Univerzitet u Beogradu
Matematički fakultet

Milan Mitić

Bezbednosni aspekti razvoja složene Java Veb aplikacije sa više nivoa pristupa

Master rad

Beograd 2017

Univerzitet u Beogradu
Matematički fakultet

Master rad

Autor: Milan Mitić
Naslov: Bezbednosni aspekti razvoja složene Java Veb aplikacije sa više nivoa pristupa
Mentor: prof dr Vladimir J. Filipović
Članovi komisije: prof dr Dušan Tošić
prof dr Saša Malkov
Datum:

Sadržaj

1. Uvod

Prilikom razvoja složenih Veb aplikacija, programeri se suočavaju sa dva ozbiljna izazova: kako obezbediti da različite vrste korisnika mogu pristupati samo onim delovima aplikacije i onim podacima za koje su im data ovalšćenja i kako obezbediti da sistem bude maksimalno zaštićen od napada iz spoljašnosti, imajući u vidu široke mogućnosti pristupa Veb aplikaciji od strane klijenta.
Programski jezik Java je jedan od najpopularnijih programskih jezika i on se često koristi za razvoj složenih Veb aplikacija. *Spring* predstavlja najčešće korišćen okvir otvorenog koda za razvoj složenih *Java* Veb aplikacija. *Spring*-ova podrška za *model-pogled-kontroler* (eng. model-view-controller - MVC) arhitekturu omogućava brz i lak razvoj složenih Veb aplikacija. *Hibernate* je okvir koji realizuje mapiranje objektno orijentisanog modela u model relacione baze podataka i koji, u kombinaciji sa *Spring* okvirom, omogućava još brži i jednostavniji razvoj složenih *Java* Veb aplikacija.

Cilj master rada je istraživanje korišćenja prethodno pomenutih tehnologija. U okviru rada će biti razvijena aplikacija korišćenjem ovih tehnologija koja će imati više vrsta korisnika, u kojoj će biti regulisana prava pristupa i koja će biti zaštićena od svih najpoznatijih vrsta napada na Veb aplikacije.

U drugom poglavlju master rada biće ukratko opisane tehnologije korišćene za izradu aplikacije. U trećem odeljku rada je opisana aplikacija. Objašnjeno je kako pomenutim tehnologijama postaviti arhitekturu aplikacije i opisane su njene funkcionalnosti. Četvrti deo master rada se odnosi na bezbednost aplikacije. Detaljno je opisan proces autentifikacije i autorizacije. Opisane su sve najpoznatije vrste napade na Veb aplikacije i kako se od njih zaštititi.

Aplikacija za potrebe ovog master rada je razvijena kao softer otvorenog koda i može se naći na adresi: *https://github.com/miticmilan89/master.*

2. Tehnologije

2.1 Java

Java je objektno-orijentisani programski jezik široke primene. Napravljen je sa idejom da se jednom napisani kôd može izvršavati svuda (eng. write once, run anywhere - WORA), što znači da se kompajlirani *Java* kôd može pokrenuti na bilo kojoj platformi koja podržava *Javu,* a da se ne mora pre toga ponovo kompajlirati. *Java* aplikacije su obično kompajlirane u bajt kôd koji se može pokrenuti na bilo kojoj *Java* virtualnoj mašini nezavisno od arhitekture računara. *Java* je jedan od najpopularnijih programskih jezika koji se koriste, pogotovo za klijent-server Veb aplikacije. Po nezvaničnoj statistici, postoji preko 9 miliona *Java* programera u svetu. *Java* programski jezik je prvobitno razvio James Gosling iz kompanije Sun Microsystems, a objavljen je 1995. godine. Nasledio je dosta sintakse od programskih jezika C i C++, ali je mnogo stroži pri prevođenju. Dizajniran je tako da bude nezavisan od platforme i sa pojednostavljenim upravljanjem memorijom. Osnovne karakteristike jezika *Java* su:

- Jednostavnost - *Java* je koncepirana tako da programeri mogu jednostavno i brzo da je nauče. Pod uslovom da programer ima određeno iskustvo u programiranju, neće mu biti teško da savlada *Javu*. Poznavanje osnova objektno-orijentisanog programiranja dodatno ubrzava proces učenja.
- Objektno-orijentisano - U *Javi* sve je objekat. Model objekta u *Javi* je jednostavan i lako se proširuje.
- Robusno - Veb okruženje sa više platformi postavlja izuzetne programske zahteve jer program mora da se pouzdano izvršava na različitim sistemima. Zbog toga je pri projektovanju *Jave* dat prioritet sposobnosti da se napravi robustan program. U cilju postizanja pouzdanosti rada programa, *Java* ograničava programere u nekoliko ključnih područija, terajući ih da programske greške isprave u ranoj fazi. Istovremeno, ona ih oslobađa briga o mnogim čestim programskim greškama. Pošto je *Java* strogo tipiziran jezik, ona proverava kôd u trenutku njegovog prevođenja, ali i u trenutku izvršavanja. Glavna prednost *Jave* je što programeri mogu da predvide kako će se ono što su napisali ponašati u razlišitim situacijama.
- Višenitno - *Java* je projektovana tako da izađe u susret realnim zahtevima pravljenja interaktivnih mrežnih programa. Da bi se to postiglo, *Java* podržava višenitno programiranje, koje omogućava da program istovremeno radi više stvari.
- Nezavisno od platforme - Osnovni problem dizajnera *Jave* je bilo stvaranje prenosivog koda koji će trajno raditi. Danas kada se napiše program koji radi, niko ne može da garantuje da će raditi i sutra, čak i na istoj mašini. Operativni sistemi se neprestano poboljšavaju, kao i procesori, a kada se sve kombinuje sa promenama u osnovnim resorsima sistema, može se dogoditi da program više ne radi kako treba. Zato su dizajneri *Jave* smislili *Java* virtuelnu mašinu pomoću koje je *Java* postala nezavisna od platforme i arhitekture računara.
- Distribuirano - *Java* je posebno namenjena distribuiranom okruženju Interneta

jer lako rukuje protokolima TCP/IP. U stvari, pristupanje resursu pomoću URL-a u osnovi se ne razlikuje od pristupanja datoteci. *Java* podržava i daljinsko izvršavanje procedura (eng. Remote Method Invocation - RMI). Dakle, program može da zahteva izvršavanje procedura koje se nalaze na bilo kojoj mrežnoj adresi. [2]

Bitan deo *Jave,* koji je dosta korišćen u aplikaciji, jesu *Java* anotacije (eng. Java Annotations). *Java* anotacije su oznake koje predstavljaju meta podatke i one mogu biti povezane sa klasama, interfejsima, metodama i atributima. One služe kako bi prižile dodatne informacije koje mogu biti korišćene od strane kompajlera ili *Java* virtuelne mašine. One služe kao zamena za XML konfiguracione fajlove, i dosta su jednostavnije i preglednije za korišćenje. Postoji nekoliko ugrađenih anotacija u *Javi.* To su:
- *@Override* - služi da se naglasi da metod potklase redefiniše metod iz natklase.
- *@SuppressWarnings* - služi da potisne upozorenja od strane kompajlera
- *@Deprecated* - služi da se naglasi da je metoda zastarela i da će možda biti uklonjena u nekoj od narednih verzija

Java pruža mogućnost pravljenja sopstvenih anotacija. Anotacione klase moraju biti obeležene anotacijom *@interface*. Pored toga, za kreiranje novih anotacija se koriste sledeće ugrađene anotacije:
- *@Target* - služi da se označi na koji tip će se odnositi anotacija. Tipovi mogu biti klase, metode, polja, konstruktori
- *@Retention* - služi da se označi na kom nivou će se anotacija koristiti. Mogući nivoi su izvorni kôd, kompajlirani kôd ili izvršni kôd.
- *@Inherited* - služi da se označi da li će anotacija biti nasleđena ili ne
- *@Documented* - označava da li će anotacija biti dokumentovana ili ne

Razvojno okruženje Spring poseduje veliki broj sopstvenih anotacija.

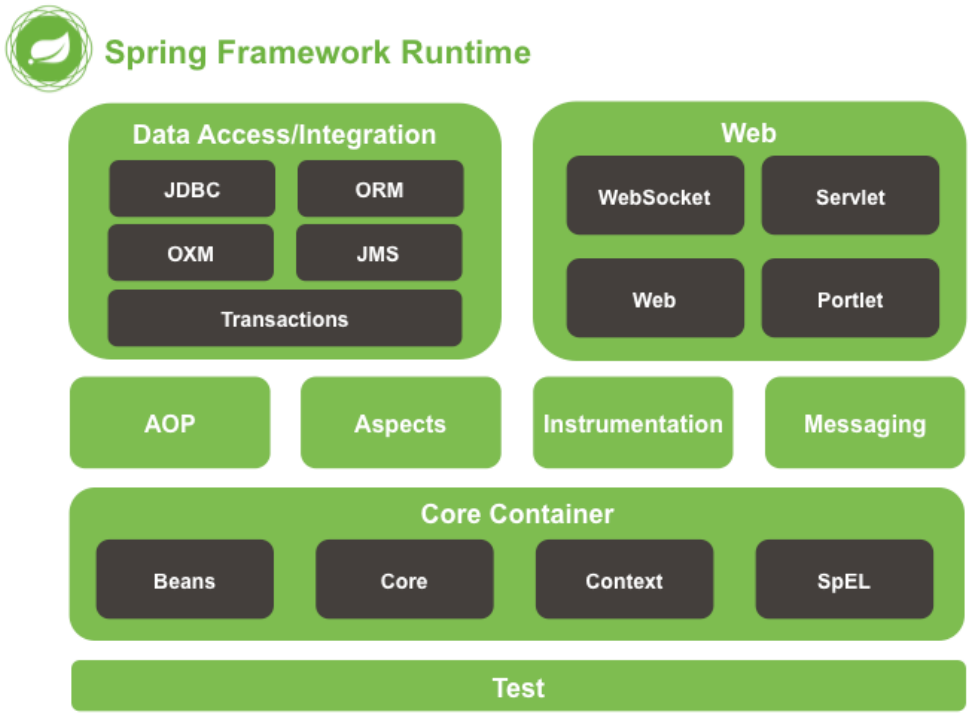
2.2 Razvojno okruženje Spring

Razvojno okruženje *Spring* je *Java* platforma koja pruža veliki broj opcija i predstavlja podršku razvoju složenih *Java* aplikacija. *Spring* upravlja infrastrukturom tako da se programer može usredsrediti na aplikaciju. Važni aspekti *Spring*-a su ubrizgavanje zavisnosti (eng. Dependency Injection) i inverzija kontrole (eng. Inversion of Control). *Java* aplikacija se sastoji od objekata koji međusobno sarađuju kako bi aplikacija zadovoljila funkcionalnost. To znači da objekti unutar aplikacije zavise jedan od drugog. Ubrizgavanje zavisnosti omogućava razvoj aplikacije korišćenjem jednostavnih *Java* objekata (eng. plain old Java object - POJOs) nad kojima se primenjuju napredniji servisi kako bi se definisala zavisnost medju njima.

Iako *Java* pruža veliki broj funkcionalnosti za razvoj aplikacija, manjkava je u smislu organizacije osnovnih gradivnih blokova u jedinstvenu celinu, pa ovaj posao pada na arhitekte i programere. Iako se mogu koristiti obrasci za projektovanje (eng. Design patterns) kao što su: Fabrika (eng. Factory), Apstraktna fabrika (eng. Abstract factory), Graditelj (eng. Builder) i Dekorater

(eng. Decorator), da bi se sastavile različite klase i primerci objekata u celinu koja čini aplikaciju, ipak su to samo obrasci sa datim imenom i opisom kada i kako ih treba primeniti, a na programeru ostaje da ih implementira unutar aplikacije. *Spring*-ova komponenta ubrizgavanje zavisnosti olakšava ovaj proces tako što obezbeđuje formalizovane načine kako razdvojene komponente sastaviti u kompletno funkcionalnu celinu. *Spring* kodira obrasce za projektovanje unutar kompeksnih objekata koji se kao takvi koriste unutar aplikacije. [3]

Razvojno okruženje Spring obezbeđuje funkcionalnosti koje su organizovane u 20 modula. Ovi moduli su grupisani u *Core Container, Data Access/Integration, Web, AOP, Instrumentation, Messaging,* i *Test* kao što je prikazano na slici 1.



slika 1. Moduli razvojnog okruženja Spring

Core Container se sastoji od *spring-core, spring-beans, spring-context, spring-context-support* i *spring-expressions* modula. Moduli *spring-core* i *spring-beans* predstavljaju osnovu razvojnog okruženja, ukjlučujući inverziju kontrole i ubrizgavanje zavisnosti. Modul *spring-context* nasleđuje prethodna dva i pruža dodatnu podršku za internacionalizaciju, propagiranje događaja, učitavanje resorsa i kreiranje konteksta. Modul *spring-context-support* pruža podršku za integraciju bibloteka treće strane (eng. third-party library). Modul *spring-expressions* služi za upravljanje objektima prilikom izvršavanja programa.

AOP i *Instrumentation* se sastoje od *spring-aop* modula koju pruža podršku za aspektno-orijentisano programiranje (eng. Aspect-Oriented Programming - AOP). U ovom delu se nalazi i *spring-aspects* modul koji pruža podršku za integraciju sa *AspectJ*. Modul *spring-instrument* sadrži klase koje će biti korišćene od strane aplikativnih servera. Modul *spring-instrument-tomcat* sadrži klase koje koristi

Tomcat server.

Messaging deo razvojnog okruženja pruža podršku za aplikacije zasnovane na razmeni poruka.

Data Access/Integration deo pruža podršku za sve različite načine upravljanja podacima. To su *JDBC* (Java Database Conectivity), *ORM* (Object-Relational Mapping), *OXM* (Object/XML mapping), *JMS* (Java Messaging Service). Modul korišćen u aplikaciji je *spring-orm* koji pruža podršku za objektno-orijentisano mapiranje. Konkretno, korišćena je podrška za *Hibernate*.

Web deo sa sastoji od *spring-web, spring-webmvc, spring-websocket* i *spring-webmvc-portlet* modula. Modul *spring-web* pruža osnove za integraciju sa Vebom kao što su *upload* datoteka, inicijalizacija *Servlet Listener*-a i podršku za HTTP. Modul *spring-webmvc* pruža podršku za model-pogled-kontroler (eng. Model-View-Controller - MVC) arhitekturu kao i *REST* Veb servise. Ovaj modul predstavlja osnovu razvijene aplikacije.

Test deo pruža podršku za testiranje aplikacije kao i za integraciju sa postojećim testnim alatima kao što su *JUnit* i *TestNG*.

2.3 JavaServer Pages - JSP

JavaServer Pages (JSP) je tehnologija za kreiranje Veb stranica sa dinamičkim sadržajem. Za razliku od običnih *HTML* stranica koje su statične i uvek imaju isti sadržaj, *JSP* stranice mogu promeniti sadržaj u zavisnosti od nekih faktora, kao što su identitet korisnika, tip pretraživača, informacije prosleđene od strane korisnika, itd... *JSP* stranice su sačinjene od statičkih elemenata kao što su *HTML* etikete, ali i od specijalnih *JSP* elemenata koji omogućavaju serveru da ubaci dinamički sadržaj u stranicu. Ovi elementi su sačinjeni od *Java* koda koji se izvršava na serverskoj strani. Kada korisnik zahteva stranicu, server izvršava ove *JSP* elemente, spaja ih sa statičkim elementima i takvu stranicu vraća pretraživaču. *JSP* obezbeđuje veliki broj etiketa, ali takođe postoje biblioteke koje obezbeđuju dodatne etikete. Jedan takva biblioteka je *JSP Standart Tag Library* (JSTL) koja enkapsulira bazične funkcionalnosti zajedničke za mnoge Veb aplikacije u vidu etiketa. Ona omogućava optimizaciju implementacije kôda stavljajući na raspolaganje širok spektar etiketa kao sto su na primer etikete za iteraciju ili proveru ispunjenosti uslova. [4] Da bi bilo moguće koristiti ove etikete, potrebno je učitati ih na početku svake JSP stranice:

```
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
```

Na primer, iteracija brojeva od 0 do 9, koja odgovara *for* petlji u *Javi,* na JSP stranici bi izgledala:

```
<c:forEach begin="0" end="9" var="i">
```

```
    ...
</c:forEach>
```

Provera da li je neki broj *n* veći od 10, koja odgovara *if* uslovu u *Javi,* na JSP stranici bi izgledala:

```
<c:if test="${ n > 10 }">
    ...
</c:if>
```

2.4 Hibernate

Hibernate je alat otvorenog koda za objektno-relaciono mapiranje u *Java* programskom jeziku. On pruža okvir za mapiranje objektno-orijentisanog modela u model relacione baza podataka. *Hibernate* ima podršku za sve najpoznatije tipove relacionih baza podataka (DB2, MySQL, Oracle,...). Njegov primarni zadatak je mapiranje *Java* klasa u tabele baze podataka i mapiranje *Java* tipova podataka u *SQL* tipove podataka. Mapiranje *Java* klasa u tabele baze podataka se vrši ili pomoću XML fajlova ili pomoću *Java* anotacija. *Hibernate* pruža *Hibernate Query Language* (HQL) koji predstavlja objeknto-orijentisanu verziju SQL-a. Pomoću njega upiti se pišu nad Java objektima, a ne nad tabelama baze podataka. Na ovaj način se programerima omogućava da pišu upite nezavisne od tipa baze podataka. Ukoliko projekat zahteva promenu tipa baze podataka, neće biti potrebe za menjanjem upita unutar projekta. [3]

2.5 HTML, CSS, JavaScript, jQuery

HTML je jezik za obeležavanje pomoću kojeg se prave Veb stranice i Veb aplikacije. CSS je jezik kojim se opisuje izgled HTML dokumenta. JavaScript je skriptni jezik koji se koristi za definisanje funkcionalnosti Veb stranica na klijentskoj strani. jQuery je JavaScript bioblioteka koja u mnogome pojednostavljuje JavaScript programiranje.

Iako JavaScript u svom imenu ima reč Java on uopšte nije sličan Javi. Java je objektno-orijentisani programski jezik, dok je JavaScript objektno-orijentisani skriptni jezik. Java aplikacije se izvršavaju na virtuelnoj mašini, dok se JavaScript izvršava u pretraživaču. Java kôd mora biti kompajliran, dok je JavaScript predstavljen tekstualno i interpretiran od strane pretraživača.

2.6 Druge tehnologije

Pored Jave i Spring-a postoje mnoge druge tehnologije pomoću kojih se mogu razviti bezbedne Veb aplikacije. Najčešće korišćene tehnologije su PHP i ASP.NET.

PHP (eng. *PHP: Hypertext Preprocessor*) je skriptni jezik koji se izvršava

na serverskoj strani i primarno je razvijen u svrhe Veb programiranja. Razvio ga je Rasmus Lerdorf 1994. godine pod uticajem *C, Java i Perl* programskih jezika. Koristi se tako što se PHP kôd uključuje unutar HTML dokumenata kako bi omogućio izvršavanje na serverskoj strani. PHP pruža veliku podršku bezbednosnom aspektu aplikacija i postoje mnoge knjige na tu temu [11].

ASP.NET je razvojno okruženje otvorenog koda koje omogućava programerima kodiranje dinimačkih Veb stranica, aplikacija i servisa. Razvijeno je od strane kompanije Microsoft 2002. godine kao naslednik ASP (eng. Active Server Pages) tehnologije. ASP.NET koristi zajedničko jezičko okruženje (eng. *Common Language Runtime - CLR*) koje omogućava programerima da pišu kôd na bilo kom jeziku koji je podržan od strane .NET okruženja. Neki od najčešće korišćenih jezika su C#, C++ i Visual Basic. ASP.NET posvećuje pažnju bezbednosti Veb aplikacija i postoje mnoge knjige na ovu temu [12].

3. Aplikacija

3.1 Opis funkcionalnosti

U ovom delu su opisane funkcionalnosti aplikacije, bez akcenta na to kako su te funkcionalnosti implementirane. Detalji implementacije su prikazani u delu o bezbednosti.

Aplikacija nema određenu, usku tematiku. Ona predstavlja moguću jaku osnovu za bilo koju vrstu aplikacije i poslovni domen gde je važna sigurnost. Sa osmišljenom tematikom, jednostavnim proširenjem baze podataka i funckionalnosti bi se moglo doći do kompletno funkcionalne aplikacije.

Aplikacija podržava dva tipa aktera:

1. Administrator (eng. Admin)
2. Korisnik (eng. User)

Administratori imaju omogućen pristup svim funkcionalnostima aplikacije. Korisnici imaju pristup određenim funkcionalnostima aplikacije. Različiti korisnici mogu imati pristupe različitim funcionalnostima i ovo je moguće konfigurisati kroz aplikaciju. Korisnik je striktno vezan za entitet aplikacije koji se naziva učesnik. S obzirom da aplikacija nema tematiku, ovaj entitet u aplikaciji ima generičko ime - učesnik. Šta bi ovaj entitet predstavljao u aplikaciji sa određenom tematikom? Ukoliko bi tematika aplikacije bila, recimo, studentski servis, entitet učesnik bi bio fakultet, dok bi korisnici bili studenti. Struktura aplikacije bi bila takva da jedna instanca aplikacije, povezana na jednu bazu podataka, omogućava finkcionisanje studentskih servisa za više fakulteta. Svaki korisnik, odnosno student, bi morao da pripada nekom od fakulteta i on bi imao pristup samo podacima tog fakulteta. Dakle, u istoj bazi podataka bi se čuvali podaci više fakulteta, dok bi unutar aplikacije ti podaci bili potpuno razdvojeni.

Kako je ovo omogućeno biće objašnjeno u delu o bezbednosti. Još jedan primer tematike za aplikaciju bi bio bankarski sistem. U tom slučaju entitet učesnik bi bila banka, dok bi korisnici bili službenici banaka. U nastavku opisa funcionalnosti koristiće se generičko ime entiteta, dakle učesnik.

U aplikaciji postoje dva formulara za prijavljivanje. Jedan je za prijavljivanje administratora, dok je drugi za prijavljivanje korisnika. Izgledaju isto, ali do njih se stiže preko različitih URL-ova.

Izgled stranice je jednostavan i potrebno je uneti samo korisničko ime i lozinku (slika 2). Aplikacija je višejezična, i na stranici postoji mogućnost promene jezika. Aplikacija podržava srpski i engleski jezik.

slika 2 - Stranica za prijavljivanje

Korisničko ime i kriptovana lozinka se čuvaju u bazi podataka. Proces prijavljivanje biće detaljno objašnjen u delu o bezbednosti. Nakon prijavljivanja, korisnik je preusmeren na početnu stranicu (slika 3).

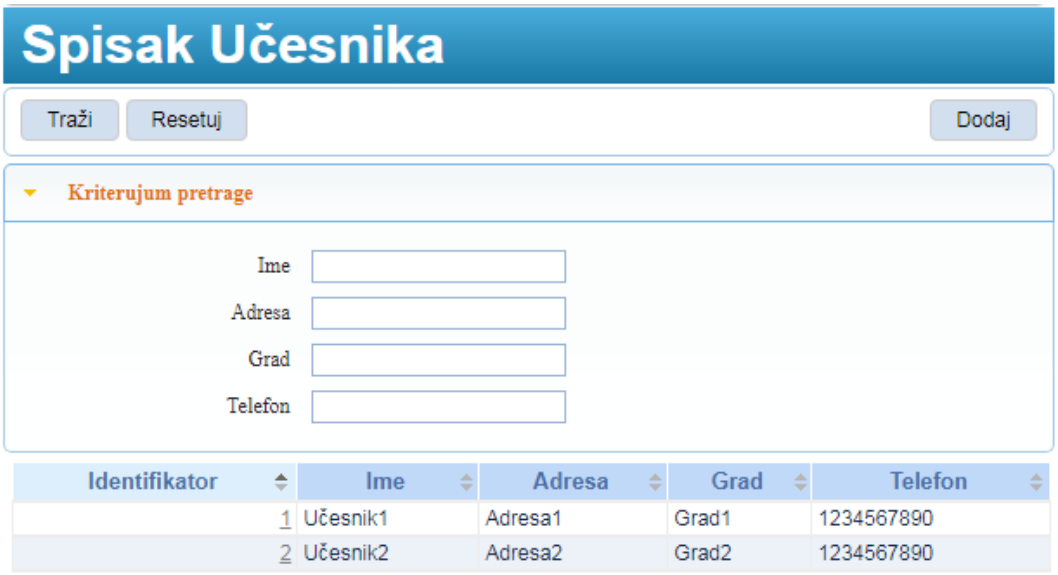
slika 3 - Početna strana aplikacije

Svakoj stranici u aplikaciji je zajednički deo koji se nalazi u gornjem desnom uglu i gde je moguće promeniti jezik i odjaviti se. U tom delu se nalaze i informacije o trenutno prijavljenom korisniku i preostalom trajanju sesije. Na svakoj stranici aplikacije se nalazi i deo za navigaciju. Administratori uvek vide sve funcionalnosti u navigaciji, dok korisnici vide samo one funcionalnosti kojima imaju pristup.

Spisak funkcionalnosti koje poseduje aplikacija

- Dodavanje/Pregled/Izmena/Brisanje učesnika
- Dodavanje/Pregled/Izmena/Brisanje korisnika
- Dodavanje/Pregled/Izmena/Brisanje uloga
- Dodavanje/Pregled/Izmena/Brisanje pravila lozinke
- Izmena lozinke
- Pregled izmena objekata aplikacije
- Pregled aktivnosti aplikacije

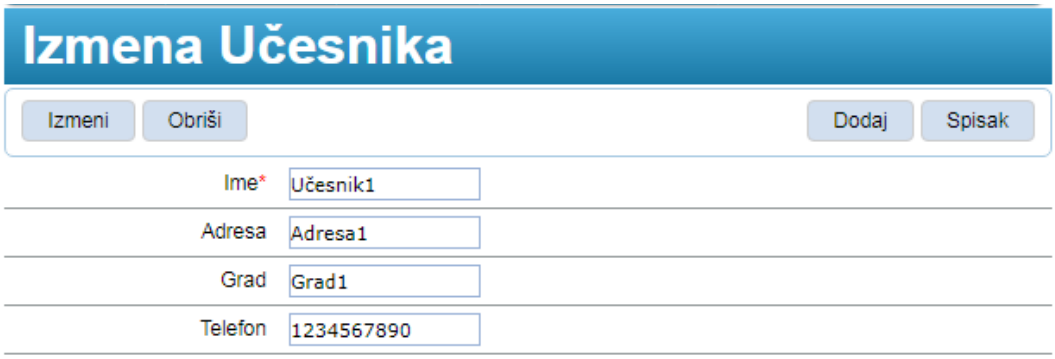
Dodavanje/Pregled/Izmena/Brisanje učesnika je funkcionalnost kojoj mogu pristupiti samo administratori. Već je objašnjeno koja je uloga entiteta učesnik u aplikaciji. Za svaki entitet kojim se može manipulisati u aplikaciji postoje dve vrste stranica. Na jednoj stranici je prikazan spisak postojećih entiteta, sa mogućnošću pretrage i pristupa određenom entitetu (slika 4).



slika 4 - Stranica za prikaz spiska entiteta

Druga stranica je stranica na kojoj se može dodati novi entitet (slika 5), odnosno pregledati ili obrisati postojeći entitet (slika 6).

slika 5 - Stranica za dodavanje novog entiteta



slika 6 - Stranica za izmenu i brisanje entiteta

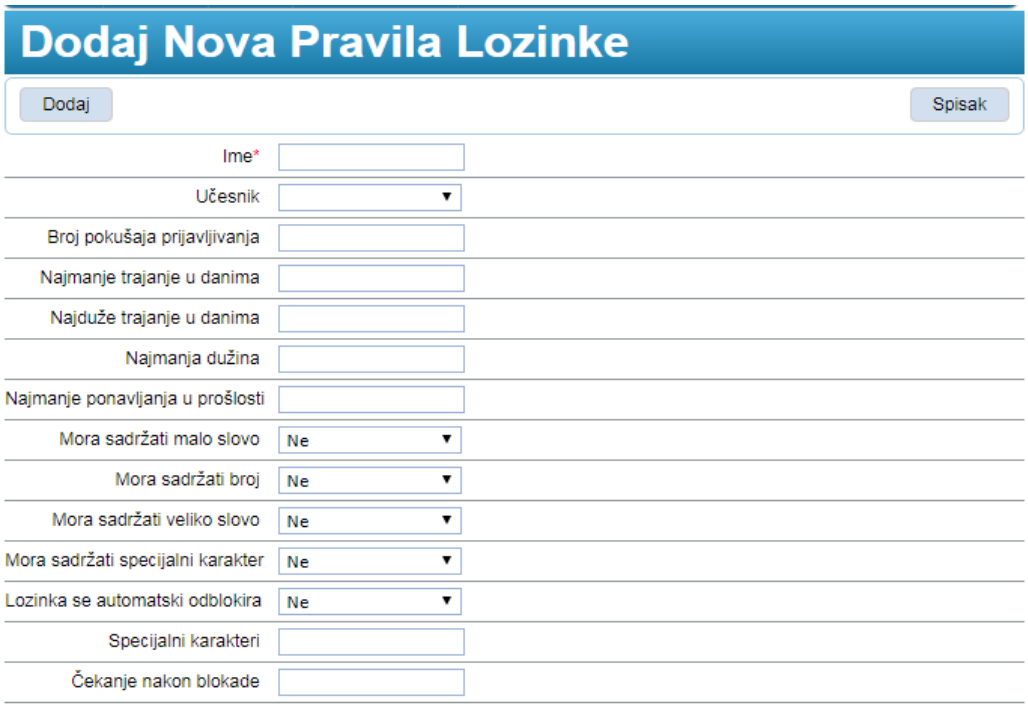
Pre nego što budu objašnjene funkcionalnosti vezane za korisnika, biće objašnjene funkcionalnosti uloga i pravila lozinke.

Kao što je rečeno, korisnik može pristupiti samo određenim funkcionalnostima aplikacije. Spisak ovih funkcionalnosti se nalazi u bazi podataka. Svaka funcionalnost sadrži spisak svih URL-ova koji odgovaraju toj funcionalnosti. Spisak funkcionalnosti u aplikaciji je sledeći:

- Pregled korisnika
- Izmena korisnika
- Pregled uloge
- Izmena uloge
- Pregled pravila lozinke
- Izmena pravila lozinke

Kojim funkcionalnostima mogu pristupiti korisnici se određuje pomoću entiteta **uloga**. Prilikom dodavanja/izmene uloge, moguće je dodeliti joj jednu ili više funkcionalnosti. Nakon toga, na formularu za dodavanje/izmenu korisnika, moguće je korisniku dodeliti jednu ili više uloga. Korisnik će moći pristupiti samo onim funkcionalnostima koje pripadaju ulogama koje su mu dodeljene. Ideja aplikacije je da, nakon što administrator korisnik napravi novog učesnika, pored toga, napravi novu ulogu za tog učesnika kojoj će dodeliti sve funkcionalnosti i napravi novog korisnika za tog učesnika kojem će biti dodeljena ta uloga. Taj korisnik će biti u mogućnosti da pravi druge korisnike, kao i da pravi nove uloge koje će dodeljivati tim korisnicima. Na taj način, on će moći da pravi nove podvrste korisnika, odnosno korisnike koji će moći pristupati različitim funkcionalnostima aplikacije.

Dodavanje/Pregled/Izmena/Brisanje pravila lozinke je funkcionalnost kojoj mogu pristupiti oba tipa korisnika. Funkcionalnost se sastoji u tome da se definišu pravila koja korsnička lozinka mora zadovoljiti (slika 7).



slika 7 - Pravila lozinke

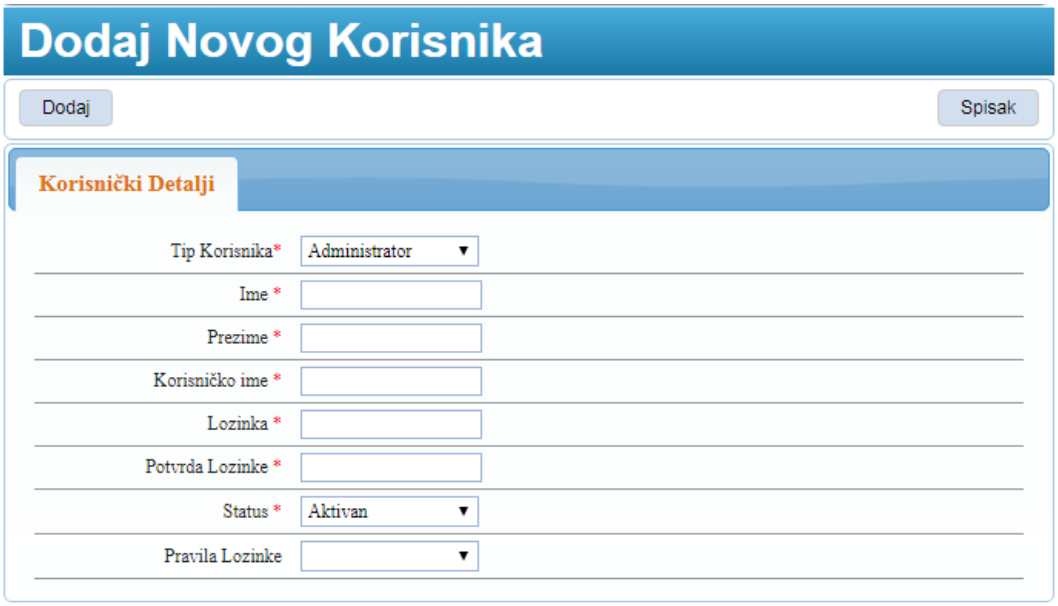
Pravilima lozinke je potrebno dati ime i odrediti za kog učesnika važe. Ukoliko učesnik nije izabran, znači da se pravila lozinke odnose na administrator korisnike. Značenje ostalih polja:

- **Broj pokušaja prijavljivanja** - koliko puta korisnik sme da pogreši lozinku pre nego što bude blokiran. Sprečava zlonamernog korisnika da sa pogađanjem lozinke prijavi na sistem.
- **Najmanje trajanje u danima** - minimalni broj dana koji mora da prođe između dve promene lozinke. Sprečava korisnika da često menja lozinku.
- **Najduže trajanje u danima** - broj dana nakon kojeg je neophodno promeniti lozinku. Podstiče korisnike da češće menjaju lozinke.
- **Najmanje dužina** - označava koliko najmanje karaktera mora sadržati lozinka.
- **Najmanje ponavljanja u prošlosti** - označava broj lozinki iz prošlosti koje korisnik ne sme izabrati kao novu lozinku. Sprečava korisnika da ponavlja lozinke iz prošlosti.
- **Mora sadržati malo slovo** - lozinka mora sadržati bar jedno malo slovo
- **Mora sadržati broj -** lozinka mora sadržati bar jedan broj
- **Mora sadržati veliko slovo -** lozinka mora sadržati bar jedno veliko slovo.
- **Mora sadržati specijalni karakter -** lozinka mora sadržati bar jedan specijalni karakter.
- **Lozinka se automatski odblokira** - označava da li se nakon blokade lozinka automatski odblokira ili ne.
- **Specijalni karakteri** - označava listu specijalnih karaktera koje korisnik mora izabrati ukoliko lozinka mora sadržati specijalni karakter
- **Čekanje nakon blokade** - označava vreme u minutima koje mora da prođe nakon blokade lozinke kako bi se lozinka odblokira. Ovo je samo u slučaju kada je označeno da se lozinka automatski odblokira.

Kao što vidimo pravila lozinke obezbeđuju dodatnu sigurnost aplikaciji. Svakom korisniku je moguće dodeliti jedno od pravila.

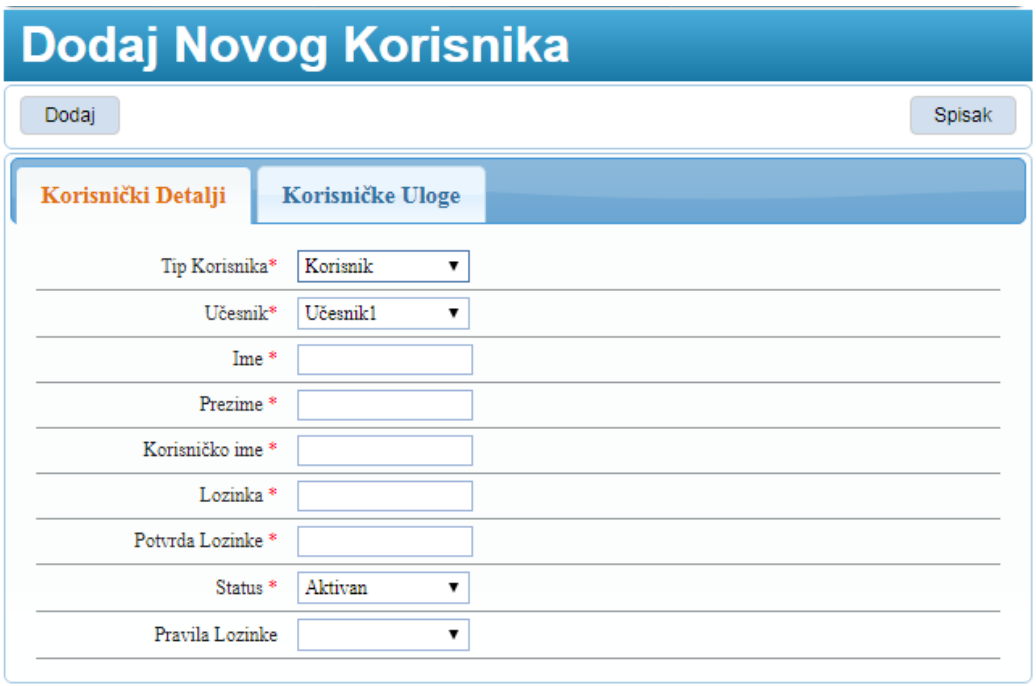
Dodavanje/Pregled/Izmena/Brisanje korisnika je funkcionalnost kojoj mogu pristupiti oba tipa korisnika. Administrator može upravljati drugim administratorima i korisnicima, dok korisnik može upravljati samo drugim korisnicima koji pripadaju istom učesniku kao on. Ovako izraženo zvuči prilično konfuzno, pa će ovo biti objašnjeno na konkretnom primeru. Recimo da je tematika aplikacije bankarski sistem. Administrator bi mogao upravljati drugim administrator korisnicima, ili korisnicima bilo koje banke. Dok bi korisnik banke mogao upravljati samo korisnicima svoje banke.

U delu koji sledi opisuje se kako izgleda formular za dodavanje novog korisnika iz ugla administrator korisnika (slika 8).



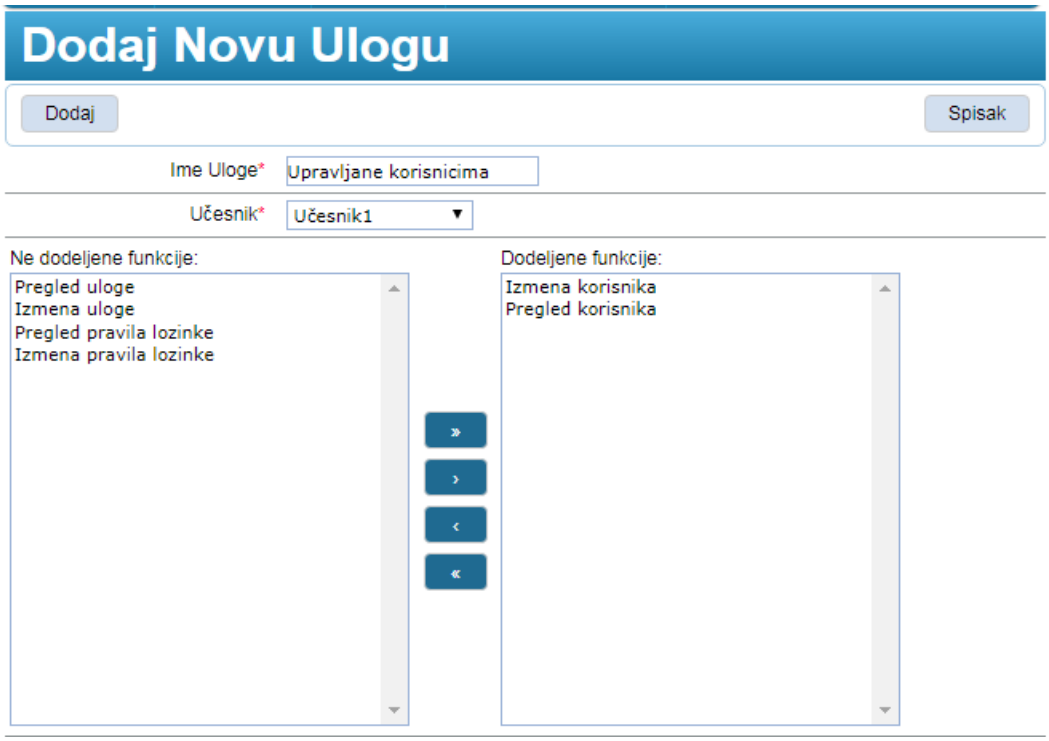
slika 8 - Dodavanje novog korisnika

S obzirom da Administrator korisnik može da dodaje nove korisnike oba tipa, on mora prvo da izabere tip korisnika. Ostali podaci bitni za korisnika su ime i prezime, zatim korisničko ime i lozinka koje će korisnik koristiti za prijavljivanje na aplikaciju. Moguće je još izabrati status korisnika i dodeliti mu neko od pravila lozinki. Ukoliko se dodaje običan korisnik, na formularu se prikazuju dodatna polja (slika 9).



slika 9 - Dodavanje novog korisnika iz ugla administratora

Na formularu se pojavilo novo polje u kome je potrebno izabrati kom učesniku će pripadati novi korisnik. I pojavila se nova sekcija u kojoj je moguće korisniku dodeliti neku od korisničkih uloga. Pre svega, dodata je jedna uloga. Uloga je predviđena za korisnike koji pripadaju učesniku "Učesnik1", i dodeljena joj je funkcionalnosti za pregled i izmenu korisnika. Uloga je nazvana "Upravljanje korisnicima" (slika 10).

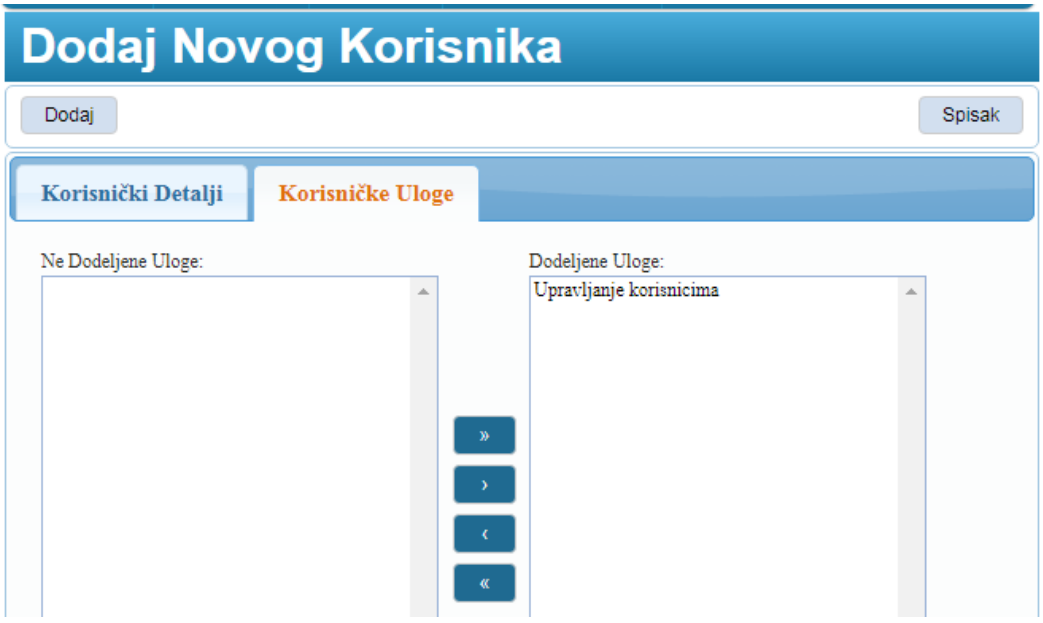


slika 10 - Dodavanje uloge za upravljanje korisnicima

Sada je moguće napraviti novog korisnika kome će se dodeliti ova uloga. Izabrano je da on pripada Učesniku "Učesnik1", i uneti su ostali potrebni podaci (slika 11).

slika 11 - Dodavanje novog korisnika

Zatim je korisniku dodeljena uloga "Upravljanje korisnicima" (slika 12).



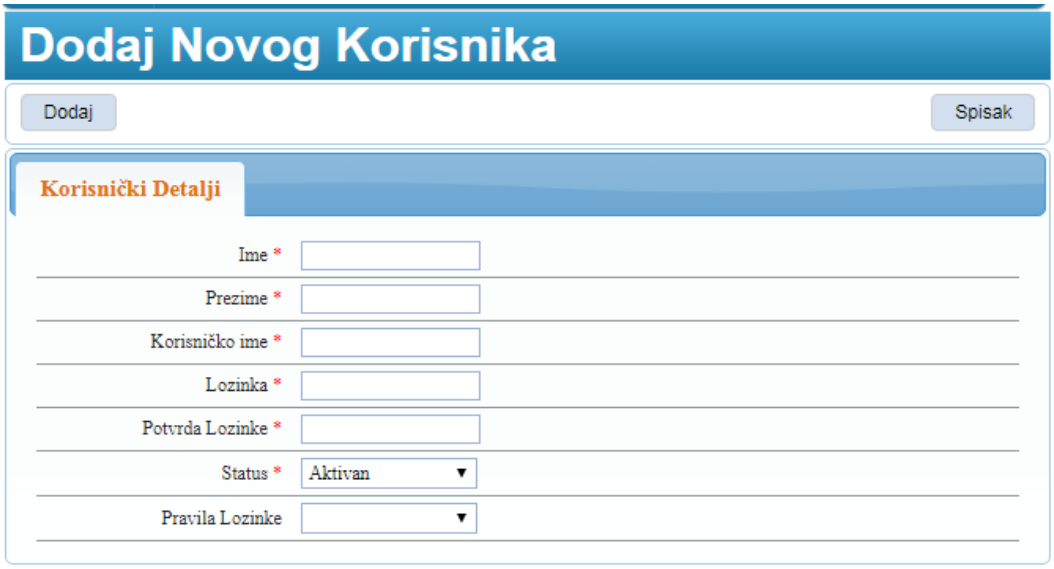
slika 12 - Dodeljivanje uloge kornisniku

Dodat je ovaj korisnik i korišćenjem njegovog korisničkog imena i lozinke je izvršeno prijavljivanje na aplikaciju. Nakon prijavljivanja dolazi se na početnu stranicu (slika 13).

slika 13 - Početna stranica korisnika

U gornjem desnom uglu ove stranice se može primetiti nova informacija koja pokazuje kom učesniku pripada trenutno prijavljeni korisnik. Druga bitna stvar koja se primećuje jeste da se u navigaciji vide samo stavke korisnik i administracija. Dakle, s obzirom da korisnik ima ulogu kojoj nisu dodeljene funkcionalnosti uloga i pravila lozinke, on te stavke u navigaciji ne vidi. Pored toga, on ni na koji način ne može pristupiti tim funcionalnostima.

Objašnjeno je kako izgleda stranica za dodavanje novog korisnika iz ugla administratora, a u delu koji sledi opisuje se kako izgleda stranica za dodavanje novog korisnika iz ugla običnog korisnika (slika 14).



slika 14 - Dodavanje novog korisnika iz ugla običnog korisnika

Sada će biti upoređena ova stranica sa istom stranicom iz ugla administrator korisnika (slika 8). S obzirom da običan korisnik može dodavati samo druge obične korisnike, na formularu ne postoji izbor tipa korisnika. Takođe, ne postoji ni izbor učesnika, jer svaki korisnik, kog on bude dodao, pripadaće istom učesniku kao i on. Još jedna stvar koja se može primetiti jeste da ne postoji sekcija za dodeljivanje korisničkih uloga. Ona je sakrivena zato što trenutno prijavljeni korisnik nema pristup funkcionalnosti uloga. Ono što je bitno naglasiti jeste da stranica za dodavanje novog korisnika iz ugla administrator korisnika i iz ugla običnog korisnika je zapravo jedna ista stranica, odnosno to je jedan JSP fajl u projektu. Nije potrebno praviti posebne stranice sa svakog tipa korisnika, već se jedna stranica dinamički prilagođava tipu prijavljenog korisnika. Kako je ovo omogućeno biće objašnjeno u delu o bezbednosti.

Izmena lozinke je funkcionalnost kojoj mogu pristupiti oba tipa korisnika. Ona pruža mogućnost izmene lozinke. Ukoliko su korisniku dodeljena pravila lozinke, on prilikom promene mora izabrati novu lozinku koja je u skladu sa tim pravilima.

Pregled izmena objekata i pregled aktivnosti aplikacije su funkcionalnosti kojima može pristupiti samo administrator korisnik. Na ovim stranicama administrator ima pregled svih pristupa aplikaciji, kao i istorijat izmene svih podataka unutar aplikacije. Svaki pristup aplikaciji, kao i svaki pokušaj izmene podataka, se čuva u bazi podataka. Ove informacije su bitne administratorima aplikacije, jer im pomažu u održavanju aplikacije. Ukoliko je došlo do neke nepredviđene greške prilikom rada aplikacije, administratori imaju sve podatke koji im kažu kako je do greške došlo, što im u mnogome pomaže u otklanjanju greške. Takođe, ove informacije omogućavaju administratorima aplikacije da opravdaju faktor ljudske greške. Recimo da se korisnik aplikacije žali da aplikacija ne funkcioniše ispravno i da je zbog toga izgubio neke važne podatke. Administratori pomoću spiska aktivnosti i izmena mogu videti način na koji je taj korisnik koristio aplikaciju i pokazati mu da je do greške došlo usled njegove greške, a ne greške u aplikaciji. Takođe, ove funkcionalnosti nam pomažu u otkrivanju napada na aplikaciju.

3.2 Arhitektura

Za potrebe ovog master rada izrađena je Veb aplikacija korišćenjem tehnologije opisane u prethodnom poglavlju. Dakle, korišćeno je razvojno okruženje *Spring* i *Java* programski jezik. Osnovu aplikacije čini *Spring Web MVC* komponenta koja obezbeđuje Model-Pogled-Kontroler (eng. Model-View-Controller - MVC) arhitekturu. Ona pruža gotove komponente koje služe da se razvije fleksibilna Veb aplikacija. Omogućava da se razdvoje različiti aspekti aplikacije (podatke, poslovnu logiku i grafički prikaz), a da pri tom ne postoji velika zavisnost između ovih elemenata. Modelom su obuhvaćeni aplikacioni podaci i uglavnom su to *Java* klase. Pogled je zadužen za obradu podataka iz modela. On generiše HTML koji se vraća klijentskom pregledaču. Kontroler je

zadužen za obradu klijentskih zahteva i generisanje modela podataka, koje šalje pogledu na dalju obradu.

Osnovu *Spring MVC* komponente čini klasa *DispatcherServlet* koja upravlja svim HTTP zahtevima (eng. request) i odgovorima (eng. response). Nakon što primi zahtev od klijenta, on konsultuje *HandlerMapping* kako bi pozvao odgovarajući kontroler. Kontroler preuzima zahtev i poziva odgovarajuću metodu. Metoda, u zavisnosti od poslovne logike, priprema model i vraća *DispatcherServlet*-u ime pogleda. *DispatcherServlet* pomoću *ViewResolver*-a pronalazi odgovarajući pogled na osnovu imena. *DispatcherServlet* prosleđuje model pogledu koji, na osnovu modela, generiše sadržaj i vraća rezultat pretraživaču.

Unutar konfiguracionog fajla **web.xml** mora se definisati koje zahteve će *DispatcherServlet* obrađivati. U aplikaciji se obrađuju svi zahtevi kojima URL počinje sa */app/*, i konfiguracija izgleda ovako:

```
<servlet>
	<servlet-name>dispatcherServlet</servlet-name>
	<servlet-class>
		org.springframework.web.servlet.DispatcherServlet
	</servlet-class>
	<init-param>
		<param-name>contextConfigLocation</param-name>
		<param-value/>
	</init-param>
	<load-on-startup>1</load-on-startup>
</servlet>
<servlet-mapping>
	<servlet-name>dispatcherServlet</servlet-name>
	<url-pattern>/app/*</url-pattern>
</servlet-mapping>
<context-param>
	<param-name>contextConfigLocation</param-name>
	<param-value>/WEB-INF/conf/spring/*.xml</param-value>
</context-param>
```

Parametar *contextConfigLocation* predstavlja lokaciju ostalih *Spring* konfiguracionih fajlova. Kao što je rečeno, *Spring* se umesto programera bavi arhitekturom aplikacije. Ono što je neophodno jeste da se definiše koje klase će činiti tu arhitekturu. U aplikaciji to je urađeno pomoću *Java* anotacija. Anotacije potrebne za *Spring MVC* arhitekturu su *@Component*, odnosno specijalni podslučajevi ove anotacije:

- *@Controller* - definiše klase koje će se baviti obradom zahteva
- *@Service* - definiše klase koje će se baviti poslovnom logikom
- *@Repository* - definiše klase koje će se baviti operacijama nad bazom

Još je potrebno definisati da aplikacija koristi prepoznavanje *Spring* klasa na osnovu anotacija, i u okviru kog paketa će se te klase nalaziti. To se definiše u konfiguracionom fajlu **spring-app-config.xml**:

```
<context:annotation-config />
<context:component-scan base-package="rs.milanmitic.master"/>
```

Dakle, prilikom startovanja aplikacije, *Spring* će u navedenom paketu pronaći klase sa navedenim anotacijama i od njih napraviti objekte. [5]

Druga stvar koju *Spring* radi umesto programera jeste ubrizgavanje zavisnosti. Ono što je potrebno jeste da se definiše koji objekti će međusobno zavisiti jedan od drugog. To se radi pomoću anotacije *@Autowired*. Ovo će biti objašnjeno na primeru. U aplikaciji je definisana klasa *CommonController* koja predstavlja komponentu *@Controller* i koja zavisi od klase *NomenclatureService*:

```
@Controller
public class CommonController extends BasicController {

	@Autowired
	private NomenclatureService nomenclatureService;
```

Zatim je definisana klasa *NomenclatureServiceImpl,* kao komponenta *@Service* koja implementira interfejs *NomenclatureService*. Još je definisano da ova klasa zavisi od klase *NomenclatureDao*.

```
@Service
public class NomenclatureServiceImpl extends MasterServiceImpl
implements NomenclatureService {

	@Autowired
	private NomenclatureDao nomenclatureDao;
```

Klasa *NomenclatureHibernateDao* implementira interfejs *NomenclatureDao* i definisana je kao komponenta *@Repository.*

```
@Repository
public class NomenclatureHibernateDao extends BasicDaoHibernate
implements NomenclatureDao {
```

Ovime je definisano da se klasa *CommonController* bavi obradom zahteva. Ukoliko je zahtev takav da zahteva obradu poslovne logike, time će se baviti klasa *NomenclatureServiceImpl*. Ukoliko poslovna logika zahteva operacije nad bazom podataka, time će se baviti *NomenclatureHibernateDao* klasa. Dakle, definisano je koje klase zavise jedna od druge a *Spring* će automatski obezbediti tu zavisnost.

Osim anotacija, postoje i drugi načini da se definišu *Spring* komponente i zavisnost među njima. To su:

- unutar XML fajlova
- pomoću specijalne klase koja sadrži anotaciju *@Configuration*

Još jedan konfiguracioni fajl, koji je bitan za arhitektutu aplikacije, je **spring-mvc-core-config.xml**. Neke bitnija podešavanja unutar ovog fajla su:

```
<context:component-scan base-package="rs.milanmitic.master.controller"/>
<mvc:annotation-driven />
```

Ovime je definisan paket u kome se nalaze svi kontroleri unutar aplikacije i to da će oni biti obeleženi anotacijom. Iako je ovo već definisano pomoću <context:annotation-config />, ova definicija je bitna zato što pomaže *HandlerMapping*-u da brže pronađe odgovarajući kontroler za klijentske zahteve.

Drugo bitno podešavanje unutrar ovog konfiguracionog fajla je:

```
<bean id="viewResolver"
class="org.springframework.web.servlet.view.tiles3.TilesViewResolver">
	<property name="redirectHttp10Compatible" value="false" />
</bean>

<bean id="tilesConfigurer"
class="org.springframework.web.servlet.view.tiles3.TilesConfigurer">
	<property name="definitions">
		<list>
			<value>/WEB-INF/views/views.xml</value>
		</list>
	</property>
</bean>
```

Ovim delom koda je definisano koji *ViewResolver* će se koristiti u aplikaciji i lokaciju XML fajla gde se nalazi spisak pogleda korišćenih u aplikaciji.

Kada korisnik pošalje zahtev za početnom stranicom aplikacije, *HandlerMapping* skenira sve klase u paketu *rs.milanmitic.master.controller* koje imaju anotaciju *@Controller* i pokušava da pronađe metod koji ima anotaciju *@RequestMapping* čija vrednost odgovara zahtevanom URL-u i pronalazi metod unutar *HomeController* klase:

```
@RequestMapping(value = "/home", method = RequestMethod.GET)
public String home(HttpServletRequest request, Model model) {
	...

	return "home";
}
```

Ovaj metod vraća pogled sa imenom "home". Sada *TilesViewResolver* pokušava u fajlu **views.xml** da pronađe koja stranica odgovara pogledu sa tim imenom, i pronalazi:

```
<definition name="home" extends="homeLayout">
	<put-attribute name="body" value="/WEB-INF/views/home.jsp" />
</definition>
```

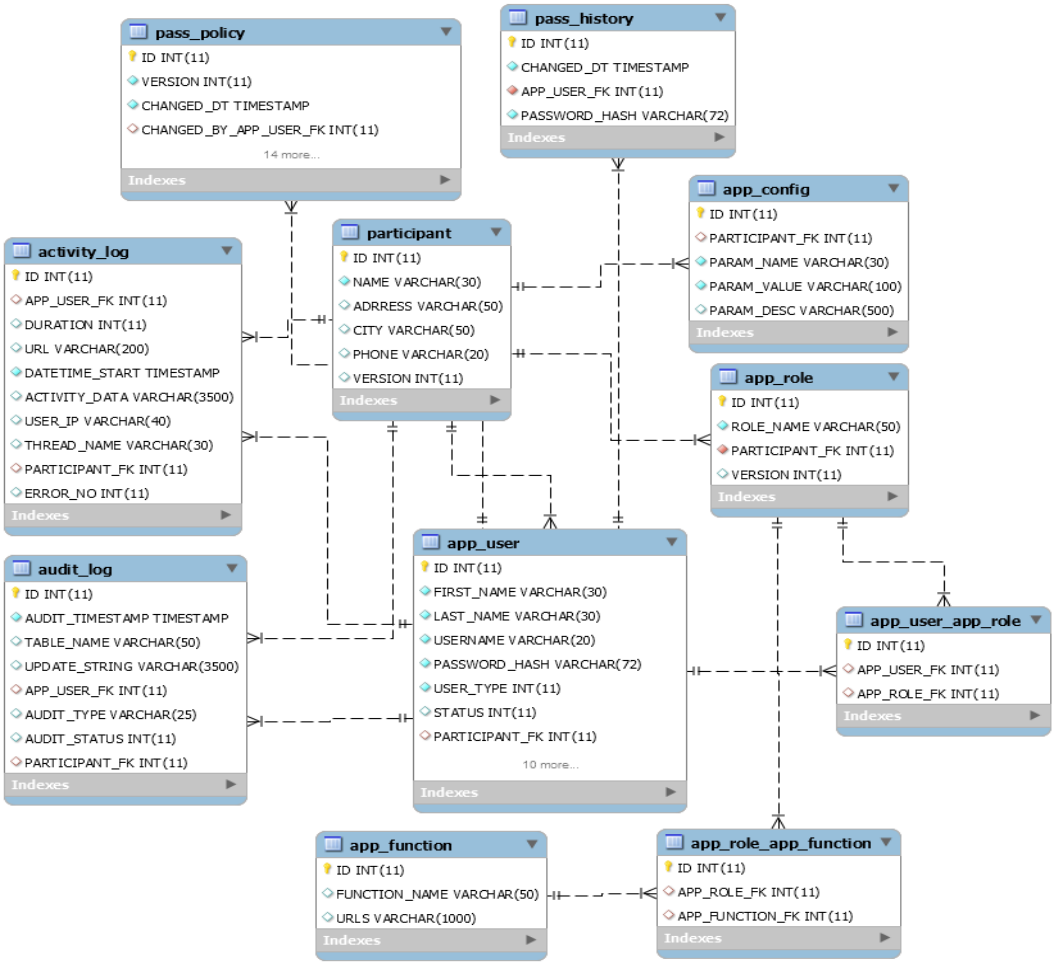
Dakle, korisniku će biti vraćena stranica home.jsp.

3.3 Baza podataka

Aplikacija koristi MySQL bazu podataka sa imenom Master. Tabele koje sadrži baza podataka su:

PARTICIPANT	- tabela koja sadrži učesnike
APP_USER	- tabela koja sadrži korisnike oba tipa
APP_ROLE	- tabela koja sadrži korisničke uloge
APP_FUNCTION	- tabela koja sadrži funkcionalnosti koje mogu biti dodeljene korisnicima
APP_ROLE_APP_FUNCTION	- tabela koja spaja uloge sa funcionalnostima
APP_USER_APP_ROLE	- tabela koja spaja korisnike sa ulogama
PASS_POLICY	- tabela koja sadrži pravila lozinke
PASS_HISTORY	- tabela koja sadrži istorijat lozinke
ACTIVITY_LOG	- tabela koja sadrži sve pristupe aplikaciji
AUDIT_LOG	- tabela koja sadrži sve izmene objekata u aplikaciji
APP_CONFIG	- tabela koja sadrži konfiguraciju aplikacije

Na EER dijagramu baze podataka (slika 15) se vide zavisnosti među tabelama.



slika 15 - EER dijagram baze podataka

Sada će biti objašnjeno kako konfigurisati aplikaciju da koristi bazu podataka i Hibernate. Prvi bitan konfuguracioni fajl je **data-access.properties**. On služi da se navede koji tip baze podataka će se koristiti. U aplikaciji to je MySQL:

```
jdbc.driverClassName=com.mysql.jdbc.Driver
jdbc.datasource=jdbc/masterDatasource
hibernate.dialect=org.hibernate.dialect.MySQLDialect
```

Drugi potreban konfiguracioni fajl je **context.xml**. Unutar njega se definišu parametri za konekciju za gore navedeni *jdbc/masterDatasource*:

```
<Resource name="jdbc/masterDatasource" auth="Container"
      type="javax.sql.DataSource" maxActive="100" maxIdle="30"
      maxWait="10000" username="root" password="root"
      driverClassName="com.mysql.jdbc.Driver"
      url="jdbc:mysql://localhost/master" />
```

Sledeće podešavanje je unutar **spring-app-config.xml** fajla:

```
<context:property-placeholder location="/WEB-INF/conf/spring/data-
access.properties" system-properties-mode="OVERRIDE"/>

<bean id="sessionFactory"
class="org.springframework.orm.hibernate4.LocalSessionFactoryBean">
      <property name="dataSource" ref="dataSource" />
      <property name="packagesToScan"
      value="rs.milanmitic.master.model" />
      <property name="hibernateProperties">
            <props>
                  <prop key="hibernate.dialect">
                        ${hibernate.dialect}
                  </prop>
                  <prop key="hibernate.show_sql">false</prop>
                  <prop key="hibernate.jdbc.batch_size">500</prop>
            </props>
      </property>
</bean>
```

Ovime je definisano da *Hibernate* koristi tip baze podataka koji je naveden u fajlu **data-access.properties**. Takođe je definisano ime paketa u kome će se nalaziti klase koje predstavljaju Hibernate objekte. To je paket *rs.milanmitic.master.model*.

Još jedna bitna definicija unutar ovog fajla je:

```
<tx:annotation-driven transaction-manager="transactionManager" />
```

Ovime je definisano da će transakcione klase biti obeležene anotacijom. Anotacija koja služi za obeležavanje transakcionih klasa je *@Transactional*. Šta predstavljaju transakcione klase? Ukoliko se pozove neka motoda unutar transakcione klase koja zahteva više operacija nad bazom podataka, to što je klasa

obeležena kao transakciona omogućava da se izvrše ili sve operacije nad bazom, ili nijedna. Na primer, ako se pozove metoda koja zahteva tri različita upisa u bazu podataka. Prva dva upisa se izvrše uspešno, ali prilikom trećeg upisa dođe do nekog oblika greške. U tom slučaju doći će do povraćaja (eng. rollback) prva dva upisa, i baza podataka će ostati nepromenjena. Ovo omogućava očuvanje logike unutar aplikacije, u smislu da neke funkcionalnosti ne budu završene samo delom, i da dođe do nekonzistentnosti podataka u bazi.

Ostalo je još jedno podešavanje u fajlu **spring-datasource-config.xml**:

```
<context:property-placeholder location="/WEB-INF/conf/spring/data-
access.properties" system-properties-mode="OVERRIDE"/>

<jee:jndi-lookup id="dataSource" jndi-name="${jdbc.datasource}"/>
```

gde je definisano koji će se parametri koristiti za konekciju na bazu podataka.

Kako definisati Hibernate objekte u projektu? Kao što je spomenuto, *Spring* će skenirati sve klase unutar paketa *rs.milanmitic.master.model* i tražiti one sa odgovarajućom anotacijom kako bi ih pretvorio u Hibernate objekte. Ta anotacija je *@Entity*. Tako je u aplikaciji klasa *Participant* koja predstavlja učesnike definisana kao:

```
@Entity(name = "Participant")
@Table(name = "PARTICIPANT")
public class Participant
```

Anotacijom: `@Table(name = "PARTICIPANT")` je definisano da naša klasa odgovara tabeli baze podataka sa imenom PARTICIPANT.

Predstavljanje aributa koji odgovara koloni koja je primarni ključ tabele je definisano na sledeći način:

```
@Id
@Column(name = "ID")
@GeneratedValue(strategy = GenerationType.AUTO)
private Long id;
```

Ovim je definisano da će atribut ove klase *id* odgovarati koloni tabele sa imenom *ID* koja je primarni ključ tabele i koja je auto-inkrementirajuća.

Definisanje kolone koja nije primarni ključ:

```
@Column(name = "NAME", nullable = true, unique = false)
@Length(max = 30)
@NotEmpty
private String name;
```

Ovim je definisan atribut *name* koji odgovara koloni *NAME* tabele *PARTICIPANT*, čija je dužina maksimalno 30 karaktera i koja je *NOT NULL*.

4. Bezbednost

4.1 Spring Security

Spring Security je okvir koji se bavi bezbednosnim delom aplikacija baziranih na *Spring*-u. On pruža mnoga bezbednosna rešenja, obezbeđuje podršku za autentikaciju i autorizaciju, kako na nivou klijentskog zahteva, tako i na nivou metoda unutar klasa.

Spring Security se prvobitno nazivao *Acegi Security*. Iako je *Acegi* imao veliku podršku za bezbednost, on je imao jedan veliki nedostatak, a to je da je zahtevao mnogo linija koda unutar XML konfiguracionih fajlova. Počevši od verzije 2.0, *Acegi Security* postaje *Spring Security*. Osim promene imena, Spring Security 2.0 je omogućavao da se sa mnogo manje linija XML kôda, uz podršku anotacija, konfiguriše bezbednost u *Spring*-u. U najnovijim verizijama *Spring Security*-a, bezbednost je obezbeđena na dva načina, filtriranjem Veb zahteva i obezbeđivanjem *Java* metoda.

4.1.1 Konfiguracija Spring Security-a

Konfiguracija *Spring Security*-a se vrši u **spring-security.xml** fajlu. Minimalna potrebna konfiguracija se sastoji iz sledećeg [6]:

```
<security:http use-expressions="true" access-denied-
page="/error403.jsp">

<security:intercept-url pattern="/app/**"
access="hasAnyRole('ROLE_ADMIN', 'ROLE_USER')"/>
```

U prvoj liniji koda omogućen je *Spring Security* za sve HTTP zahteve i definisana je stranica na koju se preusmeravaju korisnici ako je narušena bezbednost. U dugoj liniji definisano je da delovima aplikacije čiji URL počinje sa */app* mogu pristupiti samo prijavljeni korisnici. ROLE_ADMIN odgovara korisnicima Administrator tipa, dok ROLE_USER odgovara običnim korisnicima. Ovo je definisano na nivou aplikacije kao enumeracija:

```
public enum UserRole {
	ROLE_ADMIN("ROLE_ADMIN"), ROLE_USER("ROLE_USER");
```

Ovo su uloge koje *Spring* koristi da razdvoji tipove korisnika i ove uloge nisu povezane sa entitetom uloga u kontekstu logike aplikacije.

S obzirom da postoje delovi aplikacije kojima moraju da pristupe i neprijavljeni korisnici, kao na primer stranica za prijavljivanje, moraju se definisati ovi izuzeci:

```
<security:intercept-url  pattern="/app/login/Admin"
access="permitAll" />
<security:intercept-url  pattern="/app/login/User" access="permitAll" />
```

```
<security:intercept-url  pattern="/app/changeLanguage/**"
access="permitAll" method="GET" />
```

U aplikaciji to su stranice za prijavljivanje i URL koji se koristi za promenu jezika na stranici za prijavljivanje.
Zatim je obezbeđeno da *JavaScript* i *CSS* fajlovima mogu pristupiti samo prijavljeni korisnici:

```
<security:intercept-url pattern="/themes/**"
access="hasRole('ROLE_ADMIN', 'ROLE_USER')"  method="GET" />
<security:intercept-url pattern="/js/**" access="hasRole('ROLE_ADMIN',
'ROLE_USER')"  method="GET" />
```

4.2 Autentifikacija

Spomenuto je da je filtritanje zahteva jedan od osnovnih načina na koji *Spring Security* osigurava bezbednost. Jedan od takvih filtera je filter za autentifikaciju korisnika. Podrazumevani filter u *Spring*-u za autentifikaciju pomoću korisničkog imena i lozinke je *UsernamePasswordAuthenticationFilter*. Da bi on bio potpuno funkcionalan potrebno je samo malo konfiguracuje unutar **spring-security.xml** fajla:

```
<security:form-login login-page="/app/login"
username-parameter="username"
password-parameter="password"
login-processing-url="/app/login-process"
default-target-url="/app/home"
authentication-failure-url="/app/login?error=1"  />
```

Ovime je definisano sledeće:

- `login-page="/app/login"` - lokacija stranice za prijavljivanje
- `username-parameter="username"` - polje korisničkog imena
- `password-parameter="password"` - polje lozinke
- `login-processing-url="/app/login-process"` - akcija za prijavljivanje
- `default-target-url="/app/home"` - lokacija početne stranice
- `authentication-failure-url="/app/login?error=1"` - lokacija stranice u slučaju greške

Potrebno je definisati akciju odjavljivanja:

```
<security:logout invalidate-session="true" logout-url="/app/logout"
logout-success-url="/app/login"/>
```

Sledeća stavka je definisanje akcija nad bazom prilikom prijavljivanja:

```
<authentication-manager>
    <authentication-provider>
        <jdbc-user-service data-source-ref="dataSource"
        users-by-username-query=
        "select username, password, status from app_user where
         username=?"
         authorities-by-username-query=
         "select username, user_type from app_user where username =?" />
    </authentication-provider>
</authentication-manager>
```

Na kraju, potrebno je definisati stranicu za prijavljivanje koja će saržati formular koji ima polja za korisničko ime i lozinku koja su definisana sa *username-parameter* i *password-parameter*. Akcija formulara mora biti ona koja je definisana u *login-processing-url*.

Najjednostavniji oblik takvog formulara bi bio:

```
<form action="<c:url value="/app/login-process"/>" method="post" >
       <label for="username">Username</label>
       <input type="text" name="username" /><br/>
       <label for="password">Password</label>
       <input type="password" name="password" /><br/>
</form>
```

Ovo je sve što je potrebno kako bi se obezbedila potpuno funkcionalna stranica za prijavljivanje korisnika.

Međutim, osim autentifikacije korisnika postoje još neke akcije koje se moraju izvršiti u aplikaciji prilikom prijavljivanja korisnika. Prva stvar je da prilikom prijavljivanja običnog korisnika treba iz baze podataka pročitati uloge i funkcionalnosti koje su mu dodeljene. Druga stvar je da treba proveriti da li su korisniku dodeljena pravila lozinke. Ukoliko jesu, mora se proveriti da li je korisnik blokiran, da li je lozinka istekla, zatim ukoliko je pogrešio lozinku da mu se uveća broj neuspelih pokušaja i blokira se ukoliko je dostigao maksimalan broj neuspelih pokušaja. Zbog ovih dodatnih aktivnosti ne može se koristiti obezbeđena *Spring*-ova podrška za autentifikaciju, već se mora definisati sopstvena. To se radi tako što se re-definiše *Spring*-ova podrška. Prvo je potrebno napraviti *Spring* objekat koji će predstavljati novi filter za autentifikaciju. U apikaciji je nazvan *MasterAuthenticationFilter*.

```
@Component(value = "masterAuthenticationFilter")
public class MasterAuthenticationFilter extends
AbstractAuthenticationProcessingFilter
```

Klasa mora da nasleđuje neki postojeći *Spring*-ov filter. U aplikaciji to je *AbstractAuthenticationProcessingFilter*. U **spring-security.xml** fajlu mora se navesti da će se novi filter koristiti umesto podrazumevanog *Spring*-ovog filtera i to na sledeći način:

```
<security:custom-filter ref="masterAuthenticationFilter"
before="FORM_LOGIN_FILTER"/>
```

FORM_LOGIN_FILTER predstavlja *Spring*-ov filter za autentifikaciju, a ovime je navedeno da *masterAuthenticationFilter* filter ima prioritet u odnosu na njega. Na ovaj način je definisan filter koji će primati zahteve za autentifikaciju. Sledeće što je potrebno definisati jeste klasa koja će te zahteve obrađivati. U podrazumevanom procesu autentifikacije to je klasa *AuthenticationProvider*. Dakle, potrebno je napraviti klasu koja će nasleđivati ovu klasu ili neku njenu potklasu. U aplikaciji to je klasa *DaoAuthenticationProvider* koja omogućava korišćenje servisa i konektovanja na bazu podataka. U aplikaciji ta klasa je nazvana *MasterAuthenticationProvider*:

```
@Service("masterAuthenticationProvider")
@Transactional
public class MasterAuthenticationProvider extends
DaoAuthenticationProvider
```

U **spring-security.xml** fajlu se navodi ova klasa kao podrazumevana u procesu autorizacije:

```
<security:authentication-manager>
 <security:authentication-provider ref="masterAuthenticationProvider" />
</security:authentication-manager>
```

Zatim je u konstruktoru filtera potrebno povezati ga sa ovom klasom:

```
@Autowired
public MasterAuthenticationFilter(
@Qualifier("masterAuthenticationProvider") AuthenticationProvider
authenticationProvider) {
	super("/app/login-process");
	List<AuthenticationProvider> authenticationProviderList = new
			ArrayList<AuthenticationProvider>();
	authenticationProviderList.add(authenticationProvider);

	AuthenticationManager authenticationManager = new
			ProviderManager(authenticationProviderList);
	super.setAuthenticationManager(authenticationManager);

	AuthenticationSuccessHandler a = new
			SimpleUrlAuthenticationSuccessHandler("/app/home");
	super.setAuthenticationSuccessHandler(a);
}
```

Sledeća je potrebno definisati objekat koji će prenositi podatke od filtera do servisa i nazad. U *Spring*-u se ti objekti nazivaju tokeni. Token koji *Spring* koristi za prenos korisničkog imena i lozinke je *UsernamePasswordAuthenticationToken*. U aplikaciji je potrebno preneti više podataka. Od filtera do servisa potrebno je preneti i informaciju o tipu korisniku, a zatim ako autentifikacija bude uspešna, potrebno je preneti informacije o korisniku od servisa nazad do filtera. Zato je potrebno definisati poseban token koji će se koristiti i koji će nasleđivati *Spring*-ov *UsernamePasswordAuthenticationToken*. Taj token je u aplikaciji nazvan *MasterAuthenticationToken*.

```
public class MasterAuthenticationToken extends
UsernamePasswordAuthenticationToken
```

Kada je sve definisano, može se videti kako izgleda proces autentifikacije. Nakon što korisnik popuni formular za prijavljivanje i pošalje zahtev, *Spring* na osnovu konfiguracije prepoznaje definisan filter za autentifikaciju. Zahtev dolazi do filtera. Filter prepoznaje token i servis koji se koristi. Priprema token i prosleđuje servisu.

```
MasterAuthenticationToken  authRequest =
new MasterAuthenticationToken(username, pswdVal, UserType.getById(loginType));
Authentication authentication =
this.getAuthenticationManager().authenticate(authRequest);
```

Servis na osnovu tipa korisnika proverava u bazi podataka korisničko ime i lozinku, zatim pravila lozinke i ostale potrebne provere. Ukoliko je sve u redu, priprema token kao odgovor koji vraća filteru.

```
Set<GrantedAuthority> setAuths = new HashSet<GrantedAuthority>();
setAuths.add(new SimpleGrantedAuthority(UserRole.ROLE_USER.getId()));

if (UserType.ADMIN.equals(loginBean.getUserType())) {
        loginBean = autentificationService.adminLogin(authentication);
        setAuths.add(new SimpleGrantedAuthority(UserRole.ROLE_ADMIN.getId()));
} else if (UserType.USER.equals(loginBean.getUserType())) {
        loginBean = autentificationService.userLogin(authentication);
        setAuths.add(new SimpleGrantedAuthority(UserRole.ROLE_USER.getId()));
}

User user = new User(loginBean.getPrincipal().toString(),
loginBean.getCredentials().toString(), true, true, true, true, setAuths);

Authentication a = createSuccessAuthentication(loginBean.getPrincipal(),
authentication, user);
```

Bitna stavka u ovom delu koda jeste kreiranje objekta klase *User*. Ovo je *Spring*-ova klasa iz paketa *org.springframework.security.core.userdetails*, čijim se kreiranjem označava uspešna autentifikacija korisnika. Druga bitna stvar jeste dodeljivanje uloga ovom objektu. U zavisnosti od tipa, korisnik će imati ulogu ***ROLE_ADMIN*** ili ***ROLE_USER***. Ove uloge su bitne zato što su to uloge na koje se referencira prilikom definisanja pristupa u konfiguracionom fajlu **spring-security.xml**. Takođe, ovo su uloge pomoću kojih su definisana prava pristupa na svim nivoima u aplikaciji, što će biti objašnjeno malo kasnije.

Lozinka se u bazi podataka čuva u kriptovanom obliku. *Spring* pruža kvalitetnu podršku za kriptovanje lozinke pomoću interfejsa *PasswordEncoder* iz paketa *org.springframework.security.crypto.password*. Postoje *Spring*-ove klase koje predstavljaju implementaciju ovog interfejsa i to su *AbstractPasswordEncoder*, *BCryptPasswordEncoder*, *NoOpPasswordEncoder*, *Pbkdf2PasswordEncoder*, *SCryptPasswordEncoder* i *StandardPasswordEncoder*. *Spring* preporučuje korišćenje *BCryptPasswordEncoder* implementacije koja koristi jaku ***BCrypt*** funkciju za heširanje.

BCrypt je funkcija koja su 1999. dizajnirali Niels Provos i David Mazieres. Zasnovana je na *Blowfish* funkciji. Funkcija se izvršava tako što se heširanje izvršava više puta unutar petlje. Osnovni napadi na ovakve tipove algoritma su:

1. Kreiranje tabele svih mogućih lozinki. Sastoji se u tome da se kreira tabela sa svim mogućim lozinkama koje odgovaraju svim heš vrednostima nekog algoritma. Jednom kada napadač na sistem dođe u posed podataka iz baze podataka, on na osnovu heš vrednosti iz baze i pomenute tabele može otkriti lozinke korisnika i iskoristiti ih da ugrozi sistem.

2. Nagađanje lozinke. Napadač na sistem isprobava sve moguće lozinke.

BCrypt se bori protiv ovakvih napada tako što je dizajniran da bude spor. Napravljen je tako da bude dovoljno brz za korisnike, ali spor za napadače na sistem. Funkciji treba oko 100 milisekundi da se izvrši, što je oko 10 hiljada puta sporije od algoritma *SHA1*. 100 milisekundi je zanemarljivo malo vreme koje korisnik treba da sačeka da bi se prijavio na sistem, ali je dovoljno sporo da onemogući napadačima da koriste gore pomenute napade na sistem. Još jedna prednost ove funkcije jeste da je broj iteracija heširanja unutar funkcije promenljiv parametar. To znači da ukoliko bi vremenom nastali neki super kompjuteri koji mogu brže da napadnu sistem, jednostavim povećanjem broja iteracija funkcije bi se ovakvi napadi neutralisali. U *Spring*-ovoj implementaciji funkcije podrazumevani broj iteracija je 10. [7].

Da bise omogućila ova funkcija u *Spring*-u, potrebno je napraviti objekat koji nasleđuje klasu *BCryptPasswordEncoder*.

```
@Component(value = "masterUserPasswordEncoder")
public class MasterUserPasswordEncoder extends BCryptPasswordEncoder
```

Dve funkcije koje su potrebne su:

- `public String encode(CharSequence arg0)` - funkcija za kriptovanje lozinke. Ona se koristi prilikom dodavanja novog korisnika kako bi se u bazi snimila lozinka u kriptovanom obliku.

- `public boolean matches(CharSequence rawPassword, String encodedPassword)`- funkcija koja poredi lozinku sa njenom kriptovanom vrednošću. Koristi se prilikom prijavljivanja korisnika kako bi se proverilo da li je korisnik uneo tačnu lozinku.

4.3 Autorizacija

Nakon uspešne autentifikacije i prijavljivanja korisnika na sistem, potrebno je omogućiti da korisnik može pristupiti samo onim funcionalnostima za koje mu je odobren pristup. U prethodnim odeljcima je objašnjeno da se autorizacija korisnika vrši na osnovu uloga dodeljenih *Spring* objektu *User* koji se kreira nakon uspešne autentifikacije. Uloge koje je moguće dodeliti su:

- ROLE_ADMIN - administrator korisnici
- ROLE_USER - obični korisnici

Spring omogućava autorizaciju na 3 nivoa:

1. Na nivou URL-a
2. Na nivou metode
3. Na nivou JSP stranice

4.3.1 Autorizacija na nivou URL-a

Autorizaciju na nivou URL-a je moguće obezbediti na dva načina:

- U konfiguracionom fajlu
- U *Spring*-ovom filteru

Deo o autorizaciji u konfiguracionom fajlu je objašnjen u delu o konfiguraciji *Spring*-a, kada je pokazano kako definisati URL-ove kojima mogu pristupiti neprijavljeni korisnici. Takođe je pokazano kako definisati da funkcionalnim delovima aplikacije mogu pristupiti samo prijavljeni korisnici. Sada će biti objašnjeno kako definisati URL-ove kojima mogu pristipiti samo prijavljeni korisnici određenog tipa.

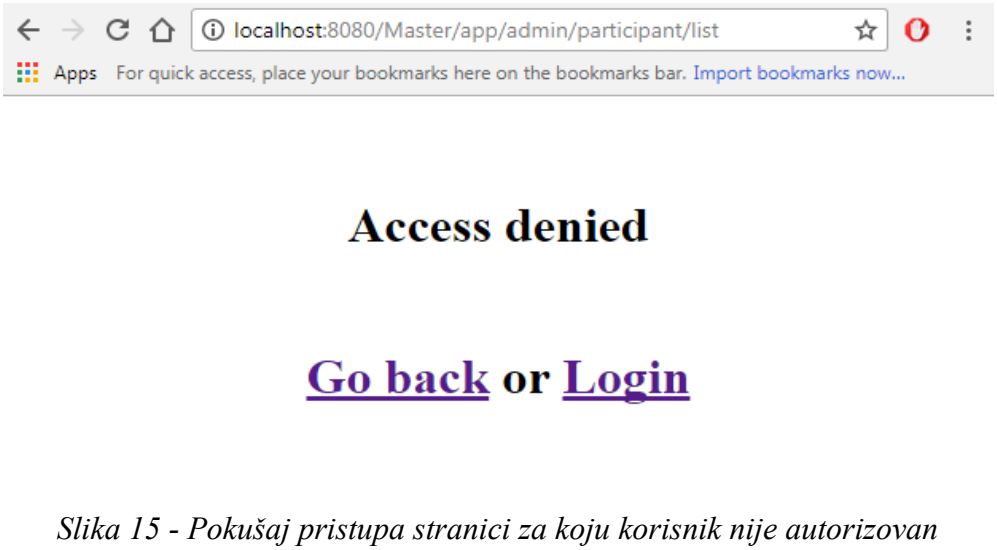
Logika aplikacije je takva da funkcionalnostima o učesniku mogu pristupati samo administrator korisnici. Kako to omogućiti? Prvo je definisano da URL-ovi svih zahteva koji se odnose na funkcionalnosti o učesniku počinju sa */admin*. S obzirom da se sva mapiranja za ove zahteve nalaze u klasi *ParticipantController*, to se može uraditi na sledeći način:

```
@Controller
@RequestMapping(value = "/admin/participant")
public class ParticipantController extends BasicController
```

Zatim u konfiguracionom fajlu **spring-security.xml se** definiše da svim zahtevima koji počinju sa */admin* mogu pristupiti samo administrator korisnici:

```
<security:intercept-url pattern="/app/admin/**"
access="hasRole('ROLE_ADMIN')" />
```

Spring će uraditi sve ostalo. Dakle, ako bi se na aplikaciju prijavio običan korisnik i pokušao da prisupi URL-u koji odgovara stranici za pregled spiska učesnika, *Spring* ne bi omogućio pristup i preusmerio bi korisnika na stranicu za narušenu bezbednost (slika 15).



Slika 15 - Pokušaj pristupa stranici za koju korisnik nije autorizovan

Sada će biti objašnjeno kako obezbediti autorizaciju preko *Spring*-ovog filtera. Već je objašnjeno kako se unutar aplikacije definiše kojim funkcionalnostima ima pristup običan korisnik. U bazi podataka se čuva spisak svih funkcionalnosti koje su predviđene za obične korisnike. Svaka funkcionalnost sadrži spisak svih URL-ova koji odgovaraju toj funkcionalnosti. U aplikaciji se definiše entitet uloga. Svakoj ulozi je moguće dodeliti jednu ili više funkcija. Zatim je konkretnom korisniku moguće dodeliti jednu ili više uloga. Nakon što se korisnik prijavi, iz baze podataka se čita spisak uloga koje su mu dodeljenje. Zatim se čita spisak funkcionalnosti koje su dodeljene tim ulogama. Od spiska funkcionalnosti se pravi spisak svih URL-ova kojima će prijavljeni korisnik imati pristup. Jedino što je preostalo je omogućiti da korisnik može pristupati samo onim URL-ovima koji su na tom spisku.

Da bi se to omogućilo, potrebni su *Servlet Filteri*. Oni omogućavaju da se presretne svaki zahtev upućen ka serveru. Dakle, ideja je da se presretne zahtev i proveri da li se URL zahteva nalazi u spisku URL-ova koji su dozvoljeni prijavljenom korisniku. Da bi se napravio *Servlet Filter* potrebno je da napraviti klasu koja implementira interfejs *javax.servlet.Filter.*

```
public class SecurityFilter implements Filter
```

Zatim je potrebno da se taj filter registruje. Unutar **web.xml** fajla je potrebno dodati:

```
<filter>
	<display-name>SecurityFilter</display-name>
	<filter-name>SecurityFilter</filter-name>
	<filter-class>rs.milanmitic.master.filter.SecurityFilter</filter-
	class>
</filter>
<filter-mapping>
	<filter-name>SecurityFilter</filter-name>
	<url-pattern>/app/*</url-pattern>
</filter-mapping>
```

U filteru je potrebno redefinisati metodu:

```
@Override
public void doFilter(ServletRequest request, ServletResponse response,
FilterChain chain)
```

Unutar ove metode potrebno je dodati proveru da korisnik ima pristup zahtevanom URL-u. To se radi tako što se iz sesije pročita spisak svih URL-ova kojima korisnik ima pristup i proveri da li je zahtevani URL na tom spisku. Ukoliko ima prava pristupa, potrebno je pozvati metod za nastavak izvršavanja zahteva:

```
chain.doFilter(request, response);
```

4.3.2 Autorizacija na nivou metode

Ovaj način autorizacije omogućava da samo određeni tipovi korisnika mogu pristupati određenim Java metodama. Ovo omogućavaju specijalne anotacije koje se dodaju na nivou metoda. Postoje tri grupa anotacija koje nam omogućavaju autorizaciju:

- *@Secured*
- *@RolesAllowed*
- *@PreAuthorize, @PostAuthorize, @PreFilter, @PostFilter*

Da bi bilo moguće koristiti ove anotacije, potrebno je omogućiti ih u **spring-security.xml** konfiguracionom fajlu:

```
<security:global-method-security secured-annotations="enabled" jsr250-
annotations="enabled" pre-post-annotations="enabled" />
```

Anotacije *@Secured* i *@RolesAllowed* obezbeđuju potpuno istu stvar. Jedina razlika između njih je to što je *@Secured* *Spring*-ova anotacija, dok je *@RolesAllowed* standardna *Java* anotacija. Obe anotacije obezbeđuju da samo određeni tipovi korisnika mogu pristupiti metodi. Koriste se na isti način, tako što kao parametar primaju spisak uloga koje imaju pristup metodi. Na primer:

```
@Secured("ROLE_ADMIN") i @RolesAllowed("ROLE_ADMIN")
```

imaju portpuno isto značenje i označavaju da samo administrator korisnici mogu pristupiti metodi. Ukoliko je kao atribut prosleđen spisak uloga:

```
@Secured({"ROLE_ADMIN", "ROLE_USER}")
```

znači da korisnik, koji pristupa metodi, mora biti ili administrator ili običan korisnik.

Kako je ovaj način autorizacije primenjen u aplikaciji? Već je spomenuto da funkcionalnostima o učesniku mogu pristupati samo administrator korisnici. To znači da i metodama koje pripadaju ovim funkcionalnostima mogu pristupiti samo administrator korisnici:

```
@Secured("ROLE_ADMIN")
SearchResults getParticipantList(Participant bean);
```

Ovime je obezbeđeno da metodi koja služi za čitanje liste učesnika mogu pristupiti samo administrator korisnici. Ukoliko bi neki drugi tip korisnika pokušao da pristupi ovoj metodi, *Spring* bi bacio izuzetak *AccessDeniedException*.

Dok anotacije *@Secured* i *@RolesAllowed* mogu samo da ograniče pristup određenim tipovima korisnika, anotacije *@PreAuthorize, @PostAuthorize, @PreFilter, @PostFilter* imaju mnogo veći izbor mogućnosti. One kao parametar mogu primiti bilo koji izraz.

- *@PreAuthorize* proverava izraz koji je prosleđen kao parametar pre izvršenja metode. Metoda će se izvršiti samo ako je taj izraz zadovoljen.
- *@PostAuthorize* proverava izraz koji je prosleđen kao parametar, ali tek nakon izvršenja metode. Obično služi da se izrazom proveri rezultat izvršene metode. Ukoliko je izraz zadovljen, metoda se izvršava normalno, Ukoliko nije, baca se izuzetak.
- *@PreFilter* omogućava da se filtriraju parametri metode. Recimo da je parametar metode lista objekata. Unutar *@PreFilter* anotacije možemo navesti izraz koji će izbaciti pojedine objekte iz liste pre nego što ih prosledi metodi.
- *@PostFilter* omogućava da se filtrira rezultat metode. Recimo da je rezultat metode lista objekata. Pomoću izraza unutar ove anotacije možemo iz te liste objekata izbaciti neke od njih i novonastalu listu vratiti kao rezultat metode.

4.3.3 Autorizacija na nivou JSP stranice

Autorizacija na nivou JSP stranice omogućava da se sadržaj stranice prikazuje drugačije u zavisnosti od tipa prijavljenog korisnika. Autorizacija se vrši pomoću specijalnih elemenata koji se koriste na JSP stranici. Da bi se omogućilo korišćenje ovih elemenata, potrebno je deklarisati biblioteku ovih elemenata na početku JSP stranice:

```
<%@ taglib prefix="sec"
uri="http://www.springframework.org/security/tags" %>
```

Najčešće korišćeni element iz ove biblioteke jeste *authorize*. On omogućava da se definišu delovi koda na JSP stranici koji će biti vidljivi samo korisnicima koji ispunjavaju određene uslove. Uslovi se definišu pomoću atributa *access* i *url*. Unutar atributa *access* se navodi izraz koji će odrediti da li je deo koda unutar elementa *authorize* vidljiv ili ne. Na primer:

```
<sec:authorize access="hasRole('ROLE_ADMIN')">
	...
</sec:authorize>
```

Ovim je definisano da će sve navedeno između otvorenog i zatvorenog elementa *authorize* biti vidljivo samo administrator korisnicima. Drugi način da se definiše uslov unutar elementa authorize jeste preko atributa *url*. Unutar ovog atributa može se navesti URL za koji korisnik mora biti autorizovan kako bi video kôd unutrar elementa. Na primer:

```
<sec:authorize url="/app/appRole/add">
	...
</sec:authorize>
```

Kôd unutrar ovog elementa biće vidljiv samo korisnicima koji su autorizovani da dodaju nove uloge.

4.4 Napadi na Veb aplikacije

Napadi na Veb aplikacije predstavljaju tehnike koje napadači koriste kako bi ugrozili bezbednost aplikacije. Najčešće cilj napada jeste ugrožavanje poverljivosti (eng. Confidentiality) i integriteta (eng. Integrity) podataka. Poverljivost podataka predstavlja sposobnost aplikacije da zaštiti podatke od neovlašćenog pristupa. Integritet podataka predstavlja sposobnost aplikacije da zaštiti podatke od neovlaščenog menjanja ili brisanja.

Napadi na Veb aplikacije se obično izvode direktno iz pregledača, najčešće na sledeća dva načina:

- menjanjem elemenata aplikacije preko grafičkog korisničkog interfejsa (eng. Graphical User Interface - GUI). Za svaki tip pregledača postoje dodaci koji omogućavaju promenu svih elemenata stranice čime se može izazvati nepredviđeno ponašanje aplikacije.
- menjanjem jedinstvenog identifikatora resorsa (eng. Uniform Resource Identifier - URI). Svaki element URI-a je moguće ručno promeniti u pregledaču i time dovesti do neočekivanog ponašanja aplikacije.

Veb aplikacije su jedan od tipova aplikacija koji se najčešće napadaju. Neki od razloga za to su sledeći:

- sveprisutnost - Veb aplikacije su danas prisutne svuda i brzo nastavljaju da se

šire putem javnih i privatnih mreža. Napadači imaju veliki izbor meta koji stalno raste.

- jednostavne tehnike - tehnike za napade na Veb aplikacije je lako razumeti, naučiti i primeniti, za razliku od tehnika za napade na neke druge tipove aplikacija
- anonimnost - Internet omogućava lak pristup aplikacijama uz malu mogućnost da napadači budu identifikovani
- prilagođeni kôd - na razvoju Veb aplikacije obično radi više programera pa je samim tim veća verovatnoća da postoji programerska greška. Dovoljno je da jedan od programera ne bude dovoljno stručan kako bi došlo do greške koja može biti meta napadača
- nedovoljno pažnje sigurnosti - prilikom razvoja mnogih Veb aplikacije ne posvećuje se dovoljno pažnje sigurnosti aplikacije
- stalna promena - Veb aplikacije se stalno menjaju i unapređuju. Kod kompleksnih aplikacija teško je odrediti uticaj promene na postojeće funkcionalnosti.
- novac - sve je veći broj Veb aplikacija koje se bave tokom novca i novčanim transakcijama [8].

Postoji veliki broj različitih vrsta napada na Veb aplikacije. Neke od najvažnijih su:

- Umetanje skriptova (eng. Cross-Site Scripting)
- Prevara unakrsnim zahtevima (eng. Cross-Site Request Forgery)
- Nasilno pregledanje (eng. Forcefull Browsing)
- Podmetanje parametara (eng. Parameter Tampering)
- Zloupotreba skrivenih polja (eng. Hidden Field Manipulation)
- Umetanje SQL upita (eng. SQL Injection)

4.4.1 Umetanje skriptova (eng. Cross-Site Scripting)

Napad se izvodi tako što se zlonamerni kôd, najčešće JavaScript, postavlja na lokacije gde drugi korisnici mogu da ga vide. Ciljna polja na formularima mogu biti adrese, komentari, oglasne table, itd... Zlonamerni kôd obično krade kolačić koji napadaču omogućava da se predstavi kao žrtva napada. Cilj napada takođe može biti da se žrtva napada natera da izvrši neželjene operacije. [8]

Način na koji se Veb aplikacija može zaštititi od ovakvih vrsta napada je korišćenjem *OWASP* biblioteke koja se naziva *AntiSamy*. *OWASP* (eng. Open Web Application Security Project) je internet zajednica koja koja sadrži besplatne članke, metodologije, dokumentacije, alate i tehnologije koje se koriste u polju bezbednosti Veb aplikacija. *OWASP AntiSamy* projekat obezbeđuje interfejs koji omogućava da *HTML/CSS* kôd, isporučen od strane korisnika, bude u skladu sa pravilima aplikacije. Drugim rečima, to je interfejs koji onemogućuje korisnike da proslede zlonameran kôd kao vrednost parametara. Pod zlonamernim kodom se uglavnom smatra *JavaScript*. *CSS* kôd se smatra zlonamernim jedino kada u sebi sadrži *JavaScript* kôd. Međutim, postoje mnoge situacije u kojima običan *HTML* i *CSS* kôd mogu biti zlonamerni, pa se mora voditi računa i o tome. [9] *OWASP AntiSamy* pruža *Java* interfejs koji je potrebno uključiti u projekat. Da bi bilo

moguće upotrebiti ga, potrebno je napraviti novi filter. Ideja ovog filtera je da presretne zahtev od korisnika i proveri da li je svaki od parametara uključenih u zahtev u skladu za unapred definisanim pravilima. *AntiSamy* interfejs omogućava ovu proveru. Dakle, prvo se definiše filter koji je u aplikaciji nazvan *AntiSamyFilter*:

```
public class AntiSamyFilter implements Filter
```

i mapira se unutar **web.xml** konfiguracionog fajla:

```
<filter>
 <filter-name>antisamyFilter</filter-name>
 <filter-class>rs.milanmitic.master.filter.AntiSamyFilter</filter-class>
 <init-param>
    <param-name>antisamy-policy-file</param-name>
    <param-value>antiSamyPolicy.xml</param-value>
 </init-param>
</filter>

<filter-mapping>
      <filter-name>antisamyFilter</filter-name>
      <url-pattern>/app/*</url-pattern>
</filter-mapping>
```

Ovime je definisano da će svaki zahtev proći kroz *AntiSamyFilter*. Prilikom mapiranja filtera definisan je i jedan parametar koji se naziva *antisamy-policy-file* i koji ima vrednost *antiSamyPolicy.xml*. Ovaj parametar predstavlja ime fajla u kojem će biti definisana pravila koja će koristiti *AntiSamy* interfejs. Dakle, moguće je unutar XML fajla definisati pravila na osnovu kojih će *AntiSamy* filter prepoznati da li je kôd zlonameran. Ukoliko jeste, *AntiSamy* će ga izmeniti tako da bude u skladu sa pravilima. Postoje predefinisani konfiguracioni fajlovi koji sadrže ova pravila:

- **antisamy-tinymce.xml** - Zasnovan je na HTML WYSIWYG (eng. What You See Is What You Get) editoru i relativno je siguran. Skup pravila unutar ovog fajla dozvoljava samo formatiranje teksta i koristan je za aplikacije koje predstvljaju blogove.
- **antisamy-ebay.xml** - Zasnovan je na pravilima koja se koriste na popularnom aukcionom sajtu EBay. Skup pravila daje malo slobode korisniku i može biti koristan ukoliko korisnik treba da prosledi HTML koji će predstavljati veći deo stranice.
- **antisamy-myspace.xml** - Zasnovan je na pravilima koja se koristite na populatnim društvenim mrežama. Relativno je opasan. Dozvoljava korisniku dosta slobode i pogodan je ukoliko korisnik prosleđuje HTML koji će predstavljati celu stranicu.
- **antisamy-slashdot.xml** - Zasnovan na pravilima koja se koriste na popularnim sajtovima za vesti. Ova pravila omogućavaju samo striktno formatiranje teksta i predstavljaju dobar izbor ukoliko korisnik prosleđuje HTML koji predstavlja neki oblik komentara.
- **antisamy.xml** - Ovo je podrazumevani skup pravila. On dozvoljava skoro sve

HTML elemente, i pogodan je ukoliko korisnik prosleđuje kompletan HTML dokument.

Međutim, s obzirom da nijedan od ovih postojećih skupa pravila nije u potpunosti bezbedan, interfejs omogućava da se definišu sopstvena pravila. Sledi objašnjenje strukture XML fajla pomoću kog se mogu definisati pravila. Sva pravila se definišu unutar elementa *anti-samy-rules*.
Unutar elementa **directives** moguće je navesti neke od definisanih direktiva od kojih svaka ima svoje značenje. Neke od njih su:

- **omitXmlDeclaration** - da li će *AntiSamy* automatski dodati XML zaglavlje ili ne
- **omitDoctypeDeclaration** - da li će *AntiSamy* automatski dodati *doctype* zaglavlje ili ne
- **maxInputSize** - maksimalna veličina parametra u bajtovima
- **useXHTML** - da li će vrednost parametra automatski biti pretvorena u XHTML
- **formatOutput** - da li će vrednost parametra biti formatirana ili ne
- **embedStyleSheets** - da li će spoljne veze ka CSS fajlovima biti uključene ili ne

Na primer:

```
<directives>
	<directive name="omitXmlDeclaration" value="true"/>
</directives>
```

Unutar elementa **common-regexps** se mogu definisati regularni izrazi koji se mogu koristiti u daljim podešavanjima. Na primer:

```
<common-regexps>
	<regexp   name="htmlTitle"   value="[\p{L}\p{N}\s\-_',:\[\]!\./\\\
(\)&amp;]*"/>
</common-regexps>
```

Unutar elementa **common-attributes** mogu se definisati pravila koja moraju da zadovoljavaju poznati atributi HTML elemenata. Na primer:

```
<attribute name="title">
	<regexp-list>
		<regexp name="htmlTitle"/>
	</regexp-list>
</attribute>
```

Ovime je definisano da vrednosti atributa *title* moraju zadovoljavati regularni izraz *htmlTitle* koji smo predhodno definisali.

Unutar elementa **tags-to-encode** se mogu navesti HTML elementi koji neće biti proveravani od strane AntiSamy-a.

Unutar elementa **tag-rules** može se definisati na koji način će elementi biti provereni od strane *AntiSamy*-a. Na primer:

```
<tag-rules>
	<tag name="script" action="remove"/>
	<tag name="iframe" action="remove"/>
	<tag name="style" action="remove"/>
	<tag name="div" action="validate"/>
</tag-rules>
```

Ovo je najbitnije podešavanje. Ovime se definiše da se svi *script, iframe* i *style* elementi uklone od strane *AntiSamy*-a, dok će sadržaj *div* elementa biti proveren. [9]

Nakon što je definisan filter i fajl sa pravilima, preostalo je da se iskoristi podrška koju omogućava *AntiSamy* i da se unutar filtera proveri da li je svaki parametar koji je korisnik prosledio u skladu sa pravilima koja smo definisali. Ovo omogućava klasa iz interfejsa *AntiSamy*, *CleanResults* i *Policy* iz paketa *org.owasp.validator.html*. Prvo se pravi instanca klase *Policy*:

```
Policy policy = Policy.getInstance(new File(path));
```

gde je *path* putanja do fajla gde su definisana pravila. Zatim se pravi instanca klase *AntiSamy*:

```
AntiSamy antiSamy = new AntiSamy(policy);
```

Na kraju, za svaki parametar koji je prosledio korisnik se mora proveriti da li je u skladu sa pravilima:

```
CleanResults cr = antiSamy.scan(parameter, antiSamy.getPolicy());
```

i ukoliko je potrebno parametar zameniti vrednošću koja zadovoljava pravila:

```
parameter = cr.getCleanHTML();
```

4.4.2 Prevara unakrsnim zahtevima (eng. Cross-Site Request Forgery)

Prevara unakrsnim zahtevima predstavlja napad koji primorava korisnika da izvrši neželjene akcije na Veb aplikaciji na kojoj je trenutno autorizovan. Ovi napadi obično za cilj imaju promenu podataka, dok je dosta teže pomoću ovog napada doći do krađe podataka. Napadač pokušava na neki način, najčešće email-om, da dostavi korisniku URL čijim bi se otvaranjem izvršile akcije po želji napadača. Ako je žrtva napada običan korisnik aplikacije, uspešan napad može naterati korisnika da izvrši operacije kao što su transfer novca, promena email adrese, itd... Ukoliko je žrtva administrator aplikacije, onda je ugrožena bezbednost cele aplikacije. [10]

Spring Security obezbeđuje kvalitetnu podršku za zaštitu od ove vrste napada. Štaviše, od verzije 3.2 *Spring Security*-a ova zaštita je podrazumevano omogućena. *Spring Security* implementira zaštitu pomoću sinhronizovanog tokena. Nepisano pravilo je da klijentski zahtevi, kojima se menjaju neki podaci,

moraju biti poslati pomoću POST metode. Dakle, metode GET, HEAD, OPTIONS i TRACE se koriste samo za čitanje podataka. Preduslov je da aplikacija podržava ovo pravilo kako bi u potpunosti bila zaštićena od napada prevarom unakrsnim zahtevima. Server generiše token koji se uključuje unutar formulara na stranici. Kada se pošalje zahtev POST metodom, token se šalje zajedno sa ostalim parametrima. Server presreće ovaj zahtev i proverava da li je token jednak onom koji je on generisao. Ukoliko nije, server generiše izuzetak tipa *CsrfException*. [1]

Iako je u novijim verzijama *Spring Security*-a ova zaštita automatski omogućena, biće pokazano kako se ona omogućuje ručno. U fajlu **spring-security.xml** potrebno je definisati sledeće:

```
<security:csrf request-matcher-ref="csrfRequestMatcher" />
```

request-matcher-ref nije neophodno definisati. On služi da se definiše klasa kojom će biti definisano koje zahteve će presresti server. Mora se navesti ime klase:

```
<bean id="csrfRequestMatcher"
class="rs.milanmitic.master.common.security.CSRFRequestMatcher" />
```

Ta klasa mora predstavljati *Spring* objekat i mora implementirati interfejs *RequestMatcher* iz paketa *org.springframework.security.web.util.matcher.*

```
@Component
public class CSRFRequestMatcher implements RequestMatcher
```

Unutar predefinisane metode *matches* je moguće definisati koje zahteve će server presresti.

Unutar svakog formulara potrebno je uključi token:

```
<input type="hidden" name="${_csrf.parameterName}" value="$
{_csrf.token}"/>
```

4.4.3 Nasilno pregledanje (eng. Forcefull Browsing)

Napad se izvodi tako što korisnik pokušava eksplicitno u pregledač da unese URI za koji nije autorizovan. Posledica napada je da korisnik može videti podatke koji ne bi smeli da mu budu dostupni ili da izvrši operacije za koje nije ovlašćen.

Već je objašnjeno na koji način je obezbeđeno da samo autorizovani korisnici mogu pristupati aplikativnim delovima aplikacije. Takođe je objašnjeno na koji način je obezbeđeno da korisnik može pristupati samo onim funkcionalnostima za koja ima privilegije.

4.4.4 Podmetanje parametara (eng. Parameter Tampering)

Napad se izvodi tako što korisnik eksplicitno menja parametre unutar URI-a. Cilj je neovlašćeno pristupanje podacima ili neovlašćeno izvršavanje operacija. Posledica napada je da korisnik ima pristup podacima aplikacije za koje nije autorizovan.

Običan korisnik može da pristupa samo onim podacima koji pripadaju njegovom učesniku. Ono što nije objašnjeno jeste na koji način je ovo omogućeno, a upravo ovo može biti meta napada podmetanjem parametara. Recimo da korisnik ima privilegiju da čita i menja podatke o drugim korisnicima. On, koristeći aplikaciju, može videti da je URI koji odgovara čitanju korisnika u obliku:

- */Master/app/appUser/view/4*

On može zaključiti da se u URI-u nalazi identifikator korisnika i može eksplicitno u pregledaču pokušati da promeni identifikator. Takvim nabrajanjem identifikatora može naići na neki koji odgovara korisniku drugog učesnika i pročitati podatke o njemu, iako ne bi smeo da ima pristup njima. Kako ovo izgleda na konkretnom primeru? Recimo da je tematika aplikacije studentski servis i da imamo dva učesnika, Matematički fakultet i Biološki fakultet. Korisnik Matematičkog fakulteta nabrajanjem identifikatora korisnika nailazi na identifikator koji odgovara korisniku Biološkog fakulteta i pristupa podacima o njemu, iako on sme da pristupa samo podacima o korisnicima Matematičkog fakulteta.

Ono što je potrbno uraditi kako bi aplikacija bila zaštićena od ove vrste napada, jeste da prilikom svakog zahteva za čitanjem ili menjanjem podataka se proverava sledeće:

1. Da li je zahtev poslao običan korisnik
2. Da li podatak ili podaci pripadaju nekom Učesniku
3. Da li podaci i korisnik odgovaraju istom Učesniku

Mora se voditi računa da se ove provere sprovode prilikom svakog zahteva.

4.4.5 Zloupotreba skrivenih polja (eng. Hidden Field Manipulation)

Zloupotreba skrivenih polja je sličan napad kao nasilno pregledanje. Razlika je u tome što se ne menjaju parametri koji su deo URI-a, već vrednosti skrivenih polja formulara. Na primer, na formularu za promenu nekog podatka postoji skriveno polje koje čuva vrednost identifikatora. Korisnik može promeniti vrednost ovog skrivenog polja i poslati takav zahtev serveru. Na taj način može promeniti podatak za koji nema privilegije. Način na koji se aplikacija obezbeđuje od ovakve vrste napada je isti kao i kod napada nasilnim pregledanjem. Dakle, eksplicitnim proverama na serverskoj strani da li prosleđeni podaci odgovaraju kontekstu prijavljenog korisnika.

Drugi način na koji je moguće obezbediti aplikaciju od ovakve vrste

napada jeste provera da li korisnik nije menjao podatke skrivenih polja. Na koji način je ovo moguće proveriti? Unapred se zna koja polja će biti skrivena na formularu. Pre dolaska na formular, generiše se heš vrednost korišćenjem sve vrednosti skrivenih polja. Pored svih vrednosti skrivenih polja, može se koristiti i neki tajni ključ. Dobijena heš vrednost se prosleđuje formularu i čuva kao još jedno skriveno polje. Kada korisnik pošalje zahtev, na serveru ponovo generišemo heš vrednost od skrivenih polja i upoređujemo sa prosleđenom heš vrednošću. Ukoliko su vrednosti različite, to znači da je korisnik menjao vrednosti skrivenih polja. Dakle, da bi korisnik mogao da promeni neko skriveno polje, on mora da izračuna novu heš vrednost i promeni i nju. A da bi to mogao da uradi, on mora da zna na osnovu kojih polja se ona generiše, kojim redosledom, kojim algoritmom i vrednost tajnog ključa što je nemoguće znati i nemoguće pogoditi.

4.4.6 Umetanje SQL upita (eng. SQL Injection)

Umetanje SQL upita je jedan od najčešće korišćenih napada na Veb aplikacije. Napad se izvodi tako što se kao vrednosti polja na formularima unose SQL upiti. Cilj je neovlašćeno pristupanje podacima, neovlašćeno menjanje podataka, izvršavanje administratorskih operacija nad bazom podataka. Najčešće korišćeni napad jeste umetanjem SQL upita 'OR 1=1'. Recimo da se upitom traže podaci o korisniku, a kao parametar se prosleđuje ime korisnika. Upit bi izgledao:

- "SELECT * FROM APP_USER WHERE USERNAME = '" + PARAM + "'

Ukoliko se kao parametar unese 'OR 1 = 1', dobija se upit koji vraća sve korisnike iz baze podataka, bez obzira na sve druge uslove u upitu. Dakle, preduslov za izvođenje ove vrste napada jeste da se na serverskoj strani upiti dobijaju spajanjem niski. Ovo je nešto što programeri nikako ne bi smeli da koriste. Umesto spajanjem niski, moraju se koristiti parametrizovani upiti. *Spring* i *Hibernate* podržavaju parametrizovane upite. Upiti se prave pomoću klase *Query* iz paketa *org.hibernate*. Parametri se dodaju pomoću metode *setParameter*:

```
String hql = "SELECT a FROM AppUser a WHERE a.username = :username ";
Query q = getCurrentSession().createQuery(hql);
q.setMaxResults(1);
q.setParameter("username", username.toLowerCase().trim());
```

5. Zaključak

Današnje vreme karakteriše velika rasprostranjenost Veb aplikacija. Ne postoji oblast interesovanja za koju ne postoji neki oblik Veb aplikacije. Najbitniji aspekt svake Veb aplikacije predstavljaju podaci, počevši od osnovnih podataka kao što su korisničko ime, lozinka i email adresa, pa sve do izuzetno poverljivih informacija kao što su broj kreditne kartice ili stanje bankovnog računa. Korisnici očekuju da će podaci koje dostave aplikaciji ostati poverljivi i zaštićeni. U ovom trenutku postoji veliki broj različitih vrsta napada na Veb aplikacije. Zato programeri moraju posvetiti veliku pažnju bezbednosti aplikacije. Veb aplikacija koja nije bezbedna vrlo brzo može postati neupotrebljiva. Razvojno okruženje *Spring* i njegova podrška za bezbednost omogućavaju programerima da brzo i lako postave osnovu Veb aplikacije koja sadrži veliki nivo bezbednosti. Spring pruža mnoga gotova rešenja, ali i mogućnost da se ta rešenja prilagode potrebama aplikacije. Programeri na ovaj način, u početnoj fazi projekta, definišu sve bezbednosne aspekte aplikacije, tako da se u narednim fazama razvoja mogu posvetiti isključivo funkcionalnim delovima aplikacije.

U master radu je pokazano, kako pomoću razvojnog okruženja *Spring* i tehnologija *Java, JSP* i *Hibernate,* napraviti Veb aplikaciju koja koristi model-pogled-kontroler arhitekturu. Detaljno je istražena i opisana *Spring*-ova podrška za bezbednost koja se naziva *Spring Security*. Opisani su procesi autentikacije i autorizacije. Na primeru konkretne aplikacije pokazano je kako pomenutu podršku implementirati, ali i prilagoditi potrebama same aplikacije. Podrška je iskorišćena da bi aplikacija podržavala više tipova korisnika i da bi određeni tipovi korisnika mogli upravljati pravima pristupa drugih korisnika. U radu su opisane osnovne vrste napada na Veb aplikacije, kako se od njih odbraniti i kako je to implementirano u aplikaciji.

Na kraju se može zaključiti da razvojno okruženje *Spring* omogućava brz razvoj Veb aplikacija. Podrška za bezbednost omogućava da se u ranoj fazi razvoja definišu svi bezbednosti aspekti koji se lako mogu prilagoditi funkcionalnim zahtevima same aplikacije.

6. Literatura

1. Craig Walls, *Spring in Action,* 4th Edition, Manning, November 2014

2. Herbert Schildt, *Java complete reference,* 9th Edition, McGraw-Hill Osborne Media, March 2014

3. Rod Johnson, *Spring Framework Reference Documentation*, https://docs.spring.io/spring-framework/docs/current/spring-framework-reference/html/index.html, 2016 (datum proslednjeg pristupa 6. septembar 2017.)

4. Hans Bergsten, *Java Server Pages*, O'Reilly Media, December 2000

5. *Spring Tutorial*, https://www.tutorialspoint.com/spring/spring_web_mvc_framework.htm, 2016 (datum proslednjeg pristupa 6. septembar 2017.)

6. Ben Alex, Luke Taylor, *Spring Security Reference Documentation*, http://docs.spring.io/spring-security/site/docs/3.0.x/reference/springsecurity.html, (datum proslednjeg pristupa 6. septembar 2017.)

7. Dustin Boswel, *Brain Dumps about Computers, Programming, and Everything Else*, http://dustwell.com/how-to-handle-passwords-bcrypt.html, 2012 (datum proslednjeg pristupa 6. septembar 2017.)

8. Joel Scambray, Vincent Liu, Caleb Sima, *Hacking Exposed Web Applications*, 3rd Edition, McGraw-Hill, October 2010

9. Arshan Dabirsiaghi, *OWASP AntiSamy Project*, https://www.owasp.org/index.php/Category:OWASP_AntiSamy_Project, (datum proslednjeg pristupa 6. septembar 2017.)

10. Dave Wichers, *OWASP Cross-Site Request Forgery (CSRF)*, https://www.owasp.org/index.php/Cross-Site_Request_Forgery_(CSRF), 2017 (datum proslednjeg pristupa 6. septembar 2017.)

11. Chris Snyder, Thomas Myer i Michaer Southwell, *Pro PHP Security*, Apress, 2005

12. Lakshmiraghavan, Badrinarayanan, *Pro ASP.NET Web API Security*, Apress, 2013