TALLINNA TEHNIKAÜLIKOOL
Infotehnoloogia teaduskond
Arvutiteaduse instituut

ITV40LT

Rannar Allorg 134554IAPB

SPRING RAAMISTIKULT SPRING BOOTI PEALE ÜLEMINEKU AUTOMATISEERIMINE JA PROTSESSI KIRJELDAMINE

bakalaureusetöö

Juhendaja: Ago Luberg
MSc
Assistent

Tallinn 2016

Autorideklaratsioon

Kinnitan, et olen koostanud antud lõputöö iseseisvalt ning seda ei ole kellegi teise poolt varem kaitsmisele esitatud. Kõik töö koostamisel kasutatud teiste autorite tööd, olulised seisukohad, kirjandusallikatest ja mujalt pärinevad andmed on töös viidatud.

Autor: Rannar Allorg

24.05.2016

Annotatsioon

Töö eesmärgiks oli töötavas Spring rakenduses Spring Booti kasutusele võtmine, protsessi põhjal juhendi koostamine ning seejärel juhendi põhjal protsessi automatiseeriva programmi loomine. Programm lähtub Spring Booti kasutamise nõuetest ja kohandab projekti Maveni faili, struktuuri ja konfiguratsiooni.

Töö tulemusena valmis juhend, millest juhindudes on võimalik ka teistes rakendustes Spring Booti kasutuselevõtt ning sealjuures on võimalik kasutada programmi, mis on võimeline osa tööst ise ära tegema. Valminud programm ei paku täielikku üleminekut, kuid vähendab arendaja jaoks masinliku töö mahtu.

Lõputöö on kirjutatud eesti keeles ning sisaldab teksti 29 leheküljel, 5 peatükki, 8 joonist.

Abstract

Automating the transition from Spring Framework to Spring Boot and description of the process

This thesis describes the process of converting Spring application to Spring Boot application and the process of creating a program for automating this transition. Spring framework requires huge amount of configuration and Spring Boot tries to reduce it. During this thesis, the author is converting an existing Spring project to Spring Boot.

The results of the thesis were a guide for going from Spring to Spring Boot and a program that helps to automate the process. Guide describes the transition in four steps: Maven, project structure, XML to Java configuration and security configuration. The guide is based on the project that the author used for this thesis, but it is written in a way that it can be helpful for other projects as well. The automating program is based on the previously created guide, but it consists of three steps: Maven, project structure and XML to Java configuration. Security configuration is not part of automation. The program is able to assist the transition by adding elements that every Spring Boot application needs and migrating Spring Bean configurations from XML to Java as much as possible.

The thesis is in Estonian language and contains 29 pages of text, 5 chapters, 8 figures.

Lühendite ja mõistete sõnastik

POJO	Tavaline Java objekt; i.k. *Plain Old Java Object*
Bean	Java objekt, mille loob või mida kasutab Spring *IoC container*. Spring rakenduse element.
JAR	Failiformaat, mis tüüpiliselt sisaldab Java klasse ja nendega seotud *metadata't*; i.k. *Java Archive*
POM	Maveni tööks vajalik komponent XML fail, mis sisaldab seadistusi selle kohta, kuidas Maven peab projekti ehitama; i.k. *Project Object Model*
Boilerplate	Tekst või kood, mida kasutatakse mitmetes kohtades olulisi muudatusi tegemata
JSP	Serveri poolne tehnoloogia, mis võimaldab dünaamiliste veebilehtede loomist; i.k. *JavaServer Pages*
XML	Märgistuskeel, mille eesmärgiks on struktureeritud informatsiooni jagamine infosüsteemide vahel; i.k. *Extensible Markup Language*
URL	Internetis asuva ressursi aadress; i.k. *Uniform Resource Locator*
CSRF	Küberrünnak, mis kasutab ära veebilehe usaldust kasutaja brauseri vastu; i.k. *Cross-Site Request Forgery*
API	Rakendusliides; i.k. *Application Programming Interface*
Java	Objektorienteeritud programmeerimiskeel
Groovy	Java platvormil töötav programmeerimiskeel

Sisukord

Jooniste loetelu

1 Sissejuhatus

Käesoleva bakalaureusetöö eesmärgiks on Spring [1] raamistikul ehitatud projekti Spring Bootile [2] üleviimine, selle protsessi kirjeldamine ja võimaluste piires automatiseerimine. Kasutusel olevad tehnoloogiad vananevad kiiresti ja tihtipeale kipuvad valmis projektid seetõttu ajast maha jääma, kuna uuendamine on enamasti küllaltki aeganõudev protsess. Sellest tulenevalt otsustas autor luua lahenduse, mis lihtsustaks Springil põhinevate projektide uuendamist. Töö käigus valmis juhend ülemineku teostamiseks ja programm, mis aitab protsessi kiirendada.

1.1 Taust ja probleem

Suurte tarkvaraprojektide puhul on tihti probleemiks tehnoloogia aegumine aastatega. Kuna uuemad tehnoloogiad üritavad parandada minevikus eksisteerinud vigu, on kasulik aeg-ajalt tehnoloogiaid uuendada. Probleemiks on aga see, et tarkvarafirmades on enamasti tööd rohkem kui arendajatel aega ja seega jäetakse uuendamine tagaplaanile. Kuna Spring on väga laialt levinud raamistik [3], siis aitab antud töö paljude arendajate tööd lihtsustada. Lisaks sellele aitab automatiseerimine vähendada kuiva töö hulka, mis nimetatud üleminekuga kaasneb.

Töö tulem on abiks kõigile arendajatele, kellele tuleb kasuks olemasolevate Spring projektide kaasajastamine. Samuti on töö abiks ka nende arendajate jaoks, kellel on Spring tehnoloogiatega vähe kogemusi, kuid tahavad mõnes olemasolevas projektis Spring Booti kasutusele võtta.

1.2 Ülesande püstitus

Uute tehnoloogiatega kaasnevad alati dokumentatsioonid sellest, kuidas konkreetset tehnoloogiat kasutada, kuid väga harva on juttu sellest, mille poolest erineb uus vanast või kuidas neid omavahel sobitada. Tulemusena tuleb kulutada palju aega dokumentatsioonide lugemisele ja erinevuste leidmisele, et uuendamise vajadusel nendest lähtuda.

Käesolev töö üritab seda probleemi Spring rakenduste puhul vähendada, andes ülevaate sellest, mis on tavalise Springi ja Spring Booti peamised erinevused ning kuidas üleminek välja näeb. Lisaks valmib programm, mis aitab lihtsustada järgnevaid samme:

- Maveni [4] konfiguratsiooni muutmine.
- Projekti struktuuri muutmine.
- XML konfiguratsioonilt üle minemine Java põhisele konfiguratsioonile.

1.3 Metoodika

Töö tulemuse saamiseks kasutas autor ühe Eestis tegutseva kindlusteenuseid pakkuva ettevõtte e-teeninduse rakendust, mille käsitsi uuendamine sai juhendi koostamise aluseks. Automatiseeriv programm on loodud Groovy programmeerimiskeeles ja selle kirjutamisel lähtus autor käesoleva töö raames valminud ülemineku juhendist.

1.4 Ülevaade tööst

Töö jaguneb kolmeks põhiliseks etapiks:

- Springi ja Spring Booti tutvustus ja võrdlus. Tuuakse välja, mida nimetatud tehnoloogiad endast kujutavad ja millised on nende erinevused. Samuti tuuakse välja põhjused, miks on kasulik ühelt teisele üleminek.
- Ülemineku kirjeldus ja analüüs. Räägitakse sellest, kuidas näiteprojekti põhjal üleminek välja nägi ja millised on olulisemad sammud, mida kindlasti teha tuleks.
- Automatiseeriva programmi tehniline teostus. Kirjeldatakse, milliseid samme automatiseeritakse ja kuidas see protsess töötab. Samuti tuuakse välja sammud, mida ei automatiseerita ja põhjused, miks käesoleva töö raames seda ei tehta.

2 Tehnoloogiate tutvustus

Käesoleva töö raames on kasutusel Java Spring veebitehnoloogiad. Selles peatükis räägib autor lühidalt nende tehnoloogiate olemusest ja tööpõhimõtetest. Lisaks sellele tuuakse välja kasutatavate tehnoloogiate peamised erinevused ja selgitatakse, miks eelistab autor ühte tehnoloogiat teisele.

2.1 Spring raamistik

Spring on populaarseim Java raamistik [5], mille eesmärgiks on lihtsustada Java rakenduste loomist. Spring koosneb erinevatest moodulitest, mille abil on võimalik täita rakenduse tehnoloogilisi vajadusi. Käesoleva töö raames on kasutusel veebirakenduste jaoks vajalikud moodulid.

Spring raamistikul loodud rakendused koosnevad tavalistest Java objektidest (*POJO*), mis on Springi poolt kokku ühendatud ning seadistatud. Need objektid kannavad nime Spring bean. Selleks, et oleks võimalik objekte omavahel siduda, on vaja konfiguratsiooni infot, mis võib Springis olla esitatud XML või Java kujul. Konfiguratsioonifailis defineeritakse, milliste klassidega tuleb Springil ise tegeleda ning milliseid parameetreid nende klasside objektid vajavad.

2.2 Spring Boot

Spring on võimalusterohke ja kergesti kasutatav raamistik, kuid loodud projektide tööle saamine nõuab suurt konfigureerimist, mis takistab projekti jaoks vajaliku funktsionaalsuse arendamist. Projekti arenduse käigus tuleb samuti arvestada sellega, et funktsionaalsuse lisamine või muutmine võib kaasa tuua konfiguratsiooni täiustamise vajaduse.

Spring Boot on Spring raamistiku moodul, mille eesmärgiks on võimalikult väikese vaevaga töötavate rakenduste ülesseadmine. Spring Booti rakendusi on võimalik luua

ilma XML konfiguratsioonita ning valmis olevat funktsionaalsust ära kasutades. Lisaks sellele pakub Spring Boot käivitatavate JAR failide kasutamist. Teisisõnu tähendab see seda, et rakenduse käivitamiseks piisab `java -jar application.jar` käsu käivitamisest. Märkimisväärne sealjuures on see, et samamoodi on võimalik käivitada ka veebirakendusi. Viimasel juhul kasutab Spring Boot sisseehitatud veebiserverit, mis on võimeline täielikult töötama ilma välise serverita.

2.3 Spring Boot'i eelised

Spring Booti kasutava ning mitte kasutava rakenduse peamine erinevus seisneb nende tööpõhimõtetes. Tavaline Spring rakendus eeldab täielikku konfiguratsiooni, mis määrab ära rakenduse tööpõhimõtted. Spring Booti käitumine on tavalisest Springist erinev. Nimelt on Spring Boot "iseteadlik" raamistik, mis tähendab, et vastavalt etteantud nõuetele seadistab ta ennast oma heaksarvamise järgi.

Spring Booti peamised eelised:

- Ehitamissüsteemi *starter* POM-id. Tegemist on sõltuvuste gruppidega, mis projekti lisamisel toovad endaga kaasa seotud sõltuvused. Näiteks on olemas sõltuvus nimega `spring-boot-starter-web`, mille koosseisu kuuluvad: `jackson`, `spring-web`, `spring-webmvc`, `spring-boot-starter`, `spring-boot-starter-tomcat` ja `spring-boot-starter-validation`.

- Koheselt valmis funktsionaalsus. Spring Boot on võimeline töötama ilma igasuguse konfiguratsioonita. Ainsad tingimused on ehitamissüsteemi sõltuvuste olemasolu ja korrektne projekti struktuur.

- Käivitatavad JAR failid. Käivitavate JAR failide loomine oli võimalik ka enne Spring Booti tulekut, aga protsess oli väga tülikas. Spring Booti abil on võimalik selline JAR kokku panna ühe Maveni käsuga. Käivitatavate JAR failide suurim kasu tuleb välja veebirakenduste puhul. Spring Boot pakub võimalust kasutada sisseehitatud servereid, mis sarnaselt Spring Booti endaga on võimelised koheselt töötama. Teisisõnu on võimalik veebirakendusi lihtsalt käivitada.

Kokkuvõttes võib öelda, et enamasti on Spring Booti kasutamine igal juhul soovitatav. Lisaks lihtsale ülesseadmisele on Spring Booti kasutavates projektides tunduvalt vähem

boilerplate koodi, mille hulk suurte projektide puhul kiirelt kasvab. Vanemate Springi projektide puhul toob üleminek kaasa peamiselt projekti konfiguratsiooni poolse lihtsustumise ja parema hallatavuse. Tavaliselt Springi projektilt ülemineku korral tuleks siiski korraks mõelda, kas üleminek tasub ära. Juhul kui projekti konfiguratsioon on väga erinev Spring Booti vaikimisi seadistustest võib üleminek osutuda väga aeganõudvaks ja kasutegur sellisel juhul oleks minimaalne.

3 Spring raamistikult Spring Bootile ülemineku kirjeldus

Spring Booti kasutusele võtmisel tuleb arvestada erinevate nõuetega. Spring Boot vajab töötamiseks vähemalt Java 7-t ja Spring raamistiku versioon `4.2.6.REALEASE` või uuem. Tehniliselt on võimalik kasutada ka Java 6-te, kuid see nõuab eraldi seadistamist. Vaatamata sellele, et Spring Boot toetab nii Java 6-te kui ka Java 7-t, on ametlik soovitus kasutada Java 8-t [6]. Juhul kui varasemalt on projektis kasutusel vanemad versioonid, tuleb arvestada lisatööga, mis tuleneb Springi ja Java uuendamisest. Näiteks võib tekkida olukordi, kus mõni meetod on muutunud mittesoovituslikuks või on midagi täielikult eemaldatud.

Käesoleva töö raames on autor otsustanud ülemineku analüüsimiseks kasutada ühe Eestis tegutseva kindlustusteenuseid pakkuva ettevõtte e-teenindust. Nimetatud rakenduses oli algselt kasutusel Spring `3.0.6.RELEASE`, Java 1.6, Maven ning kasutajaliidese loomiseks oli kasutatud JSP-sid. Antud juhul oli eesmärgiks saada projekt tööle kasutades Spring Booti, sealjuures säilitades täielikult olemasoleva funktsionaalsuse. Uuendamise protsess jaguneb neljaks suuremaks etapiks:

- Maveni pom.xml'i uuendamine.
- Projekti struktuuri muutmine
- XML konfiguratsioonilt üleminek Java-põhisele konfiguratsioonile
- Turvakonfiguratsioon

3.1 Maven

Esimene samm on `spring-boot-starter-parent`'i (*parent* POM) lisamine. *Parent* POM toob projekti esimesed eelseadistused, mille hulka kuuluvad Java 1.6 ja UTF-8. Soovitatav on Java versiooni käsitsi muutmine uuema peale. Samuti kaasneb parent POM-iga sõltuvuste haldus, tänu millele piisab *parent* POM-is versiooni defineerimisest ja kõik ülejäänud Spring Booti sõltuvuste versioonid seadistatakse automaatselt. *Parent*

POM-i kasutamine ei ole kohustuslik. Juhul, kui projektil on oma standardid, mis määratakse Maveni *parent* POM-i poolt, on Spring Booti kasutamine ikka võimalik.

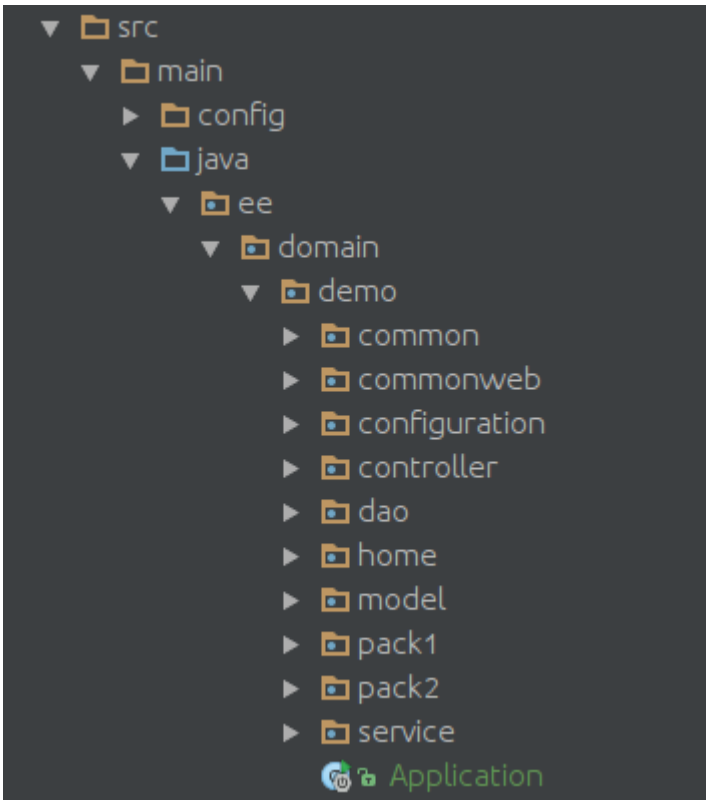
`spring-boot-maven-plugin` on Maveni pistikprogramm, mis võimaldab käivitatavate JAR failide loomist. Tegemist on rangelt soovitusliku elemendiga, kuna suur osa Spring Booti mugavusest seisneb käivitatavate JAR-ide loomises. Juhul kui eelnevalt on määratud Spring Booti *parent* POM, ei ole vaja pistikut seadistada ja piisab selle lisamisest pom.xml'i `build` märgendite vahele.

Kolmas samm on sõltuvuste korrastamine. Suurte projektide puhul kipub sõltuvuste nimekiri väga pikaks minema. Lisaks pikast nimekirjast tingitud loetavuse langemisele, kaasneb sellega ka probleem sõltuvuste versioonidega. Näiteks ühe sõltuvuse versiooni uuendamine võib nõuda teiste sõltuvuste uuendamist. Selline olukord toob kaasa ajakulu, mida on võimalik vähendada Spring Booti abil. Eelnevalt on välja toodud, et Spring Boot pakub *starter* POM-ide kasutamise võimalust. Iga starter POM esindab ühte kategooriat ja toob endaga kaasa selle kategooria jaoks vajalikud sõltuvused. Käesoleva sammu teostamiseks, tuleks välja vahetada kõik Maveni sõltuvused, mis on esindatud *starter* POM-ide koosseisus.

Välja toodud sammud ei ole ainult Maveni põhised. Sarnast protsessi on võimalik läbi viia ka Gradle [7] projektide puhul. Täpsemat informatsiooni Spring Booti ehitamissüsteemide kohta on võimalik leida ametlikust dokumentatsioonist [8].

3.2 Projekti struktuur

Spring Boot vajab töötamiseks `main()` meetodit, mida kasutatakse programmi sisendpunktina. Selleks, et Spring Boot oleks võimeline ennast automaatselt seadistama, peab `main()` meetod asuma projekti puus teistest Java klassidest kõrgemal. Näitena toob autor välja kasutatud projekti Spring Bootile sobiliku struktuuri (Joonis 1.).



Joonis 1. Lõpptulemusena projekti struktuur.

Spring Boot kasutab automaatselt staatilisi elemente, mis asuvad järgnevates kaustades [9]:

- /META-INF/resources/
- /resources/
- /static/
- /public/

Autori kasutatavas projektis asusid kasutajaliidese moodustavad failid `webapp` nimelises kaustas, mis ei vasta Spring Booti poolt soovitatavale struktuurile. Selle probleemi lahendamiseks tuli projekti seadetes määrata JSP-de asukoht.

3.3 XML konfiguratsiooni muutmine Java konfiguratsioonile

Spring Boot võimaldab luua Spring rakendusi ilma XML konfiguratsioonita. Tänu Spring Booti autokonfiguratsioonile väheneb vajaliku seadistamise maht oluliselt ning Java

konfiguratsiooni kasutamine aitab seadistust kompaktsemaks muuta. Java-põhine seadistamine kujutab endast Java annotatsioonide kasutamist, klasside ja meetodite ülekirjutamist (*override*) ja *bean*'ide loomist. Spring Booti kasutusele võtmisel Spring projektis on lihtsam määrata XML failide asukohad ja jääda neid kasutama, kuid tulevikule mõeldes on igati kasulik Java konfiguratsioonile üleminek.

Eeldades, et projektis on kasutusele võetud Spring Boot, on võimalik luua Java klasse, mille juurde on lisatud annotatsioon `@Configuration`. Sellise annotatsiooniga klasse hakkab Spring Boot käivitamisel otsima ja selliste klasside sisse saab kirjutada loogika, mis asendab XML konfiguratsioone.

Suurima osa sellisest konfiguratsioonist moodustavad *bean*'id. Enamasti on tegemist objektidega, mis tulevad projekti Maven'i sõltuvuste kaudu ja seetõttu ei ole võimalik neid komponentideks annoteerida. Iga XML-is defineeritud *bean* esitab Java objekti XML kujul. Näiteks võib lihtne XML *bean* olla defineeritud kujul (Joonis 2.):

```
<bean class="ee.domain.demo.common.utils.ApplicationContextProvider"/>
```

Joonis 2. Lihtne *bean* XML kujul.

Selline esitus ütleb Springile, et rakenduses on kasutuses bean klassiga `ApplicationContextProvider`. Java esitlus sama *bean'i* jaoks näeks välja (Joonis 3.):

```
@Bean
public ApplicationContextProvider applicationContextProvider() {
     return new ApplicationContextProvider();
}
```

Joonis 3. Lihtne *bean* Java-kujul

`@Bean` annotatsioon märgib ära, et tegemist on *bean'iga* ja meetod peab tagastama objekti, mis vastab XML-is määratud klassile. Keerukama näitena toob autor välja järgneva *beani* (Joonis 4.):

```
<bean id="webServiceTemplate" class="org.springframework.ws.client.core.WebServiceTemplate">
    <constructor-arg ref="messageFactory"/>
    <property name="marshaller" ref="marshaller"/>
    <property name="unmarshaller" ref="marshaller"/>
    <property name="messageSender">
        <bean class="org.springframework.ws.transport.http.HttpComponentsMessageSender">
            <property name="credentials">
                <bean class="org.apache.http.auth.UsernamePasswordCredentials">
                    <constructor-arg value="${username}:${password}"/>
                </bean>
            </property>
        </bean>
    </property>
    <property name="defaultUri" value="${url}"/>
</bean>
```

Joonis 4. Keerukam *bean* XML kujul.

`class` atribuut määrab ära, millise klassi konkreetne `bean` tagastab. `id` vastab `bean`'i nimele, mis Java's on meetodi nimi. Kõik, mis jääb `<bean>` märgendite vahele kujutavad endast *bean'i* omadusi. Parema ülevaate andmiseks, toob autor *bean'i* sisu samm haaval välja:

- `ref` atribuut viitab teisele `bean`'ile ja atribuudi väärtus on viidatava *bean'i* nimi.
- `constructor-arg` märgend tähistab *bean'i* konstruktori argumente.
- `property` märgendid tähistavad *bean'i* väljade väärtusi. Antud näites lisatakse väljadele `marshaller` ja `unmarshaller` väärtuseks `bean` nimega `marshaller`.
- `bean` märgend `property` sees tähendab, et `property` väärtuseks saab teine `bean`. Sama asja on võimalik teha ka `ref` atribuudi abil.
- `${...}` tähendab, et projekti seadetes on defineeritud väärtus, mida Spring saab sellele kohale lisada.

Sellisele XML *bean'ile* vastav Java konfiguratsioon (Joonis 5.):

```java
@Value("${username}")
private String username;

@Value("${password}")
private String password;

@Value("${url}")
private String defaultURL;

@Bean
public WebServiceTemplate webServiceTemplate() {
    WebServiceTemplate webServiceTemplate = new WebServiceTemplate(messageFactory());
    webServiceTemplate.setMarshaller(marshaller());
    webServiceTemplate.setUnmarshaller(marshaller());
    webServiceTemplate.setMessageSender(messageSender());
    webServiceTemplate.setDefaultUri(defaultURL);

    return webServiceTemplate;
}
```

Joonis 5. Keerukam *bean* Java kujul.

XML konfiguratsioon ei seisne ainult *bean'ide* loomises. Springi põhiste veebirakenduste käitumine on kirjeldatud `web.xml` nimelises failis. Autori kasutatavas rakenduses leidusid näiteks: `<filter>` ja `<listener>` märgendid. Nende üleviimiseks Java konfiguratsioonile on vajalik vastavate klasside laiendamine ja seejärel meetodite üle kirjutamine. Autori projekti loodi selleks `WebConfiguration` klass, mis laiendab `WebApplicationInitializer` klassi ja kirjutab üle `onStartup` meetodi. Kuna selline konfiguratsioon on väga projektikeskne, siis otsustas käesoleva töö autor loobuda selle teema sügavuti uurimisest.

3.4 Turvakonfiguratsioon

Turvakonfiguratsioon on üks vähestest asjadest, mida tuleb vastavalt projektile käsitsi seadistada. Kui Maveni sõltuvuste hulgas on `spring-boot-starter-security`, lisandub rakendusele automaatselt lihtne autentimise nõue. Soovitud lehele ligi pääsemiseks tuleb hüpikakna kaudu ennast sisse logida Spring Boot'i poolt pakutud andmetega. Selline lähenemine ei ole üldiselt kasutamiskõlbulik, kuna kasutajainfo peaks olema kasutaja poolt valitud ning enamasti leidub ka selliseid lehti, mille juurde pääsemiseks ei ole vaja sisse logida.

Turvalisuse seadistuse üleviimisel Java konfiguratsioonile tuleb selgeks teha projekti turvanõuded. Teisisõnu peab olema teada, millised lehed on nähtavad ainult sisse loginud kasutajale. Autori projektis tuli seadistada URL-id, mida CSRF-i eest kaitsta ning URL-

id, mille nägemiseks peab kasutaja olema sisse logitud. Kuna autori kasutuses olev projekt kasutas autentimiseks pangalinke ning ID-kaarti, siis tuli Spring Booti vaikimisi seatud kasutaja tuvastuse seadistus üle kirjutada. Turvakonfiguratsiooni lisamiseks on autor oma projekti põhjal toonud välja järgnevad sammud:

- Esimese sammuna tuleb luua klass, mis laiendab `WebSecurityConfigurerAdapter` klassi ja omab `@EnableWebMvcSecurity` annotatsiooni.
- Järgmisena tuleb üle kirjutada `configuration(HttpSecurity)` meetod.
- Meetodi sees defineeritakse uus `RequestMatcher`, mille abil määratakse ära URL-id, mis peavad CSRF-i eest kaitstud olema.
- Määratakse URL-id, millele pääsemiseks peab kasutaja sisse logitud olema.
- Lisatakse *bean*, mis tegeleb kasutaja välja logimisega
- Klassi lisatakse `configureGlobal(AuthenticationManagerBuilder)` meetod, mille abil määratakse ära sisse logimisega tegelevad *bean*'id

4 Automatiseerimine

Töö käigus selgus, et Spring Booti kasutusele võtmine sisaldab suurel hulgal käsitööd, mille korduv tegemine on tüütu. Selle probleemi lahendamiseks valmib käesoleva töö raames programm, mis aitab Springis loodud veebirakendusi Spring Bootile üle viia.

4.1 Automatiseeritavad etapid

Töö kolmandas peatükis jagas autor ülemineku protsessi neljaks suuremaks etapiks. Automatiseerimisele lähevad nendest kolm:

- Maveni uuendamine.
- Projekti struktuuri muutmine.
- Java konfiguratsioonile üleminek.

Neljas etapp (turvakonfiguratsioon) jäeti automatiseerimise protsessist välja, kuna Java kujul turvakonfiguratsioon Spring Bootis, erines oluliselt XML konfiguratsioonist tavalises Springis. Lisaks sellele võib vale turvaseadistus kaasa tuua suuri kahjusid rakendust kasutavale ettevõttele ja on parem kui sellise konfiguratsiooniga tegeleb kogenud arendaja.

4.2 Automatiseerimisega seotud probleemid

Peamised probleemid tekivad etappides, kus on kindel sisend ja kindel väljund. Antud töö puhul on nendeks etappideks Maveni uuendamine ja XML-ilt Javale üleminek. Ühise tunnusjoonena mõlema etapi puhul toob autor välja fakti, et kood on kirjutatud inimeste poolt, mis toob kaasa koodi stiili varieeruvuse kohtades, kus on võimalik kirjutada sama asja erinevatel viisidel.

4.2.1 Maveni uuendamisega seotud probleemid

Maveni uuendamisel kerkisid esile järgnevad probleemid:

- Sõltuvuste konfliktid. Autori poolt kasutatud projekt oli vana ja sellest tulenevalt olid paljud sõltuvused endiselt vanade versioonide peal. Spring Boot kasutab uuemaid sõltuvusi. Tulemusena võivad tekkida versioonide erinevusest tulenevad konfliktid.

- Sõltuvuste muutumine. Mõnede sõltuvuste puhul võib `artifactId` või `groupId` uuema versiooniga muutuda. Näiteks „Opensymphony >> Sitemesh" on versioon kahes `groupId`'ga „Opensymphony". Paraku ei toeta see versioon Java konfiguratsiooni ja kasutada tuleb „Sitemesh 3-e", mis on Maveni hoidlas `groupId`'ga „org.sitemesh".

4.2.2 XML konfiguratsioonilt Java konfiguratsioonile ülemineku probleemid

Peamised probleemid seoses konfiguratsioonide üleminekuga:

- XML-i tõlgendamine Java koodi. XML konfiguratsioonis võib esineda elemente, mida ei ole võimalik otse Javasse tõlkida. Näiteks nõuab mõne XML märgendi üleviimine uue Java klassi loomist, mis omakorda peab kindlat klassi laiendama. Seejärel tuleb üle kirjutada vastav meetod, mille sees vajalik seadistus teha.

- Võimalus saada sama tulemus erinevate kirjapiltide korral. Programmi jaoks on vaja ühtset esitusviisi, et oleks võimalik igal juhul sama tulemuseni jõuda. Projekti käsitsi üleviimise käigus selgus, et ühtne esitusviis ei ole alati reaalsus. Näiteks on XML-is võimalik kujutist (*map)* esitada kahel viisil (Joonised 6 ja 7).

```
<map>
    <entry>
        <key>
            <util:constant static-field="javax.xml.bind.Marshaller.JAXB_FORMATTED_OUTPUT"/>
        </key>
        <value type="java.lang.Boolean">true</value>
    </entry>
</map>
```

Joonis 6. Kujutis staatilise võtmega.

```
<map>
    <entry key="testKey" value="testValue"/>
</map>
```

Joonis 7. Tavapärane kujutis.

- Väärtuste tüüpide määramine. Esineb olukordi, kus märgendi juurde on kirjutatud, millise andmetüübiga on tegemist või on täpselt teada, et konkreetse välja puhul

on alati tegemist näiteks sõne andmetüübiga. Enamikel juhtudel ei ole võimalik üheselt määrata, millise andmetüübiga on tegemist, mistõttu ei ole võimalik töötavat Java koodi genereerida.

4.3 Programmi analüüs

Käesoleva töö raames tegeleb autor kolme etapi automatiseerimisega. Igas etapis toimuv automatiseerimine on ennekõike arendaja töö lihtsustamiseks, mitte töö ära tegemiseks. Töötava tulemuse saamiseks peab arendaja tegema lõpetavad viimistlused.

4.3.1 Maven

Spring Bootile omane `pom.xml` jaguneb kolmeks:

- *Parent* POM
- `Build` pistik
- *Starter* POM-id

Töö käigus valmiv programm peab olema võimeline leidma projektist `pom.xml` nimelise faili. Seejärel tuleb lisada *parent* POM ja *build* pistik. Kolmas samm on sõltuvuste uuendamine. Kõigepealt üritatakse leida kõik sõltuvused, mida on võimalik asendada *starter* POM-i abil. Ülejäänud sõltuvuste jaoks küsitakse Maveni API käest uusimad versioonid ning liigutatakse need eraldi faili. Eraldi faili liigutamine tuleneb asjaolust, et autori kasutatavas projektis tekkis suur hulk selliseid sõltuvusi, mida projekt ei vajanud.

4.3.2 Projekti struktuur

Projekti käivitamiseks Spring Bootiga, tuleb lisada rakendusse `main()` meetodit sisaldav klass, mis on defineeritud projekti puus enne teisi Java klasse (Joonis 1). Faili loomiseks peab `java` kaust asuma „`src/main/java`“, et oleks võimalik üles leida `java` kaust. Kui kaust on käes, peab programm olema võimeline leidma sobiva asukoha `main()` meetodi loomiseks, olenemata pakettide arvust.

4.3.3 Java *beanide* loomine

Bean’ide kohta on teada, et XML konfiguratsioonid asuvad */resources/* nimelises kaustas ning *bean*’id võivad olla defineeritud mitmetes XML failides.

Esimese sammuna tuleb üles otsida eelnimetatud XML failid. Otsimine toimub rekursiivselt `.xml` faili lõppu otsides. Sedasi on võimalik leida ka projekti puus sügavamal asuvad failid.

Järgmisena hakatakse otsima XML failidest `<bean>` märgendeid. *Bean*'i leidmisel leitakse tema klass, nimi ja väärtused. Viimaseid võib olla erinevaid tüüpe:

- Tavaline juhtum. On antud välja nimi ja väärtus (Joonis 1).
- Kogumid. Erinevate kogumitena on esindatud:
    - Loend
    - Hulk
    - Kujutis (Joonis 7)
- *Properties* objekt. Tegemist on Java objektiga, kuhu on võimalik lisada võti-väärtus paare sõnedena (Joonis 8).

```
<property name="properties">
    <props>
        <prop key="location.field.props">/WEB-INF/bundles/Field</prop>
    </props>
</property>
```

Joonis 8. *Property* näide.

Viimase sammuna tuleb luua konfiguratsiooni klass, kuhu sisse kirjutatakse saadud *bean*'id Java kujul.

4.3.4 Vigade haldus

Valmiva programmi näol on tegemist käsurearakendusega, mistõttu kuvatakse kogu informatsioon konsooli. Kriitilise vea saamisel lõpetab programm oma töö ning jätkamiseks tuleb programm uuesti käivitada.

Sisemiselt suudab rakendus tuvastada mõningaid olukordi, kus programm ei osanud korrektselt andmeid kätte saada. Näiteks kui Java *bean*'i loomisel leiab programm, et mõned väärtused on puudu, lisatakse tulemusse kommentaar, kuhu on märgitud koos lühikese selgitusega „TODO:".

4.3.5 Testimine

Rakenduse testimisel on eesmärk testida programmi korrektset käitumist, mitte saadud lõpptulemust. Kuna programmi eesmärk on arendaja töö lihtsustamine ja käesoleva töö raames ei ole oluline, et tulemus oleks kohe täielikult töötav, ei saa testida rakenduse tööd enne ja pärast programmi käivitamist.

Testimisel arvestatakse programmi erinevad mooduleid ja funktsionaalsust. Kasutusel on JUnit raamistik, mille abil koostatakse automaattestid, mis käivitatakse programmi JAR faili genereerimisel. Programmi testimine näeb ette järgnevaid samme:

- Maven
    - *Parent* POM lisamine
    - *Build* pistiku lisamine
    - *Starter* POM-ide lisamine. Siin testitakse ennekõike funktsionaalsust, kus programm asendab ära sellised sõltuvused, mis on võimalik saada *starter* POM-idest, ning sealjuures eemaldab sellised sõltuvused, mis on juba olemasoleva *starter* POM-iga kaasa tulnud. Testi, mida kasutatakse sobiva starteri leidmiseks on näha Joonis 9 peal.

```
@Test
void findReplacementTest() {

    Dependency dependency = new Dependency()
    dependency.setGroupId("org.hibernate")
    dependency.setArtifactId("Hibernate-validator")

    Map<String, List<String>> startersMap = new HashMap<>()
    new File('starters').eachLine { line ->
        String[] splitted = line.toLowerCase().trim().split(",")
        startersMap.put(splitted[0], splitted[1..splitted.length - 1] as List<String>)
    }

    assertEquals("spring-boot-starter-validation", dependencyConverter.findReplacement(dependency, startersMap))
}
```

Joonis 9. Sobiva *starter* POM-i leidmise test.

- Projekti struktuur
    - Testitakse, kas programm on võimeline leidma korrektset asukohta `main()` meetodi jaoks. Selleks luuakse ajutisi kaustade struktuure, kuhu lisatakse Java klasse ja seejärel lastakse programmil tagastada `main()` meetodit sisaldava klassi asukoht. Näitena on toodud Joonis 10.

```
File createMockProject() {
    File dir = new File("new3")
    dir.mkdir()

    new File("new3/src/main/java/folder/domain").mkdirs()
    new File("new3/src/main/java/folder/domain/controller").mkdir()
    new File("new3/src/main/java/folder/domain/pack/service").mkdirs()
    new File("new3/src/main/java/folder/domain/dir").mkdir()

    new File("new3/src/main/java/folder/domain/controller/a.java").createNewFile()
    new File("new3/src/main/java/folder/domain/pack/service/b.java").createNewFile()
    new File("new3/src/main/java/folder/domain/c.txt").createNewFile()
    new File("new3/src/main/java/folder/domain/dir/d.java").createNewFile()

    dir
}

@Test
void findRootFolderForMainTest() {
    File dir = createMockProject()

    File expected = new File(dir.getAbsolutePath() + "/src/main/java/folder/domain")
    assertEquals(expected, sc.findRootFolderForMain(dir.getAbsolutePath()))

    dir.deleteDir()
}
```

Joonis 10. `main()` meetodiga klassi loomise asukoha test.

- XML-ilt Java-le üleminek
    - XML failide leidmine. Siin kasutatakse ära juba eelnevalt kirjeldatud funktsionaalsust kaustade loomiseks.
    - *Bean*'ide leidmine. Selle jaoks luuakse XML fail koos *bean*'idega ja testitakse, kas programm tagastab sama arvu *bean*'e.
    - Erinevate *bean*'i sätete tuvastamine. Peatükis 4.3.3 tõi autor välja erinevad võimalused *bean*'i sätete esitamiseks. Programm peab olema võimeline nimetatud sätteid tuvastama.
    - Korrektse Java koodi genereerimine XML põhjal. Testitakse etteantud XML *bean*'i ja vastavust saadud Java *bean*'ile.

4.4 Programmi kasutamine

Tulenevalt automatiseerimise etappideks jagamisest on ka programmi töö jagatud kolmeks komponendiks. Esimene komponent tegeleb Maven'iga, teine projekti struktuuriga ja kolmas *bean*'idega. Tänu programmi modulaarsusele on võimalik käivitada komponente eraldi või tervet programmi korraga. Selleks tuleb käivitada „boot-converter.jar" nimeline fail. Linuxi keskkonnas näeb käivitamine välja järgnevalt:

```
java -cp boot-converter.jar Application /home/rannar/projectname
```

Ühe komponendi käivitamiseks tuleb käsule otsa lisada *-m,* millele järgneb tühik ja soovitav komponendi nimi (*maven/structure/bean).* Programmikood on kättesaadav aadressil: https://bitbucket.org/rannarallorg/boot_converter.

4.5 Võimalikud edasiarendused

Käesoleva bakalaureusetöö raames oli automatiseerimise skoop küllaltki kitsas ja edasiarendusteks on võimalusi mitmeid. Peamine edasiarenduse koht on XML-i üleviimine Java-le. Näiteks tuleb leida kõik võimalused XML kujul Springi konfigureerimiseks ja nende Java vasted. Selleks tuleks Springi dokumentatsioonist otsida üles kõik võimalused, mida pakutakse Springi XML kujul konfigureerimiseks. Seejärel tuleks leida kuidas on võimalik samu konfiguratsioone esitada Java abil. Tulemusena oleks võimalik tekitada programm, mis suudab analüüsida XML faile ja nende põhjal genereerida Java konfiguratsiooni. Samm kaugemale minnes oleks võimalik arvestada ka selliseid seadistusi, mis Spring Booti poolt on juba vaikimisi määratud.

Kasutajaliideste arendus on kindlasti märkimisväärne suund. Juhul kui käesolev programm võetakse kasutusele mõnes ettevõttes, tuleks aega panustada ka kasutajaliidese arendusele, et luua võimalikult lihtne töö protsess. Kasutajaliideste alla võib kuuluda nii graafilise kasutajaliidese loomine kui ka tehtud töö raportite koostamine.

Kui ühendada graafiline kasutajaliides ja parendatud programm, mis suudab kõikide pakutavate konfiguratsioonidega tegeleda, oleks võimalik luua viisard, mille abil saab kasutaja Spring rakenduse Spring Bootile üle viia ilma ühegi koodirea kirjutamiseta. Viisardi põhimõte seisnes selles, et ülemineku käigus võib tekkida olukordi, kus programm ei saa leida õiget käitumist ja oleks vaja kasutaja sisendit. Näiteks kui programm leiab mõne Maveni sõltuvuse, mis tundub ebavajalik, siis saab kasutaja anda

nõusoleku sõltuvuse eemaldamiseks või keelduda kustutamisest. Samuti võib kasutaja lahendada peatükis 4.2.2 väljatoodud andmetüüpide probleemi, kus programm ei suuda tuvastada, millist andmetüüpi tuleb Java koodi genereerimisel kasutada.

5 Kokkuvõte

Käesoleva töö eesmärgiks oli töötavas Spring rakenduses kasutusele võtta Spring Boot ning selle protsessi automatiseerimine. Töö käigus selgus, et protsessi täielik automatiseerimine ei ole võimalik ja lõpptulemuse saamiseks on vaja arendaja sekkumist, kuna programmil on raske tuvastada funktsionaalsust, mida tuleb eemaldada või juurde lisada. Samuti on keeruline XML-i põhjal töötava Java koodi genereerimine, kui XML-i ja Javat ei saa panna üks-ühele vastavusse. Kirjeldati Spring Booti peamiseid eeliseid ning selgitati, millal on olemasoleva projekti üleviimine otstarbekas.

Töö raames valmis käsureaprogramm, mis kohandab Maveni `pom.xml` faili, projekti struktuuri ja *bean*'ide konfiguratsiooni Spring Bootile vastavaks ning seeläbi kiirendab ülemineku protsessi. Lisaks sellele valmis Spring Booti kasutuselevõtu juhend, millest on tulevikus võimalik teiste rakenduste korral lähtuda. Töö tulemus on abiks arendajatele, kes tegelevad mõne vanema Spring projektiga ning soovivad tehnoloogiat Spring Bootile uuendada. Seatud eesmärkidest sai täidetud nii juhend kui ka automatiseerimise osaline lahendus.

Edasiarenduse raames on eesmärk uurida lähemalt Springi ja Spring Booti konfigureerimise võimalusi ning leida võimalusi automatiseeriva programmi parendamiseks. Suurim potentsiaal on programmi moodulil, mis tegeleb XML konfiguratsiooni üleviimisega Java konfiguratsioonile. Seal on võimalik parendada *bean*'ide loomise võimalusi ning edasiarendusena implementeerida ka ülejäänud konfiguratsiooni üleviimine. Tulemusena võiks valmida programm, mille tööd tuleb arendajal ainult kontrollida, ilma suuremaid muudatusi tegemata.

Kasutatud kirjandus

[1] „Spring Framework Reference Documentation,“ [Võrgumaterjal]. Available: http://docs.spring.io/spring/docs/current/spring-framework-reference/htmlsingle/. [Kasutatud 2016].

[2] „Spring Boot Reference Guide,“ [Võrgumaterjal]. Available: http://docs.spring.io/spring-boot/docs/current-SNAPSHOT/reference/htmlsingle/. [Kasutatud 2016].

[3] „Web framework rankings,“ [Võrgumaterjal]. Available: http://hotframeworks.com/#top-frameworks. [Kasutatud 2016].

[4] „Apache Maven Project,“ [Võrgumaterjal]. Available: https://maven.apache.org/. [Kasutatud 2016].

[5] „Top 4 Java Frameworks,“ [Võrgumaterjal]. Available: http://zeroturnaround.com/rebellabs/top-4-java-web-frameworks-revealed-real-life-usage-data-of-spring-mvc-vaadin-gwt-and-jsf/. [Kasutatud 2016].

[6] „Spring Boot System Requirements,“ [Võrgumaterjal]. Available: http://docs.spring.io/spring-boot/docs/current/reference/html/getting-started-system-requirements.html. [Kasutatud 05 2016].

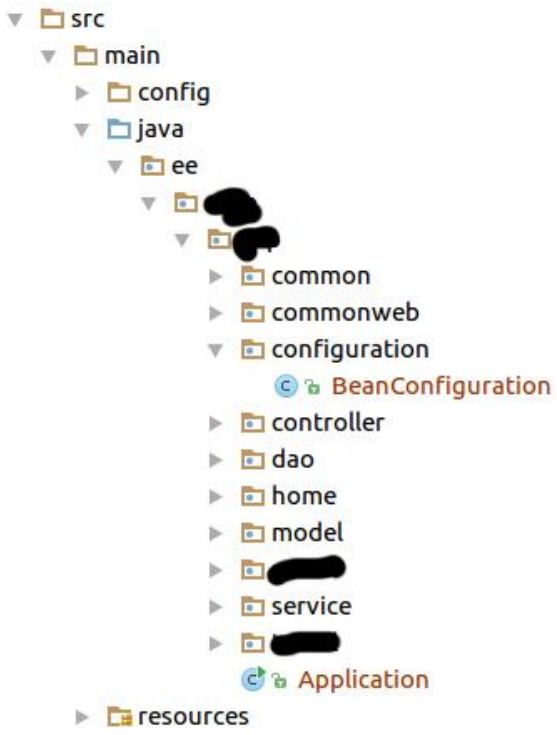
[7] „Gradle Introduction,“ [Võrgumaterjal]. Available: https://docs.gradle.org/current/userguide/introduction.html. [Kasutatud 2016].

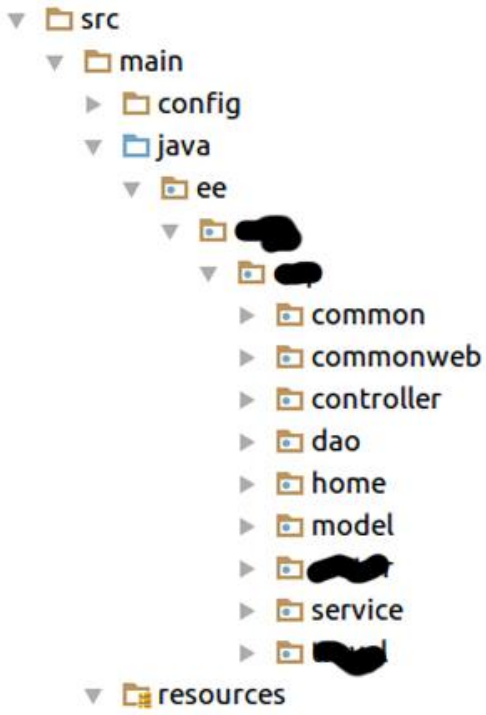
[8] „Spring Boot Build systems,“ [Võrgumaterjal]. Available: http://docs.spring.io/spring-boot/docs/current/reference/html/using-boot-build-systems.html. [Kasutatud 05 2016].

[9] „Serving Static Web Content with Spring Boot,“ [Võrgumaterjal]. Available: https://spring.io/blog/2013/12/19/serving-static-web-content-with-spring-boot. [Kasutatud 05 2016].

Lisa 1 – Projekti `src` kaust enne ja pärast

Vasakul on kujutatud projekti `src` kausta peale programmi käivitamist ja paremal on enne käivitamist.





Lisa 2 – *Bean*'i konfiguratsiooni näide

XML kujul *bean*'i konfiguratsioon:

```
<!-- Enable autowired loggers -->
<bean class="ee[redacted]common.utils.AnnotationLoggerBeanPostProcessor" />
<!-- Resolves exceptions -->
<bean id="exceptionResolver" class="ee[redacted]handlers.ExceptionResolver">
    <property name="exceptionMappings">
        <map>
            <entry key="java.lang.Throwable" value="redirect:/error/any" />
            <entry key="ee[redacted]exceptions[redacted] value="redirect:[redacted] />
        </map>
    </property>
</bean>
```

Sama *bean* Java-kujul:

```
@Configuration
public class BeanConfiguration {

@Bean
public AnnotationLoggerBeanPostProcessor annotationLoggerBeanPostProcessor() {
    return new AnnotationLoggerBeanPostProcessor();
}

@Bean
public ExceptionResolver exceptionResolver() {
    ExceptionResolver exceptionResolver = new ExceptionResolver();
    // TODO: check types
    Map<String, String> tempMap = new HashMap<String, String>();
    tempMap.put("java.lang.Throwable", "redirect:/error/any");
    tempMap.put("ee[redacted]common.exceptions[redacted]", "redirect:[redacted]");
    exceptionResolver.setExceptionMappings(tempMap);
    return exceptionResolver;
}
```