnaround (2018) The curious coder's java web frameworks comparison.
Preuzeto 10. veljače 2018 s: https://zeroturnaround.com/rebellabs/java-web-frameworks-index-by-rebellabs/

Popis slika

Popis tablica