UNIVERZA V MARIBORU
FAKULTETA ZA ELEKTROTEHNIKO,
RAČUNALNIŠTVO IN INFORMATIKO

Nik Orter

Izdelava spletne aplikacije s pomočjo okvirja Grails

Diplomsko delo

Maribor, avgust 2015

Izdelava spletne aplikacije s pomočjo okvirja Grails

Diplomsko delo

Študent:	Nik Orter
Študijski program:	Visokošolski študijski program
	Računalništvo in informacijske tehnologije
Mentor:	red. prof. dr. PETER KOKOL, univ. dipl. inž. el.
Lektor:	Denis Kalamar, prof., mag. inž.

Univerza v Mariboru

Fakulteta za elektrotehniko,
računalništvo in informatiko
Smetanova ulica 17
2000 Maribor, Slovenija

FERI

Številka: E1065855
Datum in kraj: 27. 05. 2015, Maribor

Na osnovi 330. člena Statuta Univerze v Mariboru (Ur. l. RS, št. 46/2012)
izdajam

SKLEP O DIPLOMSKEM DELU

1. **Niku Orterju,** študentu visokošolskega strokovnega študijskega programa RAČUNALNIŠTVO IN INFORMACIJSKE TEHNOLOGIJE, se dovoljuje izdelati diplomsko delo.

2. **MENTOR:** **red. prof. dr. Peter Kokol**

3. **Naslov diplomskega dela:**
   IZDELAVA SPLETNE APLIKACIJE S POMOČJO OKVIRJA GRAILS

4. **Naslov diplomskega dela v angleškem jeziku:**
   DEVELOPMENT OF WEB APPLICATION WITH GRAILS FRAMEWORK

5. Diplomsko delo je potrebno izdelati skladno z "Navodili za izdelavo diplomskega dela" in ga oddati v treh izvodih (dva trdo vezana izvoda in en v spiralo vezan izvod) ter en izvod elektronske verzije do 30. 09. 2015 v referatu za študentske zadeve.

Pravni pouk: Zoper ta sklep je možna pritožba na senat članice v roku 3 delovnih dni.

Dekan:
red. prof. dr. Borut Žalik

UNIVERZA V MARIBORU FAKULTETA ZA ELEKTROTEHNIKO, RAČUNALNIŠTVO IN INFORMATIKO

Obvestiti:
- kandidata,
- mentorja,
- odložiti v arhiv.

Zahvala

»Sedaj smo vsi povezani z internetom kot nevroni v velikanskih možganih.«
~ Stephen Hawkings

Zahvaljujem se mentorju red. prof. dr. Petru Kokolu za
svetovanje in pomoč pri izdelavi diplomskega dela.
Zahvaljujem se Denisu Kalamarju, prof., mag. inž.
za lektoriranje diplomskega dela.

Izdelava spletne aplikacije s pomočjo okvirja Grails

Ključne besede: Grails, Groovy, Spring, Hibernate, spletna aplikacija

UDK: 004.774.6(043.2)

Povzetek

Diplomsko delo opisuje okvir Grails. Predstavljene so glavne lastnosti okvirja in njegovi najbolj pomembni gradniki. V nadaljevanju predstavljamo programski jezik Groovy, ki je privzet programski jezik za okvir Grails, ter opisujemo glavne tehnologije, ki jih uporablja okvir za svoje delovanje. Na koncu diplomskega dela predstavljamo uporabo okvirja v praksi.

Development of web application with Grails framework

Key words: Grails, Groovy, Spring, Hibernate, Web application

UDK: 004.774.6(043.2)

Abstract

Thesis described framework Grails. It presented main properties of framework and most important components of framework. Below we present programming language Groovy which is default programming language for Grails framewok and then we describe main technologies that are used for framework workflow. At the end of thesis we present usage of framework in real project.

Kazalo

KAZALO SLIK

SEZNAM UPORABLJENIH KRATIC

XML – Extensible Markup Languaage
MVC – Model View Controller
JVM – Java Virtual Machine
ORM – Object Relational Mapping
GORM – Grails Object Relational Mapping
GSP – Groovy Server Page
GGTS – Groovy/Grails Tool Suite
STS – SpringSource Tool Suite
GDK – Groovy Development Kit
API – Application Programming Interface

1 UVOD

Živimo v času, v katerem je potrebno narediti veliko v zelo kratkem času. Tako je tudi pri spletnih aplikacijah, kjer moramo od njene ideje do realizacije priti v čim krajšem času. Hitrost razvoja aplikacije je zelo odvisna od produktivnosti razvijalcev, v ta namen so si razvijalci začeli pomagati z okvirji, ki jim omogočajo hitreje razviti aplikacijo. Razvilo se je zelo veliko različnih spletnoaplikacijskih okvirjev, katere lahko uporabimo pri izdelavi spletnih aplikacij in s tem prihranimo na času, ki ga porabimo za izdelavo spletne aplikacije. Da dvignemo produktivnost razvijalcev s pomočjo okvirja, je zelo pomembno, da so okvirji lahko razumljivi in da uporabljajo že znane sintakse. Eden takšnih okvirjev je Grails, ki z uporabo različnih mehanizmov zelo poenostavi implementacijo spletne aplikacije. Vsak okvir pa ima svoje prednosti in slabosti, zato moramo vedeti, kdaj uporabiti okvir, ki najbolje zadosti našim potrebam.

V diplomski nalogi smo se osredotočili na preučevanje okvirja Grails in na njegovo uporabo v praksi. Preučili smo, kako se Grails integrira s tehnologijami Groovy, Spring, Hibernate in ostalimi. Primerjali smo programski jezik Groovy, katerega uporablja okvir Grails s programskim jezikom java. Raziskali smo, kako se ukvarja s persistenco podatkov in kako je z razširitvami okvirja. Po pridobljenem znanju o okvirju Grails pa smo izdelali primer spletne aplikacije, ki uporablja okvir Grails, in s tem na primeru pokazali praktično uporabo okvirja. Na koncu smo raziskali teze, kot so: »Kdaj uporabiti okvir Grails«, »Kakšne so glavne prednosti okvirja« in »Kakšne so glavne pomanjkljivosti okvirja Grails«.

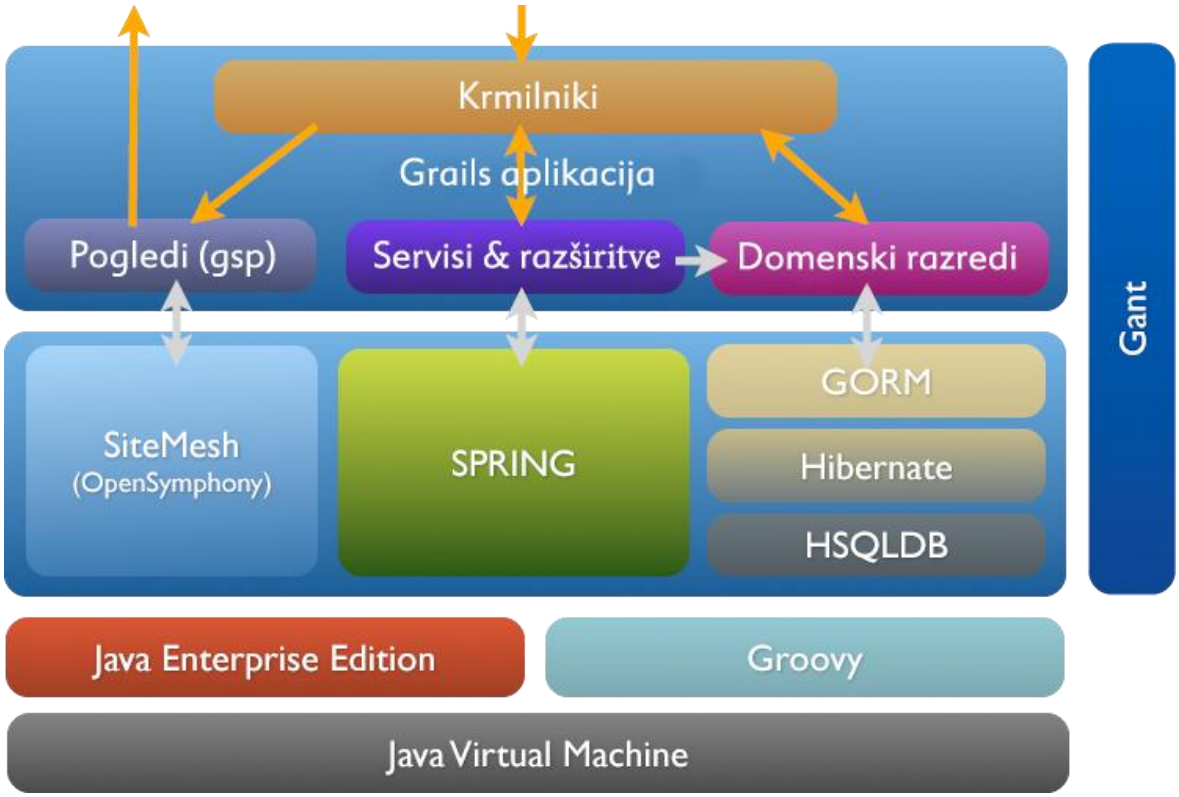
V drugem poglavju smo opisali najbolj pomembne komponente okvirja Grails. Opisali smo njegovo delovanje in predstavili razvijalna okolja, katera nudijo podporo za okvir Grails. V nadaljevanju drugega poglavja smo opisali tehnologije, ki jih uporablja okvir Grails za svoje delovanje. Vsako tehnologijo smo opisali na splošno in specifično glede na to, kako okvir Grails uporablja omenjeno tehnologijo.

V tretjem poglavju smo opisali uporabo okvirja v praksi, pri čemer smo izdelali spletno aplikacijo s pomočjo okvirja Grails in slednjo tudi predstavili.

2 METODOLOGIJE

2.1 Okvir Grails

Grails je zmogljiv spletnoaplikacijski okvir, namenjen platformi Java. Glavni namen okvirja Grails je povečati produktivnost razvijalcev in s tem prihraniti na času, ki je potreben za izdelavo spletne aplikacije. Grails uporablja že obstoječe tehnologije, kot sta Hibernate in Spring. Lepo se integrira z JVM (Java Virtual Machine), kar zagotavlja uporabo zmogljivih funkcij, vključno z integriranim ORM, domenskimi specifični jeziki, metaprogramiranjem v času izvajanja in v času prevajanja ter asinhronim programiranjem [1]. Arhitektura okvirja Grails je prikazana na Slika 2.1.



Slika 2.1: Arhitektura okvirja

Okvir Grails uporablja programski jezik Groovy, ki je nekakšna izboljšana verzija programskega jezika Java. Grails je bil znan kot Groovy on Rails, ampak so morali to ime kasneje spremeniti zaradi zahteve Davida Heinemeierja Hanssona, ustanovitelja Ruby on Rails. [2]

Grails je bil razvit predvsem z naslednjimi cilji [2]:

- Zasnovati spletnoaplikacijski okvir, ki bo lahko tekel na javanski platformi.
- Uporabiti obstoječe javanske tehnologije, kot sta Hibernate in Spring.
- Ponuditi konsistentni okvir za razvoj spletnih aplikacij.
- Olajšati razvoj spletne aplikacije z vključevanjem v naprej pripravljenega spletnega strežnika in avtomatskim nalaganjem virov.

Za zagotavljanje visoke produktivnosti so pri okvirju Grails dosegli naslednje prednosti [2]:

- Ni potrebno dodatno konfigurirati XML, kot je to potrebno pri tradicionalnih javanskih okvirjih.
- Okvir je narejen tako, da je po namestitvi že takoj pripravljen za uporabo, pri tem nam ni potrebno kar koli dodatno konfigurirati.
- Funkcionalnosti različnih razredov lahko združimo z razredom mixins.
- Sloj »Object Relational Mapping« (ORM), ki je narejena nad Hibernetom in je enostaven za uporabo.
- Izrazni pogled tehnologije, imenovan Groovy Server Pages (GSP).
- Vpeljava injiciranja odvisnosti z vgrajenim kontejnerjem Spring.
- Sistem, zgrajen na osnovi Gradle.
- Vgrajen kontejner Tomcat, ki je narejen za sprotno osveževanje aplikacije med njenim razvojem.
- Podpora za internacionalizacijo, ki je zgrajena na jedru Spring in konceptu MessageSource.

Grails je del platforme Java, kar omogoča zelo lahko integracijo s knjižnicami Jave, okvirji in obstoječo bazo kod. Grails ponuja transparentno integracijo razredov, ki so preslikani s Hibernate ORM. To pomeni, da lahko obstoječe aplikacije, ki uporabljajo Hibernate, uporabijo okvir Grails brez ponovnega prevajanja kode ali ponovne konfiguracije Hiberneta. Grails uporablja okvir Spring, v bistvu bi lahko rekli, da je okvir Grails aplikacija Spring MVC. Okvir Spring se lahko uporablja za vključevanje dodatnih Spring beanov in jih je mogoče integrirati v kontekst aplikacije. Okvir SiteMesh pa se uporablja za upravljanje predstavitvene plasti in za poenostavitev razvoja strani preko robustnega sistema predlog.

2.1.1 Servisi

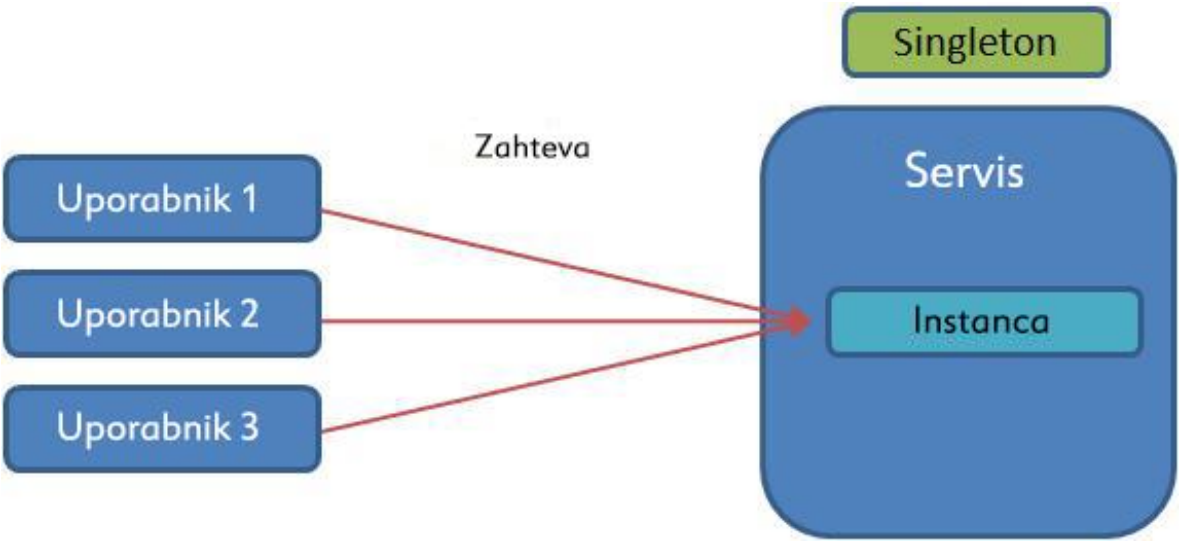
Servisi v okvirju Grails so namenjeni logiki aplikacije, ostale akcije, kot so preusmeritve in odzivi na zahteve, pa spadajo v krmilnike. Razvijalci okvirja Grails odsvetujejo, da je jedro aplikacijske logike vpeto v krmilnike, saj s tem ne spodbujamo ponovno uporabo kode.

Razred Groovy v aplikaciji postane avtomatsko servis Grails, če ga damo v mapo grails-app/services in če se ime razreda konča na Service.

Servisi se v večini primerov ukvarjajo z usklajevanjem logike med domenskimi razredi in se zato pogosto ukvarjajo s persistenco, ki vključuje velike operacije. Glede na naravo servisov mnogokrat pridemo do potrebe po uporabi transakcij. V takšnem primeru lahko programsko dodamo zahtevo po transakciji s pomočjo metode withTransaction, ampak to ni najboljši način, saj se čez čas začnemo ponavljati in ne izkoristimo celotne moči, ki nam jo ponuja transakcijska abstrakcija Spring.

V starejših verzijah okvirja Grails je ukaz create-service generiral začetni razred z lastnostjo static transactional = true, kar je bilo kasneje odstranjeno, saj so to privzete nastavitve in je eksplicitna deklaracija odveč. Eksplicitno deklariramo transakcijski atribut le v primeru, če vse metode v razredu niso transakcijske, npr., ko v razredu ne dostopamo do podatkovne baze. Servis torej avtomatsko postane transakcijski, če nima postavljene lastnosti »transactional« ali pa v servisu obstaja vsaj ena implementacija Spring anotacije »@Transactional«.

Servisi v okvirju Grails so avtomatsko registrirani, kot Spring bean, in so privzeto »Singletone«, kar pomeni, da obstaja samo ena instanca servisa (Slika 2.2). Servis postane Spring bean v vsakem primeru, ne glede, ali je transakcijski ali ne. [3]



Slika 2.2: Servis Singleton

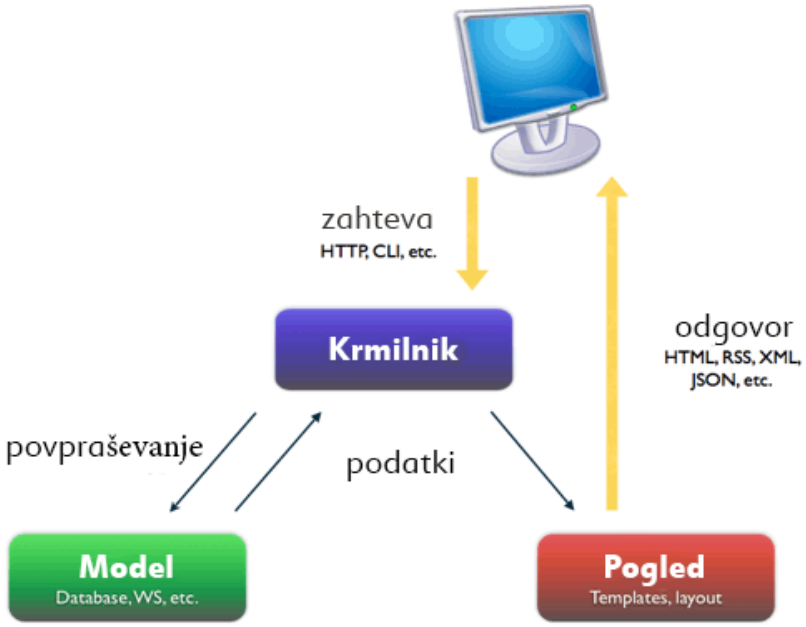
Dostop do servisnih metod privzeto ni sinhroniziran, tako da nič ne preprečuje hkratno izvedbo metod. Ker so servisi privzeto singletone in se lahko uporabljajo sočasno, moramo biti zelo pazljivi s shranjevanjem podatkov v servisu. Lahko pa spremenimo takšno obnašanje, tako da servis uvrstimo v točno določeno področje. Grails ponuja podporo za naslednja področja:

- prototype: nov servis se ustvari vedno, kadar ga injiciramo v drug razred
- request: nov servis se ustvari za vsako zahtevo
- flash: nov servis bo ustvarjen za trenutno in naslednjo zahtevo
- flow: v spletnem toku servis obstaja za čas toka
- conversation: v spletnem toku servis obstaja v času pogovora
- session: servis se ustvari za vsako sejo uporabnika
- singleton (privzeto): obstaja samo ena instanca servisa

Glavna prednost servisov Grails je zmožnost uporabe funkcije okvirja Spring, ki omogoča injiciranje odvisnosti. Grails podpira »injiciranje odvisnosti s konvencijo«, kar pomeni, da lahko dosežemo injiciranje odvisnosti tako, da poimenujemo lastnost v razredu enako, kot je poimenovan servis, in s tem avtomatsko pridobimo instanco servisa. Kontejner najprej ustvari bean, nato izvede injiciranje odvisnosti, kot je specificirano v konfiguraciji .

2.1.2 Krmilniki

Krmilnik upravlja zahteve in ustvarja ali pripravi odgovor na zahtevo. Krmilnik lahko ustvari odgovor direktno ali pa to prenese na pogled. Primer vloge krmilnika v strukturi MVC je prikazan na Slika 2.3.



Slika 2.3: Vloga krmilnika v strukturi MVC

Za kreiranje krmilnika enostavno dodamo razred v mapo grails-app/controllers in poskrbimo, da se ime krmilnika konča na »Controller«, s tem je naš izdelan razred avtomatsko predstavljen kot krmilnik. [3]

Krmilnik lahko ima več javnih metod, pri čemer vsaka od njih vodi na določen naslov URL. V prejšnjih verzijah okvirja Grails so bile akcije implementirane s pomočjo »Closures«. To se še vedno podpira, ampak se ne priporoča, saj ima uporaba javnih metod naslednje prednosti:

- bolj optimalno uporabljamo pomnilnik
- uporabimo lahko krmilnike brez stanj
- lahko prekrmilimo akcije od podrazreda in kličemo prekrmiljeno metodo iz nadrazreda
- metode so lahko predstavljene s standardnimi mehanizmi proxy

V okvirju Grails lahko v krmilniku implementiramo naslednje operacije:

- Upravljanje usmerjevalne logike
- Sklicevanje operacij GORM za manipuliranje podatkov v podatkovni bazi
- Prikaz rezultatov uporabnikom

Ni priporočljivo, da implementiramo vse zgoraj naštete stvari v krmilniku. Grails omogoča razvijalcem implementacijo vseh zgoraj naštetih operacij v krmilniku, ampak naj bi se tega izogibali. Pravi namen krmilnika je:

- Sprejeti zahtevo od uporabnika
- Sklicati najbolj primerno poslovno logiko
- Sklicati pogled za prikaz rezultatov

Poslovno logiko naj bi implementirali v servisi in potem do nje dostopali preko krmilnika. Za prikaz rezultatov pa poskrbimo v Groovy Server Pages, preko krmilnika pa samo podamo rezultat, ki ga dobimo od servisa. [4]

2.1.3 Domenski razredi

Domenski razredi predstavljajo persistenco entitete, ki je preslikana v podatkovno bazo. Domenski razredi so razredi Groovy, ki se nahajajo v mapi grails-app/domain. Polja, katerim ne določimo, ali so zasebna ali javna, so avtomatsko persistentna. Ime razreda predstavlja ime tabele v podatkovnem modelu, ki se generira iz domenskih razredov. Imena posameznih polj pa predstavljajo imena posameznih stolpcev v tabeli.
Domenskim razredom lahko dodamo omejitve s pomočjo omejitev, katere deklariramo v statični spremenljivki constrains. Nekateri primeri vsebujejo ničelne in unikatne omejitve. Te omejitve pridejo v poštev pri validaciji, katera ni ločena od domenskega razreda, ampak se proži z metodo validacije ali z metodo shranjevanja, katera je prožilec in edini način za shranjevanje instance pri uspešni validaciji. [3]

2.1.4 Podpora IDE

Pri razvijanju aplikacije Grails ni nujno, da uporabljamo ukazno vrstično programiranje, saj obstaja veliko orodij, ki podpirajo programiranje v programskem jeziku Groovy in nudijo podporo okvirju Grails. Med najbolj znanimi orodji so Groovy/Grails Tool Suite (GGTS), Intellij IDEA in NetBeans IDE. Vsak izmed njih podpira kreiranje aplikacije Grails in uporabo razširitev ter generiranje artefaktov in zagon aplikacije direktno iz razvojnega orodja v razvijalnem načinu. GGTS je podaljšek razvijalnega okolja SpringSource Tool Suite (STS), katero je po osnovi zgrajeno iz razvijalnega okolja Eclipse. Vključuje razširitev Groovy-Eclipse, ki služi preverjanju sintakse in samodokončanje kode Groovy. IntelliJ je komercialno razvijalno okolje, ki nudi podporo programskima jezikoma Java in Groovy, ampak v brezplačni verziji ne nudi podporo za okvir Grails. Lahko pa za razvoj aplikacije Grails uporabimo urejevalnike besedila. Med najbolj priljubljenima sta TextMate za uporabnike okolja Mac OS in Sublime Text za uporabnike okolja Windows. [3]

2.1.5 Groovy server pages (GSP)

Groovy server pages ali GSP je privzeta predstavitvena tehnologija, ki je uporabljena za prikaz pogledov v Grailsu. Zelo je podobna tehnologiji JSP, kjer lahko kombiniramo statično in dinamično vsebino. Okvir Grails je poskrbel za zelo preprosto uporabo in dovolj zmogljivo, da zadosti za večino primerov uporabe. [5]
V GSP lahko z uporabo kode Groovy dostopamo do domenskih razredov, naredimo zanke, v katerih lahko izpišemo vsebino glede na rezultat, ki ga vrne krmilnik. Ustvarimo lahko obrazce, preko katerih se po potrditvi izvrši zahteva na določen krmilnik, katero lahko pošljemo sinhrono ali asinhrono.

Datoteke Gsp imajo prostor v mapi grails-app/views, kamor lahko shranimo vse naše poglede. Pogledi so prikazani avtomatsko s pomočjo konvencije ali pa z metodo render. Gsp ima po navadi model, kar je zbirka podatkov, katero lahko uporabimo in prikažemo. Model v stran Gsp posredujemo preko krmilnika. [6]

2.1.6 Preslikave URL

Grails že od namestitve aplikacije zagotavlja delujočo preslikavo URL. Privzeto imamo nastavljeno preslikavo URL, ki je sestavljena tako, da URL na prvem mestu vsebuje ime krmilnika, na drugem pa ime akcije, na katero se sklicujemo, na primer naslov URL, ki se sklicuje na KnjigaController in akcijo index bi se glasil /knjiga/index. Ime akcije je opcijsko, v primeru, da ga ne navedemo, se pokliče privzeta akcija krmilnika. Če v naslov URL podamo številko, se bo poklicala navedena akcija, v *post* parametrih pa se bo podal id, ki smo ga navedli v naslovu URL. V primeru, da želimo definirati svojo preslikavo URL, pa to enostavno definiramo v datoteki URL mappings. Definiranje preslikav URL je zelo pogosto pri pisanju aplikacij. Tehnika, ki se uporablja pri okvirju Grails za preslikave URL, je zelo močna in enostavna. Za konfiguracijo preslikav URL moramo uporabiti programski jezik Groovy, ni pa potrebno ničesar definirati v datotekah XML. [7]

```
class UrlMappings {

    static mappings = {
        "/$controller/$action?/$id?(.$format)?"{
            constraints {
                // apply constraints here
            }
        }

        "/"(view:"/index")
        "500"(view:'/error')
        "404"(view:'/notFound')
    }
}
```

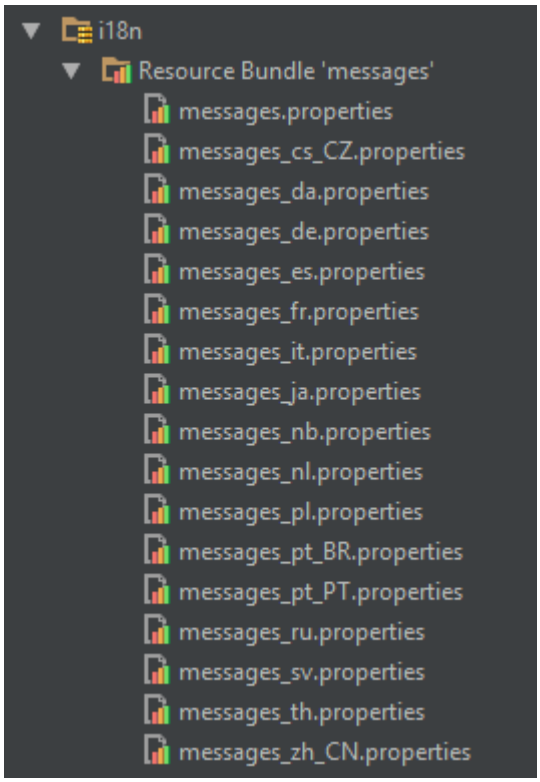
Slika 2.4: Privzeta preslikava URL

Ključni del na Slika 2.4 je niz /$controller/$action/$id?(.$format)?. Vprašaji, ki so v nizu, pomenijo, da je parameter opcijski, torej sta v našem primeru krmilnik in akcija obvezna parametra, id in format pa opcijska. Če preslikava vsebuje opcijske parametre, morajo ti biti zapisani na koncu niza.

V preslikavo URL lahko dodamo tudi, katere metode zahtev so dovoljene za določeno preslikavo URL. Obstajajo opcije za post, get, put in update. Podamo lahko tudi, kateri krmilnik in katera akcija se pokličeta na določenemu naslovu URL. [7]

2.1.7 Lokalizacija

Ena najboljših lastnosti spletnih aplikacij je, da zelo lahko dosežemo zelo veliko maso ljudi. Pri uvajanju spletnih aplikacij za široko občinstvo se mora spletna aplikacija pogosto prilagoditi različnim državam, kar pomeni, da moramo podpirati več jezikov. Na primer, če bo do naše spletne aplikacije dostopal nekdo s Portugalske, bo celotna spletna aplikacija v portugalskem jeziku. Včasih lokalizacija zahteva več kompleksnosti, kot le prikaz teksta v določenem jeziku, saj moramo glede na lokacijo zahteve uporabiti drugačno poslovno logiko. Lokalizacijo lahko izvedemo na več načinov: en način je, da za vsako jezikovno skupino ustvarimo novo aplikacijo. Ta pristop lahko prinese veliko težav. Vzdrževanje vseh verzij aplikacije bi vzelo precej časa. Boljši pristop je, da ustvarimo eno verzijo aplikacije, ki je fleksibilna in sposobna prikazovati sporočila v več jezikih. Za uporabo lokalizacije je najbolje, da vsa sporočila v različnih jezikih hranimo v eni datoteki in ne implementiramo vse v datoteke gsp. Pri ustvarjanju aplikacije Grails okvir samodejno ustvari datoteke, ki so namenjene lokalizaciji. Datoteke imajo svoj prostor v zbirki i18n (Slika 2.5). [7]



Slika 2.5: Privzeto ustvarjene datoteke za lokalizacijo

2.1.8 Ajax

Ajax je zelo pomemben del programiranja spletnih aplikacij. Originalno je bil razvit s strani Microsofta za podporo spletne verzije Outlooka. Prednost Ajaxa, zaradi katere se najbolj uporablja, je princip, da se stran v celoti ponovno ne nalaga, ampak lahko osvežimo samo določen del strani. [7]

Grails ima implementirane značke, katere poenostavijo delo z Ajaxom. Značke Grails za Ajax:

- remoteField
- remoteFunction (Slika 2.6)
- remoteLink
- formRemote (Slika 2.7)

```
<g:remoteFunction controller="izposojaRezervacija"
action="odjavaRezervacije"
onSuccess="updateTable('tr' + knjigaId)"
params="parameters"/>
```

Slika 2.6: Uporaba značke remoteFunction

```
<g:formRemote  name="iskanje"
               url="[controller: 'knjiga', action:'iskanjePoKriteriju']"
               update="container">
    <input type="text" name="naslov" placeholder="Naslov">
    <input type="text" name="avtor" placeholder="Avtor">
    <button type="submit" id="login-button">Išči</button>
</g:formRemote>
```

Slika 2.7: Uporaba značke formRemote

Na Slika 2.7 vidimo primer uporabe značke formRemote, katera ustvari zahtevo Ajax za krmilnik knjiga in akcijo iskanjePoKriteriju. Zahteva se pošlje asinhrono, zato se ne posodobi celotna stran, ampak samo del strani, ki je definiran v atributu update.

Te značke so bile vključene v prvih verzijah okvirja Grails, sedaj se njihova uporaba ne priporoča. Značke so bile razvite pred jQueryjem in podobnimi knjižnicami, ki zagotavljajo zelo optimalno uporabo Ajaxa.

2.2 Programski jezik Groovy

2.2.1 Splošno o programskemu jeziku Groovy

Groovy je močan dinamični programski jezik s statično tipizacijo in statičnimi prevajalskimi zmogljivostmi. Narejen je za platformo Java, pri čemer je cilj dvigniti programersko produktivnost s pomočjo znane in zelo lahko razumljive sintakse. Zelo dobro se integrira s katerim koli javanskim programom in s tem prinese programu dodatne funkcionalnosti, vključno s skriptnimi zmogljivostmi, uporabo domensko specifičnih jezikov, prevajanje v času zagona in prevajanja, metaprogramiranje in funkcijsko programiranje. [8] Večina javanske programske kode je sintaktično pravilne kode Groovy, le da je semantika malo drugačna. Koda Groovy pa je lahko bolj kompaktna, saj ne zahteva vseh elementov, ki jih zahteva javanski jezik. [9]

2.2.2 Metaprogramiranje in MOP

Objektno metaprogramiranje Groovy je tisto glavno, kar dela jezik Groovy tako zelo močen. Vsak razred dobi svoj metarazred, kateri prestreže vse klice metod in omogoči prilagajanje glede na poklicane metode ter hkrati omogoča dodajanje in odstranjevanje metod. To naredi jezik Groovy dinamičen v nasprotju z jezikom Java, kateri prevede razrede v bytecode in ne omogoča sprememb med izvajanjem. Ker Groovy MOP prestreza vse klice metod, lahko simulira dodajanje metod, kot da bi bile prevedene na začetku, kar naredi razrede Groovy odprte in se jih lahko spreminja kadar koli.

Pri sklicevanju na metodo v programskem jeziku Groovy v bistvu odpošljemo metodo metarazredu objekta. Klici so implementirani z refleksijo, kar je počasnejše od direktnega klica metod, ampak z vsako novo verzijo Groovy se zmogljivost refleksije izboljšuje. Jezik Groovy ima par optimizacijskih postopkov, kateri zmanjšujejo slabo odzivnost. Prejšnje verzije programskega jezika Groovy so bile počasne, medtem ko so novejše pridobile na zmogljivosti in hitrosti, zaradi česar je Groovy primerljiv s programskim jezikom Java. Ker sta za velik del počasnega delovanja aplikacije kriva latenca omrežja in dostop do podatkovne baze, se majhno povečanje skupnega odzivnega časa zaradi jezika Groovy niti ne pozna. [3]

2.2.3 Prednosti programskega jezika Groovy

Programski jezik Groovy ima kar nekaj prednosti [9]:

- Pisanje skript je zelo lahko z nekaterimi izboljšavami v razredih knjižice JDK
- Ima zelo enostavno sintakso
- Ima boljšo podporo za funkcijske izraze in večvrstične nize
- Podpira prekrivanje operatorjev
- Ima vgrajeno podporo za generiranje in razčlenjevanje datotek XML in JSON
- Uporaba Traitsov za večkratno dedovanje
- Deluje na Javi 6 in naprej. Deluje tudi na virtualni napravi, kot so naprave Android.
- Uporaba testov Spock, ki so hitrejši in učinkovitejši

2.2.4 Slabosti programskega jezika Groovy

Pri vsakem programskem jeziku pride do slabosti, ki pa jih pri Groovyju ni toliko. Slabosti programskega jezika Groovy so naslednje [10]:

- Enkrat do dvakrat počasnejši od programskega jezika Java
- Ker je Groovy dinamični jezik, med prevajanjem ne najde nekaterih napak, tako je težje odkrivati napake, katere nastanejo med izvajanjem aplikacije
- Ni primeren za večje projekte, saj lahko med izvajanjem prihaja do več napak, kar pomeni, da moramo napisati več testov

2.2.5 Razlike med programskima jezikoma Java in Groovy

2.2.5.1 Kombinirane metode

V programskem jeziku Groovy se metode, na katere se bomo sklicevali, izberejo med izvajanjem. To pomeni, da se bodo metode izbrale glede na tip argumentov med izvajanjem. V Javi je to ravno nasprotno, metode se izberejo med prevajanjem glede na deklaracijski tip. [11]

2.2.5.2 Obseg vidljivosti paketa

V Groovyju se z izpuščanjem modifikatorja »private« rezultat ne pojavi v paketu privatno, kot se to zgodi v Javi. Namesto tega se ustvari lastnost, ki je privatna, in dve metodi, kateri vračata oz. nastavljata to lastnost. S posebno notacijo (»@PackageScope«) pa lahko definiramo, če želimo, da je polje privatno. [11]

2.2.5.3 Lambda izrazi

Java 8 podpira uporabo izrazov lambda (Slika 2.8).

```
Runnable runnable = () -> System.out.print("Java lambda");
```

Slika 2.8: Uporaba izraza lambda v Javi

Java 8 izraze lambda lahko razume kot anonimne notranje razrede. Groovy te sintakse ne podpira, ampak samo ti. »closures« (Slika 2.9). [11]

```
Runnable runnable = { println("Grails clousure") }
```

Slika 2.9: Uporaba clousure v Groovy

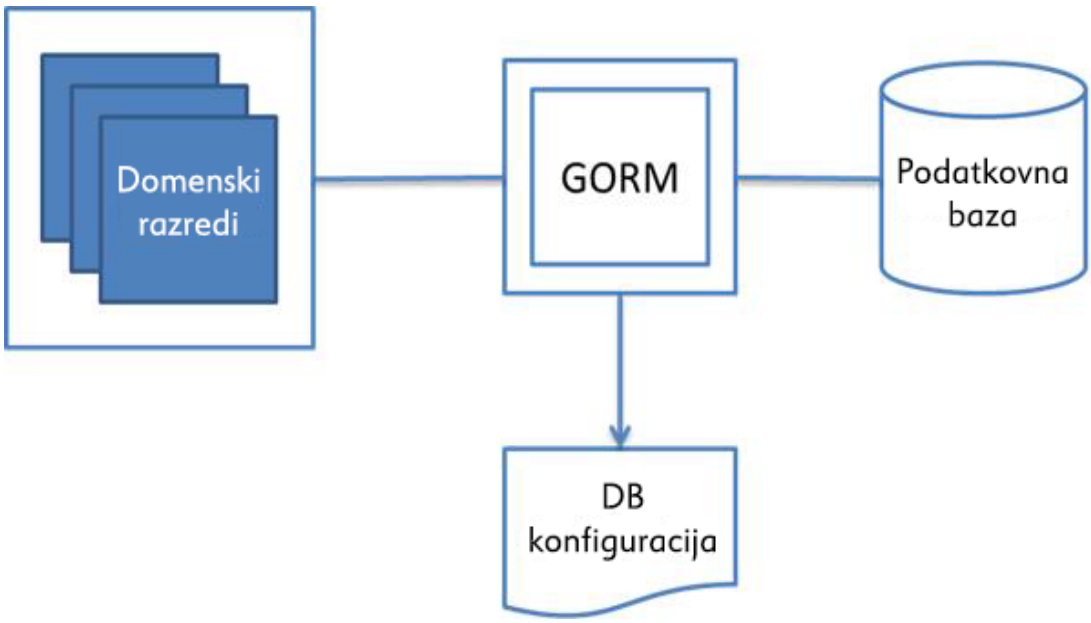
2.2.5.4 Nizi Groovy

Niz, deklariran z dvojnima narekovajema, je v jeziku Groovy predstavljen kot vrednost GString. Pri uporabi tipa string lahko pride do napake v času prevajanja ali napačnega obnašanja pri uporabi znaka dolar v nizu. Groovy po navadi avtomatsko pretvori med »GStringom« in »Stringom«. [11]

2.3 Persistenca

2.3.1 Splošno o persistenci v okvirju Grails

Strategija persistence, ki je uporabljena v okvirju Grails, se imenuje objektno relacijska preslikava Grails ali krajše GORM. V zgodnjih verzijah okvirja Grails je bil to le ovoj okrog Hibernate ORM-a, kar kasneje ni zadostilo potrebam, saj je GORM dovoljeval dostop do drugih podatkovnih virov, vključno s podatkovnimi shrambami noSql, kot je MangoDB. S tem so razvili nov GORM API, kateri zagotavlja konsistentno programersko izkušnjo. Vloga GORM-a v okvirju Grails je povezovanje domenskih razredov s podatkovno bazo (Slika 2.10). [3]



Slika 2.10: Vloga GORM-a v okvirju Grails

V okvirju Grails lahko uporabimo Hibernate, s tem se podatki dobro prilegajo relacijskemu modelu. Za manj strukturirane podatke ali podporo bolj dinamičnih shem pa lahko uporabimo noSql podatkovne shrambe. Razširitve Grails noSql dobro delujejo v povezavi s Hibernetom in tako lahko shranimo vse podatke v relacijsko podatkovno bazo, v shrambe NoSql ali kombiniramo oboje skupaj. [3]

2.3.2 Poizvedbe

Za pridobivanje shranjenih podatkov obstaja veliko načinov z uporabo GORM-a in v večini primerih je sintaksa ista, če uporabljamo razširitve Hibernate ali noSql. Metoda get pridobi instanco s podanim idjem, če le-ta obstaja, ali pa vrne null, če ta ne obstaja. Metoda read narediti enako, le da pri tem konfigurira instanco, tako da je samo za branje. Do podatkov ne dostopamo, dokler ne zahtevamo dostopa do razrednih lastnosti, kar sproži sprožilec za ti. »lazy« nalaganje podatkov. To lahko sproži izjemo, če zahtevana vrstica ne obstaja.

Obstajajo tudi metode, ki vračajo več instanc naenkrat. Metoda list vrne vse instance, če je klicana brez parametrov. Obstajajo dinamični iskalniki, kot sta findBy in findAllBy, kateri omogočajo pridobivanje podatkov pod podanimi kriteriji. Dinamični iskalniki podpirajo zelo veliko možnosti poizvedb, ampak včasih za pridobitev željenih rezultatov ne zadostuje klic samo ene metode, ampak mora biti poizvedba kompleksnejša, pri čemer si pomagamo z uporabo kriterijev DSL, kateri omogočajo, da lahko podamo veliko število kriterijev za filtriranje. [3]

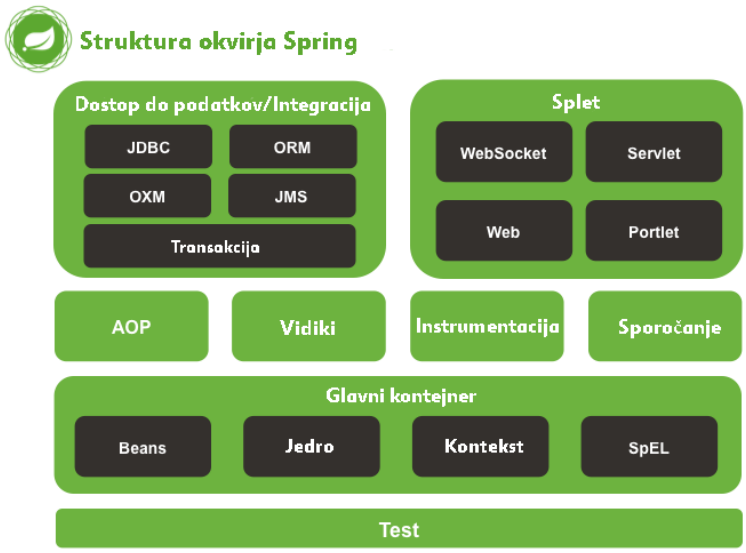
2.3.3 Shranjevanje, posodabljanje in brisanje

Shranjevanje ene instance domenskega razreda lahko dosežemo z metodo save, katera bo v primeru napake pri validaciji vrnila vrednost null, v primeru uspešnega shranjevanja pa bo vrnila instanco, ki se je posnela v podatkovno bazo. Posodabljanje poteka podobno kot shranjevanje. Najprej pridobimo instanco s pomočjo metode get, jo spremenimo in shranimo nazaj v podatkovno bazo, s tem smo posodobili instanco v podatkovni bazi. Brisanje podatkov iz podatkovne baze je zelo enostavno, saj na domenskem razredu obstaja metoda delete, ki jo prikličemo na instanci in s tem zbrišemo podatke iz podatkovne baze. [3]

2.4 Spring

2.4.1 Splošno

Okvir Spring je odprtokodni aplikacijski okvir in je inverzija kontrolnega kontejnerja za platformo Java. Glavne funkcije okvirja Spring so lahko uporabljene v vsaki aplikaciji, napisani v javanskem jeziku, obstaja pa razširitev za izdelavo spletnih aplikaciji na vrhu platforme Java EE. Popularen je postal v javanski skupnosti kot alternativa za modele Enterprise JavaBeans. Arhitektura okvirja Sping je razvidna na Slika 2.11. [13]



Slika 2.11: Arhitektura okvirja Spring

2.4.2 Moduli

Okvir Spring vključuje module, kateri zagotavljajo širok nabor funkcionalnosti [13]:

- Aspektno orentirano programiranje
- Avtorizacijo in avtentikacijo
- Dostop do podatkov
- Inverzijo kontrolnega kontejnerja
- Model-pogled-krmilnik
- Transakcijsko opravljanje

2.4.3 Transakcijski servisi

Vsi razredi, ustvarjeni v mapi grails-app/services, in razredi, katerim se ime konča na Service, so avtomatsko registrirani kot servis. To pomeni, da je privzeto servis registriran kot Spring bean, tip tega beana pa je singleton. Vse javne metode v razredu so transakcijske, kar naredi razred za zelo dober prostor za poslovno logiko, posebno, kadar delamo s podatkovno persistenco. Če želimo, da servis ne bi bil transakcijski, lahko to specifično povemo s podanim parametrom transactional = false.

Pri okvirju Spring nismo omejeni na uporabo privzetih transakcijskih nastavitev ali prisiljeni v uporabo kompleksnih konfiguracij za prilagajanje obnašanja transakcijskih servisov. Prilagajanje je zelo preprosto: enostavno uporabimo notacijo @Transactional (Slika 2.12). V primeru, da podamo takšno notacijo, okvir Grails predvideva, da bomo vse sami konfigurirali in s tem nam ne doda privzete konfiguracije. Notacijo lahko uporabimo nad razredom ali nad posamezno metodo, lahko pa tudi uporabimo kombinacijo konfiguriranja nad metodo in razredom (Slika 2.12). [3]

```
@Transactional
class BookController {

    def metoda1() {
        //some code
    }

    @Transactional
    def metoda2() {
        //some code
    }
}
```

Slika 2.12: Uporaba notacije @Transactional na metodi in razredu

2.4.4 Okvir MVC

Okvir Spring ima lasten okvir MVC, kateri prvotno ni bil načrtovan. Razvijalci Spring so se odločili napisati lasten spletni okvir zaradi pomanjkljivosti drugih spletnih okvirjev MVC. Predvsem pa so hoteli ločiti med predstavitveno plastjo, plastjo, kjer se upravljajo zahteve, in modelom. Spring MVC je okvir, ki temelji na zahtevi. Definira strategijo vmesnikov, kateri morajo biti odgovorni za vse, kar zahteva moderni okvir, ki temelji na zahtevi. Cilj vsakega vmesnika je biti kar se da enostaven in razumljiv ter da je uporabnikom, ki uporabljajo MVC Spring, implementacija na njihovih projektih enostavna. [13]

2.4.5 Inverzija kontrolnega kontejnerja

Glavna funkcija okvirja Spring je inverzija kontrolnega kontejnerja, kar prinaša dosleden način konfiguriranja in upravljanja objektov Java s pomočjo refleksije. Kontejner je odgovoren za upravljanje življenjskega cikla objektov, kot so na primer akcije za ustvarjanje objektov, klicanje inicializacijskih metod in konfiguriranje teh objektov s pomočjo povezovanja objektov.

Objekt, ki je ustvarjen s pomočjo kontejnerja, se imenuje bean. Kontejner je lahko konfiguriran z nalaganjem datotek XML ali z zaznavanjem specifičnih notacij Java na konfiguriranih objektih. Ti podatkovni viri vsebujejo definicijo objektov, kateri dajo informacije, ki so potrebne za ustvarjanje objektov.

Objekti so lahko pridobljeni na dva načina, to sta pregled odvisnosti in injiciranje odvisnosti. Pri pregledu odvisnosti gre za vzorec, pri katerem klicatelj vpraša kontejner objekta za objekt s specifičnim imenom ali tipom. Pri injiciranju odvisnosti pa gre za vzorec, pri katerem kontejner poda ime objekta drugim objektom s pomočjo konstruktorja, lastnosti ali posebne metode. [13]

2.4.6 Aspektno orientirano programiranje

Spring je za svoje potrebe razvil svoj okvir AOP. Motivacija za izdelavo ločenega okvirja AOP je prišla iz prepričanja, da je možno zagotoviti osnovne operacije AOP brez pretirane kompleksnosti v implementaciji in konfiguraciji. Sping okvir AOP je izkoristil vse prednosti kontejnerja Spring.

Sping okvir AOP temelji na vzorcu proxy in je kofiguriran v času izvajanja. S tem se znebi potrebe po kompilaciji. Če primerjamo okvir AspactJ, je Springov okvir AOP manj zmogljiv, ampak je ob enem tudi bolj enostaven. Springov okvir AOP je bil zasnovan tako, da je sposoben delati z medsektorskimi principi znotraj okvirja Spring. Vsak objekt, kateri je ustvarjen in konfiguriran s pomočjo kontejnerja, ga lahko obogatimo z uporabo Spring AOP-ja. Okvir Spring uporablja Spring AOP interno za upravljanje transakcij, varnosti, oddaljenega dostopa in JMX. [13]

2.5 Hibernate

2.5.1 Splošno

Hibernate je zelo močna in popularna objektno relacijsko preslikana knjižica in predstavlja osnovni standard persistentnega sloja v okvirju Grails. Hibernate ustvari povezavo med objektno orientirano kodo in relacijskim podatkovnim modelom. Poskuša zagotoviti kolikor se da transparenten API, kjer se razvijalcem ni potrebno ukvarjati z implementacijskimi detajli za shranjevanje in pridobivanje podatkov. [3] Na Slika 2.13 je vidna arhitektura Hibernata.

Slika 2.13: Ahitektura Hiberneta

Hibernate ne sili v spremembo obnašanja objektov, niti ne potrebujemo implementirati dodatnih vmesnikov za delovanje Hiberneta. Vse, kar moramo narediti, je, da ustvarimo dokument XML, v katerega zapišemo razrede, katere hočemo shraniti v podatkovno bazo in določiti, kako so ti razredi povezani s tabelami in stolpci v podatkovni bazi. Potem pa lahko od Hibernata zahtevamo, da vrne podatke iz podatkovne baze kot objekt ali shranimo objekt v podatkovno bazo.

Med izvajanjem, Hibernet prebere dokument XML, v katerem smo podali razrede, za katere hočemo, da so persistentni in dinamično zgradi razrede Java, da lahko upravlja preslikavo med podatkovno bazo in svetom Java. Obstaja preprost, intuitiven API za opravljanje poizvedb za objekte, ki so predstavljeni z strani podatkovne baze. [14]

2.5.2 Preslikava domenskih razredov

Domenski razredi so jedro za vsako poslovno aplikacijo, držijo stanja o poslovnih procesih in implementirajo obnašanje. Med seboj so povezani preko razmerij: ena – mnogo, mnogo – mnogo in ena – ena.

Domenski razredi se v glavnem preslikajo tako, da je vsak domenski razred predstavljen kot tabela v podatkovni bazi, razredne lastnosti pa se v podatkovni bazi predstavijo kot stolpci. Relacijske odvisnosti med tabelami se definirajo kot atributi v razredu in se nato preslikajo v podatkovno bazo kot tuji ključi. [3]

Aplikacije Hibernata po navadi uporabljajo datoteke XML za definiranje preslikave med programsko kodo in podatkovno bazo. Po dodajanju notacij v verziji Jave 1.5 je Hibernate dodal bolj inovativen pristop preslikave, kateri ohranja metapodatke skupaj s kodo. Hibernate ustvari metamodel domenskega razreda, ki je neodvisen od njegovega vira. Programsko lahko sami ustvarimo celoten podatkovni model Hibernate z direktno uporabo API-jev. [14]

Najbolj osnoven domenski razred je POGO z enim ali več javnimi polji (Slika 2.14).

```
class Avtor {
    String ime
    String priimek
}
```

Slika 2.14: Domenski razred POGO

Privzeto so vse lastnosti obravnavane kot persistentne in so preslikane v primerne tipe stolpcev. Če imamo lastnosti, za katere nočemo, da so persistentne, jih lahko izključimo z rezervirano besedo »transients« . [3]

2.5.3 Seja

Seja Hibernate je osrednja točka dostopa do persistence v Hibernetu. Razvijalci Grails redko dostopajo neposredno do seje, ampak preko GORM-a in razreda HibernateTemplate, ki je uporabljen za poizvedbe in se zelo pogosto uporablja. Seja uporablja povezavo JDBC, katera se odpre na zahtevo. Prav tako zagotavlja prvostopenjski predpomnilnik za instance in preslikane zbirke. Seja je vzpostavljena s pomočjo SessionFactory, katera upravlja drugostopenjski predpomnilnik. Vsa persistenca se upravlja s pomočjo seje. Ima metode za pridobivanje individualnih instanc (Slika 2.15), ustvarjanje novih in posodabljanje obstoječih (save in update), brisanje instanc (delete) in posodabljanje v naprej naloženih instanc. [3]

```
Knjiga knjiga = Knjiga.get(id)
```

Slika 2.15: Pridobivanje individualne instance

Za obsežnejše poizvedbe obstajajo metode createCriteria (Slika 2.16), createQuery in CreateSqlQuery.

```
def criteria = Knjiga.createCriteria()
def books = criteria.list {
    ilike("naslov", "${naslov}%")
    avtor {
        ilike("ime", "${avtor}%")
    }
}
books
```

Slika 2.16: Uporaba metode createCriteria

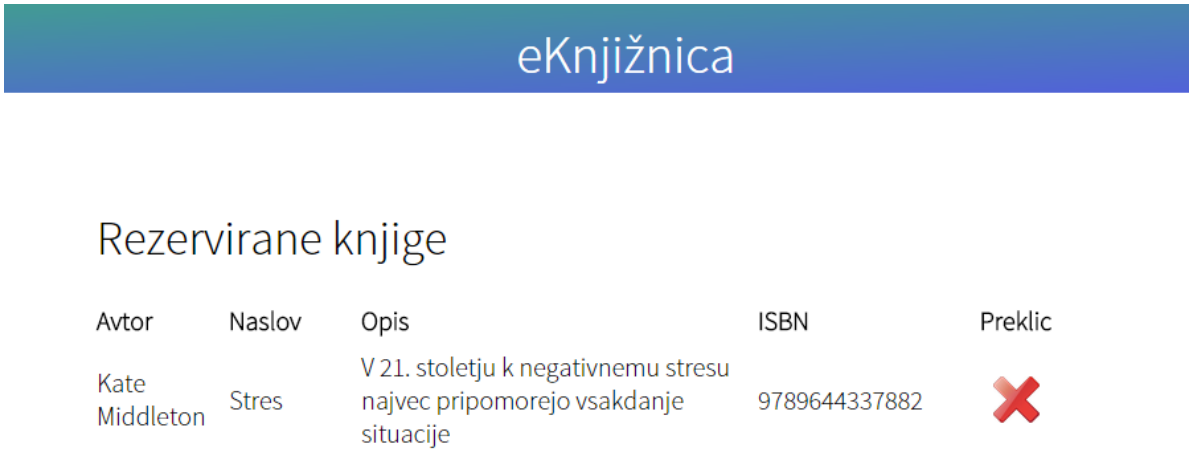
2.5.4 Beleženje Sql

Obstajata dva načina beleženja stavkov Sql, ki jih izvrši Hibernate za naše poizvedbe. Najlažji način je, da dodamo logSql=true v sekcijo dataSource. To deluje, ampak je bolj omejen način uporabe, saj se v tem načinu samo izpiše na stdout. Na primer, če bi to zagnali na strežniku Tomcat, bi naša datoteka, v katero beležimo izpise, hitro postala zelo velika. Boljši način je uporaba beleženja Log4j. [3]

3 UPORABA OKVIRJA GRAILS V PRAKSI

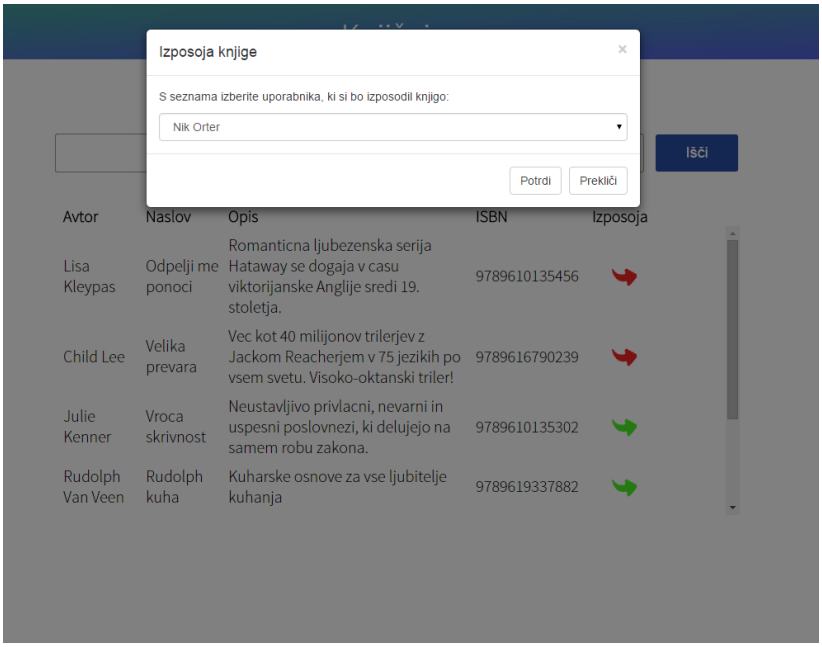
3.1 Opis aplikacije

Izdelali smo aplikacijo za vodenje izposojenega gradiva v knjižnici. Spletna aplikacija eKnjižnica omogoča tri poglede aplikacije, in sicer administrator, moderator in uporabnik. Uporabnikom je omogočena rezervacija gradiva in pregled izposojenega in rezerviranega gradiva (Slika 3.1).



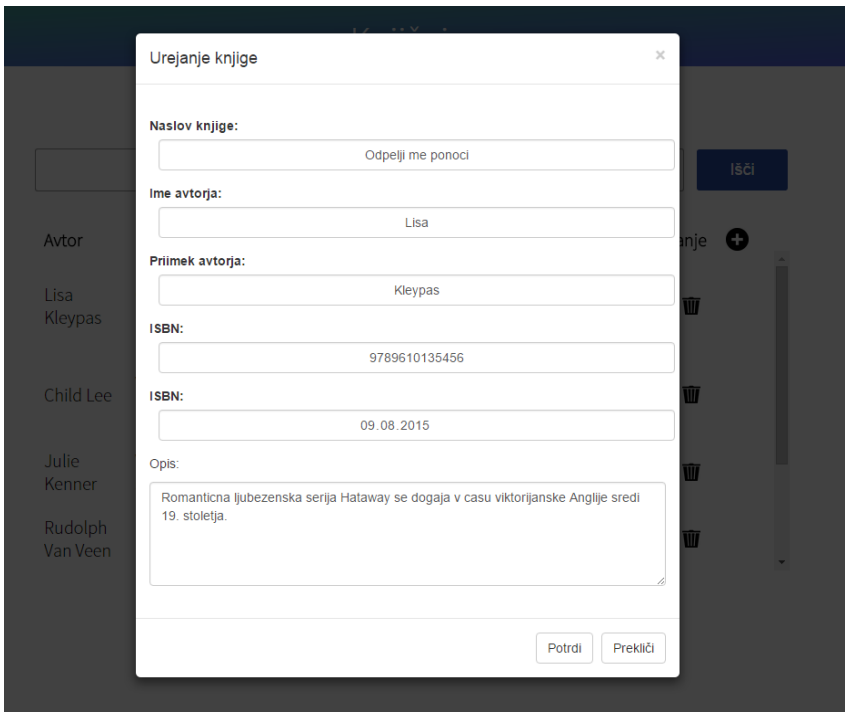
Slika 3.1: Pregled rezerviranega gradiva

Način moderatorja je namenjen knjižničarki, ki izposoja knjige v knjižnici. Omogočene so akcije za vrnitev in izposojo knjig (Slika 3.2).



Slika 3.2: Uporabniški vmesnik moderatorja za vrnitev knjig

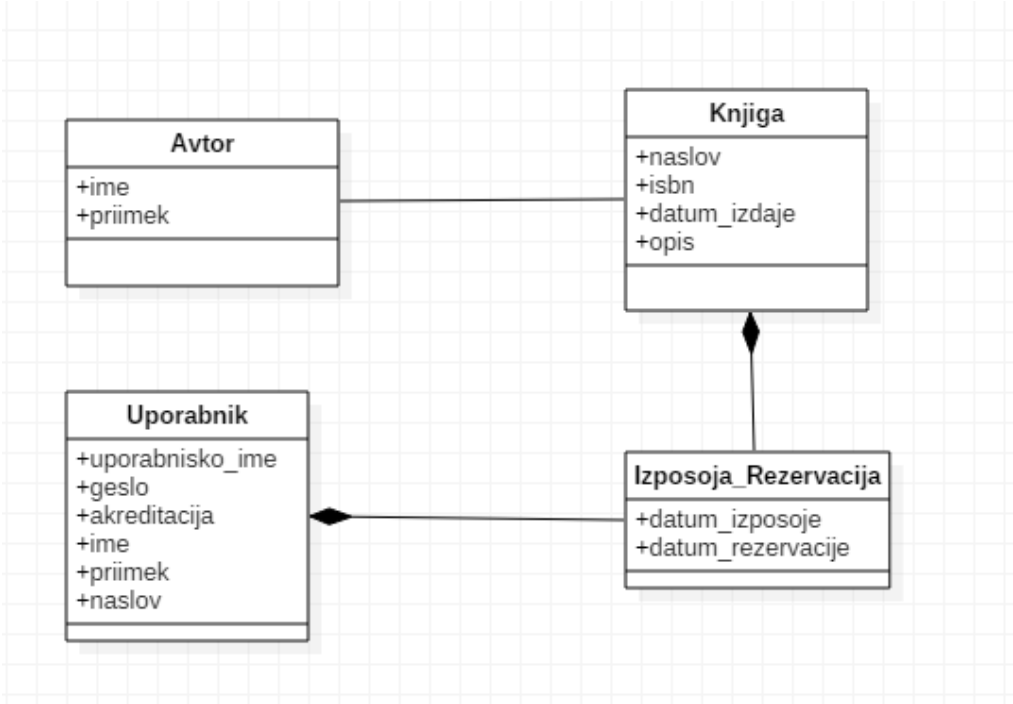
Tretji pogled je administratorjev s polnim dostopom. Imel bo vse možnosti, do katerih lahko dostopa moderator, ter možnost urejanja in dodajanja uporabnikov in knjig (Slika 3.3). Namen izdelave aplikacije je preizkusiti funkcionalnosti okvirja Grails in preveriti, kako deluje v praksi.



Slika 3.3: Uporabniški vmesnik za urejanje knjig

3.2 Diagram UML

Izdelali smo diagram UML (Slika 3.4), ki predstavlja razredni diagram. Vsak posamezni razred v razrednem diagramu predstavlja domenski razred, kateri hrani podatke. Tako smo ustvarili razrede Avtor, Knjiga, Izposoja_Rezervacija in Uporabnik. Vsakemu razredu smo dodali atribute, ki jih bo potreboval vsak posamezni razred.

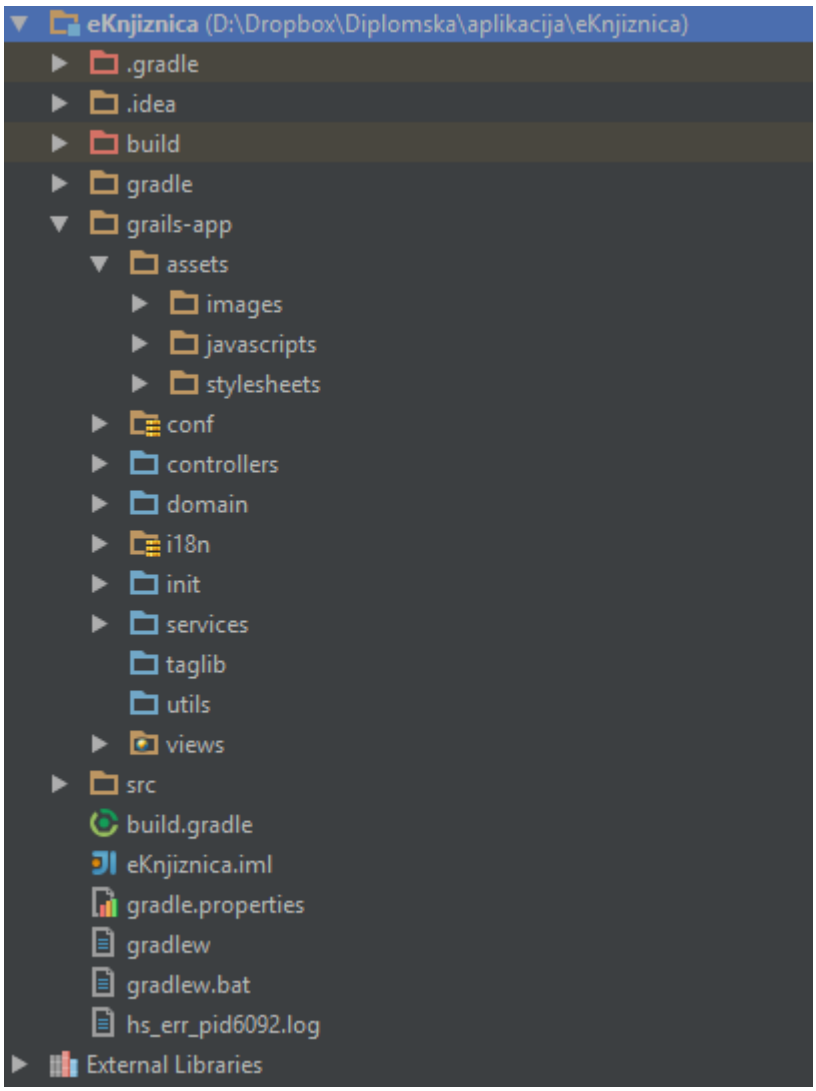


Slika 3.4: Razredni diagram

Med razredi smo označili potrebne odvisnosti. Med avtorjem in knjigo je asociacija, to pomeni, da lahko živi vsak za sebe. Izposoja_Rezervacija pa je odvisna od knjige in uporabnika, kar pomeni, da v primeru, da en od njiju ne obstaja, tudi Izposoja_Rezervacija ne more obstajati. Izposoja_rezervacija bo hranila podatke o izposojah in rezervacijah. Izdelan diagram UML je služil kot osnova za izdelavo domenskih razredov in boljši pregled odvisnosti med razredi.

3.3 Namestitev okvirja Grails

Okvir Grails namestimo na računalnik tako, da najprej z interneta prenesemo inštalacijsko datoteko in ga namestimo. Po namestitvi moramo v sistemskih spremenljivkah dodati pot do datoteke, v kateri je nameščen okvir Grails. Potem lahko preko terminala ustvarimo projekt Grails, kar storimo s pomočjo ukaza create-app. Po zagonu ukaza create-app bo okvir Grails pripravil celoten projekt in celotno strukturo projekta (Slika 3.5).



Slika 3.5: Struktura projekta Grails

Okvir Grails že v naprej pripravi celoten projekt z že vgrajenim strežnikom Tomcat. Projekt lahko zaženemo in začetna spletna aplikacija že deluje.

3.3.1 Domenski razredi

Razrede, ki smo jih prikazali v diagramu UML, jih sedaj ustvarimo v aplikaciji Grails. S pomočjo ukaza create-domain-class ustvarimo domenske razrede. V našem primeru smo ustvarili domenske razrede Uporabnik, Knjiga, Izposoja_Rezervacia in Avtor. Vse ustvarjene razrede okvir Grails doda v mapo domain. Po kreiranju vseh domenskih razredov smo vsakemu posameznemu razredu dodali atribute, katere smo potrebovali. Vsak atribut je določenega podatkovnega tipa, npr. String, Date, Integer, Double, … Primer atributov za domenski razred Knjiga je prikazan na Slika 3.6.

```
String naslov
String isbn
@BindingFormat('yyyy-MM-dd')
Date datum_izdaje
String opis
```

Slika 3.6: Primer atributov razreda Knjiga

Po kreiranju vseh domenskih razredov in dodajanju njihovih atributov smo vsakemu posameznemu razredu dodali še omejitve; to smo storili tako, da smo definirali konstanto »constraints« in v to konstanto dodali omejitve (Slika 3.7).

```
static constraints = {
    naslov size:5..40
    isbn unique: true
    opis blank:true, nullable: true, size:5..500
    izposoje_rezervacije nullable: true
}
```

Slika 3.7: Primer omejitev razreda Knjiga

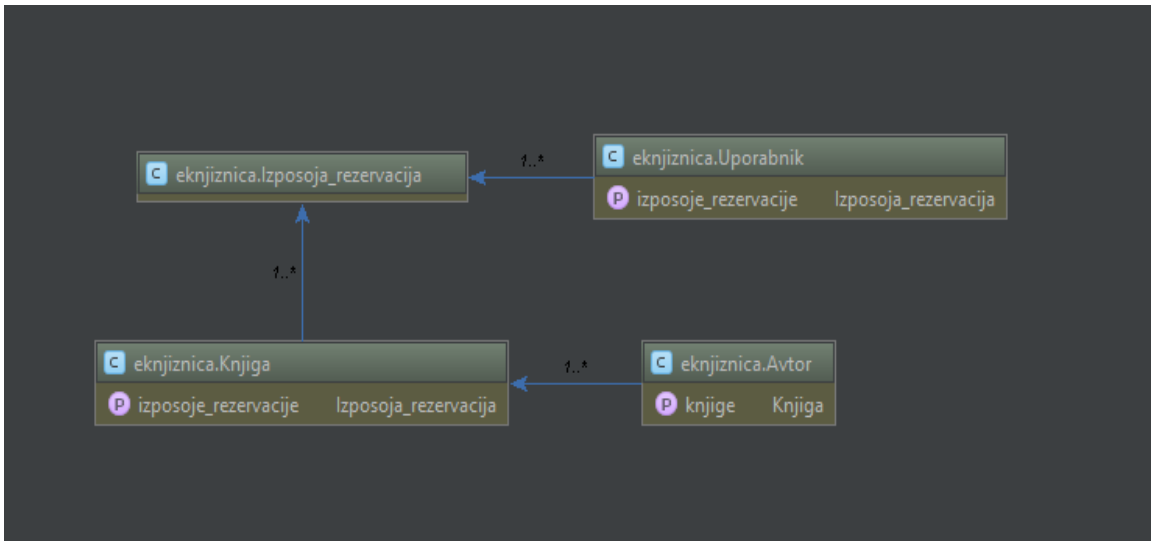
Med razredi smo morali določiti še odvisnosti. To storimo s pomočjo konstante hasMany, v katero moramo dodati še instanco drugega razreda. Konstanta hasMany pomeni, da je lahko razred povezan z več instancami drugega razreda. V našem primeru smo dodali konstanto hasMany na razred avtor, s čimer smo povedali, da ima lahko avtor več knjig. Konstanto hasMany smo dodali še na Uporabnik in Knjiga. Druga konstanta, s katero določimo odvisnost, je belongsTo, s čimer povemo, da neki razred pripada drugemu razredu in ne more obstajati brez njega.

Primer celotnega domenskega razreda s konstantama hasMany in belongsTo je prikazan na Slika 3.8.

```
class Knjiga {

    String naslov
    String isbn
    @BindingFormat('yyyy-MM-dd')
    Date datum_izdaje
    String opis

    static belongsTo = [avtor:Avtor]

    static hasMany = [izposoje_rezervacije:Izposoja_rezervacija]

    static constraints = {
        naslov size:5..40
        isbn unique: true
        opis blank:true, nullable: true, size:5..500
        izposoje_rezervacije nullable: true
    }
}
```

Slika 3.8: Domenski razred Knjiga

Primer odvisnosti, ki smo jih dobili s pomočjo konstant hasMany in belongsTo, pa je prikazana na Slika 3.9.



Slika 3.9: Odvisnosti domenskih razredov

3.3.2 Izdelava servisov

Poslovno logiko aplikacije smo implementirali v servisih. Preko ukazne vrstice smo z ukazom create-service ustvarili servise. Okvir Grails je ustvaril servis in rezred, ki služi za testiranje servisa. Ustvarili smo servise UporabnikService, KnjigaService in IzposojaRezervacijaService. V vsakem servisu smo implementirali metode, preko katerih smo vračali, shranjevali, posodabljali in brisali podatke iz podatkovne baze, do katere smo dostopali s pomočjo tehnologije GORM.

Pri servisu UporabnikService (Slika 3.10) smo potrebovali štiri metode:

- shraniUporabnika: metoda, v kateri shranimo uporabnika v podatkovno bazo.
- posodobiUporabnika: metoda, v kateri posodobimo uporabnika na podane parametre.
- getUporabnikPoImenuInGeslu: metoda, ki vrne uporabnika glede na podano uporabniško ime in geslo.
- getUporabnikPoImenuInPriimku: ta metoda vrne uporabnika glede na podana ime in priimek.

```
@Transactional
class UporabnikService {

    def getUporabnikPoUpImenuInGeslu(String uporabniskoime, String geslo) {
        def criteria = Uporabnik.createCriteria()
        def result = criteria.list {
            eq("uporabnisko_ime",uporabniskoime)
            and {
                eq("geslo",geslo)
            }
        }
        result
    }

    def getUporabnikPoImenuInPriimku(String ime,String priimek){
        def criteria = Uporabnik.createCriteria()
        def uporabniki = criteria.list {
            ilike("ime", "${ime}%")
            and {
                ilike("priimek", "${priimek}%")
            }
        }
        uporabniki
    }

    def shraniUporabnika(Object params){
        Uporabnik uporabnik = new Uporabnik(params)
        uporabnik.save(failOnError: true)
    }

    def posodobiUporabnika(Object params){
        Uporabnik uporabnik = Uporabnik.get(params.idUporabnik)
        uporabnik.properties = params
        uporabnik.save(failOnError: true)
    }
}
```

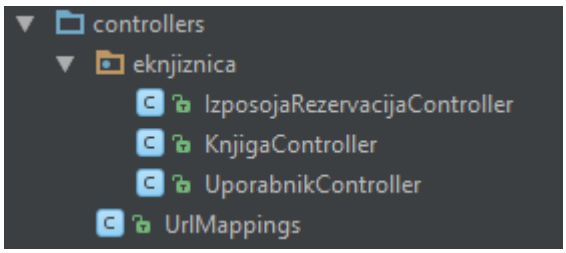
Slika 3.10: Servis UporabnikService

3.3.3 Izdelava krmilnikov

Za obdelavo zahtev uporabnika in posredovanje rezultatov pogledu smo napisali krmilnike, ki nam bodo služili za tri namene:

- sprejeti zahtevo od uporabnika
- sklicati najprimernejšo poslovno logiko
- sklicati pogled, v katerem prikažemo rezultat

Krmilnike smo ustvarili s pomočjo ukaza create-controller, kateri ustvari krmilnik in pripravi razrede za testiranje krmilnika. V naši aplikaciji smo ustvarili tri krmilnike (Slika 3.11): UporabnikController, KnjigaController in IzposojaRezervacijaController.



Slika 3.11: Ustvarjeni krmilniki v mapi controllers

Najprej smo injicirali servise, ki jih potrebujemo v vsakem krmilniku; to smo enostavno dosegli tako, da smo definirali spremenljivko, ki ima isto ime kot servis, npr. servis KnjigaService smo deklarirali z »def knjigaService« in servis se je avtomatsko injiciral. Po injiciranju servisov smo implementirali metode, katere bodo skrbele za usmerjanje po pogledih v aplikaciji. Na primer v krmilniku KnjigaController (Slika 3.12) smo ustvarili metode:

- index: privzeta metoda
- iskanjePoKriteriju: metoda, v kateri kličemo servis, ki vrne knjige s podanimi parametri, nato pa rezultat posredujemo pogledu, v katerem bomo izpisali ta rezultat.
- urejanjeKnjig: metoda, ki prikaže pogled za urejanje knjig.
- isciKnjige: enaka metoda kot iskanjePoKriteriju, le da rezultat posredujemo drugemu pogledu.
- izbrisiKnjigo: metoda, ki kliče servis, v katerem odstranimo knjigo s podanim idjem.
- shraniKnjigo: metoda, v kateri kličemo servisno metodo za posodabljanje ali kreiranje nove knjige, odvisno od tega, ali imamo podan id knjige ali ne.

```
class KnjigaController {

    def knjigaService

    def index() {
    }

    def iskanjePoKriteriju(){
        Object books = knjigaService.getKnjigaPoNaslovuInAvtorju(params.naslov,params.avtor)
        render (view: "rezultatIskanja", model: [result:books])
    }

    def urejanjeKnjig(){
    }

    def isciKnjige(){
        Object books = knjigaService.getKnjigaPoNaslovuInAvtorju(params.naslov,params.avtor)
        render (view: 'knjigeZaUrejanje',  model: [result:books])
    }

    def izbrisiKnjigo(){
        knjigaService.izbrisiKnjigo(params.id)
        render ""
    }

    //v tej metodi posodobimo knjigo v primeru da ?e obstaja ali dodamo novo
    def shraniKnjigo(){
        if(params.idKnjiga){
            knjigaService.posodobiKnjigo(params)
        }
        else{
            knjigaService.novaKnjiga(params)
        }
        render ""
    }
```

Slika 3.12: Krmilnik KnjigaController

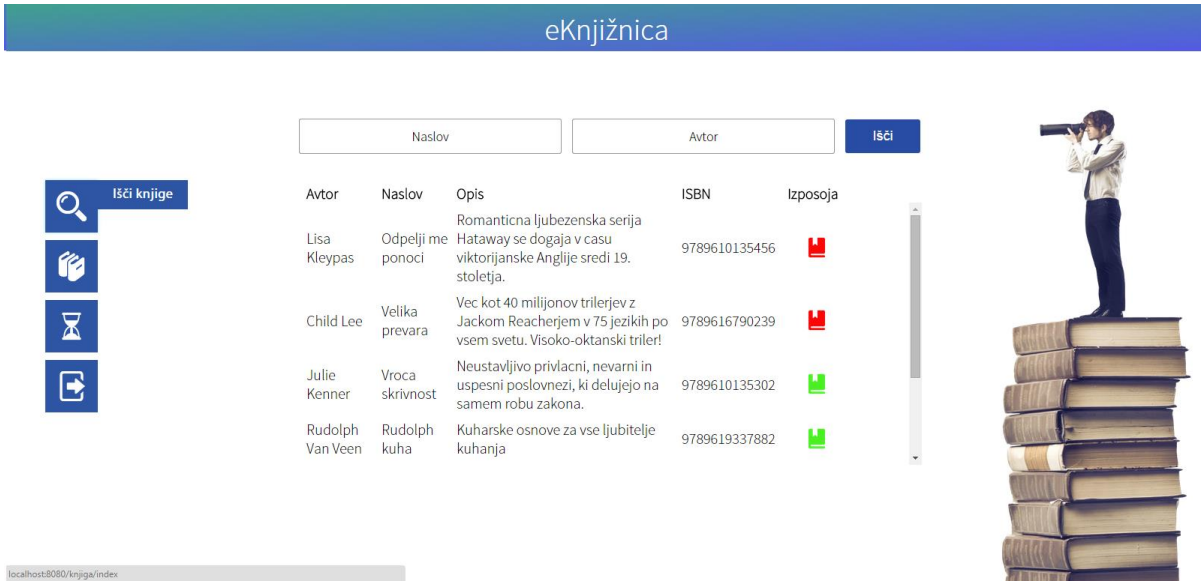
Kot je razvidno na Slika 3.12, v krmilniku skrbimo za izbiranje pravilne poslovne logike, kar pomeni, da pokličemo servis, kateri bo izvedel željeno akcijo, potem pa glede na rezultat pravilno posredujemo izbranemu pogledu ali pa preusmerimo na drug krmilnik (Slika 3.13).

```
def prijava(){
    def result = uporabnikService.getUporabnikPoUpImenuInGeslu(params.uporabniskoime,params.geslo)
    if(result) {
        session.status = ((Uporabnik)result[0]).akreditacija
        session.ide = ((Uporabnik)result[0]).getId()
        redirect(controller: 'knjiga',action: 'index')
    }
    else {
        flash.message = "Napačno uporabniško ime ali geslo"
        redirect(action: 'index')
    }
}
```

Slika 3.13: Preusmeritev na drug krmilnik glede na rezultat

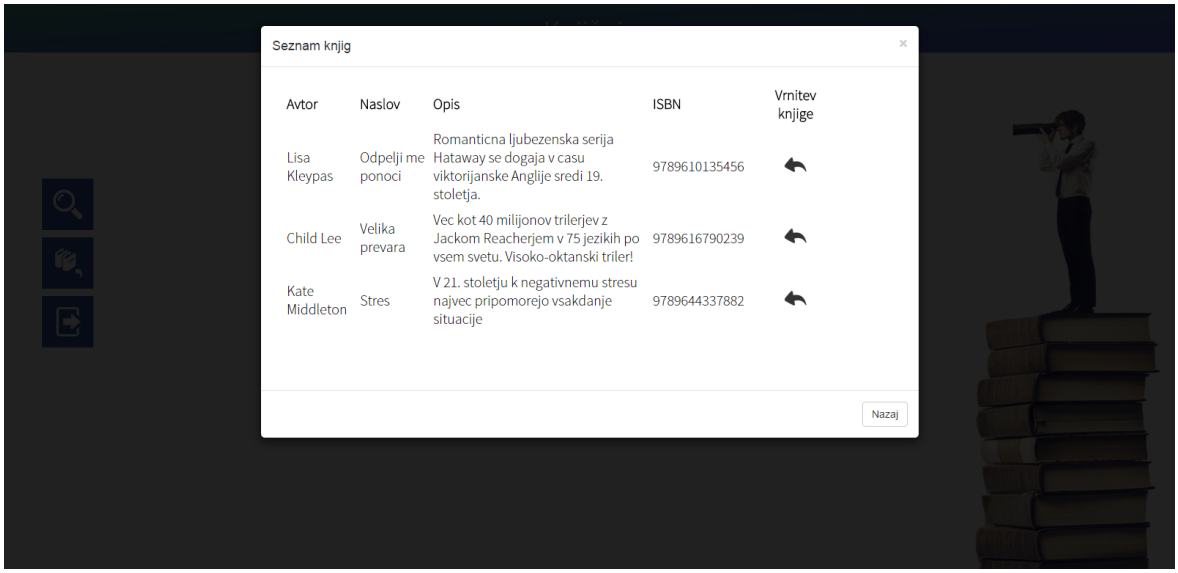
3.3.4 Izdelava GSP

Uporabniški vmesnik aplikacije smo izdelali s pomočjo Groovy server pages. GSP omogoča, da lahko kombiniramo statično in dinamično vsebino. Naš uporabniški vmesnik je sestavljen iz več pogledov: pogled za uporabnika, kateri ima dostop do iskanja knjig (Slika 3.14), pregled rezerviranih knjig in pregled izposojenih knjig.



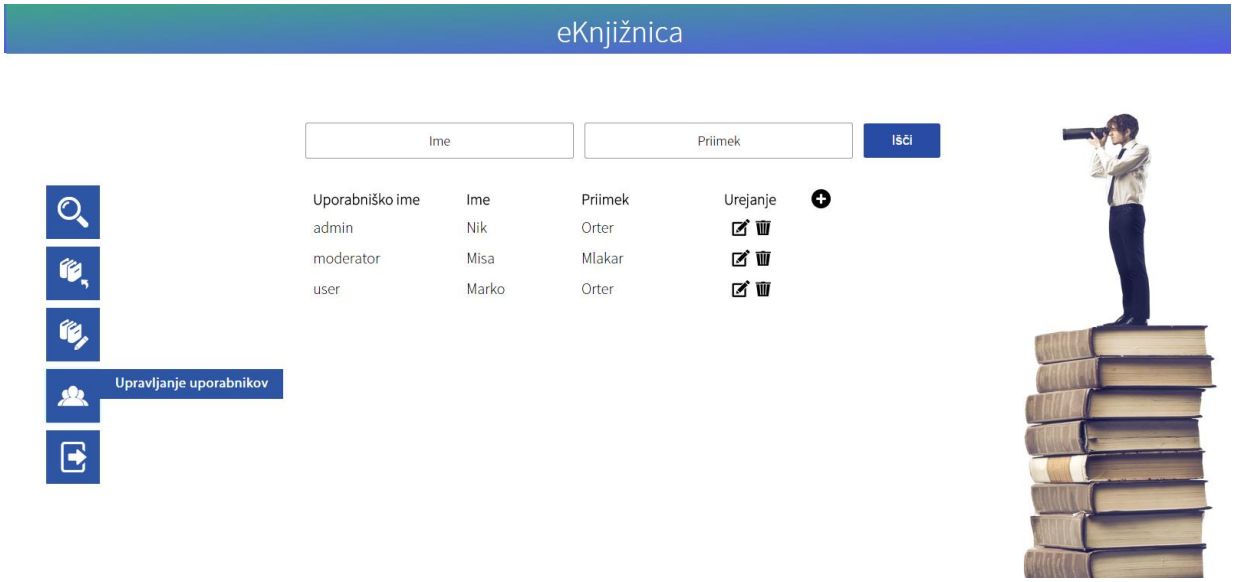
Slika 3.14: Uporabniški vmesnik za iskanje knjig

Pri pogledu moderatorja je omogočeno izdajanje in vrnitev knjig (Slika 3.15).



Slika 3.15: Vrnitev knjig izbranega uporabnika

Pri pogledu administratorja imamo omogočene enake funkcionalnosti kot pri moderatorju. Dodane funkcionalnosti so urejanje, posodabljanje in brisanje knjig in uporabnikov (Slika 3.16).



Slika 3.16: Uporabniški vmesnik za urejanje uporabnikov

Izdelavo uporabniškega vmesnika smo pričeli z izdelavo predloge, ki bo uporabljena na večini pogledih. V mapi layouts smo ustvarili novo datoteko gsp, ki bo predstavljala predlogo. V glavi predloge smo najprej vključili stile css, ki smo si jih definirali v zunanjih datotekah css (Slika 3.17).

```
<head>
    <title><g:layoutTitle default="MainLayout"/></title>

    <asset:stylesheet src="menustyle.css"/>
    <asset:stylesheet src="layoutstyle.css"/>
    <script src="https://ajax.googleapis.com/ajax/libs/jquery/1.11.3/jquery.min.js"></script>
    <g:layoutHead/>
</head>
```

Slika 3.17: Glava predloge

Vključili smo tudi knjižico jQuery, da tega ne bo potrebno storiti na vsaki strani, katera bo uporabljala predlogo. Z značko layoutHead definiramo, kje bo prišla glava dokumenta gsp, kateri bo uporabljal predlogo. Z layoutTitle pa smo določili, kam pride naslov dokumenta gsp, v primeru, da naslov ne bo podan, bo privzeto nastavljen na MainLayout.

V telesu predloge smo najprej ustvarili glavo z logotipom eKnjižnica, nato pa dodali še seznam (Slika 3.18), ki predstavlja meni, kateri se prilagaja glede na vlogo uporabnika.

```
<div id="main">

    <ul id="navigationMenu">
        <li>
            <g:link controller="knjiga" action="index" class="home">
                <span>Išči knjige</span>
            </g:link>
        </li>
    %{--Uporabnik--}%
        <g:if test="${session.status == enums.Akreditacija.uporabnik}">
            <li>
                <g:remoteLink controller="izposojaRezervacija" action="izposojeno" update="page_body" class="books">
                    <span>Izposojene knjige</span>
                </g:remoteLink>
            </li>
            <li>
                <g:remoteLink controller="izposojaRezervacija" action="rezervirano" update="page_body" class="reserved"
                              href="#">
                    <span>Rezervirane knjige</span>
                </g:remoteLink>
            </li>
        </g:if>

    %{--Administrator in Moderator--}%
        <g:if test="${session.status == enums.Akreditacija.administrator || session.status == enums.Akreditacija.moderator}">
            <li>
                <g:remoteLink controller="izposojaRezervacija" update="page_body" action="vrnitevKnjig"
                              class="returnbook">
                    <span>Vrnitev knjig</span>
                </g:remoteLink>
            </li>
        </g:if>
```

Slika 3.18: Seznam, ki predstavlja meni

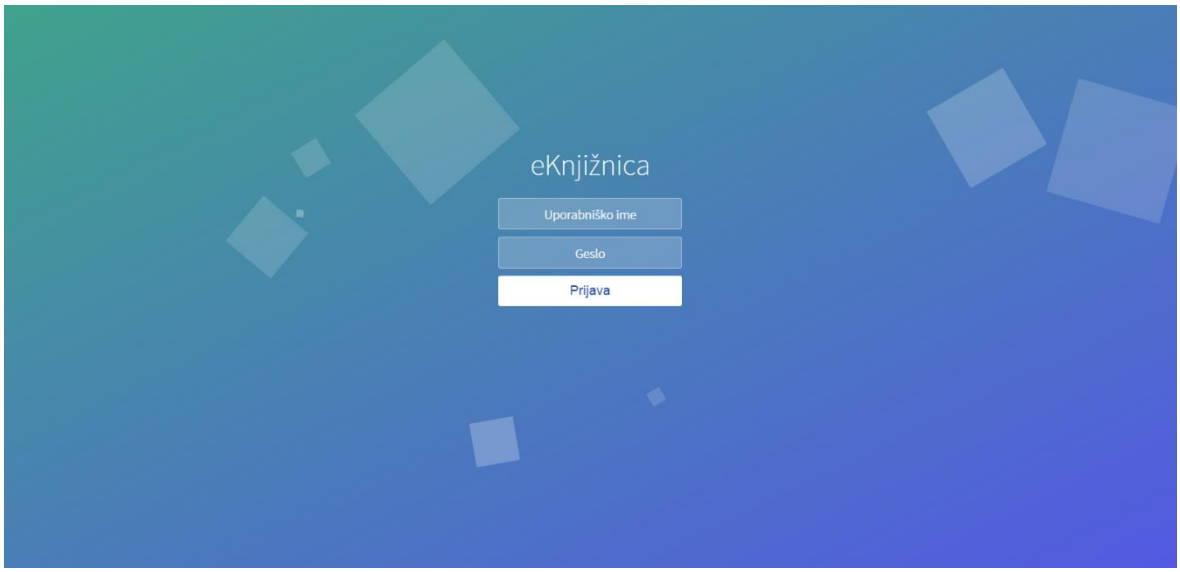
Akreditacijo uporabnika smo preverjali s stavkom if, v katerem preverjamo, ali je v sejni spremenljivki akreditacija nastavljena na uporabnik, administrator ali moderator.

S pomočjo značke gsp remoteLink smo dosegli, da se bo vsebina nalagala asinhrono, s tem bomo lahko posodabljali samo dele strani in ne vedno znova celotne strani. V znački remoteLink so podani atributi za krmilnik in metodo, ki se pokliče ob pritisku na povezavo, podan je tudi atribut update, ki pove, kateri del strani se posodobi ob uspešnem prejemu odgovora krmilnika. Predlogo smo v drugi datoteki gsp vključili s pomočjo meta značke (Slika 3.19).

```
<meta name="layout" content="app_layout" />
```

Slika 3.19: Vključitev izdelane predloge

Po izdelavi predloge za našo spletno aplikacijo smo izdelali prijavno okno (Slika 3.20), ki pa ni uporabljala naše predloge, saj se razlikuje od ostalih pogledov. Za prijavo smo uporabili obrazec, preko katerega lahko uporabnik vnese uporabniško ime in geslo.



Slika 3.20: Prijavni obrazec

Po potrditvi obrazca se ustvari zahteva, nato pa se pokliče krmilnik z imenom prijava. Glede na to, ali je prijava uspešna oz. neuspešna, se prikaže primeren pogled. Na Slika 3.21 je prikazana izvorna koda obrazca.

```
<g:form action="prijava" class="form">
    <input type="text" name="uporabniskoime" placeholder="Uporabniško ime">
    <input type="password" name="geslo" placeholder="Geslo">
    <button type="submit" id="login-button">Prijava</button>
</g:form>
```

Slika 3.21:Izvorna koda obrazca

V primeru neuspešne prijave se preko krmilnika pošlje nazaj sporočilo, da je bila prijava neuspešna, kar se izpiše na uporabniškem vmesniku. Sporočilo je poslano iz krmilnika v spremenljivki flash (Slika 3.22), do katere lahko dostopamo v pogledu.

```
<div id="notification" style="color: red">${flash.message}</div>
```

Slika 3.22: Uporaba spremenljivke flash za izpis sporočila

Izdelavo uporabniškega vmesnika za iskanje knjig smo izdelali s pomočjo obrazca remoteForm, ki ustvari asinhorno zahtevo ajax. Po prejetju odgovora pa rezultat prikaže v kontejnerju, ki je v ta namen že vnaprej definiran. Implementacija obrazca in kontejnerja je vidna na Slika 3.23.

```
<g:formRemote  name="iskanje" url="[controller: 'knjiga', action:'iskanjePoKriteriju']" update="container">
    <input type="text" name="naslov" placeholder="Naslov">
    <input type="text" name="avtor" placeholder="Avtor">
    <button type="submit" id="login-button">Išči</button>
</g:formRemote>
<div id="container"></div>
```

Slika 3.23: Obrazec za iskanje

Krmilnik v odgovoru vrne pogled, ki je definiran v drugi datoteki gsp. V našem primeru je to datoteka rezultatIskanja.gsp, v kateri poskrbimo za prikaz iskanih podatkov. Prikaz vseh iskanih podatkov dosežemo tako, da ustvarimo tabelo, v njej pa ustvarimo zanko, ki ustvari vrstice z iskanimi podatki. Podatke, ki jih moramo prikazati, vrne krmilnik. Implementacija zanke je vidna na Slika 3.24.

```
<g:each in="${result}" var="b">
    <tr>
        <td>${b.avtor.ime} ${b.avtor.priimek}</td>
        <td>${b.naslov}</td>
        <td>${b.opis}</td>
        <td>${b.isbn}</td>
        <g:if test="${eknjiznica.Izposoja_rezervacija.findByKnjiga(b)}">
            <g:if test="${session.status == enums.Akreditacija.uporabnik}">
                <td class='dodatnistolpec'><img src='${g.resource(dir: 'images', file: 'reservation_red.png', absolute: true)}'
                                width='30px' height='30px'/>
            </g:if>
            <g:else>
                <td class='dodatnistolpec'><img src='${g.resource(dir: 'images', file: 'out_icon_red.png', absolute: true)}'
                                width='30px' height='30px'/>
            </g:else>
        </g:if>
        <g:else>
            <g:if test="${session.status == enums.Akreditacija.uporabnik}">
                <td class='dodatnistolpec' id="td${b.id}"><g:remoteLink controller="izposojaRezervacija"
                                                                        action="novaRezervacija"
                                                                        onSuccess="updateTable(td${b.id})"
                                                                        id="${b.id}"><img
                            src='${g.resource(dir: 'images', file: 'reservation_green.png', absolute: true)}'
                            width='30px'
                            height='30px'/></g:remoteLink></td>
            </g:if>
            <g:else>
                <td  id="td${b.id}" class='dodatnistolpec'><a href="#" OnClick="setKnjigaId(${b.id})" id="openmodal" data-toggle="modal"
                                    data-target="#myModal"><img
                            src='${g.resource(dir: 'images', file: 'out_icon_green.png', absolute: true)}'
                            width='30px'
                            height='30px'/></a></td>
            </g:else>
        </g:else>
    </tr>
</g:each>
```

Slika 3.24:Zanka, v kateri izpišemo rezultat

Za dostop do slik smo uporabili funkcijo grails g.resource, ki vrne absolutno pot do slike na strežniku. Ker dostopamo do istega pogleda iz načina uporabnika in administratorja, smo morali s pomočjo sejne spremenljivke preverjati akreditacijo in izpisati pravilne atribute. V primeru uporabnika se v zadnjem stolpcu pokaže opcija za rezervacijo gradiva, ki deluje tako, da se ustvari nova zahteva, ki ustvari novo rezervacijo in jo asocira na uporabnika, ki je prijavljen v sistem. V primeru moderatorja pa se pokaže možnost za izposojo knjige, ki deluje tako, da se odpre modalno okno, preko katerega izberemo uporabnika, ki si želi izposoditi knjigo. Po potrditvi uporabnika se ustvari zahteva ajax, ki ustvari novo izposojo in nanjo asocira podanega uporabnika.

Modalno okno smo v aplikaciji konstanto uporabljali, zato smo si naredili predlogo, preko katere lahko ustvarimo modalno okno in podamo podatke, ki se naj v tem oknu tudi prikažejo. V mapi layouts smo ustvarili novo datoteko gsp, ki bo predstavljala predlogo. Za razliko od glavne predloge, ki služi kot predloga aplikacije, kjer smo definirali prostor za glavo, naslov in telo, tukaj definiramo značke pageProperty (Slika 3.25), ki bodo povedale, kje je prostor za vsebino, ki jo želimo spreminjati.

```
<!-- Modal -->
<div class="modal fade" id="<g:pageProperty name="page.id" default="myModal"/>" role="dialog">
    <div class="modal-dialog">

        <!-- Modal content-->
        <div class="modal-content">
            <div class="modal-header">
                <button type="button" class="close" data-dismiss="modal">&times;</button>
                <h4 class="modal-title"><g:pageProperty name="page.title"/></h4>
            </div>

            <div class="modal-body">
                <g:pageProperty name="page.body"/>
            </div>

            <div class="modal-footer">
                <g:pageProperty name="page.footer"/>
            </div>
        </div>

    </div>
</div>
```

Slika 3.25: Predloga za modalno okno

Predlogo za modalno okno v pogledu uporabimo tako, da ustvarimo značko applyLayout in zraven kot parameter podamo ime predloge. Vsebino, ki želimo, da je prikazana na modalnem oknu, pa definiramo s pomočjo značk content, v atribut tag pa podamo ime pagePropertyja, kateremu želimo nastaviti vrednost. Prikaz implementacije uporabe modalnega okna je viden na Slika 3.26.

```
<g:applyLayout name="modal_window">
    <content tag="title">
        <h4 class="modal-title">Potrditev</h4>
    </content>
    <content tag="body">

        <p>Ali želite preklicati rezervacijo za knjigo <span id="naslovKnjige"></span>?</p>

    </content>
    <content tag="footer">
        <a onclick="prekliciRezervacijo();" class="btn btn-default" data-dismiss="modal">Da</a>
        <button type="button" class="btn btn-default" data-dismiss="modal">Ne</button>
    </content>
</g:applyLayout>
```

Slika 3.26: Uporaba modalnega okna

Dodajanje in urejanje uporabnikov v načinu administratorja smo implementirali tako, da se najprej prikaže seznam uporabnikov, med katerimi lahko tudi iščemo. Vsak uporabnik ima v zadnjem stolpcu opcijo za brisanje ali urejanje uporabnika. Pri akciji za brisanje se asinhrono pošlje zahteva, ki jo prejme krmilnik in pokliče servis za brisanje podanega

uporabnika. Pri izbiri urejanja se prikaže novo modalno okno, v katerem so prikazani podatki izbranega uporabnika. To dosežemo tako, da kličemo funkcijo Javascript, v katero podamo podatke uporabnika (Slika 3.27).

```
<a href="#"
   onclick="nastaviParametre('${u.id}','${u.ime}','${u.priimek}','${u.naslov}','${u.akreditacija}','${u.uporabnisko_ime}')"
   data-toggle="modal" data-target="#myModal">
```

Slika 3.27: Sklic funkcije Javascript za nastavitev parametrov

Po potrditvi podatkov se pošlje asinhrona zahteva krmilniku, ki posodobi uporabnika. Za shranjevanje in posodabljanje uporabnika uporabljamo eno metodo. V primeru, da gre za posodabljanje v krmilnik, podamo id uporabnika, v primeru, da gre za dodajanje uporabnika, pa v metodo ne podamo id uporabnika.

4 REZULTATI

V diplomskem delu smo ugotovili, da je okvir Grails zelo visoko produktivni okvir, kateri zagotovi višjo produktivnost pri razvoju spletne aplikacije. Prišli smo do sklepa, da je okvir Grails najbolj primerno uporabiti, kadar delamo na manjših ali srednje velikih projektih in za protitipiranje aplikacij, saj lahko v zelo kratkem času ustvarimo delujoč prototip aplikacije. Pri raziskavi okvirja Grails lahko izpostavimo glavne prednosti, ki so:

- Zelo hiter razvoj spletne aplikacije
- Veliko razširitev, katere lahko zelo poenostavijo razvoj
- Zelo dobra dokumentacija
- Uporaba programskega jezika Groovy, ki je zelo priročen in lahko razumljiv
- Zelo enostavna in hitra namestitev okvirja
- GORM, ki olajša delo s podatkovno bazo
- Ne prepričuje, da bi direktno uporabljali okvirje, katere uporablja Grails za delovanje
- Spodbuja uporabo principa DRY, kar pomeni, da se koda čim manj ponavlja
- Ni potrebne dodatne konfiguracije za delo z XML ali JSON
- Ime zelo dobro podporo za servise REST
- Zelo dobro se integrira s programskih jezikom Java, tako lahko pišemo aplikacijo v Javi ali v Groovyju, lahko pa tudi uporabimo kombinacijo obeh

Seveda pa ima vsak okvir svoje pomankljivosti. Pri raziskavi in implementaciji smo odkrili naslednje pomankljivosti:

- Če delamo na aplikaciji, ki zahteva več procesov, se lahko pojavi problematika pri uporabi GORM-a
- Pri uporabi spremenljivke def je težje vzdrževanje aplikacije
- Uporaba programskega jezika Groovy se pozna na odzivnosti aplikacije
- Polno podporo za aplikacijo Grails omogoča le plačljiva različica IntelliJ

5 SKLEP

V diplomskem delu smo najprej raziskali in opisali okvir Grails ter tehnologije, ki jih okvir uporablja za svoje delovanje. Ugotovili smo, da okvir Grails uporablja preverjene in testirane okvirje, ki so že nekaj časa v uporabi in so za to stabilni in izpopolnjeni. Preučili smo tehnologije Hibernate in Spring, ki sta jedrni tehnologiji za delovanje okvirja Grails. Kasneje smo uporabo okvirja Grails prikazali v praksi. Izdelali smo spletno aplikacijo za knjižnico. Spletna aplikacija omogoča osnovne funkcionalnosti, ki jih potrebujejo v knjižnici za izposojo in rezervacijo knjig. Iz implementacije aplikacije smo se naučili, da je spletni okvir zelo primeren za hitro protitipiranje aplikacij in da z njim dvignemo produktivnost razvoja aplikacije.

VIRI IN LITERATURA

[1] Grails, „A powerful Groovy-based web application framework for JVM,“ [Elektronski]. Dostopno na: https://grails.org/. [Poskus dostopa 10. 7. 2015].

[2] Wikipedia, „Grails framework,“ [Elektronski]. Dostopno na: https://en.wikipedia.org/wiki/Grails_(framework). [Poskus dostopa 10. 7. 2015].

[3] B. Beckwith, Programming Grails, O'Relly Media, 2013.

[4] G. cookbook, „Grails Tutorial for Beginners - Grails Service Layer,“ [Elektronski]. Dostopno na: http://grails.asia/grails-tutorial-for-beginners-grails-service-layer/. [Poskus dostopa 6. 8. 2015].

[5] G. cookbook, „Grails tutorial for Beginners - Basic Groovy Server Pages (GSP),“ [Elektronski]. Dostopno na: http://grails.asia/grails-tutorial-for-beginners-basic-groovy-server-pages-gsp/. [Poskus dostopa 4. 8. 2015].

[6] Grails, „The Web Layer - Reference Documentation,“ [Elektronski]. Dostopno na: http://grails.github.io/grails-doc/2.3.11/guide/theWebLayer.html#GSPBasics. [Poskus dostopa 3. 8. 2015].

[7] R. Graeme in J. S. Brown, The definitive guide to Grails 2, Aspress, 2012.

[8] Groovy-lang.org, „A multi-faceted language for Java platform,“ [Elektronski]. Dostopno na: http://www.groovy-lang.org/. [Poskus dostopa 12. 7. 2015].

[9] Wikipedia, „Groovy (programming language),“ [Elektronski]. Dostopno na: https://en.wikipedia.org/wiki/Groovy_(programming_language). [Poskus dostopa 10. 7. 2015].

[10] P. Ledbrook, „Groovy in light of Java 8,“ [Elektronski]. Dostopno na: http://blog.cacoethes.co.uk/groovyandgrails/groovy-in-light-of-java-8. [Poskus dostopa 20. 7. 2015].

[11] Stackoverflow, „What is your opinion of Groovy,“ [Elektronski]. Dostopno na: http://stackoverflow.com/questions/562630/what-is-your-opinion-of-groovy. [Poskus dostopa 20. 7. 2015].

[12] Groovy-lang.org, „Differences with Java,“ [Elektronski]. Dostopno na: http://www.groovy-lang.org/differences.html. [Poskus dostopa 7. 20. 2015].

[13] Wikipedia, „Spring framework,“ [Elektronski]. Dostopno na: https://en.wikipedia.org/wiki/Spring_Framework. [Poskus dostopa 5. 8. 2015].

[14] O'reilly.com, „What is Hibernate,“ [Elektronski]. Dostopno na: http://archive.oreilly.com/pub/a/onjava/2005/09/21/what-is-hibernate.html. [Poskus dostopa 10. 8. 2015].

[15] Wikipedia, „Hibernate (Java),“ [Elektronski]. Dostopno na: https://en.wikipedia.org/wiki/Hibernate_(Java). [Poskus dostopa 10. 8. 2015].

Univerza v Mariboru

Fakulteta za elektrotehniko,
računalništvo in informatiko
Smetanova ulica 17
2000 Maribor, Slovenija

FERI

IZJAVA O AVTORSTVU

Spodaj podpisani/-a NIK ORTER

z vpisno številko E10658555

sem avtor/-ica diplomskega dela z naslovom: Izdelava spletne aplikacije s pomočjo okvirja Grails

(naslov diplomskega dela)

S svojim podpisom zagotavljam, da:

- sem diplomsko delo izdelal/-a samostojno pod mentorstvom (naziv, ime in priimek)

red. prof. dr. Peter Kokol, univ. dipl. inž. el.
in somentorstvom (naziv, ime in priimek)

- so elektronska oblika diplomskega dela, naslov (slov., angl.), povzetek (slov., angl.) ter ključne besede (slov., angl.) identični s tiskano obliko diplomskega dela.
- soglašam z javno objavo elektronske oblike diplomskega dela v DKUM.

V Mariboru, dne 24.8.2015

Podpis avtorja/-ice:

Univerza v Mariboru

Fakulteta za elektrotehniko,
računalništvo in informatiko
Smetanova ulica 17
2000 Maribor, Slovenija

FERI

IZJAVA O ISTOVETNOSTI TISKANE IN ELEKTRONSKE VERZIJE ZAKLJUČNEGA DELA IN OBJAVI OSEBNIH PODATKOV DIPLOMANTOV

Ime in priimek avtorja-ice: NIK ORTER

Vpisna številka: E10065855

Študijski program: Računalništvo in informacijske tehnologije

Naslov zaključnega dela: Izdelava spletne aplikacije s pomočjo okvirja Grails

Mentor: red. prof. dr. PETER KOKOL, univ. dipl. inž. el.

Somentor: /

Podpisani-a NIK ORTER izjavljam, da sem za potrebe arhiviranja oddal elektronsko verzijo zaključnega dela v Digitalno knjižnico Univerze v Mariboru. Zaključno delo sem izdelal-a sam-a ob pomoči mentorja. V skladu s 1. odstavkom 21. člena Zakona o avtorskih in sorodnih pravicah dovoljujem, da se zgoraj navedeno zaključno delo objavi na portalu Digitalne knjižnice Univerze v Mariboru.

Tiskana verzija zaključnega dela je istovetna z elektronsko verzijo elektronski verziji, ki sem jo oddal za objavo v Digitalno knjižnico Univerze v Mariboru.

Zaključno delo zaradi zagotavljanja konkurenčne prednosti, varstva industrijske lastnine ali tajnosti podatkov naročnika: _____ ne sme biti javno dostopno do _____(datum odloga javne objave ne sme biti daljši kot 3 leta od zagovora dela).

Podpisani izjavljam, da dovoljujem objavo osebnih podatkov, vezanih na zaključek študija (ime, priimek, leto in kraj rojstva, datum zaključka študija, naslov zaključnega dela), na spletnih straneh in v publikacijah UM.

Datum in kraj: 25.8.2015, Ravne na Kor. Podpis avtorja-ice: _____

Podpis mentorja: _____
(samo v primeru, če delo ne sme biti javno dostopno)

Podpis odgovorne osebe naročnika in žig:
(samo v primeru, če delo ne sme biti javno dostopno) _____

Univerza v Mariboru

Fakulteta za elektrotehniko,
računalništvo in informatiko
Smetanova ulica 17
2000 Maribor, Slovenija

FERI

IZJAVA O USTREZNOSTI ZAKLJUČNEGA DELA

Podpisani mentor :

red. prof. dr. Peter Kokol, univ. dipl. inž. el.

(ime in priimek mentorja)

in somentor (eden ali več, če obstajata):

_____

(ime in priimek somentorja)

Izjavljam (-va), da je študent

Ime in priimek: NIK ORTER

Vpisna številka: E1065855

Na programu: Računalništvo in informacijske tehnologije

izdelal zaključno delo z naslovom:

Izdelava spletne aplikacije s pomočjo ogrodja Grails

(naslov zaključnega dela v slovenskem in angleškem jeziku)

Development of web application with Grails framework

v skladu z odobreno temo zaključnega dela, Navodilih o pripravi zaključnih del in mojimi (najinimi oziroma našimi) navodili.

Preveril (-a, -i) in pregledal (-a, -i) sem (sva, smo) poročilo o plagiatorstvu.

Datum in kraj: 24.8.2015, Maribor Podpis mentorja:

Datum in kraj: Podpis somentorja (če obstaja):

Priloga:

- Poročilo o preverjanju podobnosti z drugimi deli.«