Универзитет у Београду

Математички факултет

Мастер рад

Електронске лекције о радном оквиру Spring

Ана Нитковић 1047/2017

Ментор

др Мирослав Марић

Септембар 2021.

Садржај

Увод **1**

1 Електронске лекције **2**

2 О радном оквиру *Spring* **4**
2.1 *Spring* пројекти 4

3 Подешавање окружења **8**

4 Контејнер и зрна **10**
4.1 Креирање и дохватање зрна 11
4.2 Опсег зрна 12
4.2.1 Опсег `singleton` 13
4.2.2 Опсег `prototype` 14

5 Уметање зависности **15**
5.1 Аутоматско уметање зависности 17
5.1.1 Уметање зависности преко конструктора класе 18
5.1.2 Уметање зависности преко `set` метода 19
5.1.3 Уметање зависности преко поља класе 19
5.1.4 Проблеми приликом аутоматског уметања зависности . . 19
5.2 Уметање зависности - *Java* конфигурација 21
5.3 Уметање зависности - `xml` конфигурација 22
5.3.1 Уметање зависности преко конструктора класе 23
5.3.2 Уметање зависности преко `set` метода 24

6 Аспектно оријентисано програмирање **25**
6.1 Дефинисање аспеката - *Java* конфигурација 26
6.2 Дефинисање аспеката - `xml` конфигурација 28

7 Радни оквир *Spring Boot* **31**

8 Апликација *Spring Initializr* **32**

9 Радни оквир *Spring MVC* **35**

Закључак **46**

Литература **47**

Увод

Развој интернета довео је до тога да је интернет постао основни начин дељења и приступа информацијама. Све је већи број платформи за учење путем интернета и учење на такав начин постаје све популарније. Једна од таквих платформи је и еШкола веба[1]. У оквиру платформе може се приступити великом броју курсева из области веб технологија који су бесплатни и јавно доступни. Међу њима се налази и курс о радном оквиру *Spring*, који је креиран за потребе овог рада.

Програмски језик *Java* је због своје једноставности и независности од платформе на којој се извршава, као и разноврсних примена један од најпопуларнијих програмских језика. Због тога се јавила потреба за развојем алата и радних оквира који ће развој апликација у програмском језику *Java* учинити једноставнијим. *Spring* је један од најпопуларнијих радних оквира за програмски језик *Java*. Чини га скуп екстензија и библиотека које нуде решења за најчешће проблеме у развоју софтвера. На тај начин је омогућено програмерима да се фокусирају на решавање проблема у вези са самом логиком апликације. *Spring* је модуларан радни оквир са минималном зависношћу између модула. Својом структуром изузетно доприноси писању чистог кода који се лако тестира и одржава, што представља решење за један од највећих изазова у писању великих апликација.

Spring је радни оквир који се константно развија, прати развој програмског језика *Java* и има велику базу корисника који га унапређују, што доприноси његовој једноставности и популарности.

Циљ овог рада је упознавање са основним принципима на којима се радни оквир *Spring* заснива. На почетку рада биће приказан кратак преглед *Spring* пројеката. Затим ће детаљно ће бити обрађене две теме, уметање зависности и аспектно оријентисано програмирање, које су кључне за разумевање свих *Spring* пројеката. Након тога биће представљен пројекат *Spring Boot*, а затим ће бити обрађен модул *Spring MVC* који се користи за израду веб апликација. Ради лакшег разумевања, сви појмови и њихова објашњења биће праћени одговарајућим примерима.

На крају рада биће приказана апликација која обухвата обрађене концепте.

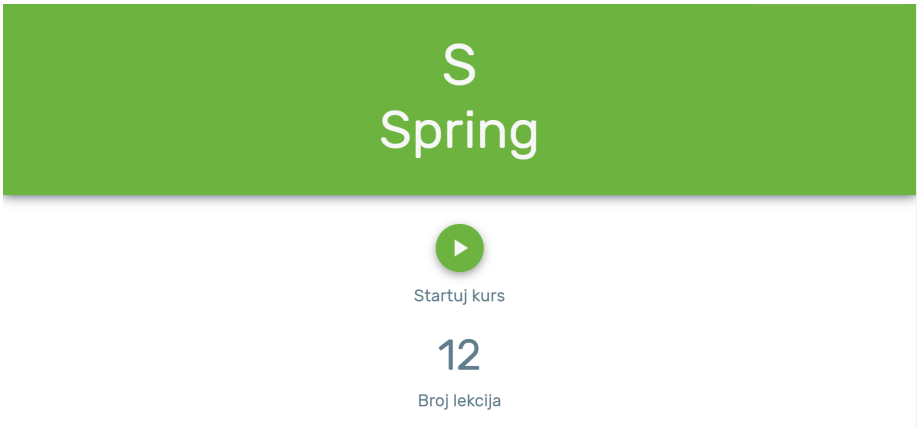
[1] `http://edusoft.matf.bg.ac.rs/eskola_veba/`

1 Електронске лекције

Платформа еШкола веба је електронска школа веб програмирања на којој се налазе курсеви о популарним веб технологијама. Сваки курс је организован у виду електронских лекција, које су бесплатне и доступне на српском језику. Изглед почетне стране платформе приказан је на слици 1. Више о платформи еШкола веба се може видети у [1].

Слика 1: *Почетна страна платформе еШкола веба*

Међу доступним курсевима налази се и курс о радном оквиру *Spring*[2]. Курс је намењен свима који желе да науче како да користе радни оквир *Spring*, при чему је посебна пажња посвећена програмирању веб апликација. Приликом покретања курса отвара се страница са слике 2.



Слика 2: *Насловна страна курса о радном оквиру* Spring

У оквиру курса о радном оквиру *Spring* креиране су следеће електронске лекције:

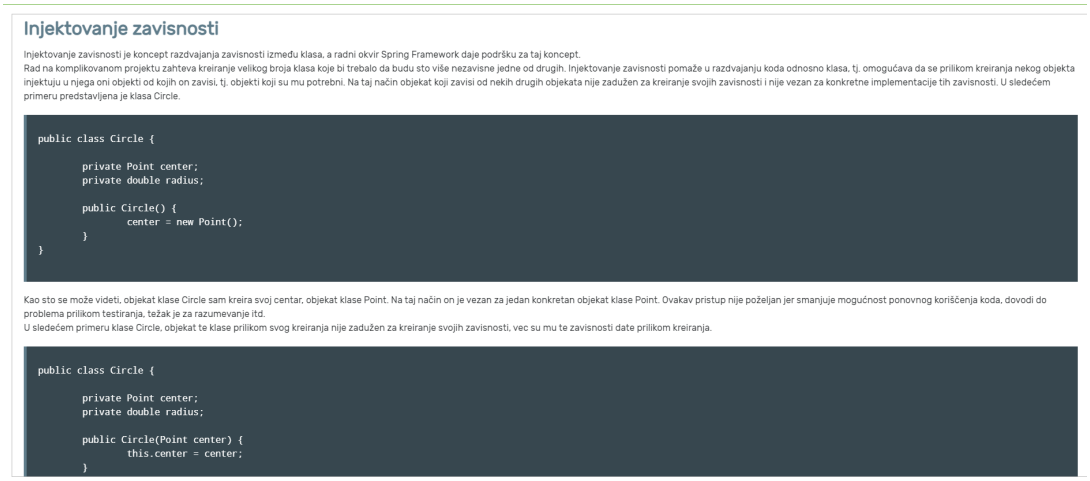
- Радни оквир *Spring*;
- Подешавање окружења;

[2] `http://edusoft.matf.bg.ac.rs/eskola_veba/#/course-details/spring`

- Контејнер и зрна;
- Уметање зависности;
- Уметање зависности - `xml` конфигурација;
- Уметање зависности - *Java* конфигурација;
- Аутоматско уметање зависности;
- Аспектно оријентисано програмирање;
- Дефинисање аспеката - *Java* анотације;
- Дефинисање аспеката - `xml` конфигурација;
- Радни оквир *Spring Boot*;
- Апликација *Spring Initializr*;
- Радни оквир *Spring MVC*.

Лекције су пажљиво осмишљене, како би корисници могли детаљно да разумеју основне принципе на којима се радни оквир *Spring* заснива. Сви концепти и појмови су детаљно објашњени и праћени су одговарајућим примерима. Из тог разлога, лекције би требало да представљају добру основу за наставак учења радног оквира *Spring*.

Изглед једне од доступних лекција приказан је на слици 3.



Слика 3: *Лекција у оквиру курса*

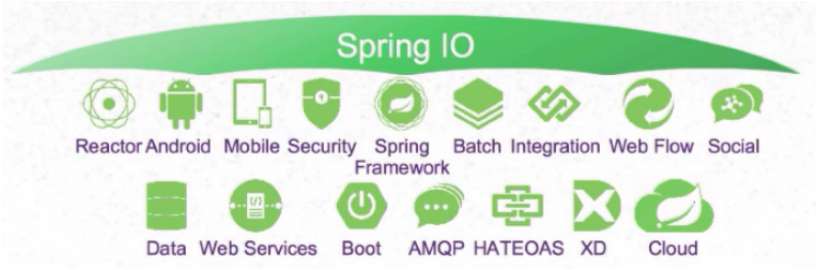
2 О радном оквиру *Spring*

Spring је један од најпопуларнијих радних оквира за програмски језик *Java*. *Spring* је радни оквир отвореног кода, потпуно је бесплатан за коришћење и поседује велики број корисника који га константно унапређују. Изворни кôд је доступан на платформи *GitHub*[3].

Творац радног оквира *Spring* је *Rod Jonhson*[4], а прва верзија објављена је у октобру 2002. године [2]. Направљен је са циљем да значајно олакша развој *Java* апликација. Од тада се константно развија, чини га велики број радних оквира који дају решења за све познате проблеме у развоју софтвера. Без обзира на тип апликације коју је потребно направити, постоји *Spring* пројекат који може олакшати развој такве апликације. *Spring* је модуларан радни оквир, а зависност између модула је минимална, што омогућава коришћење само оног дела радног оквира који одговара потребама пројекта који се развија. *Spring* је такође неинвазиван радни оквир, осим *Spring* анотација не постоји много индикација да је у питању *Spring* апликација.

На званичном сајту[5] може се пронаћи комплетна *Spring* документација, линкови за преузимање *Spring* библиотека, као и разни туторијали и примери апликација писаних помоћу радног оквира *Spring*.

На слици 4 дат је приказ свих *Spring* пројеката.



Слика 4: Spring *пројекти*

2.1 *Spring* пројекти

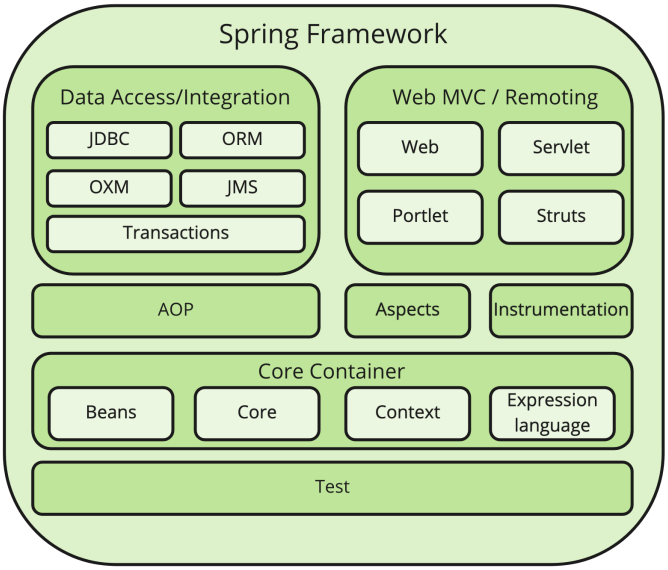
Радни оквир *Spring Framework*

Радни оквир *Spring Framework* представља основу фамилије радних оквира *Spring*. Овај радни оквир садржи основне библиотеке на којима се заснивају сви други радни оквири. Састоји се од 20 различитих модула чији шематски приказ је дат на слици 5.

[3] `https://github.com/spring-projects`

[4] *Roderick Rod Johnson* је познати Аустралијски информатичар. `https://en.wikipedia.org/wiki/Rod_Johnson_(programmer)`

[5] `https://spring.io`



Слика 5: *Шематски приказ радног оквира* Spring Framework

- *Core Spring Container* - Централни део радног оквира *Spring Framework* јесте *Spring* контејнер. Његов задатак јесте креирање и повезивање објеката који чине једну апликацију, као и управљање комплетним животним циклусом тих објеката. У основи контејнера јесте принцип уметања зависности.

- *AOP Module* - *Spring* даје подршку за аспектно оријентисано програмирање, што омогућава да се функционалности које прожимају апликацију одвоје од објеката на које се примењују.

- *Data Access and Integration* - Рад са библиотеком *JDBC*[6] доводи до писања већег броја линија кода, као и кода који се понавља. Овај модул олакшава приступ подацима, као и комуникацију између бизнис слоја апликације и слоја презистенције. Такође, нуди слој апстракције око *Java* библиотека које се користе за асинхрону интеграцију са другим апликацијама.

- *Web and Remoting* - За развој веб апликација најчешће се користи *Model View Controller* (*MVC*) архитектура, која раздваја део апликације који се односи на кориснички интерфејс од дела који садржи логику апликације. У оквиру овог модула налази се подршка за такву архитектуру.

- *Instrumentation* - Овај модул даје подршку за додавање агената у *JVM*[7].

- *Testing* - Модул за тестирање *Spring* апликација, који нуди подршку за писање јединичних и интеграционих тестова.

[6] *Java Database Connectivity*, `https://en.wikipedia.org/wiki/Java_Database_Connectivity`

[7] *Java Virtual Machine*, `https://en.wikipedia.org/wiki/Java_virtual_machine`

Радни оквир *Spring Boot*

Као што радни оквир *Spring Framework* има циљ да олакша развој *Java* апликација, тако *Spring Boot* пројекат има циљ да олакша развој апликација писаних помоћу *Spring* радних оквира. *Spring Boot* омогућава креирање *Spring* апликације коју је могуће брзо покренути. *Spring Boot* аутоматски конфигурише како *Spring* библиотеке, тако и све друге библиотеке које је могуће интегрисати са радним оквиром *Spring*. *Spring Boot* такође пружа почетне зависности (библиотеке) прилагођене потребама апликације.

Другим речима, *Spring Boot* садржи уобичајене конфигурације свега што је програмеру *Spring* апликације потребно, пружајући потпуно функционалну апликацију, коју је само потребно надоградити и прилагодити својим потребама.

Радни оквир *Spring Data*

Spring Data је радни оквир који олакшава рад са релационим и нерелационим базама података. *Spring Data* олакшава интеграцију са другим радним оквирима за перзистенцију, од којих је најпопуларнији *Hibernate*[8]. Једна од многих занимљивих карактеристика пројекта јесте и аутоматско генерисање упита на основу имена метода.

Радни оквир *Spring Cloud*

Spring Cloud је пројекат који се надограђује на *Spring Boot* и омогућава брз развој дистрибуираних система, омогућујући корисницима лак развој и подизање сервиса/апликација у дистрибуираним системима.

Радни оквир *Spring Cloud Data Flow*

Spring Cloud Data Flow је радни оквир за процесирање серије или тока података у архитектурама базираним на микросервисима[9].

Радни оквир *Spring Security*

Spring Security представља радни оквир за аутентификацију и контролу приступа, који значајно олакшава имплементацију сигурносних аспеката *Java* апликација. Поред разноврсне подршке за аутентификацију и ауторизацију, *Spring Security* пружа и подршку за заштиту од уобичајених начина за злонамеран хакерски напад, као и имплементацију сигурносних протокола и стандарда. *Spring Security* се базира на принципу аспектно оријентисаног програмирања које је детаљно објашњено у оквиру овог рада.

[8] `https://hibernate.org/`

[9] `https://sr.wikipedia.org/sr-ec/Mikroservisi`

Радни оквир *Spring Integration*

Spring Integration пројекат има за циљ олакшавање интеграције пословних апликација пружајући подршку за имплементацију устаљених образаца у овом домену. Овај радни оквир омогућава лагани систем за размену порука између *Spring* апликација, али и интеграцију са спољним системима користећи концепт декларативних адаптера.

Радни оквир *Spring Web Flow*

Spring Web Flow се надограђује на *Spring MVC* (који је део радног оквира *Spring Framework*) и пружа подршку за лак развој веб апликација које имају контролисан ток, односно низ одређених корака који који воде ка извршавању неког пословног задатка. Пример овакве апликације може бити апликација за куповину путем интернета или пријављивање на лет, а заједничка им је потреба за одржавањем стања.

Радни оквир *Spring Session*

Spring Session пројекат пружа *API*[10] и имплементацију управљања корисничким сесијама, са циљем да превазиђе ограничења *HTTP*[11] сесија. Предност овог пројекта јесте то да управљање сесијом није везано за конкретну имплементацију контејнера.

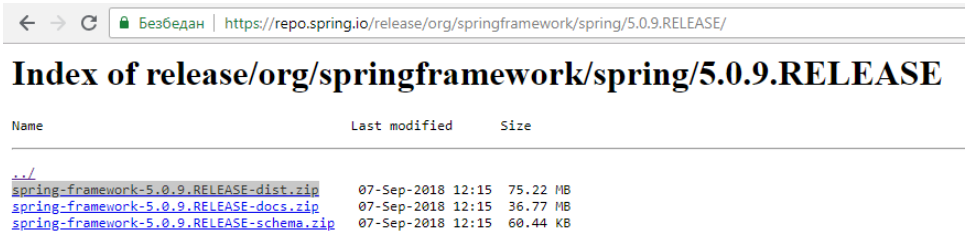
[10] *Application Programming Interface* - програмски интерфејс апликације

[11] *Hypertext Transfer Protocol* је мрежни протокол који представља најчешћи метод преноса информација на интернету

3 Подешавање окружења

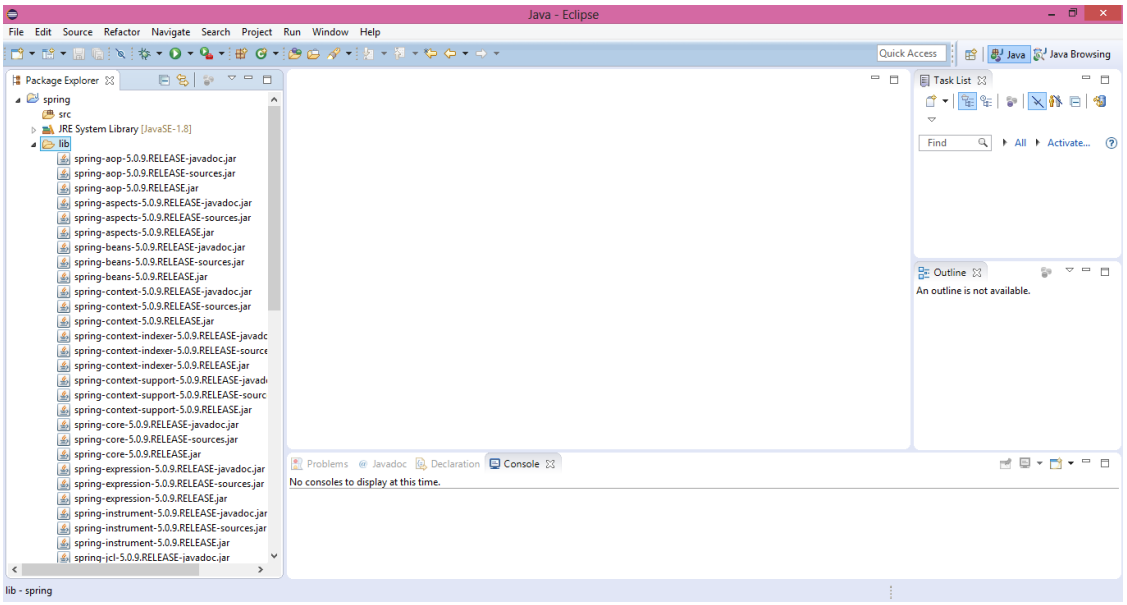
У овом поглављу биће представљен поступак подешавања окружења за коришћење радног оквира *Spring Framework* у развојном окружењу *Eclipse*[12].

Како би се користио радни оквир *Spring Framework*, потребно је преузети одговарајуће `jar` фајлове, а затим их додати у *Eclipse* пројекат.



Слика 6: *Верзије радног оквира* Spring Framework

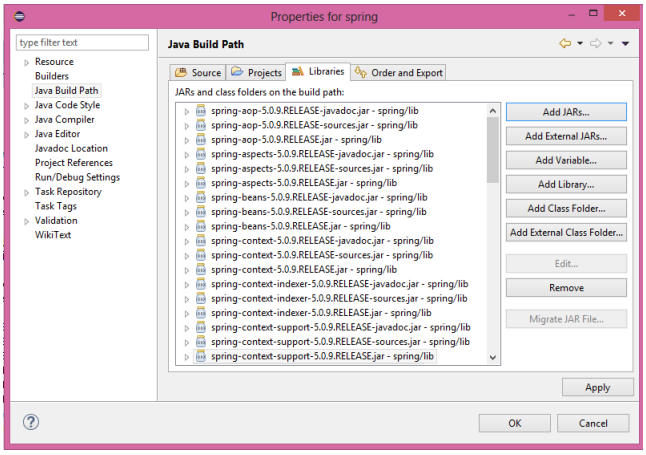
Потребно је приступити веб страници која представља *Spring* репозиторијум[13] (слика 6), у оквиру кога се налазе све досадашње верзије. Препоручује се одабир последње доступне верзије, која је уједно и последња стабилна верзија радног оквира *Spring Framework*.



Слика 7: *Додавање* `jar` *фајлова у* Eclipse *пројекат*

Преузету `zip` датотеку је потребно распаковати и `jar` фајлове из фолдера `libs` копирати у *Eclipse* пројекат. У оквиру *Eclipse* пројекта неопходно је направити фолдер `lib` и у њега копирати `jar` фајлове, што је илустровано на слици 7.

[12] `https://www.eclipse.org/`
[13] `https://repo.spring.io/ui/native/release/org/springframework/spring/`



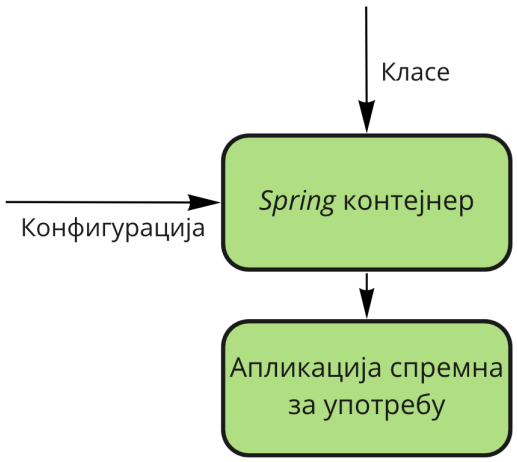
Слика 8: *Библиотеке радног оквира* Spring Framework

На слици 8 приказане су библиотеке радног оквира *Spring Framework* након успешног додавања у *Eclipse* пројекат. Када су `jar` фајлови додати могуће је користити све функционалности радног оквира *Spring Framework*.

4 Контејнер и зрна

Свака комплексна апликација састоји се од великог броја објеката који комуницирају једни са другима како би успешно обавили своје задатке.

У основи радног оквира *Spring Framework* налази се контејнер (енг. *container*) који је задужен за креирање и повезивање објеката који чине једну апликацију. Објекти нису задужени за креирање других објеката од којих зависе, већ им контејнер даје те објекте приликом њиховог стварања. Објекти који живе у контејнеру и којима управља *Spring* називају се зрна (енг. *beans*). Како би контејнер могао да креира зрно потребно је да му се дају информације о животном циклусу зрна, као и информације о његовим зависностима. Илустрација *Spring* контејнера дата је на слици 9.



Слика 9: *Spring контејнер*

Интерфејс `org.springframework.context.ApplicationContext` представља *Spring* контејнер. Контејнер у радном оквиру *Spring Framework* има неколико имплементација:

- `AnnotationConfigApplicationContext` - учитава контекст апликације из једне или више конфигурацијских *Java* класа;
- `AnnotationConfigWebApplicationContext` - учитава контекст веб апликације из једне или више конфигурацијских *Java* класа;
- `ClassPathXmlApplicationContext` - учитава контекст из једне или више `xml` datoteka које се налазе на датој путањи;
- `FileSystemXmlApplicationContext` - учитава контекст из једне или више `xml` датотека смештених на фајл систему;
- `XmlWebApplicationContext` - учитава контекст из једне или више `xml` датотека смештених у оквиру веб апликације.

Могуће је користити више конфигурацијских датотека или класа које одговарају различитим слојевима или модулима апликације.

4.1 Креирање и дохватање зрна

Spring нуди неколико начина дефинисања информација које су потребне контејнеру како би успешно креирао зрна. Како би се разумео концепт контејнера и његове карактеристике, у овом поглављу биће приказан традиционални начин дефинисања зрна коришћењем `xml` датотеке. Осим `xml` датотека, конфигурацијске податке је могуће обезбедити коришћењем *Spring* анотација и *Java* класа, који ће бити представљени кроз поглавља који се односе на уметање зависности и аспектно оријентисано програмирање. Такође, биће наведене предности и мане сва три начина дефинисања конфигурацијских података.

У конфигурацијској `xml` датотеци информације о зрнима чувају се у оквиру етикете `<beans>`, а информације о појединачним зрнима чувају се у оквиру етикете `<bean>`. У примеру 1 атрибут `class` у етикети `<bean>` дефинише класу чији објекат ће бити креиран од стране *Spring* контејнера. Атрибут `id` на јединствен начин одређује зрно.

Пример 1: Конфигурацијска `xml` датотека

```
<?xml version = "1.0" encoding = "UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:context="http://www.springframework.org/schema/context"
xmlns:aop="http://www.springframework.org/schema/aop"
xsi:schemaLocation="http://www.springframework.org/schema/aop
http://www.springframework.org/schema/aop/spring-aop.xsd
http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.xsd
http://www.springframework.org/schema/context
http://www.springframework.org/schema/context/spring-context.xsd">

   <bean id = "..." class = "..."> </bean>

</beans>
```

У примеру 2 је приказана класа `Student`, која има три поља - `brojIndeksa`, `ime` и `prezime`, као и метод `toString()` који исписује податке о студенту.

Пример 2: Класа `Student`

```
public class Student {
    private int brojIndeksa;
    private String ime;
    private String prezime;

    public String toString() {
      return "Student " + ime + " " + prezime + ", sa brojem indeksa " +
          brojIndeksa;
    }
}
```

Како би контејнер креирао објекат класе `Student`, потребне информације се налазе у оквиру `xml` датотеке `beans.xml` приказане у примеру 3. Етикета `<property>` у оквиру етикете `<bean>` има атрибуте `name` и `value` који служе за доделу вредности пољима класе `Student`. Вредност атрибута `name` јесте назив поља класе, а атрибут `value` садржи вредност тог поља.

Пример 3: Конфигурацијска датотека са подацима о објекту класе `Student`

```
<beans xmlns="http://www.springframework.org/schema/beans" ... >
    <bean id="student" class="Student">
        <property name="brojIndeksa" value="20171047"/>
        <property name="ime" value="Pera"/>
        <property name="prezime" value="Peric"/>
    </bean>
</beans>
```

У примеру 4, у оквиру `main` метода *Spring* апликације прво се креира контејнер који прави објекат класе `Student` чији се подаци налазе у `xml` датотеци[14]. Затим се помоћу методе `getBean()` дохвата тај објекат, а затим се коришћењем методе `toString()` исписују подаци о студенту.

Пример 4: Позив метода `toString()` класе `Student`

```
public static void main (String [] args) {
    ApplicationContext context =
                  new ClassPathXmlApplicationContext("beans.xml");
    Student student = (Student) context.getBean("student");
    System.out.println(student.toString());
}
```

Резултат извршавања наредбе `System.out.println(student.toString())` је: `Student Pera Peric, sa brojem indeksa 20171047`.

4.2 Опсег зрна

У оквиру етикете `<bean>` могуће је навести атрибут `scope` чија вредност представља опсег зрна. Атрибут `scope` може да има пет вредности:

- `singleton` - Контејнер креира само једно зрно дате класе. Сваки пут када се од контејнера затражи такво зрно, контејнер даје референцу на претходно креирано зрно. Вредност `singleton` се углавном користи за објекте чије стање није потребно чувати.

- `prototype` - Сваки пут када се од контејнера затражи зрно чији атрибут `scope` има вредност `prototype`, контејнер креира ново зрно. Вредност `prototype` се углавном користи за објекте чије стање је потребно чувати.

- `request` - Контејнер креира ново зрно за сваки *HTTP* захтев.

[14] Пракса је да се наведена датотека назива `beans.xml`, али је име потпуно прозвољно. Важно је само да дефинисани ресурс буде доступан апликацији приликом њеног извршавања.

- `session` - Контејнер креира ново зрно за сваку *HTTP* сесију.
- `global session` - Контејнер креира ново зрно за сваку глобалну *HTTP* сесију.

Подразумевана вредност атрибута `scope` је `singleton`.

4.2.1 Опсег `singleton`

У примеру 5 приказана је класа `Poruka`.

Пример 5: Класа `Poruka`

```
public class Poruka {
    private String tekst;

    public void postaviTekst(String tekst) {
      this.tekst = tekst;
    }

    public void ispisiTekst() {
      System.out.println("Poruka: " + tekst);
    }
}
```

Информације које су потребне контејнеру како би креирао објекат класе `Poruka` налазе се у оквиру `xml` датотеке приказане у примеру 6. Атрибут `scope` има вредност `singleton`.

Пример 6: Вредност `singleton` атрибута `scope`

```
<beans xmlns="http://www.springframework.org/schema/beans" ... >
    <bean id = "poruka" class = "Poruka" scope = "singleton"></bean>
</beans>
```

Уколико се од контејнера више пута затражи зрно класе `Poruka`, контејнер сваки пут враћа зрно које је већ креирао, што илуструје метода у примеру 7.

Пример 7: Испис података о зрну са опсегом `singleton`

```
public static void main(String[] args) {
    ApplicationContext context =
                  new ClassPathXmlApplicationContext("beans.xml");
    Poruka prvaPoruka = (Poruka)context.getBean("poruka");

    prvaPoruka.postaviTekst("Prva poruka!");
    prvaPoruka.ispisiTekst();

    Poruka drugaPoruka = (Poruka)context.getBean("poruka");
    drugaPoruka.ispisiTekst();
}
```

Резултат наредбе `prvaPoruka.ispisiTekst()` је `Prva poruka!`, док је резултат наредбе `drugaPoruka.ispisiTekst()` такође `Prva poruka!`.

4.2.2 Опсег `prototype`

У оквиру датотеке приказане у примеру 8 налазе се информације о објекту класе `Poruka` чији атрибут `scope` има вредност `prototype`.

Пример 8: Вредност `prototype` атрибута `scope`

```
<beans xmlns="http://www.springframework.org/schema/beans" ... >
    <bean id = "poruka" class = "Poruka" scope = "prototype">
        <property name = "tekst" value = "Standardna poruka!">
    </bean>
</beans>
```

Уколико се од контејнера затражи објекат класе `Poruka`, контејнер на сваки захтев креира ново зрно. Наведено понашање илуструје пример 9.

Пример 9: Испис података о зрну са опсегом `prototype`

```
public static void main(String[] args) {
    ApplicationContext context =
                  new ClassPathXmlApplicationContext("beans.xml");
    Poruka prvaPoruka = (Poruka)context.getBean("poruka");

    prvaPoruka.setTekst("Prva poruka!");
    prvaPoruka.ispisiTekst();

    Poruka drugaPoruka = (Poruka)context.getBean("poruka");
    drugaPoruka.ispisiTekst();
}
```

Резултат наредбе `prvaPoruka.ispisiTekst()` је `Prva poruka!`, а резултат наредбе `drugaPoruka.ispisiTekst()` је `Standardna poruka!`

Пример зрна чији атрибут `scope` има вредност `prototype` може бити корпа у оквиру апликације за куповину путем интернета. Сваки корисник би требало да има посебну корпу, како би поруџбина била успешно обављена.

За разлику од осталих опсега, *Spring* не управља у потпуности животним циклусом објеката чији опсег има вредност `prototype`. *Spring* креира, конфигурише и прослеђује такве објекте, али није задужен за њихово уништавање.

5 Уметање зависности

Рад на компликованом пројекту захтева креирање великог броја класа које би требало да буду што више независне једне од других, како би се пројекат што лакше одржавао и унапређивао.

Уметање зависности (енг. *dependency injection*) је концепт раздвајања зависности између класа, а радни оквир *Spring* даје подршку за тај концепт. Уметање зависности помаже у раздвајању кода односно класа, тј. омогућава да се приликом креирања неког објекта уметну у њега они објекти од којих он зависи. На тај начин објекат који зависи од неких других објеката није задужен за креирање својих зависности и није везан за конкретне имплементације тих зависности.

У примеру 10 представљена је класа `Kasetofon`, а у примеру 11 класа `Kaseta`.

Пример 10: Класа `Kasetofon` са конструктором без параметара

```
public class Kasetofon {
    private Kaseta kaseta;

    public Kasetofon() {
        kaseta = new Kaseta();
    }

    public void pokreni() {
        kaseta.pusti();
    }
}
```

Пример 11: Класа `Kaseta`

```
public class Kaseta {
    private String naslov = "April u Beogradu";
    private String izvodjac = "Zdravko Colic";

    public void pusti() {
        System.out.println("Svira kaseta izvodjaca " + izvodjac + " pod
            nazivom " + naslov);
    }
}
```

Као што се може видети, објекат класе `Kasetofon` садржи објекат класе `Kaseta` и задужен је за његово креирање. На тај начин је везан за један конкретан објекат класе `Kaseta`. Овакав приступ није пожељан јер смањује могућност поновног коришћења кода, доводи до проблема приликом тестирања и тежак је за разумевање.

У примеру 12 приказана је измењена класа `Kasetofon` где објекат те класе није задужен за креирање својих зависности, већ су му те зависности дате приликом креирања.

Пример 12: Класа `Kasetofon` са конструктором за раздвајање зависности

```
public class Kasetofon {
    private Kaseta kaseta;

    public Kasetofon(Kaseta kaseta) {
        this.kaseta = kaseta;
    }
}
```

Осим уметања зависности преко конструктора, зависности је могуће уметнути и коришћењем `set` метода, јер можда нису доступне у тренутку креирања објекта. Уметање зависности преко `set` метода илустровано је примером 13.

Пример 13: Класа `Kasetofon` са методом `ubaciKasetu`

```
public class Kasetofon {
    private Kaseta kaseta;

    public void ubaciKasetu(Kaseta kaseta) {
        this.kaseta = kaseta;
    }
}
```

Како би се обезбедила још већа независност између наведених класа, `Kaseta` се може дефинисати као интерфејс, као у примеру 14.

Пример 14: Интерфејс `Kaseta`

```
public interface Kaseta {
    void pusti();
}
```

На овај начин обезбеђује се могућност поновног коришћења истог дела кода, кôд постаје читљивији и олакшава се његово тестирање. Такође, различити тимови могу да раде на развоју различитих делова апликације без међусобне зависности, уколико је комуникација између тих делова апликације дефинисана интерфејсима.

Spring контејнер задужен је за креирање објеката апликације, чиме се обезбеђује да објекти не знају одакле долазе њихове зависности нити како изгледају њихове имплементације. Уметање зависности јесте основни део радног оквира *Spring* на коме се заснивају све *Spring* апликације.

Spring нуди три начина повезивања зрна:

- Експлицитна `xml` конфигурација;
- Експлицитна *Java* конфигурација;
- Аутоматско повезивање зрна.

Аутоматски начин повезивања зрна захтева најмање конфигурације, па се из тога разлога највише препоручује. Уколико је потребно креирати зрна класа

које нису написане као део пројекта и чији изворни кôд није доступан, то се може постићи експлицитном `xml` или *Java* конфигурацијом. Више о уметању зависности се може прочитати у [3] и [4].

5.1 Аутоматско уметање зависности

Аутоматско креирање и повезивање објеката *Spring* врши кроз два корака:

- Скенирање компонената - *Spring* аутоматски открива зрна која ће бити креирана у *Spring* контејнеру.
- Аутоматко повезивање - *Spring* аутоматски повезује зрна.

Како би *Spring* открио класу чији објекат треба да буде креиран у *Spring* контејнеру потребно је да класа има анотацију `@Component`, као у примерима 15 и 16.

Пример 15: Класа `Kasetofon` означена анотацијом `@Component`

```
@Component
public class Kasetofon {
    private Kaseta kaseta;

    public Kasetofon(Kaseta kaseta) {
        this.kaseta = kaseta;
    }
}
```

Пример 16: Класа `Kaseta` означена анотацијом `@Component`

```
@Component
public class Kaseta {
    private String naslov = "Jesen u mom sokaku";
    private String izvodjac = "Predrag Zivkovic Tozovac";

    public void pusti() {
        System.out.println("Svira kaseta izvodjaca " + izvodjac + " pod
            nazivom " + naslov);
    }
}
```

Како би се покренуло аутоматско скенирање и како би *Spring* пронашао класе које имају анотацију `@Component` и креирао одговарајућа зрна, потребно је креирати конфигурацијску класу која осим анотације `@Configuration` треба да има и анотацију `@ComponentScan`, као у примеру 17.

Пример 17: Класа `Konfiguracija`

```
@Configuration
@ComponentScan
public class Konfiguracija {
}
```

Конфигурацијска класа из примера 17 користи се приликом креирања контејнера, а то се постиже наредбом илустрованом у примеру 18.

Пример 18: Наредба за покретање аутоматског скенирања

```
ApplicationContext context =
          new AnnotationConfigApplicationContext(Konfiguracija.class);
```

Приликом скенирања *Spring* тражи класе које имају анотацију `@Component` у оквиру истог пакета у коме се налази конфигурацијска класа. Могуће је навести више пакета кроз које ће се вршити скенирање, навођењем низа њихових имена у оквиру анотације `@ComponentScan(basePackages = { ... })`.

Свако зрно које се налази у контејнеру има свој `ID`. Уколико се експлицитно не наведе, зрно добија `ID` који одговара називу одговарајуће класе.

Да би се зрну доделио другачији `ID`, потребно је његову вредност проследити анотацији `@Component` као у примеру 19.

Пример 19: Класа `Kaseta` са дефиницијом негенеричког идентификатора

```
@Component("traka")
public class Kaseta {
    ...
}
```

Постоје три начина аутоматског уметања зависности:

- Уметање преко конструктора класе;
- Уметање преко `set` метода;
- Уметање преко поља класе.

Зависности које су обавезне, тј. зависности без којих објекат не може да обавља своје примарне задатке потребно је уметнути преко конструктора класе, како би се обезбедила њихова сигурна доступност и спречиле `null` вредности за такве зависности. Оне зависности које су опционе и које можда неће бити доступне приликом креирања објекта могуће је уметнути и коришћењем `set` метода.

5.1.1 Уметање зависности преко конструктора класе

Како би *Spring* креирао зрно коришћењем конструктора одговарајуће класе и проследио његове зависности, потребно је да конструктор има анотацију `@Autowired`, као у примеру 20.

Пример 20: Класа `Kasetofon` са конструктором анотираним са `@Autowired`

```
@Component
public class Kasetofon {
    private Kaseta kaseta;
```

```
    @Autowired
    public Kasetofon(Kaseta kaseta) {
        this.kaseta = kaseta;
    }
}
```

5.1.2 Уметање зависности преко `set` метода

Осим конструктора, анотација `@Autowired` може бити примењена на `set` методе класе, као у примеру 21. На тај начин, одговарајуће зависности се убацују позивањем `set` метода.

Пример 21: Класа `Kasetofon` са `set` методом анотираном са `@Autowired`

```
@Component
public class Kasetofon {
    private Kaseta kaseta;

    @Autowired
    public void ubaciKasetu(Kaseta kaseta) {
        this.kaseta = kaseta;
    }
}
```

Осим на `set` методе, анотација `@Autowired` може бити примењена на било који други метод класе.

5.1.3 Уметање зависности преко поља класе

Још један начин аутоматског уметања зависности јесте уметање зависности преко поља класе, као у примеру 22. Све што је потребно јесте да одговарајућа поља имају анотацију `@Autowired`.

Пример 22: Класа `Kasetofon` са пољем анотираним са `@Autowired`

```
@Component
public class Kasetofon {
    @Autowired
    private Kaseta kaseta;

    ...
}
```

5.1.4 Проблеми приликом аутоматског уметања зависности

Аутоматско уметање зависности за сва три наведена начина ради исправно само у случају да постоји једно одговарајуће зрно које је потребно уметнути. У случају постојања више од једног зрна, долази до избацивања изузетка типа `NoUniqueBeanDefinitionException`, јер није могуће одредити које зрно је потребно уметнути. *Spring* нуди два начина за решавање оваквог проблема.

Уколико постоји више имплементација интерфејса `Kaseta`, проблем уметања зависности могуће је решити коришћењем анотације `@Primary`. На тај начин *Spring* зна коју имплементацију је потребно уметнути. У примеру 23 приказана је класа `RadioBeogradKaseta` која имплементира интерфејс `Kaseta` и означена је анотацијом `@Primary`. То значи да ће и у случају постојања неке друге имплементације интерфејса `Kaseta`, `Spring` увек уметати објекат класе `RadioBeogradKaseta`.

Пример 23: Класа `RadioBeogradKaseta` означена анотацијом `@Primary`

```
@Component
@Primary
public class RadioBeogradKaseta implements Kaseta {
    ...
}
```

Осим анотације `@Primary`, проблем уметања зависности могуће је решити и дефинисањем квалификатора зрна коришћењем анотације `@Qualifier`. Аргумент који се прослеђује анотацији `@Qualifier` јесте `ID` зрна које је потребно уметнути. У примеру 24 приказана је класа `Kasetofon` чије је поље типа `Kaseta` означено анотацијом `@Qualifier` којој је прослеђен `ID` зрна `radioBeogradKaseta`. То значи да ће *Spring* уметнути имплементацију интерфејса `Kaseta` из примера 23, чак и ако она није анотирана са `@Primary`.

Пример 24: Класа `Kasetofon` са пољем означеним анотацијом `@Qualifier`

```
@Component
public class Kasetofon {
    @Autowired
    @Qualifier("radioBeogradKaseta")
    private Kaseta kaseta;

    ...
}
```

Такође, осим коришћења `ID` зрна, анотацији `@Qualifier` се као аргумент може проследити било који други назив, као у примеру 25.

Пример 25: Прослеђивање произвољног аргумента анотацији `@Qualifier`

```
@Component
@Qualifier("rb")
public class RadioBeogradKaseta implements Kaseta {
    ...
}
```

Програмски језик *Java* подржава примену више анотација истог типа само ако такве анотације имају анотацију `@Repeatable`, што није случај са анотацијом `@Qualifier`. Стога, уколико је потребно додатно сузити избор приликом избора зрна које је потребно уметнути, то се може постићи креирањем и применом нових анотација који имају анотацију `@Qualifier`. Примери 26 и 27 илуструју креирање нових анотација.

Пример 26: Креирање анотације `@RadioBeogradKaseta`

```
@Retention(RetentionPolicy.RUNTIME)
@Target({ElementType.TYPE, ElementType.CONSTRUCTOR,
        ElementType.FIELD, ElementType.METHOD})
@Qualifier
public @interface RadioBeogradKaseta {
}
```

Пример 27: Креирање анотације `@DomacaKaseta`

```
@Retention(RetentionPolicy.RUNTIME)
@Target({ElementType.TYPE, ElementType.CONSTRUCTOR,
        ElementType.FIELD, ElementType.METHOD})
@Qualifier
public @interface DomacaKaseta {
}
```

Нове анотације креиране у примерима 26 и 27 могу се применити на класу `RadioBeogradKaseta`, као у примеру 28. У примеру 29 илустрована је примена приказаних анотација на пољу класе `Kasetofon`.

Пример 28: Примена креираних анотација на класу `RadioBeogradKaseta`

```
@Component
@RadioBeogradKaseta
@DomacaKaseta
public class RadioBeogradKaseta implements Kaseta {
    ...
}
```

Пример 29: Примена креираних анотација на пољу класе `Kasetofon`

```
@Component
public class Kasetofon {
    @Autowired
    @RadioBeogradKaseta
    @DomacaKaseta
    private Kaseta kaseta;

    ...
}
```

5.2 Уметање зависности - *Java* конфигурација

У одељку 5.1 приказан је начин креирања зрна скенирањем компонената и аутоматским повезивањем. Иако овај начин захтева најмање конфигурације, није га могуће применити на класе које нису део пројекта, јер им у том случају није могуће додати анотацију `@ComponentScan`. Овај проблем може се решити коришћењем експлицитне *Java* конфигурације.

Како би се уметање зависности одрадило коришћењем *Java* конфигурације, потребно је креирати класу која има анотацију `@Configuration`. Да би се декларисало зрно, у оквиру класе са анотацијом `@Configuration` потребно је написати метод који враћа објекат који ће бити регистрован као зрно. Тај метод треба да има анотацију `@Bean`. На основу `@Bean` анотације *Spring* зна да објекат који тај метод враћа треба да буде регистрован као зрно у *Spring* контејнеру.

У примеру 30 експлицитна *Java* конфигурација илустрована је коришћењем претходно дефинисаних класа `RadioBeogradKaseta` и `Kasetofon`.

Пример 30: Класа `Konfiguracija`

```
@Configuration
public class Konfiguracija {
    @Bean
    public Kaseta kaseta() {
        return new RadioBeogradKaseta();
    }

    @Bean
    public Kasetofon kasetofon() {
        return new Kasetofon(kaseta());
    }
}
```

5.3 Уметање зависности - `xml` конфигурација

`xml` конфигурација представљала је основни начин дефинисања и повезивања зрна у првим верзијама радног оквира *Spring*. Уметање зависности `xml` конфигурацијом могуће је постићи на два начина:

- Уметање зависности преко конструктора класе;
- Уметање зависности преко `set` метода.

Оба начина биће представљена коришћењем класа `Krug` и `Tacka` дефинисаних у примерима 31 и 32.

Пример 31: Класа `Krug`

```
public class Krug {

    private Tacka centar;
    private double poluprecnik;

    public Krug(Tacka centar, double poluprecnik) {
        this.centar = centar;
        this.poluprecnik = poluprecnik;
    }
```

```
    public void postaviCentar(Tacka centar) {
        this.centar = centar;
    }

    public void postaviPoluprecnik(double poluprecnik) {
        this.poluprecnik = poluprecnik;
    }

    public String toString() {
        return "Centar: " + centar + ", poluprecnik: " + poluprecnik;
    }
}
```

Пример 32: Класа `Tacka`

```
public class Tacka {

    private int x;
    private int y;

    public Tacka(int x, int y) {
        this.x = x;
        this.y = y;
    }

    public String toString() {
        return "(" + x + "," + y + ")";
    }
}
```

5.3.1 Уметање зависности преко конструктора класе

Како је дефинисано у примеру 31, класа `Krug` има конструктор који прихвата објекат класе `Tacka`. Потребно је у оквиру `xml` датотеке декларисати зрно класе `Tacka` и уметнути га у зрно класе `Krug`. То се постиже коришћењем етикете `<constructor-arg>` у оквиру етикете `<bean>` која декларише зрно класе `Krug`. Етикета `<constructor-arg>` има атрибут `ref` чија је вредност `id` зрна класе `Tacka`. Контејнер може да прослеђује конструктору и аргументе примитивног типа. То се постиже коришћењем атрибута `type` и `value` у оквиру етикете `<constructor-arg>`. Међутим, уколико постоји више аргумената истог типа, како би се избегла двосмисленост потребно је користити атрибут `index` чија вредност представља редни број аргумента. Дакле, приликом креирања зрна класе `Krug` контејнер користи конструктор класе `Krug` за уметање потребних зависности.

У случају класа `Krug` и `Tacka` датотека `beans.xml` треба да има облик као у примеру 33.

Пример 33: Уметање зависности преко конструктора класе

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans" ... >
    <bean id = "krug" class = "Krug">
        <constructor-arg ref="centar"/>
        <constructor-arg type="double" value="3.5"/>
    </bean>
    <bean id = "centar" class = "Tacka">
        <constructor-arg index="0" value="5"/>
        <constructor-arg index="1" value="3"/>
    </bean>
</beans>
```

5.3.2 Уметање зависности преко `set` метода

Уметање зависности преко `set` метода се остварује тако што контејнер приликом креирања зрна позива `set` методе како би уметнуо одговарајуће зависности. То се постиже коришћењем етикете `<property>`. Атрибут `name` у оквиру етикете `<property>` дефинише назив поља коме се додељује вредност, а у зависности од типа поља атрибути `ref` и `value` се користе за додељивање вредности. У случају класа `Krug` и `Tacka` датотека `beans.xml` треба да има облик као у примеру 34.

Пример 34: Уметање зависности преко `set` метода

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans" ... >
    <bean id = "krug" class = "Krug">
        <property name="centar" ref = "tacka"/>
        <property name="poluprecnik" value="3.5"/>
    </bean>
    <bean id = "tacka" class = "Tacka">
        <property name="x" value="5" />
        <property name="y" value="3" />
    </bean>
</beans>
```

6 Аспектно оријентисано програмирање

Једна од кључних компоненти радног оквира *Spring* јесте и аспектно оријентисано програмирање.

Аспектно оријентисано програмирање је концепт који омогућава да неке функционалности које прожимају целу апликацију буду издвојене на једно место, не оптерећујући методе којима то није примарни задатак. Неки од примера тих функционалности су сигурност, логовање, транскације, кеширање, итд. Те функционалности се дефинишу у оквиру посебних класа које се зову аспекти. Оваквим приступом избегнуто је понављање кода који обавља исту врсту посла, а методе обављају само своје примарне задатке. Један од начина за решавање овог проблема било би креирање класе која садржи функционалности које су заједничке за многе делове апликације, а затим креирати остале класе које је наслеђују и којима су те функционалности потребне. Међутим, овакав приступ не би био добар јер програмски језик *Java* не подржава вишеструко наслеђивање, па уколико је потребно да те класе наслеђују и неке друге класе, то не би било могуће.

Дакле, потребно је овакве функционалности одвојити од бизнис логике саме апликације. Један од примера за разумевање овог концепта може бити апликација у оквиру које постоје два типа корисника, администратор и клијент. Када се корисник успешно пријави, у контејнеру се прави зрно које представља тог корисника. Потребно је нпр. омогућити позив неке методе само ако је тренутно пријављени корисник администратор. У оквиру посебне методе која ће се извршити пре, од контејнера се може затражити зрно које представља тренутно пријављеног корисника. Само у случају да је пријављени корисник администратор, дозволиће се позив тражене методе.

Аспектно оријентисано програмирање омогућава да се дефинише када и где ће неке заједничке функционалности бити примењене, без потребе за модификацијом класа на које се примењују.

Дакле, за разлику од уметања засвисности чиме се постиже раздвајање објеката који чине једну апликацију, аспектно оријентисано програмирање раздваја функционалности које су заједничке за многе делове апликације од објеката на које се примењују.

У наставку су описани основни појмови у аспектно оријентисаном програмирању.

- *Advice* означава посао који је потребно обавити, као и када се тај посао обавља. Постоји неколико *advice* типова који одређују када је потребно обавити одређени посао.
  - *Before* - дефинисани посао се обавља пре позива неке методе.
  - *After* - дефинисани посао се обавља након позива неке методе, без обзира на то да ли се та метода успешно завршила.
  - *After-returning* - дефинисани посао се обавља након што се нека метода успешно завршила.

  - *After-throwing* - дефинисани посао се обавља након што се извршавање неке методе завршило избацивањем изузетка.
  - *Around* - дефинисани посао се обавља око извршавања неке методе, тј. посао се обавља и пре и после извршавања те методе.

- *Joinpoint* представља једно место у апликацији на коме се примењује неки *advice*. То може бити позив неке методе, избацивање изузетка, промена вредности неког поља, итд.

- *Pointcuts* представља скуп више места у апликацији на коме се примењује неки *advice*.

- *Aspect* - Након што се дефинишу *pointcuts* и *advice*, тиме је дефинисано све што је потребно да би се неки посао успешно обавио. *Pointcuts* и *advice* заједно чине један *aspect*.

Постоји много радних оквира који дају подршку за аспектно оријентисано програмирање, нудећи различите могућности за дефинисање аспеката. *Spring* је радни оквир чија се подршка за аспектно оријентисано програмирање заснива на дефинисању аспеката, тј. посла које се може применити само на извршавање метода. Иако је аспекте могуће применити само на методе, то задовољава потребе већине апликација које се ослањају на аспектно оријентисано програмирање.

Како би дефинисао *pointcuts* радни оквир *Spring* користи *pointcut* означиваче радног оквира *AspectJ*[15]. У примерима који следе биће коришћен означивач `execution()` који обухвата сваки *joinpoint* који се односи на извршавање неке методе. Такође, радни оквир *Spring* користи и анотације радног оквира *AspectJ* како би дефинисао аспекте.

Spring нуди подршку за аспектно оријентисано програмирање кроз неколико начина дефинисања аспеката. У овом раду биће представљена два основна начина:

- *Java* конфигурација;

- `xml` конфигурација.

Више о аспектно оријентисаном програмирању се може прочитати у [3] и [4].

6.1 Дефинисање аспеката - *Java* конфигурација

Најважнија карактеристика радног оквира *AspectJ* јесте могућност дефинисања аспеката коришћењем анотација. У примеру 35 приказана је класа `Putnici` која има анотацију `@Aspect`. Анотација `@Aspect` означава да та класа није стандарна *Java* класа, већ представља један аспект.

[15] `https://www.eclipse.org/aspectj/`

Пример 35: Класа `Putnici` са анотацијом `@Aspect`

```
@Aspect
public class Putnici {
    @Before("execution(let())")
    public void zauzetiMesta() {
        System.out.println("Dragi putnici, molimo zauzmite vasa mesta!");
    }

    @AfterReturning("execution(let())")
    public void aplauz() {
        System.out.println("Avion je sleteo. Putnici tapsu!!!");
    }

    @AfterThrowing("execution(let())")
    public void prinudnoSletanje() {
        System.out.println("Avion je prinudno sleteo!");
    }
}
```

Класа `Putnici` има три методе које имају анотације `@Before`, `@AfterReturning` и `@AfterThrowing`. Тиме је дефинисано када те методе треба да се изврше у односу на извршавање методе `let()` класе `Avion`, која је приказана у примеру 36.

Пример 36: Класа `Avion`

```
public class Avion {
    public void let() {
        System.out.println("Avion leti!")
    }
}
```

Иако класа `Putnici` има анотацију `@Aspect`, она неће бити дефинисана као аспект у *Spring* контејнеру. Како би се то постигло конфигурацијској класи је потребно додати анотацију `@EnableAspectJAutoProxy`, као у примеру 37.

Пример 37: Класа `Konfiguracija` са анотацијом `@EnableAspectJAutoProxy`

```
@Configuration
@EnableAspectJAutoProxy
public class Konfiguracija {
    @Bean
    public Putnici putnici() {
        return new Putnici();
    }

    @Bean
    public Avion avion() {
        return new Avion();
    }
}
```

Метода `main` приказана је у примеру 38.

Пример 38: `main` метода *Spring* апликације

```
public static void main(String[] args) {
    ApplicationContext context =
              new AnnotationConfigApplicationContext(Konfiguracija.class);

    Avion avion = (Avion)context.getBean("avion");
    avion.let();
}
```

Резултат наредбе `avion.let()` је:

```
Dragi putnici, molimo zauzmite vasa mesta!

Avion leti!

Avion je sleteo. Putnici tapsu!!!
```

Дакле, пре извршавања методе `let()` позива се метода `zauzetiMesta()`, а након извршавања те методе позива се метода `aplauz()`.

6.2 Дефинисање аспеката - `xml` конфигурација

Уколико се користи `xml` конфигурација за креирање и повезивање зрна у *Spring* контејнеру, потребно је да одговарајућа `xml` датотека садржи етикету `<aop:aspectj-autoproxy>`. За пример класа `Putnici` и `Avion`, `xml` датотека треба да има облик као у примеру 39.

Пример 39: Конфигурацијска `xml` датотека

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans" ... >
    <aop:aspectj-autoproxy />
    <bean class="putnici" />
    <bean class="avion" />
</beans>
```

Spring нуди неколико елемената који се могу користити за дефинисање аспеката коришћењем `xml` конфигурације.

- `<aop:aspect>` - дефинише аспект;
- `<aop:after>` - дефинише *advice* типа `after`;
- `<aop:after-returning>` - дефинише *advice* типа `after-returning`;
- `<aop:after-throwing>` - дефинише *advice* типа `after-throwing`;
- `<aop:around>` - дефинише *advice* типа `around`;

- `<aop:before>` - дефинише *advice* типа `before`;
- `<aop:poincut>` - дефинише *pointcut*;
- `<aop:config>` - садржи остале `<aop:>` елементе.

У примеру 40, класа `Putnici` написана је без употребе анотација.

Пример 40: Класа `Putnici` без анотација

```
public class Putnici {
    public void zauzetiMesta() {
        System.out.println("Dragi putnici, molimo zauzmite vasa mesta!");
    }

    public void aplauz() {
        System.out.println("Avion je sleteo. Putnici tapsu!!!");
    }

    public void prinudnoSletanje() {
        System.out.println("Avion je prinudno sleteo!");
    }
}
```

Како би се дефинисао аспект дефинисан класом `Putnici`, потребно је да конфигурациона `xml` датотека садржи део из примера 41.

Пример 41: Дефинисање аспеката у конфигурацијској датотеци

```
<aop:config>
    <aop:aspect id="primerAspekta" ref="putnici">
        <aop:before
            pointcut="execution(let(..))"
            methods="zauzetiMesta" />
        <aop:after-returning
            pointcut="execution(let(..))"
            methods="aplauz" />
        <aop:after-throwing
            pointcut="execution(let(..))"
            methods="prinudnoSletanje" />
    </aop:aspect>
</aop:config>
```

Метода `main` приказана је у примеру 42.

Пример 42: `main` метода *Spring* апликације

```
public static void main(String[] args) {
    ApplicationContext context =
                  new ClassPathXmlApplicationContext("beans.xml");
    Avion avion = (Avion)context.getBean("avion");
    avion.let();
}
```

Резултат позива методе `avion.let()` је:

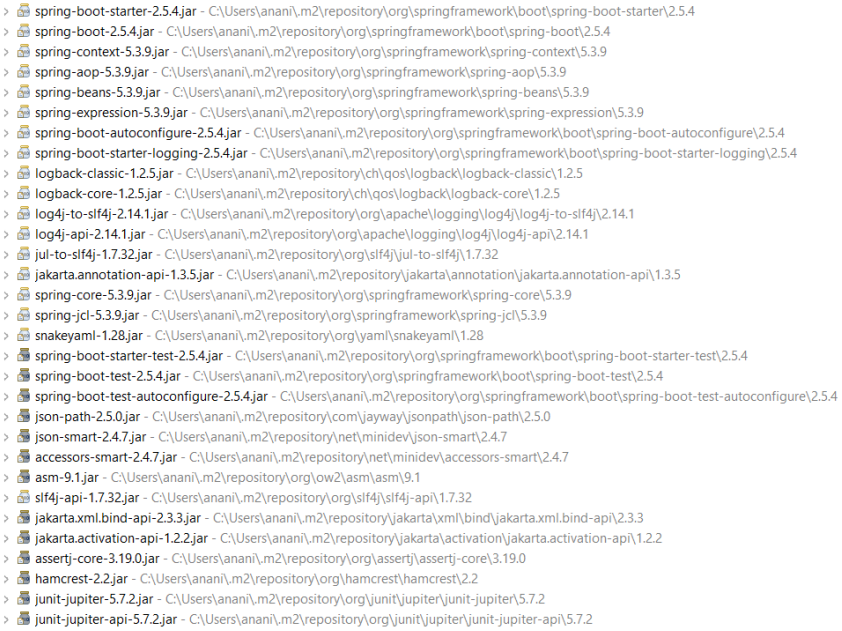
```
Dragi putnici, molimo zauzmite vasa mesta!

Avion leti!

Avion je sleteo. Putnici tapsu!!!
```

7 Радни оквир *Spring Boot*

Spring Boot представља надоградњу радног оквира *Spring Framework*. Направљен је са циљем да значајно олакша и убрза развој *Spring* апликација. *Spring Boot* води рачуна о свим техничким детаљима пројекта и омогућава програмерима да се на почетку рада на пројекту фокусирају на логику апликације коју развијају. Сваки *Spring Boot* пројекат има као основну зависност `spring-boot-starter-parent` пројекат. Пројекат `spring-boot-starter-parent` нуди почетне зависности које су потребне свима, без обзира на тип пројекта који се развија. Такође води рачуна и о компатибилности између верзија зависности. Још једна његова предност је и аутоматска конфигурација. *Spring Boot* ће за сваку зависност аутоматски покушати да креира и повеже одговарајућа зрна, чиме се значајно смањује експлицитна `xml` и *Java* конфигурација. На слици 10 приказане су неке од основних зависности које нуди `spring-boot-starter-parent` пројекат.



Слика 10: *Почетне зависности* Spring Boot *пројекта*

Spring Boot нуди велики број почетних пројеката намењених за различите врсте апликација. На почетку рада на пројекту потребно је одабрати почетне пројекте који одговарају наменама апликације која се развија, што омогућава брзо започињање пројекта. За креирање веб апликација *Spring Boot* нуди `spring-boot-starter-web` пројекат који користи *Spring MVC* и *Tomcat*[16] као уграђени веб сервер.

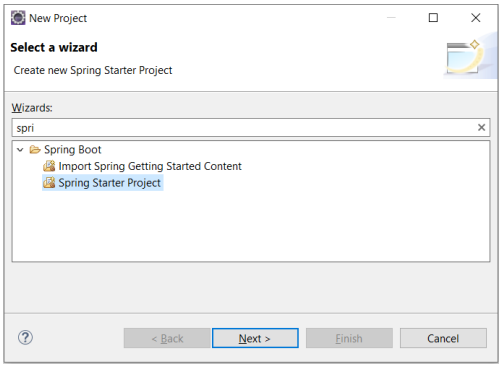
Више о радном оквиру *Spring Boot* може се прочитати у [5] и [6].

[16] `http://tomcat.apache.org/`

8 Апликација *Spring Initializr*

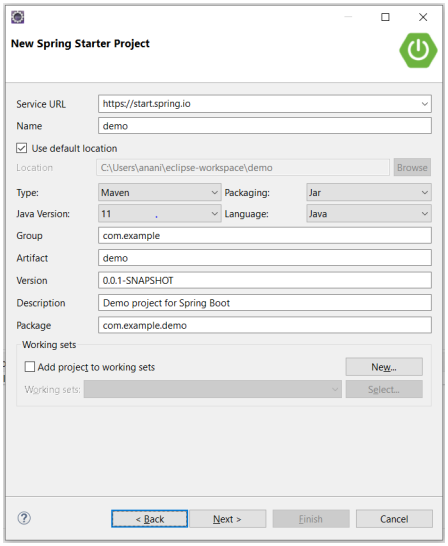
Spring Initializr је веб апликација[17] намењена за аутоматско генерисање *Spring Boot* пројекта.

У наредном делу биће приказано конфигурисање *Spring Boot* пројекта уз помоћ апликације *Spring Initializr*, коришћењем алата *Spring Tools*[18] у развојном окружењу *Eclipse*.



Слика 11: *Одабир типа пројекта*

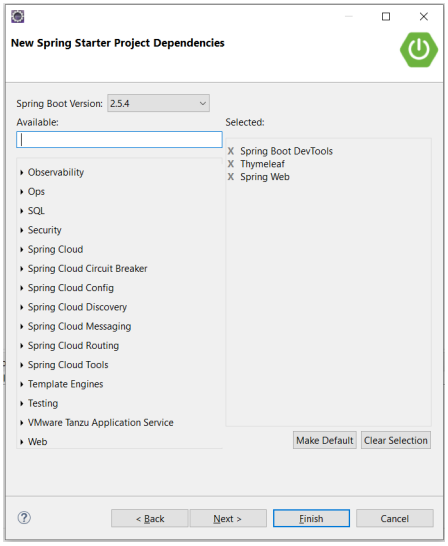
Како би се конфигурисала *Spring Boot* апликација потребно је одабрати опцију `File`, а затим `New` → `Project`. Након тога, отвара се прозор као на слици 11, у оквиру кога је као тип пројекта потребно одабрати `Spring Starter Project`. Након одабира пројекта, отвара се прозор у оквиру кога је потребно унети основне информације о пројекту. Изглед прозора приказан је на слици 12.



Слика 12: *Унос основних информација о пројекту*

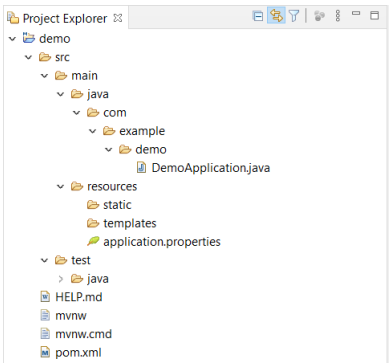
[17] Апликацији се може приступити на адреси `https://start.spring.io`
[18] `https://spring.io/tools`

Након уноса основних информација о пројекту и одабира опције `Next`, отвара се прозор у оквиру кога је потребно одабрати *Spring Boot* верзију, као и зависности пројекта. Изглед прозора приказан је на слици 13.



Слика 13: *Одабир зависности пројекта*

Одабиром опције `Finish` позива се апликација *Spring Initializr* како би се конфигурисао пројекат на основу претходно одабраних подешавања. На слици 14 приказана је структура конфигурисаног пројекта. Изворни кôд апликације смештен је у оквиру фолдера `src/main/java`, тестови су смештени у фолдеру `src/test/java`, док су ресурси смештени у оквиру фолдера `src/main/resources`. `mvnw` и `mvnw.cmd` представљају *Maven*[19] скрипте које се могу користити за изградњу пројекта. У оквиру датотеке `pom.xml` могу се прегледати дефинисане зависности пројекта. Фолдер `static` садржи све статичке компоненте, као што су нпр. слике. Фолдер `templates` садржи обрасце који се користе за генерисање садржаја у оквиру веб прегледача.

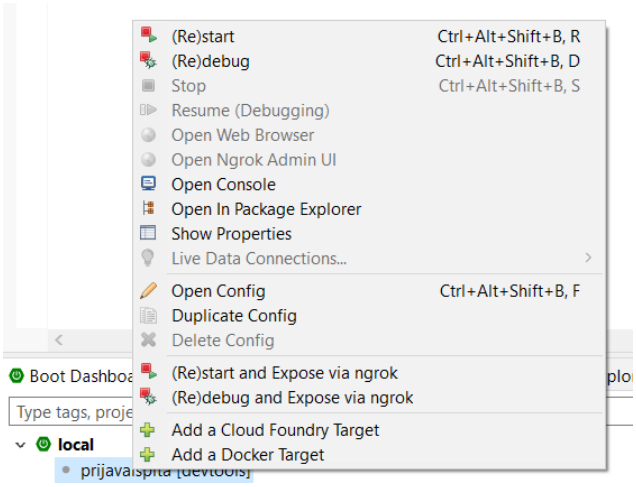


Слика 14: *Структура пројекта*

Приликом одабира зависности пројекта могуће је одабрати и *Spring Dev Tools*. *Spring Dev Tools* нуди неколико функционалности које убрзавају развој

[19] `https://maven.apache.org/`

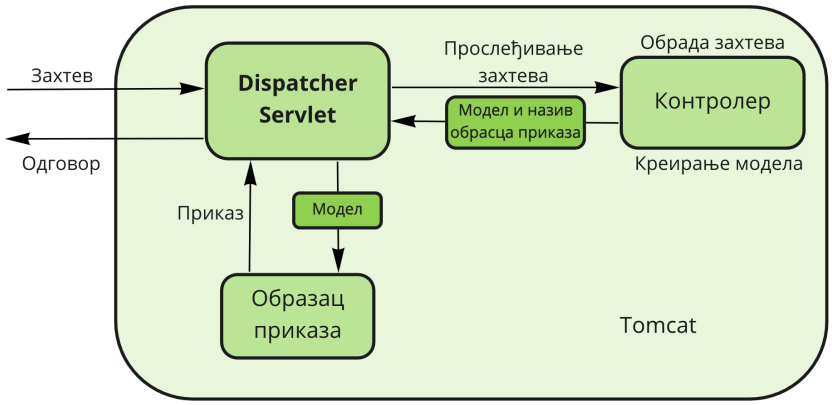
апликације, међу којима су аутоматско поновно покретање апликације након сваке измене, као и аутоматско освежавање измењене странице. У оквиру алата *Spring Tools* доступан је и *Spring Boot Dashboard* преко којег је могуће једноставно покренути и зауставити *Spring Boot* апликацију (слика 15).



Слика 15: Spring Boot Dashboard

9 Радни оквир *Spring MVC*

MVC (Model View Controller) архитектура је једна од најпопуларнијих архитектура за креирање веб апликација, а *Spring MVC* је најпопуларнији радни оквир за развој *Java* веб апликација. *MVC* архитектура раздваја апликацију на три основна дела: модел, приказ и контролер. На тај начин постиже се раздвајање слоја бизнис логике од презентационог слоја апликације. На слици 16 приказана је *Spring MVC* архитектура.



Слика 16: *Spring MVC архитектура*

У центру *Spring MVC* архитектуре налази се `DispatcherServlet` који представља имплементацију шаблона *Front Controller*. Сервлети често треба да одраде неколико уобичајених задатака пре него што обраде захтев, као што је нпр. провера дозвола пријављеног корисника. Како се овакве фунционалности не би понављале кроз сервлете, за њих је задужен *Front Controller*.

`DispatcherServlet` прима захтеве које шаље прегледач. Након што прими захтев, `DispatcherServlet` проналази одговарајући контролер који ће тај захтев обрадити. Контролер враћа модел и образац приказа. `DispatcherServlet` затим позива разрешивач приказа коме шаље модел који је контролер вратио. Разрешивач приказа врши потребно мапирање и враћа приказ назад у `DispatcherServlet`. На крају, `DispatcherServlet` шаље приказ прегледачу.

Више о радном оквиру *Spring MVC* може се прочитати у [7], [8] и [9].

У примеру 43 представљен је једноставан *Spring MVC* контролер.

Пример 43: Класа `HomeController`

```
@Controller
public class HomeController {
    @RequestMapping(value = "/welcome")
    @ResponseBody
    public String dobrodosli() {
        return "Dobrodosli na sajt za prijavu ispita!";
    }
}
```

Анотација `@Controller` означава да је у питању класа која представља контролер у *MVC* архитектури. На основу анотације `@Controller` аутоматски ће бити креирано одговарајуће зрно у *Spring* контејнеру. Метод `dobrodosli()` има две анотације, `@RequestMapping` и `@ResponseBody`. Анотација `@RequestMapping` има атрибут `value` чија је вредност `/welcome`. На тај начин се дефинише мапирање захтева у метод `dobrodosli()`, тј. када прегледач пошаље захтев на путању `/welcome` извршиће се метод `dobrodosli()`. Анотација `@ResponseBody` означава да ће вредност коју враћа метод `dobrodosli()` бити послата као одговор прегледачу.

Како би се покренула апликација која користи `HomeController`, потребно је написати класу која у оквиру `main` метода креира и учитава *Spring* контејнер. То се може постићи позивањем статичког метода `run` класе `SpringApplication`, што је илустровано у примеру 44.

Пример 44: Класа `PrijavaIspitaApplication`

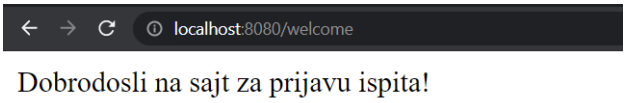
```
@SpringBootApplication
public class PrijavaIspitaApplication {

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

Анотација `@SpringBootApplication` се састоји од следеће три анотације:

- `@Configuration` - означава да је у питању конфигурацијска класа;
- `@EnableAutoConfiguration` - омогућава аутоматско скренирање;
- `@ComponentScan` - омогућава скенирање класа које се налазе у истом пакету као и класа на коју се анотација примењује.

На слици 17 приказан је изглед странице која се приказује уколико се апликација покрене и приступи се путањи `/welcome`.



Слика 17: *Страница на путањи* `/welcome`

Осим анотације `@RequestMapping`, *Spring* нуди следеће анотације које се могу користити за мапирање различитих типова *HTTP* захтева:

- `@GetMapping` - користи се за мапирање `GET` захтева;
- `@PostMapping` - користи се за мапирање `POST` захтева;

- `@PutMapping` - користи се за мапирање `PUT` захтева;
- `@DeleteMapping` - користи се за мапирање `DELETE` захтева;
- `@PatchMapping` - користи се за мапирање `PATCH` захтева.

За разлику од примера 43, садржај који прегледач приказује најчешће се генерише помоћу обрасца приказа. Постоји неколико типова образаца приказа које *Spring* подржава, од којих су најпознатији *JSP*[20] и *Thymeleaf*[21]. *Thymeleaf* је образац приказа који је базиран на *HTML*[22] језику и биће коришћен у примерима који следе.

Thymeleaf није део радног оквира *Spring*, па није могуће приступити подацима које је контролер сместио у модел. Пре прослеђивања захтева разрешивачу приказа *Spring* копира податке из модела у атрибуте захтева којима *Thymeleaf* може приступити.

У примеру 45 приказан је контролер који прима захтев послат на путању `/welcome` и као одговор прегледачу шаље приказ који има назив `welcome`. Разрешивач приказа мапира назив приказа који враћа метод `dobrodosli()` у одговарајући приказ смештен у фолдеру `src/main/resources/templates`.

Пример 45: Класа `HomeViewController`

```
@Controller
public class HomeViewController {
    @RequestMapping(value = "/welcome-view")
    public String dobrodosli() {
        return "welcome";
    }
}
```

У примеру 46 приказана је датотека `welcome.html`.

Пример 46: Датотека `welcome.html`

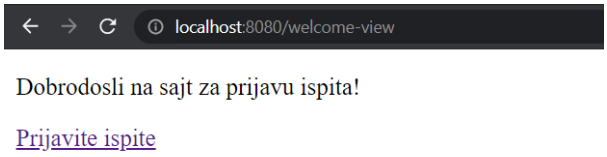
```
<!DOCTYPE html>
<html>
  <head>
    <title>Dobrodosli</title>
  </head>
  <body>
    <div>
      <p> Dobrodosli na sajt za prijavu ispita! </p>
      <a th:href="@{/form}">Prijavite ispite</a><br/>
      </div>
  </body>
</html>
```

[20] *Jakarta Server Pages*, технологија за динамичко креирање веб страница, `https://en.wikipedia.org/wiki/Jakarta_Server_Pages`

[21] `https://www.thymeleaf.org/`

[22] *HyperText Markup Language*, `https://en.wikipedia.org/wiki/HTML`

На слици 18 приказан је изглед странице која се приказује уколико се приступи путањи `/welcome-view`.



Слика 18: *Страница на путањи `/welcome-view`*

У примеру 47 приказана је класа `Student` која као поља има име, презиме, број индекса и листу испита које је студент пријавио.

Пример 47: Класа `Student`

```
public class Student {

    private String ime;
    private String prezime;
    private String indeks;
    private List<String> prijavljeniIspiti;

    public String getIme() {
        return ime;
    }

    public void setIme(String ime) {
        this.ime = ime;
    }

    public String getPrezime() {
        return prezime;
    }

    public void setPrezime(String prezime) {
        this.prezime = prezime;
    }

    public String getIndeks() {
        return indeks;
    }

    public void setIndeks(String indeks) {
        this.indeks = indeks;
    }

    public List<String> getPrijavljeniIspiti() {
        return prijavljeniIspiti;
    }
```

```
    public void setPrijavljeniIspiti(List<String> prijavljeniIspiti) {
        this.prijavljeniIspiti = prijavljeniIspiti;
    }
}
```

Како би студент могао да пријави испит, потребно је креирати одговарајућу форму која ће се приказати након одабира опције `Prijavite ispite`. Потребно је креирати нови контролер који ће бити задужен за обраду захтева који је послат на путању `/form`, као у примеру 48.

Пример 48: Класа `FormController` са методом за обраду `GET` захтева

```
@Controller
public class FormController {
    @ModelAttribute
    public void napraviListuIspita(ModelMap model) {
        List<String> ispiti = Arrays.asList("Linearna algebra",
                              "Geometrija 1", "Programiranje 2",
                              "Kompleksne funkcije", "Analiza 1");
        model.put("ispiti", ispiti);
    }

    @GetMapping("/form")
    public ModelAndView prikaziFormu(ModelMap model) {
        model.put("student", new Student());
        return new ModelAndView("form", model);
    }
}
```

Класа `FormController` из примера 48 има два метода, `napraviListuIspita(ModelMap model)` и `prikaziFormu(ModelMap model)`. Метод `napraviListuIspita(ModelMap model)` има анотацију `@ModelAttribute` која означава да је у питању метод који смешта податке у модел. Такав метод ће бити позван пре позива метода који има анотацију за мапирање захтева. Метод `prikaziFormu(ModelMap model)` додаје у модел атрибут `student` и враћа објекат класе `ModelAndView` који ће бити прослеђен разрешивачу приказа како би се извршило потребно мапирање. Објекат класе `ModelAndView` садржи модел и назив обрасца приказа.

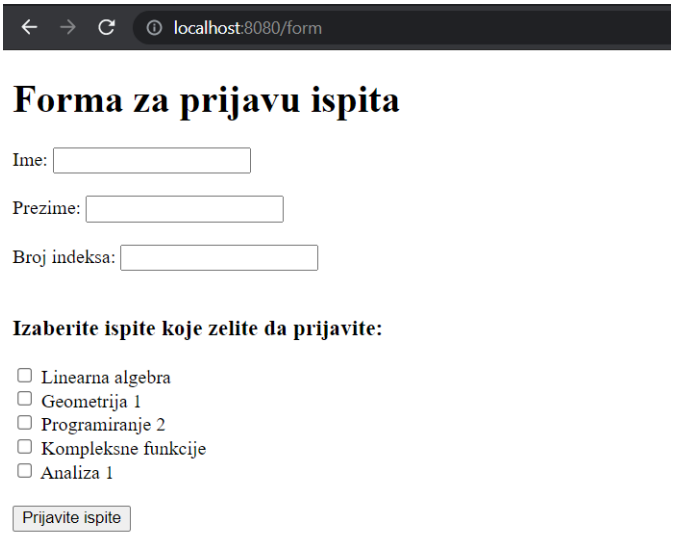
На слици 19 приказан је изглед странице која се приказује након одабира опције `Prijavite ispite`.

Одговарајући *Thymeleaf* образац (датотека `form.html`) приказан је у примеру 49.

Пример 49: Образац `form.html`

```
<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:th="http://www.thymeleaf.org">
<head>
```

```
    <title>Prijava ispita</title>
</head>
<body>
    <h1>Forma za prijavu ispita</h1>
    <form method="POST" th:object="${student}">
        <label for="ime">Ime: </label>
        <input type="text" th:field="*{ime}"/>
        <br/><br/>
        <label for="prezime">Prezime: </label>
        <input type="text" th:field="*{prezime}"/>
        <br/><br/>
        <label for="indeks">Broj indeksa: </label>
        <input type="text" th:field="*{indeks}"/>
        <br/><br/>
        <div class="grid">
            <div class="lista-ispita" id="ispiti">
                <h3>Izaberite ispite koje zelite da prijavite:</h3>
                <div th:each="ispit: ${ispiti}">
                    <input th:field="*{prijavljeniIspiti}"
                        type="checkbox" th:value="${ispit}" />
                    <span th:text="${ispit}"></span><br/>
                </div><br/>
                <button>Prijavite ispite</button>
            </div>
        </div>
    </form>
</body>
</html>
```



Слика 19: *Страница на путањи* `/form`

Датотека `form.html` из примера 49 осим стандардних атрибута садржи додатне атрибуте који се користе за генерисање садржаја приказа. Атрибут `th:object` има вредност `$student` и омогућава приступ атрибуту захтева који има назив `student`. Атрибут `th:field` служи за постављање вредности поља за унос на вредност поља објекта које је наведено као вредност атрибута. Атрибут `th:each` служи за итерацију кроз колекцију елемената.

Атрибут `method` у оквиру етикете `form` има вредност `POST`, па је у оквиру класе `FormController` потребно написати метод који ће бити задужен за обраду `POST` захтева који прегледач шаље након што студент попуни форму и одабере опцију `Prijavite ispite`. Овај метод приказан је у примеру 50.

Пример 50: Класа `FormController` са методом за обраду `POST` захтева

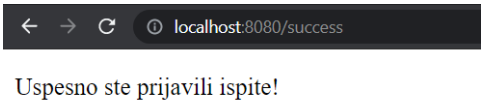
```
@Controller
public class FormController {
    @ModelAttribute
    public void napraviListuIspita(ModelMap model) {
        List<String> ispiti = Arrays.asList("Linearna algebra",
                              "Geometrija 1", "Programiranje 2",
                              "Kompleksne funkcije", "Analiza 1");
        model.put("ispiti", ispiti);
    }

    @GetMapping("/form")
    public ModelAndView prikaziFormu(ModelMap model) {
        model.put("student", new Student());
        return new ModelAndView("form", model);
    }

    @PostMapping("/form")
    public String sacuvajPrijavu(Student student) {
        return "redirect:success";
    }

    @RequestMapping("/success")
    public String redirektuj() {
        return "success";
    }
}
```

Класа `FormController` је проширена са два метода, `sacuvajPrijavu(Student student)` и `redirektuj()`. Након што студент одабере опцију `Prijavite ispite`, вредности са форме се мапирају у објекат класе `Student`, а затим се позива метод `sacuvajPrijavu(Student student)` који преусмерава корисника на страницу `/success`. Након позива метода `sacuvajPrijavu(Student student)`, *Spring* шаље одговор прегледачу са статусом `302`, односно редирекцију за нову страницу. Претраживач затим шаље нови захтев на путању `/success`. На слици 20 приказан је изглед странице која се приказује након што студент одабере опцију `Prijavite ispite`.



Слика 20: *Страница на путањи* `/success`

У имплементацији форме из примера 49 не постоји провера исправности података које је студент унео, па се може десити да студент одабере опцију `Prijavite ispite`, а да претходно није унео све потребне податке.

Spring подржава *Java Bean Validation API* који нуди велики број анотација које се примењују на поља класе и дефинишу правила за валидацију. Како би се користиле доступне анотације, потребно је додати одговарајућу зависност у оквиру датотеке `pom.xml`, као у примеру 51.

Пример 51: Део `pom.xml` датотеке

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

Следеће анотације представљају један део скупа доступних анотација.

- `@NotNull` - означава да вредност поља не може бити `null`;
- `@Size` - дефинише величину вредности поља;
- `@Past` - означава да вредност поља треба да буде датум из прошлости;
- `@Future` - означава да вредност поља треба да буде датум из будућности;
- `@Max` - дефинише максималну вредност поља;
- `@Min` - дефинише минималну вредност поља;
- `@Pattern` - дефинише образац који се користи за проверу исправности вредности поља;
- `@NotBlank` - примењује се на поља типа `String` и означава да вредност поља не може бити `null` и мора да садржи бар један карактер који није белина;
- `@Email` - означава да вредност поља треба да буде исправна *e-mail* адреса;
- `@Positive` - означава да вредност поља треба да буде позитиван број.

Пример 52 илуструје употребу анотација за валидацију над пољима класе `Student`. Атрибут `message` садржи поруку која ће бити приказана уколико унети подаци нису исправни.

Пример 52: Класа `Student` са анотацијама за валидацију поља

```
public class Student {

    @NotBlank(message = "Polje ime mora biti popunjeno")
    private String ime;
    @NotBlank(message = "Polje prezime mora biti popunjeno")
    private String prezime;
    @NotBlank(message = "Polje indeks mora biti popunjeno")
    private String indeks;
    @Size(min = 1, message = "Morate odabrati bar jedan ispit")
    private List<String> prijavljeniIspiti;

    ...
}
```

Како би се одрадила валидација унетих података, параметру `student` метода `sacuvajPrijavu(Student student)` потребно је додати анотацију `@Valid`. Уколико вредности поља објекта `student` нису у складу са дефинисаним правилима, подаци о грешкама биће уписани у објекту класе `Errors`.

У примеру 53 приказан је коначан облик класе `FormController`.

Пример 53: Класа `FormController` са методом за валидацију

```
@Controller
public class FormController {
    @ModelAttribute
    public void napraviListuIspita(ModelMap model) {
        List<String> ispiti = Arrays.asList("Linearna algebra",
                              "Geometrija 1", "Programiranje 2",
                              "Kompleksne funkcije", "Analiza 1");
        model.put("ispiti", ispiti);
    }

    @GetMapping("/form")
    public ModelAndView prikaziFormu(ModelMap model) {
        model.put("student", new Student());
        return new ModelAndView("form", model);
    }

    @PostMapping("/form")
    public String sacuvajPrijavu(
            @Valid @ModelAttribute("student") Student student,
            Errors errors) {
        if (errors.hasErrors()) {
            return "form";
        }
        return "redirect:success";
    }
```

```
    @RequestMapping("/success")
    public String redirektuj(Student student, ModelMap model) {
        return "success";
    }
}
```

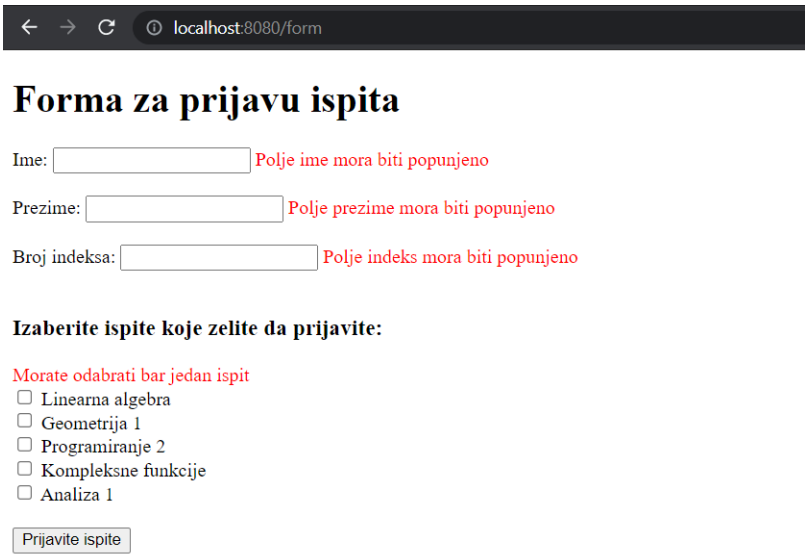
У примеру 54 приказан је изглед датотеке `form.html` која садржи потребне *Thymeleaf* атрибуте како би се студенту приказале поруке о неисправности унетих података.

Пример 54: Датотека `form.html` са атрибутима за приказ грешака

```
<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhtml"
       xmlns:th="http://www.thymeleaf.org">
<head>
    <title>Prijava ispita</title>
</head>
<body>
    <h1>Forma za prijavu ispita</h1>
    <form method="POST" th:object="${student}">
        <label for="ime">Ime: </label>
        <input type="text" th:field="*{ime}"/>
        <span style="color:red"
            th:if="${#fields.hasErrors('ime')}"
            th:errors="*{ime}"></span>
        <br/><br/>
        <label for="prezime">Prezime: </label>
        <input type="text" th:field="*{prezime}"/>
        <span style="color:red"
            th:if="${#fields.hasErrors('prezime')}"
            th:errors="*{prezime}"></span>
        <br/><br/>
        <label for="indeks">Broj indeksa: </label>
        <input type="text" th:field="*{indeks}"/>
        <span style="color:red"
            th:if="${#fields.hasErrors('indeks')}"
            th:errors="*{indeks}"></span>
        <br/><br/>
        <div class="grid">
            <div class="lista-ispita" id="ispiti">
                <h3>Izaberite ispite koje zelite da prijavite:</h3>
                <span style="color:red"
                    th:if="${#fields.hasErrors('prijavljeniIspiti')}"
                    th:errors="*{prijavljeniIspiti}">
                </span>
                <div th:each="ispit: ${ispiti}">
                    <input th:field="*{prijavljeniIspiti}"
```

```
                    type="checkbox" th:value="${ispit}" />
                <span th:text="${ispit}"></span>
                <br />
            </div><br />
            <button>Prijavite ispite</button>
        </div>
    </div>
</form>
</body>
</html>
```

На слици 21 приказан је изглед странице у случају када студент није унео исправне податке.



Слика 21: *Приказ порука о неисправности унетих података*

Закључак

У данашње време развој великих пословних апликација је незамислив без употребе радних оквира и библиотека. Радни оквир *Spring* се могућностима које пружа посебно издваја у свету *Java* апликација, па је његово познавање веома тражено на тржишту рада.

Електронске лекције о радном оквиру *Spring* имају циљ да корисницима олакшају усвајање његових концепата и да их оспособе да самостално развијају једноставне веб апликације. Лекције су пажљиво осмишљене тако да свака лекција садржи теоријско објашњење теме коју обрађује, као и примере који илуструју дата објашњења. Циљ оваквог приступа је да лекције буду разумљиве свима који се први пут сусрећу са радним оквиром *Spring*. Детаљним објашњењем напредних концепата програмирања и основа радног оквира *Spring*, корисник би требало да стекне фундаментално знање које му омогућава да развија функционалне веб апликације, а требало би да представља и добру основу за наставак учења напреднијих концепата радног оквира *Spring*. Апликација креирана на крају рада, као и *Spring* алати који се користе за развој апликације показују колико радни оквир *Spring* олакшава развој *Java* апликација, што га чини једним од најпопуларнијих радних оквира.

Литература за учење радног оквира *Spring* је обимна и најчешће доступна на енглеском језику, што захтева добро разумевање стране литературе. Предност електронских лекција, које су креиране у оквиру израде овог мастер рада, у односу на велики број материјала доступних на интернету је то што су лекције написане на српском језику, што би требало да значајно убрза процес учења свим корисницима.

Литература

[1] Jurić Nemanja, Marić Miroslav. *eŠkola Veba*. Matematički fakultet, Beograd, 2016.

[2] Rod Johnson. *Expert One-on-One J2EE Design and Development*. Wrox Press Ltd., GBR, 2002.

[3] Craig Walls. *Spring in Action*. Manning Publications Co., USA, 4th edition, 2014.

[4] Званична *Spring Framework* документација. `https://docs.spring.io/spring-framework/docs/5.3.9/reference/html/web.html`. Приступано 31. августа 2021.

[5] Craig Walls. *Spring Boot in Action*. Manning Publications Co., USA, 2015.

[6] Званична *Spring Boot* документација. `https://docs.spring.io/spring-boot/docs/current/reference/html/`. Приступано 31. августа 2021.

[7] Ranga Rao Karanam. *Naučite Spring 5*. Kompjuter biblioteka, Beograd, Srbija, 2017.

[8] Craig Walls. *Spring in Action*. Manning Publications Co., USA, 5th edition, 2018.

[9] Craig Walls. *Spring in Action*. Manning Publications Co., USA, 6th edition, 2021.