Razvoj web aplikacija pomoću razvojnog okvira Java Spring Boot

Kurilić, Jelena

Razvoj web aplikacija pomoću razvojnog okvira Java Spring Boot

Kurilić, Jelena

SVEUČILIŠTE U ZAGREBU

PRIRODOSLOVNO–MATEMATIČKI FAKULTET

MATEMATIČKI ODSJEK

Jelena Kurilić

RAZVOJ WEB APLIKACIJA POMOĆU RAZVOJNOG OKVIRA JAVA SPRING BOOT

Diplomski rad

Voditelj rada:
dr. sc. Goran Igaly

Zagreb, veljača, 2021

Ovaj diplomski rad obranjen je dana _____ pred ispitnim povjerenstvom u sastavu:

1. _____, predsjednik
2. _____, član
3. _____, član

Povjerenstvo je rad ocijenilo ocjenom _____.

Potpisi članova povjerenstva:

1. _____
2. _____
3. _____

Mami, tati, sestri i Mateju koji su bili velika podrška tijekom cijelog studija

Sadržaj

Uvod

Web aplikacija je aplikacija koja prati klasičnu klijent-server paradigmu i pokreće se u pregledniku. Preglednik prikazuje podatke primljene od poslužitelja i vraća korisničke podatke natrag. Glavna prednost ovog pristupa je da klijent ne ovisi o operativnom sustavu, što omogućuje korištenje web aplikacija na različitim platformama. Zbog svoje univerzalnosti, web aplikacije postale su vrlo popularne 90-tih godina prošlog stoljeća.

Većina web aplikacija zahtjeva implementaciju niza osnovnih ponavljajućih zadataka kao što je na primjer konekcija s bazom podataka. Kako razvojni programer ne bi iznova razvijao funkcionalnosti, ponavljajući dijelovi koda se generaliziraju što dovodi do razvoja aplikacijskih okvira. Danas za većinu programskih jezika postoji barem jedan aplikacijski okvir tako se za razvoj Java web aplikacija često koristi Spring Boot okvir. Spring Boot nudi novu paradigmu koja osigurava brže pisanje kraćeg i kompaktnijeg koda.

Na početku ovog rada reći ćemo nešto o razvojnim okvirima i o njihovim prednostima i manama. U drugom poglavlju navest ćemo kratku povijest Spring-a i Spring Boot-a te opisat osnovne pojmove vezane uz Spring okvir i izradu web aplikacije. Zatim ćemo opisati Spring Boot aplikacijski okvir i njegove značajke: automatsku konfiguraciju, *starter* ovisnosti i aktuator. Na kraju ćemo predstaviti razvijenu aplikaciju, njenu funkcionalnost i implementaciju.

Poglavlje 1

Razvojni okviri

1.1 Definicija i karakteristike

Razvojni okvir (eng. *application framework*) je softverska biblioteka koja pruža temeljnu strukturu i podršku za razvoj aplikacija u određenom okruženju. Razvojni okvir predstavlja temelj za izgradnju aplikacija. Glavni cilj razvojnog okvira je smanjiti općenita pitanja s kojima se programeri susreću tijekom razvoja aplikacije korištenjem koda koji se može dijeliti između različitih modula aplikacije. Razvojni okviri ne koriste se samo u razvoju korisničkog sučelja (eng. *graphical user interface*), već i u drugim područjima poput razvoja web aplikacija. [1]

Prilikom razvijanja web aplikacija programer se susreće s nizom osnovnih zadataka koji se ponavljaju. Ponavljajući dijelovi koda često se odvajaju u posebne metode i procedure, generaliziraju se i omogućuje se njihovo korištenje na više različitih projekata. Postupak generalizacije i ponovne upotrebe dovodi do stvaranja većine razvojnih okvira čija je primarna zadaća pomoći i olakšati razvojnom programeru razvoj aplikacije.

Karakteristike razvojnog okvira

Postupak generalizacije se na prvi pogled čini jednostavnim: uoče se dijelovi koda koji se stalno ponavljaju i odvoji ih se u elemente koji se mogu ponovno koristiti. Treba paziti da odvojeni dijelovi zajedno moraju činiti smislenu cjelinu, odnosno trebaju se moći lako iskoristiti u razvijanju različitih aplikacija tako da olakšaju cjelokupni razvoj. Kako bi razvojni okvir služio svojoj svrsi, poželjno je da ima sljedeće karakteristike [9]:

- Mora imati jednostavan mehanizam proširenja - mora omogućiti lako dodavanje novih mogućnosti i usluga od strane programera. Od aplikacijskog okvira se zahtijeva

da je proširiv, što znači da nove funkcionalnosti mogu lako dodati, a da se pritom ne naruši osnovna struktura.

- Mora biti jednostavan - pretjerana složenost okvira ne smije ugroziti njegovu primarnu i stvarnu korist. Aplikacijski okvir može biti složen, ali i dalje u konačnici mora dovesti do jednostavnijeg i bržeg razvoja aplikacije.
- Mora imati dobru dokumentaciju - sve funkcije i dijelovi okvira moraju biti dokumentirani. Dobra dokumentacija omogućava lakše učenje i korištenje aplikacijskog okvira, posebno ako se radi o složenom okviru.
- Treba biti široko primjenjiv - mora imati mogućnosti primjene u različitim razvojnim scenarijima.
- Treba generički pristupiti domeni problema - ako se aplikacijski okvir fokusira na poslovnu logiku određene aplikacije, tada će biti manje široko primjenjiv. Ako okvir sadrži poslovnu logiku, tada ona mora biti opcionalna ili zamjenjiva inače će njegova primjenjivost biti znatno ograničena.

Java razvojni okviri

Java razvojni okvir (eng. *Java Framework*) je razvojni okvir koji se koristi za stvaranje aplikacija pomoću programskog jezika Java. Java je objektno orijentirani programski jezik koji je danas u širokoj upotrebi. Jedan od glavnih razloga njegove opće prihvaćenosti je potpora za višeplatformnost. Java je robustan, siguran i relativno jednostavan jezik s obzirom na druge objektno orijentirane programske jezike kao što su C, C++. Danas se koristi za razvoj desktop, mobilnih i web aplikacija.

Današnje Java web aplikacije se uglavnom razvijaju uz pomoć nekog razvojnog okvira zbog sve veće potražnje za brži razvojem aplikacija i prototipova. Postoje brojni Java okviri koji se mogu upotrijebiti u tu svrhu, jedni od najpopularnijih su Spring, Grails, Play, Struts, JavaServer Faces (JSF), Google Web Toolkit (GWT). Ovisno o potrebama i aplikaciji koja se razvija odabire se odgovarajući aplikacijski okvir.

1.2 Prednosti i mane

Zbog sve bržeg razvoja tehnologije dolazi do zahtjeva za brzi razvoj softvera i aplikacija. Jedna od najznačajnijih stvari koji donosi korištenje aplikacijskog okvira je upravo brži razvoj. Vrijeme potrebno za razvoj aplikacije pomoću okvira može biti značajno manje. Razvojni okvir najčešće sadrži već definirane klase i funkcije koje se mogu koristi pa je potrebno napisati znatno manje koda za istu funkcionalnost.

Također, korištenje već dobro testiranih i provjerenih dijelova koda osigurava sigurnost razvoja. Razvojni programeri se tako mogu usredotočiti na pisanje poslovne logike aplikacije, razvoj postaje jednostavniji i brži, a samim time i učinkovitiji.

Glavna karakteristika brojnih okvira je dobra dokumentaciju koja je jednostavna, razumljiva i sveobuhvatna što omogućuje lako snalaženje te brzo i jednostavno učenje.

Popularni okviri često imaju dobar tim za podršku koji komunicira s korisnicima te prati njihove zahtjeve. Postoje i velike forumske zajednice gdje je moguće postavljati tehnička pitanja i dobiti pomoć od drugih korisnika. Ako aplikacijski okvir ima velik broj aktivnih korisnika, dolazi do brzog otkrivanja grešaka i poboljšanja sigurnosne implementacije.

Brojni popularni okviri su besplatni, a pošto omogućavaju brži razvoj, trošak samog razvoja za krajnjeg klijenta bit će manji.

Iako aplikacijski okviri garantiraju kraće vrijeme razvoja, treba naglasiti da su neki okviri složeni te je potrebno puno vremena da se nauče i tada je sveukupno vrijeme razvoja duže. Posebno ako se uči okvir samo da bi se primijenio na nekoliko projekata i ne planira se koristiti u budućnosti, tada će ukupni razvoj biti duži i skuplji.

Nadalje, učenjem pojedinog aplikacijskog okvira uči se sam okvir, a ne i programski jezik. Programer koji koristi okvir i vrlo malo zna o jeziku koji stoji iza njega teško će moći nadograditi nove funkcionalnosti koje okvir ne pruža. Oslanja se na funkcionalnosti koje postoje i u opasnosti je da izgubi razumijevanje o tome kako stvari zapravo funkcioniraju ”ispod haube”.

Još jedna mana je nemogućnost mijenjanja osobina i ponašanja samog okvira. Programeru nedostaje opcija koja bi mu omogućila promjenu temeljnog ponašanja okvira. Neki okviri od programera zahtijevaju korištenje određenih alata i oblikovnih obrazaca. Korištenjem određenog okvira programer je prisiljen poštivati njegova ograničenja i raditi onako kako on zahtijeva, stoga je jako bitno odabrati razvojni okvir koji odgovara potrebama.

Poglavlje 2

Spring

2.1 Povijest Spring-a i Spring Boot-a

U listopadu 2002, Rod Johnson je napisao knjigu *Expert One-On-One J2EE Design and Development*. Knjiga je pokrivala razvoj *Java enterprise* aplikacija u tom vremenu i ukazivala na nedostatke *Java Enterprise Edition* i *Enterprise Java Beans* komponenti. U knjizi je predloženo jednostavno rješenje bazirano na Java klasama i oblikovnom obrascu injektiranje ovisnosti (eng. *dependency injection*). U knjizi je opisao kako se može izgraditi kvalitetna, skalabilna aplikacija bez korištenja *Enterprise Java Beans*. Za izgradnju takve aplikacije u knjizi su opisana brojna Java sučelja i klase za višekratnu upotrebu. Principi koji su opisani u toj knjizi i danas se koriste prilikom izgradnje kvalitetnih Java web aplikacija.

Ubrzo nakon izlaska knjige počinje se raditi na projektu otvorenog koda (eng. *open-source project*) koji je bio baziran na knjizi. Projekt dobiva naziv Spring, predstavljao je novi početak nakon stagnacije tradicionalnog J2EE (*Java 2 Enterprise Edition*). Projekt je izašao u javnost 2003., a Spring 1.0 početkom 2004. godine. Vrlo brzo Spring je postao široko prihvaćen jer je obećavao jednostavnost razvoja i istovremeno je rješavao neke vrlo zamršene probleme. Spring se nakon toga počeo naglo razvijati: Spring 2.0 listopad 2006, Spring 2.5 studeni 2007, Spring 3.0 prosinac 2009, Spring 4.0 prosinac 2013 i Spring 5.0 studeni 2017.

Činilo se da nakon svakog izdanja Spring nikako ne može biti bolji, Spring Boot dokazao je da je u Spring-u ostalo još puno potencijala. Spring Boot je jedna od najznačajnijih stvari koja se dogodila Spring-u. U listopadu 2012. stvorio se zahtjev u kojem se govori o mogućnosti značajnog pojednostavljenja Spring okvira što je dovelo do početka razvoja Spring Boot projekta početkom 2013. U travnju 2014. godine izašao je Spring Boot 1.0.

Spring Boot je omogućio nekoliko velikih stvari, jedna od njih je automatska konfiguracija - nema potrebe pisati konfiguraciju za uobičajene konfiguracijske scenarije, Spring će to učiniti sam. Zatim je tu dio *starter* ovisnost koji se brine o uključivanju svih potrebnih biblioteka odgovarajućih verzija te aktuator koji daje uvid u unutarnji rad pokrenute aplikacije.

Sljedeća velika verzija Spring Boot-a, Spring Boot 2.0 izašla je u studenom 2017. godine.

2.2 Uvod u Spring

Aplikacijski okvir Spring donio je brojne pozitivne stvari. Nekoliko je temeljenih ideja Spring-a, ali sve su usredotočene na jedan cilj: pojednostaviti razvoj aplikacija u Javi. Kako bi to postigao, Spring koristi sljedeće četiri strategije [10]:

- lagan i jednostavan razvoj koristeći obične Java objekte
- labavo povezivanje objekata ostvareno kroz injektiranje ovisnosti (eng. *dependency injection*) i korištenjem sučelja
- deklarativno programiranje kroz aspekte i uobičajene konvencije
- eliminiranje ponavljajućeg koda koristeći aspekte i predloške.

Spring omogućuje rad s običnim Java objektima (eng. *Plain Old Java Objects* - POJO) koji ne trebaju nasljeđivati Spring-ovu klasu ili implementirati Spring-ovo sučelje. Klase koje se koriste u Spring aplikaciji često nemaju nikakvog pokazatelja da se radi o Spring-ovoj komponenti osim eventualno anotacije. Kod 2.1 prikazuje primjer Spring-ove klase.

```
public class HelloWorldBean {

    public String sayHello() {
        return "Hello World";
    }
}
```

Kod 2.1: Klasa u Spring aplikaciji

Iako Java objekti imaju poprilično jednostavnu formu, oni čine temelj za gradnju Spring aplikacija.

Injektiranje ovisnosti

Svaka netrivijalna aplikacija sastoji se od dvije ili više klasa koje međusobno surađuju kako bi ostvarile neku poslovnu logiku. Uobičajeno je svaki objekt zadužen za dobivanje reference drugog objekta s kojim surađuje. To može dovesti do čvrstog povezivanje objekata (eng. *highly coupled*) i stvaranja koda kojeg je teško testirati. Međutim, potrebna je određena povezanost među objektima.

Koristeći obrazac injektiranja ovisnosti, objekt dobiva referencu na objekt s kojim surađuje od koordinatora svih objekata u sustavu. Od objekta se ne očekuje da se brine o stvaranju i održavanju povezanosti s drugim objektima.

Spring nudi spremnik koji je zadužen za stvaranje i upravljanje aplikacijskim komponentama. Povezanost unutar spremnika je ostvarena pomoću obrasca injektiranja ovisnosti. Komponente ne stvaraju i održavaju životni ciklus drugih komponenti o kojima ovise već spremnik to čini za njih. [10]

Aspektno orijentirano programiranje

Aspektno orijentirano programiranje (eng. *aspect oriented programming* - AOP) nadopunjuje objektno orijentirano programiranje (OOP) pružajući drugi način razmišljanja o strukturi programa. Ključna jedinica modularnosti u OOP-u je klasa, dok je u AOP-u aspekt. OOP omogućuje izdvajanje pojedinih funkcionalnosti u klase koje se mogu ponovno koristiti. Nekad to nije dovoljno, odnosno aplikacije često moraju izvršiti određenu logiku prilikom poziva svojih metoda (npr. provjera autorizacije korisnika ili upravljanje transakcijama). Za takve potrebe razvijena je aspektno orijentirana paradigma. Aspekt predstavlja zasebnu funkcionalnu jedinicu koja nije nužno dio glavne logike aplikacije, ali je rasprostranjena po dijelovima rješenja.

AOP okvir predstavlja jednu od ključnih komponenti Spring-a. [10]

2.3 Spring spremnik

U aplikaciji koja se temelji na Spring-u, aplikacijski objekti žive u Spring spremniku. Spremnik predstavlja srž Spring okvira, stvara objekte, povezuje ih, konfigurira i upravlja njihovim životnim ciklusom. Ti objekti nazivaju se Spring *beans*.

Spremnik dobiva upute koje objekte treba stvoriti, konfigurirati i sastaviti na temelju konfiguracije. Koristeći Java objekte i konfiguracijske upute stvara se aplikacija.

Spring dolazi s nekoliko implementacija spremnika koje se mogu svrstati u dvije vrste:

- *Bean factory* - definiran je sučeljem *org.springframework.beans.factory.BeanFactory*. Predstavlja najjednostavniji spremnik koji pruža osnovnu potporu za injektiranu ovisnost.
- Aplikacijski kontekst (eng. *application context*) - definiran je sučeljem *org.springframework.context.ApplicationContext*. Temelji se na *Bean factory*-ju te proširuje njegove značajke. [7]

Iako je moguće koristiti obje vrste spremnika, aplikacijski kontekst sadrži *Bean factory* s dodatnim funkcionalnostima stoga se češće koristi u današnjim aplikacijama.

Konfiguracija *bean*-ova

Da bi Spring spremnik znao kako stvoriti i upravljati *bean*-ovima, potrebne su mu upute. Spring može dobiti konfiguracijske upute na sljedeća tri načina [10]:

- XML konfiguracija
- Java konfiguracija
- Implicitno otkrivanje *bean*-ova i automatsko povezivanje.

XML konfiguracija

Od početka Spring-a XML je bio primarni način zadavanja konfiguracije. Bezbroj redaka XML-a napisano je u svrhu konfiguriranja Spring-a te je za mnoge Spring postao sinonim za XML konfiguraciju.

Osnovna XML konfiguracija izgleda ovako:

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

    <!-- ovdje idu konfiguracijski detalji -->

</beans>
```

To je ponavljajući dio XML konfiguracije koji sadrži svaka XML datoteka. <*beans*> predstavlja korijenski element XML konfiguracijske datoteke. Za definiranje *bean*-a unutar XML konfiguracije potrebno je koristiti <*bean*> element unutar <*beans*> elementa.

Na primjer, za kreiranje *bean*-a *InventoryService* potrebno je sljedeće:

```
<bean id="inventoryService"
 class="com.example.InventoryService" />
```

Dok je za kreiranje *bean*-a *ProductService* koji sadrži referencu na *InventoryService* potrebno sljedeće:

```
<bean id="productService"
 class="com.example.ProductService" />
 <constructor-arg ref="inventoryService" />
</bean>
```

Java konfiguracija

Java konfiguracija omogućuje pisanje Spring-ove konfiguracije unutar Java koda uz pomoć Java anotacija.

Anotiranje klase s *@Configuration* omogućuje Spring spremniku da ju može koristiti kao izvor definiranja *bean*-ova. Anotacija *@Bean* govori Spring-u da će metoda vratiti objekt koji bi trebao biti registriran kao *bean* u Spring spremniku. Najjednostavniji primjer Java konfiguracije za kreiranje *bean*-a *InventoryService* bio bi sljedeći:

```
@Configuration
public class ServiceConfiguration {

 @Bean
 public InventoryService inventoryService() {
    return new InventoryService();
 }

}
```

Dok je za kreiranje *bean*-a *ProductService* koji sadrži referencu na *InventoryService* potrebno sljedeće:

```
 @Bean
 public ProductService productService() {
    return new ProductService(inventoryService());
 }
```

Implicitno otkrivanje *bean*-ova i automatsko povezivanje

Otkrivanje i povezivanje *bean*-ova odvija se pomoću sljedećih Spring-ovih tehnika:

- skeniranje komponenti (eng. *component scanning*) - Spring može otkriti koje komponente su dostupne i stvoriti ih kao *bean*-ove u Spring aplikacijskom kontekstu.

- automatsko povezivanje (eng. *autowiring*) - Spring automatski injektira *bean*-ove u pojedine komponente.

Implicitno otkrivanje *bean*-ova temelji se na anotaciji *@Component*. Ova jednostavna anotacija identificira klasu kao komponentu i služi Spring-u kao uputa za stavaranje *bean*-a u aplikacijskom kontekstu. Na primjer, za stvaranje *bean*-a *InventoryService* potrebno je samo klasi dodati anotaciju *@Component*:

```
@Component
public class InventoryService {
    ...
}
```

Međutim, skeniranje komponenti nije automatski uključeno. Potrebno je reći Spring-u da traži klase označene s *@Component* i od njih stvori *bean*-ove. Sljedeća klasa predstavlja minimalnu konfiguraciju koja to omogućuje:

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

Automatsko povezivanje *bean*-ova temelji se na *@Autowired* anotaciji. Ako je konstruktor klase *ProductService* anotiran s *@Autowired*, tada *bean ProductService* automatski dobiva referencu na *InventoryService*:

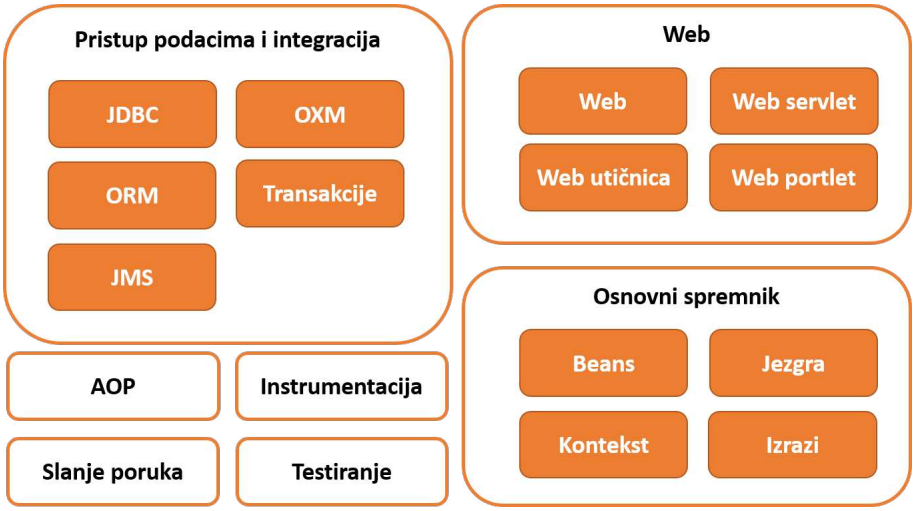
```
@Component
public class ProductService {

    private InventoryService inventoryService;

    @Autowired
    public ProductService(InventoryService inventoryService) {
        this.inventoryService = inventoryService;
    }
}
```

2.4 Spring moduli

Spring okvir sadrži brojne značajke koje su dobro organizirane u dvadesetak modula. Moduli se mogu grupirati na temelju njihovih primarnih značajki u osnovni spremnik (eng. *Core Container*), pristup podacima i integracija (eng. *Data Access and Integration*), web, aspektno orijentirano programiranje (eng. *Aspect Oriented Programming*), instrumentacija (eng. *Instrumentation*), slanje poruka (eng. *Messaging*) i testiranje (eng. *Test*) kao što je prikazano na slici 2.1. [8]



Slika 2.1: Spring moduli

Osnovni spremnik

Osnovni Spring spremnik predstavlja središnji dio Spring okvira koji upravlja *bean*-ovim. U ovome modulu se nalazi *Bean factory* kao i nekoliko implementacija Spring aplikacijskog konteksta. Nadalje, modul sadrži i usluge poput email-a, JNDI (*Java Naming and Directory Interface*) pristupa, EJB (*Enterprise Java Beans*) integracije i vremenskog planiranja. [8]

Pristup podacima i integracija

Rad s JDBC-om (*Java Database Connectivity*) često uzrokuje pisanje ponavljajućeg koda (stvaranje konekcije, pisanje upita, zatvaranje konekcije itd.). Spring-ov JDBC i pristup podacima (eng. *Data access objects* - DAO) omogućuje stvaranje čišćeg i jednostavnijeg koda te izbjegavanje čestih grešaka prilikom rada s bazom podataka.

U ovom modulu se nalazi i podrška za objektno-relacijsko mapiranje (ORM). Spring ne nudi vlastiti implementaciju ORM rješenja, ali nudi podršku za nekoliko popularnih ORM okvira kao što su: Hibernate, Java Persistence API, Java Data Objects i iBATIS SQL Maps. [8]

Web

Paradigma Model-View-Controller (MVC) je uobičajeni pristup izradi web aplikacija u kojem je korisničko sučelje odvojeno od aplikacijske logike.

Iako se Spring integrira s nekoliko popularnih MVC okvira, njegov modul za web i pristup udaljenim resursima dolazi s MVC okvirom u kojem su prisutne standardne Springove značajke.

Također, Spring nudi i podršku za izradu REST (eng. *Representational state transfer*) aplikacijskog programskog sučelja. [8]

Aspektno orijentirano programiranje i instrumentacija

Spring pruža bogatu podršku za aspektno orijentirano programiranje u svom AOP modulu. Ovaj modul služi kao osnova za razvijanje vlastitih aspekata za Spring aplikaciju.

Spring instrumentacijski modul nudi podršku za dodavanje dodatnih resursa Java virtualnom stroju. Ovaj modul se koristi u raznim aplikacijskim poslužiteljima. [8]

Slanje poruka

Ovaj modul sadrži apstrakcije poput *Message*, *MessageChannel*, *MessageHandler* i druge koje služe kao temelj za izgradnju aplikacije temeljene na porukama. [8]

Testiranje

Prepoznajući važnost pisanja testova, Spring pruža modul posvećen testiranju Spring aplikacija. U ovom modulu nalazi se kolekcija lažnih (eng. *mock*) implementacija objekata za pisanje jediničnih testova (eng. *unit test*).

Na razini testiranja integracije, modul osigurava podršku za učitavanje *bean*-ova u aplikacijskom kontekstu i rad s njima. [8]

2.5 Spring projekti

Osim Spring okvira i njegovih različitih modula, postoje i drugi okviri nazvani Spring projekti. Ovi projekti pružaju rješenja za brojna pitanja s kojima se susreću današnje poslovne aplikacije.

Neki od najpoznatijih su: Spring Web Flow, Spring Web Services, Spring Security, Spring Integration, Spring Batch, Spring Data, Spring Social, Spring Mobile, Spring Boot itd.

Spring Security

Spring Security je okvir koji pruža cjelovito sigurnosno rješenje za Spring aplikaciju. Omogućuje provjeru autentičnosti i autorizacije na razini web zahtjeva i na razini pozivanja metoda. Po uzoru na Spring okvir, Spring Security koristi injektiranu ovisnost i aspektno orijentirano programiranje.

Na početku Spring Security-a bilo je potrebno napisati stotine linija XML konfiguracije da bi se omogućila sigurnost web aplikacije. Sa Spring-om 3.2 i Java konfiguracijom dovoljno je napisati sljedeće [4]:

```
import org.springframework.context.annotation.Configuration;
import org.springframework.security.config.annotation.web.configuration.
    EnableWebSecurity;
import org.springframework.security.config.annotation.web.configuration.
    WebSecurityConfigurerAdapter;

@Configuration
@EnableWebSecurity
public class SecurityConfig extends WebSecurityConfigurerAdapter {
}
```

Pomoću anotacije *@EnableWebSecurity* uključuje se web sigurnost. U slučaju Spring MVC aplikacije moguće je koristiti i *@EnableWebMvcSecurity*.

Spring Security mora biti konfiguriran unutar *bean*-a koji implementira *WebSecurityConfigurer* ili proširuje *WebSecurityConfigurerAdapter*. Bilo koji *bean* u aplikacijskom kontekstu koji implementira *WebSecurityConfigurer* može pridonijeti konfiguraciji Spring Security-a.

Također, Spring nudi mogućnost prilagodbe web sigurnosti. To se ostvaruje nadjačavanjem nekih od tri metoda *configure()* klase *WebSecurityConfigurerAdapter*. U tablici 2.1 opisane su navedene metode.

Metoda	**Opis**
configure(WebSecurity)	konfiguracija *filter chain*-a
configure(HttpSecurity)	konfiguracija presretanja zahtjeva
configure(AuthenticationManagerBuilder)	konfiguracija korisničkih detalja

Tablica 2.1: *configure()* metode klase *WebSecurityConfigurerAdapter*

Jedan način nadjačavanja metode *configure(HttpSecurity)* može biti sljedeći:

```
@Override
protected void configure(HttpSecurity http) throws Exception {
    http
    .authorizeRequests()
    .antMatchers("/profile/me").authenticated()
    .anyRequest().permitAll();
}
```

HTTP zahtjev "/profile/me" mora biti autentificiran, dok ostali ne moraju.

U sljedećem primjeru, *SecurityConfig* nadjačava *configure(AuthenticationManagerBuilder)* metodu za spremanje dva korisnika u memoriju.

```
@Override
protected void configure(AuthenticationManagerBuilder auth) {
    auth
        .inMemoryAuthentication()
        .withUser("user").password("password").roles("USER").and()
        .withUser("admin").password("password").roles("USER", "ADMIN");
}
```

Spring Data

Spring Data olakšava rad sa svim vrstama baza podataka u Spring-u. Iako je relacijska baza podataka sveprisutna u poslovnim aplikacijama već dugi niz godina, moderne aplikacije sve češće koriste NoSQL baze podataka. Spring Data nudi mehanizam automatskog spremišta za mnoge baze podataka, bez obzira radi li se o dokumentskoj bazi podataka poput MongoDB, grafovskoj bazi podataka poput Neo4j, ili tradicionalnoj relacijskoj bazi podataka.

Ovaj projekt sastoji se od brojnih potprojekata koji su specifični za određenu bazu podataka. Neki od potprojekata su: Spring Data JPA, Spring Data MongoDB, Spring Data Redis, Spring Data Neo4j.

Spring Data JPA

Prilikom rada s JDBC-om potrebno je mapirati objekte u odgovarajuće tablice što može biti poprilično zamorno. U tome mogu pomoći ORM opcije kao što su Hibernate i JPA.

Java Persistence API je Java specifikacija koja se koristi za mapiranje između Java objekata i relacijske baze podataka. Pošto je JPA samo specifikacija, potrebna joj je implementacija. Hibernate je ORM alat koji implementira JPA specifikaciju.

Spring Data JPA omogućuje jednostavnu implementaciju repozitorija i poboljšanu podršku pristupa podacima temeljenim na JPA.

Implemetacija jednog jednostavnog JPA entiteta u Spring-u izgleda ovako:

```
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;

@Entity
public class Customer {

  @Id
  @GeneratedValue(strategy=GenerationType.AUTO)
  private Long id;
  private String firstName;
  private String lastName;
  private String email;

  protected Customer() {}

  public Customer(String firstName, String lastName, String email) {
    this.firstName = firstName;
    this.lastName = lastName;
    this.email = email;
  }

  public Long getId() {
    return id;
  }

  public String getFirstName() {
    return firstName;
  }

  public String getLastName() {
    return lastName;
  }

  public String getEmail() {
    return email;
  }
}
```

Klasa *Customer* označena je s *@Entity*, što znači da je riječ o JPA entitetu. Atribut id je označen s *@Id* tako da ga JPA prepoznaje kao ID objekta. Također, id je označen s *@GeneratedValue* - ID se generira automatski. Preostala dva atributa su bez anotacija, mapiraju se u stupce tablice koji dijele ista imena kao i sami atributi.

Za pisanje Spring Data JPA repozitorija potrebno je proširiti neko od ponuđenih sučelja

npr. *JpaRepository*. Primjer repozitorija za *Customer* entitet bio bi sljedeći:

```
public interface CustomerRepository
    extends JpaRepository<Customer, Long> {
}
```

CustomerRepository nasljeđuje *JpaRepository*. Pritom je *JpaRepository* parametriziran tako da se zna da je to repozitorij za *Customer* objekt i da *Customer* ima ID tipa *Long*. Također, *CustomerRepository* nasljeđuje i 18 metoda za izvršavanje uobičajenih operacija kao što su spremanje, brisanje, pronalazak *Customer*-a itd. Nije potrebno pisati implementaciju navedenog sučelja, već to radi Spring Data.

Za dodavanje nove funkcionalnosti unutar repozitorija potrebno je samo navesti metodu odgovarajućeg imena s potrebnim parametrima. Na primjer, za dohvaćanje *Customer*-a po imenu potrebno je sljedeće:

```
public interface CustomerRepository
    extends JpaRepository<Customer, Long> {
    Customer findByFirstName(String firstName);
}
```

Prilikom stvaranja implementacije repozitorija, Spring Data raščlanjuje nazive metoda u sučelju repozitorija i pokušava razumjeti svrhu metode. Metoda uvijek mora počinjati s glagolom. Zatim slijedi subjekt koji je opcionalan potom riječ *By* i na kraju dolazi predikat. Predikat sadrži jedan ili više uvjeta koji ograničavaju rezultat. Svaki uvjet mora biti vezan za neki atribut i imati operator usporedbe. Ukoliko operator nije naveden uzima se operator jednakosti. Uvjeti unutar predikata povezuju se pomoću *And* i *Or*. [5] U slučaju metode *findByFirstName(String firstName)*, koristi se glagol *find*, a predikat je *FirstName*.

U situacijama kada se željeni podaci ne mogu adekvatno izraziti, može se upotrijebiti anotacija *@Query* kako bi se napisao željeni upit. [6] Na primjer, za metodu *findAllGmailCustomers()* može se koristiti sljedeći *Query*:

```
@Query("select c from Customer c where c.email like '%gmail.com'")
List<Customer> findAllGmailCustomers();
```

2.6 Spring MVC

Spring Web MVC okvir pruža arhitekturu Model-View-Controller (MVC) i gotove komponente koje se mogu koristiti za razvoj fleksibilnih i labavo povezanih web aplikacija. MVC obrazac rezultira odvajanjem različitih aspekata aplikacije (ulazna logika, poslovna logika i logika korisničkog sučelja), istovremeno pružajući labavu povezanost između tih elemenata.

Model je objekt koji prenosi podatke između upravljača i pogleda.

Pogled (eng. *view*) je odgovoran za prikazivanje podataka iz modela i generiranje HTML izlaza koji klijentov preglednik može protumačiti.

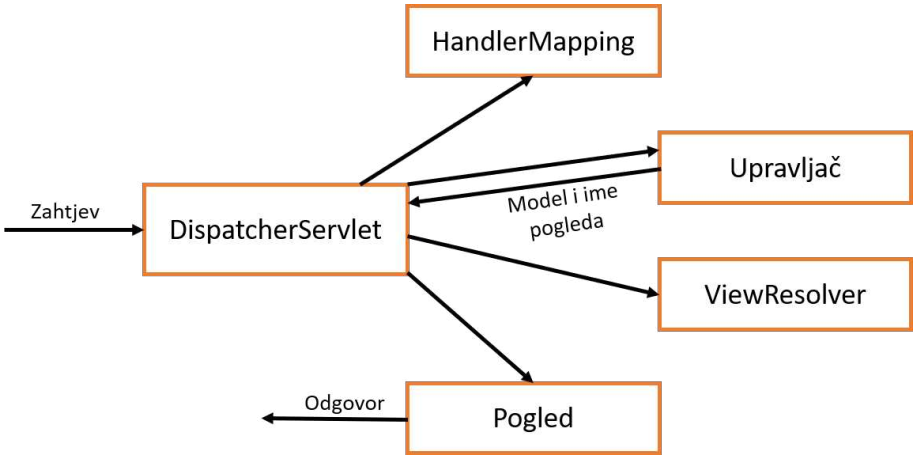
Upravljač (eng. *controller*) je odgovoran za poslovnu logiku aplikacije. Obrađuje pristigle zahtjeve te šalje podatke pogledu. [10]

DispatcherServlet

Spring MVC dizajniran je oko *DispatcherServlet*-a koji obrađuje sve HTTP zahtjeve. Slijed događaja koji odgovara dolaznom HTTP zahtjevu za *DispatcherServlet* je sljedeći:

- Nakon primanja HTTP zahtjeva, *DispatcherServlet* savjetuje se s *HandlerMapping*-om o pozivanju odgovarajućeg upravljača.
- Upravljač preuzima zahtjev i poziva odgovarajuću metodu servisa koja obavlja potrebnu poslovnu logiku. Provedena logika rezultira modelom kojeg upravljač, zajedno s imenom odgovarajućeg pogleda, vraća *DispatcherServlet*-u.
- *DispatcherServlet* traži pomoć *ViewResolver*-a za pronalazak odgovarajućeg pogleda na temelju dobivenoga imena.
- Nakon što je pogled pronađen, *DispatcherServlet* prosljeđuje podatke modela pogledu koji se zatim prikazuje u pregledniku.

Opisani postupak prikazan je dijagramom na slici 2.2. [10]



Slika 2.2: Obrada dolaznog HTTP zahtjeva

Sve spomenute komponente, *HandlerMapping*, *Controller* i *ViewResolver*, dijelovi su *WebApplicationContext* koji nasljeđuje *ApplicationContext* s nekim dodatnim značajkama potrebnim za web aplikacije.

Upravljač

DispatcherServlet šalje upravljaču zahtjev za izvršavanje određene funkcionalnosti. Klasa koja predstavlja upravljač u Spring aplikaciji anotirana je s *@Controller*. Anotacija *@RequestMapping* koristi se za povezivanje zahtjeva s odgovarajućim upravljačem.

Implemetacija jednog jednostavnog upravljača u Spring-u izgleda ovako:

```
@Controller
public class HelloController {

   @RequestMapping(value = "/hello", method = RequestMethod.GET)
   public String printHello(Model model) {
      model.addAttribute("message", "Hello!");
      return "hello";
   }
}
```

U ovom primjeru, GET zahtjev s putanjom ”/hello” biti će obrađen u *HelloController*-u, a poslovna logika koja se treba provesti dana je metodom *printHello(Model model)*. Kada upravljač završi s obradom vraća model koji sadrži atribut *message* i ime pogleda ”hello” *DispatcherServlet*-u.

Pogled

Logika rukovanja zahtjevima u upravljaču odvojena je od prikazivanja pogleda. Kada bi metode upravljača bile izravno odgovorne za proizvodnju HTML-a, tada bi se za održavanje pogleda moralo ulaziti u logiku zahtjeva. Stoga, upravljač zna jedino logičko ime pogleda, dok Spring pomoću *ViewResolver*-a određuje koji pogled se treba prikazati.

Upravo to omogućuje korištenje različitih tehnologija za razvijanje pogleda, od standardne Java Server Pages tehnologije do različitih mehanizama predložaka kao što su Thymeleaf, Groovy, FreeMarker, Jade.

Iako Java Server Pages dominira u brojnim aplikacijama, Thymeleaf mu u posljednje vrijeme postaje ozbiljna konkurencija. Predlošci koje nudi Thymeleaf su prirodni, mogu se uređivati i koristiti svugdje gdje bi se koristio i obični HTML.

Thymeleaf

Predlošci Thymeleaf-a su HTML datoteke. Thymeleaf dodaje atribute standardnom skupu HTML oznaka putem prilagođenog prostora imena.

Na primjer, *home.html* datoteka koja koristi Thymeleaf izgleda ovako:

```
<html xmlns="http://www.w3.org/1999/xhtml"
```

```
      xmlns:th="http://www.thymeleaf.org">
<head>
    <title>Spittr</title>
    <link rel="stylesheet" type="text/css"
          th:href="@{/resources/style.css}"></link>
</head>
<body>
    <h1>Welcome to My App</h1>
    <a th:href="@{/home}">Home</a> |
    <a th:href="@{/register}">Register</a>
</body>
</html>
```

Ovaj predložak relativno je jednostavan, jedini Thymeleaf atribut koji se koristi je *th:href* koji sliči HTML-ovom *href* atributu te se koristi na isti način. To će dovesti do prikaza standardnog *href* atributa s vrijednosti koja se dinamički kreira u vrijeme prikazivanja.

Za prikaz vrijednosti modela može se koristiti *th:text="${nazivAtributa}"*. Na primjer, za prikaz atributa *serverTime* koristi se sljedeći HTML kod:

```
Trenutno vrijeme je <span th:text="${serverTime}" />
```

Ako je atribut modela kolekcija objekata, tada se koristi *th:each* atribut. Na primjer, za iteriranje listom *customers* i prikazivanje informacija o svakom klijentu potrebno je sljedeće:

```
<tbody>
    <tr th:each="customer: ${customers}">
        <td th:text="${customer.id}" />
        <td th:text="${customer.email}" />
    </tr>
</tbody>
```

Thymeleaf nudi i uvjetne naredbe *if* i *unless*. Atribut *th:if = "${uvjet}"* koristi se za prikaz određenog dijela pogleda ako je uvjet zadovoljen. Atribut *th:unless= "${uvjet}"* koristi se za prikaz određenog dijela pogleda ako uvjet nije zadovoljen. Primjer korištenja uvjetnih naredbi može biti sljedeći:

```
<td>
    <span th:if="${customer.gender} == 'M'" th:text="Male" />
    <span th:unless="${customer.gender} == 'M'" th:text="Female" />
</td>
```

Još jedna uvjetna naredba koja je podržana je *switch-case*. Prethodni kod može se napisati i ovako:

```
<td th:switch="${customer.gender}">
    <span th:case="'M'" th:text="Male" />
    <span th:case="'F'" th:text="Female" />
</td>
```

U slučaju HTML forme mogu se koristiti atributi *th:action = "@{url}"* i *th:object = "${objekt}"*. *th:action* sadrži putanju na koju će se poslati podaci uneseni u formi. *th:object* koristi se za određivanje objekta na koji će se vezati poslani podaci. Pojedinačna polja se označavaju s *th:field=*{ime}*, gdje ime predstavlja odgovarajući atribut objekta. Na primjer:

```
<form action="#" th:action="@{/saveCustomer}" th:object="${customer}"
   method="post">
    <table border="1">
        <tr>
            <td><label th:text="#{msg.firstName}" /></td>
            <td><input type="text" th:field="*{firstName}" /></td>
        </tr>
        <tr>
            <td><label th:text="#{msg.lastName}" /></td>
            <td><input type="text" th:field="*{lastName}" /></td>
        </tr>
        <tr>
            <td><input type="submit" value="Submit" /></td>
        </tr>
    </table>
</form>
```

Poglavlje 3

Spring Boot

Spring Boot je aplikacijski okvir koja se temelji na Spring okviru i proširuje ga s dodatnim mogućnostima. Jedna od njegovih glavnih prednosti je mogućnost pisanja aplikacije bez ili s jako malo potrebne Spring konfiguracije. Spring Boot skenira dostupne ovisnosti i određuje koje *bean*-ove je potrebno kreirati da bi se ostvarile željene funkcionalnosti. Programer ne mora više trošiti vrijeme na pisanje ponavljajućeg koda već odmah može početi razvijati poslovnu logiku aplikacije.

Spring Boot donosi brojne mogućnosti u razvoju Spring aplikacija, među njima najznačajnije su sljedeće:

- automatska konfiguracija - osigurava automatsko stvaranje konfiguracija za funkcionalnosti koje su zajedničke brojnim Spring aplikacijama.
- *starter* ovisnost - uključivanjem *starter* ovisnosti za određenu funkcionalnost, aplikacija dobiva sve potrebne biblioteke odgovarajućih verzija.
- komandna linija - opcionalna funkcionalnost, omogućava pisanje cijele aplikacije samo pomoću aplikacijskog koda bez tradicionalnog *build*-a.
- aktuator - daje uvid u pozadinu rada pokrenute Spring Boot aplikacije.

Svaka od navedenih značajki pojednostavljuje i ubrzava razvojni proces na jedinstven način.

3.1 Automatska konfiguracija

Spring Boot automatska konfiguracija je *runtime* proces tijekom kojeg se, na temelju određenih uvjeta, odlučuje koja će se Spring konfiguracija primijeniti. Sljedeći primjeri ilustriraju rad Spring automatske konfiguracije:

- Je li Spring *JdbcTemplate* klasa dostupna? Ako je i ako postoji *DataSource bean*, onda automatski konfiguriraj *JdbcTemplate bean*
- Je li Thymeleaf dostupan? Ako je, onda konfiguriraj Thymeleaf postavke.
- Je li Spring Security dostupan? Ako je, onda konfiguriraj osnovne postavke web sigurnosti.

Spring Boot donosi gotovo 200 takvih odluka svaki put kad se aplikacija pokrene. Ove odluke pokrivaju područja sigurnosti, integracije, web razvoja i baze podataka.

Automatsko konfiguriranje aplikacije ostvaruje se na temelju dodanih *jar* ovisnosti. Na primjer, ako je prisutna MySQL ovisnost, ali ne postoji konfiguracija veze s MySQL bazom podataka, tada Spring Boot sam konfigurira bazu podataka u memoriji. U tu svrhu potrebno je dodati *@EnableAutoConfiguration* ili *@SpringBootApplication* anotaciju u glavnoj klasi programa (Kod 3.1). Dodavanjem jedne od ovih anotacija omogućuje se automatsko konfiguriranje Spring Boot aplikacije.

```
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.EnableAutoConfiguration;

@SpringBootApplication
public class DemoApplication {

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

}
```

Kod 3.1: Korištenje anotacije *@SpringBootApplication*

Način rada

Kada se aplikaciji doda Spring Boot, među ovisnostima nalazi se *jar* datoteka *spring-boot-autoconfiguration.jar* koja sadrži nekoliko konfiguracijskih klasa. Svaka od ovih konfiguracijskih klasa ima priliku doprinijeti konfiguraciji aplikacije. Postoji konfiguracija za Thymeleaf, konfiguracija za Spring Data JPA, konfiguracija za Spring MVC i konfiguracija za brojne druge stvari koje se mogu iskoristiti u Spring aplikaciji.

Ove konfiguracije koriste Spring-ovu podršku za uvjetnu konfiguraciju (eng. *conditional configuration*) koja je predstavljena u Spring-u 4.0. Uvjetna konfiguracija je konfiguracija koja je dostupna u aplikaciji samo ako je zadovoljen određeni uvjet.

Kod 3.2 prikazuje jedan primjer korištenja uvjetne konfiguracije. *MyService bean* bit će kreiran jedino ako je *JdbcTemplate* dostupan, inače ne.

```
@ConditionalOnClass(JdbcTemplateCondition.class)
public MyService myService() {
    ...
}
```

Kod 3.2: Uvjetna konfiguracija

Spring Boot definira još nekoliko konfiguracijskih uvjeta koje se koriste u auto-konfiguraciji. Odabirom odgovarajuće anotacije koristi se pojedini uvjet. Tablica 3.1 prikazuje neke od Spring Boot uvjetnih anotacija.

Anotacija	**Primijeni konfiguraciju ako ...**
@*ConditionalOnBean*	navedeni *bean* je konfiguriran
@*ConditionalOnMissingBean*	navedeni *bean* nije konfiguriran
@*ConditionalOnClass*	navedena klasa je dostupna u *classpath*-u
@*ConditionalOnMissingClass*	navedena klasa nije dostupna u *classpath*-u
@*ConditionalOnJava*	verzija Jave odgovara navedenoj verziji ili raspon verzija
@*ConditionalOnResource*	navedeni resurs je dostupan u *classpath*-u
@*ConditionalOnWebApplication*	aplikacija je web aplikacija
@*ConditionalOnNotWebApplication*	aplikacija nije web aplikacija

Tablica 3.1: Spring Boot uvjetne konfiguracije

Nadjačavanje (eng. *overriding*) automatske konfiguracije

U većini slučajeva, automatska konfiguracija daje sve što je potrebno za razvoj aplikacije. Osnovni primjer kada automatska konfiguracija ne zadovoljava potrebe je primjena sigurnosti. Sigurnost nije jednoznačna, postoje odluke koje se tiču sigurnosti koje Spring Boot ne može donijeti. Iako Spring Boot pruža osnovnu auto-konfiguraciju za sigurnost, postoji način da ju se nadjača.

Nadjačavanje automatske konfiguracije je jednostavno pisanje konfiguracije kao da automatska konfiguracija ne postoji. Konfiguracija može biti napisana u bilo kojoj formi koju Spring podržava: XML konfiguracija, Java konfiguracija itd.

U nekim slučajevima potrebno je promijeniti samo nekoliko detalja konfiguracije. Bilo bi poželjno da se tada ne mora nadjačavati cijela konfiguracija, već da se promijene samo potrebni detalji, kao što je broj porta poslužitelja ili veza do baze podataka.

Spring Boot omogućava podešavanje određenih svojstva pomoću okolinskih varijabli (eng. *environment variables*), svojstvenih datoteka (eng, *property files*), YAML datoteka i na nekoliko drugih načina.

Na primjer, za promjenu porta poslužitelja na 8081 potrebno je dodati sljedeću liniju u *application.properties* datoteci:

```
server.port = 8081
```

Isto se može promijeniti i u *application.yml* datoteci:

```
server:
  port: 8081
```

Za promjenu url-a baze podataka potrebno dodati sljedeće u *application.properties*:

```
spring.datasource.url = jdbc:mysql://localhost:3306/test
```

Ekvivalentna stvar u *application.yml*:

```
spring:
    datasource:
        url: jdbc:mysql://localhost:3306/test
```

3.2 Starter ovisnosti

Upravljanje ovisnostima na velikim projektima težak je zadatak, posebice ako se radi ručno. Treba izabrati odgovarajuće biblioteke odgovarajućih verzija koje će dobro raditi s ostalim ovisnostima. Spring Boot rješava ovaj problem pružanjem skupa ovisnosti koji se nazivaju *strater* ovisnosti. Pomoću njih aplikacija dobiva sve potrebne biblioteke za određenu funkcionalnost.

Sve Spring Boot *starter* ovisnosti su oblika *spring-boot-starter-**, gdje * označava vrstu aplikacije. Spring Boot nudi više od 30 *starter* ovisnosti.

Web starter

Za izgradnju REST aplikacijskog programskog sučelja s Spring MVC-om koji za reprezentaciju resursa koristi JSON i ima ugrađeni Tomcat poslužitelj, potrebno je uključiti barem sljedeće ovisnosti:

- *org.springframework:spring-core*
- *org.springframework:spring-web*
- *org.springframework:spring-webmvc*
- *com.fasterxml.jackson.core:jackson-databind*
- *org.apache.tomcat.embed:tomcat-embed-core*
- *org.apache.tomcat.embed:tomcat-embed-el*
- *org.apache.tomcat.embed:tomcat-embed-logging-juli*

Spring Boot *starter* ovisnosti mogu znatno smanjiti broj ručno dodanih ovisnosti. Umjesto ručnog pisanja svih potrebnih ovisnosti dovoljno je dodati web *starter* ovisnost u Maven POM datoteci:

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

Test starter

Za testiranje Spring aplikacija obično se uključuju sljedeće biblioteke: Spring Test, JUnit, Hamcrest i Mockito. Spring Boot test *starter* se može iskoristiti za automatsko uključivanje tih biblioteka na sljedeći način:

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

Starter za sigurnost

Za dodavanje sigurnost unutar Spring Boot aplikacije, potrebno je dodati *starter* za sigurnost:

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

Dodavanjem navedene ovisnosti automatski se uključuje klasa *SecurityAutoConfiguration* koja sadrži osnovnu konfiguraciju sigurnosti.

3.3 Aktuator

Spring Boot aktuator sadrži niz dodatnih značajki koje služe za nadzor i upravljanje Spring Boot aplikacijom. Putem aktuatora moguće je saznati kako su *bean*-ovi povezani unutar Spring aplikacijskog konteksta, vidjeti koje vrijednosti imaju okolinske varijable, dobiti dijagnostiku rada aplikacije i još mnogo toga. Ove funkcionalnosti pružaju se preko odgovarajućih HTTP krajnjih točki.

HTTP metoda	**Krajnja točka**	**Opis**
GET	/autoconfig	Pruža izvješće o automatskoj konfiguraciji.
GET	/configprops	Prikazuje popis svih konfiguracijskih svojstava.
GET	/beans	Prikazuje sve *bean*-ove u aplikaciji.
GET	/env	Vraća sve okolinske varijable.
GET	/env/name	Vraća vrijednost navedene okolinske varijable.
GET	/health	Prikazuje stanje aplikacije.
GET	/info	Prikazuje proizvoljne podatke o aplikaciji.
GET	/mappings	Opisuje povezivanje pojedinog URI-a i upravljača.
GET	/metrics	Izvještaj o raznim mjernim podacima aplikacije, poput upotrebe memorije i brojanja HTTP zahtjeva.
GET	/metrics/name	Izvještaj o navedenom mjernom podatku aplikacije.
POST	/shutdown	Isključuje aplikaciju.
GET	/trace	Pruža osnovne informacije o HTTP zahtjevima.

Tablica 3.2: Krajnje točke aktuatora

Za uključivanje aktuatora u Spring Boot aplikaciju potrebno je dodati aktuator *starter* ovisnost. U slučaju Maven projekta potrebno je sljedeće:

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

Krajnje točke aktuatora omogućuju nadzor i interakciju s aplikacijom. Spring Boot nudi brojne ugrađene krajnje točke, ali i mogućnost stvaranja vlastitih. Do pojedine HTTP točke dolazi se tako da se na URL aplikacije doda put do pojedine krajnje točke s prefiksom */actuator*. Na primjer, krajnja točka */bean* daje uvid u Spring-ov aplikacijski kontekst. Za pristup toj točki potrebno je na URL aplikacije dodati */actuator/bean*.

U tablici 3.2 opisane su najčešće korištene HTTP krajnje točke.

3.4 Priprema okruženja i osnovna aplikacija

Spring Boot projekt je zapravo Spring projekt koji koristi Spring Boot *starters* i auto-konfiguraciju. Stoga, bilo koja tehnika ili alat koji se koriste za stvaranje novog Spring projekta može se koristiti i u slučaju Spring Boot-a.

Spring Initializr

Spring Initializr je web aplikacija koja generira Spring Boot projektnu strukturu. Stvara osnovnu strukturu projekta i Maven ili Gradle *build* specifikaciju, sve što preostaje je pisanje koda aplikacije.

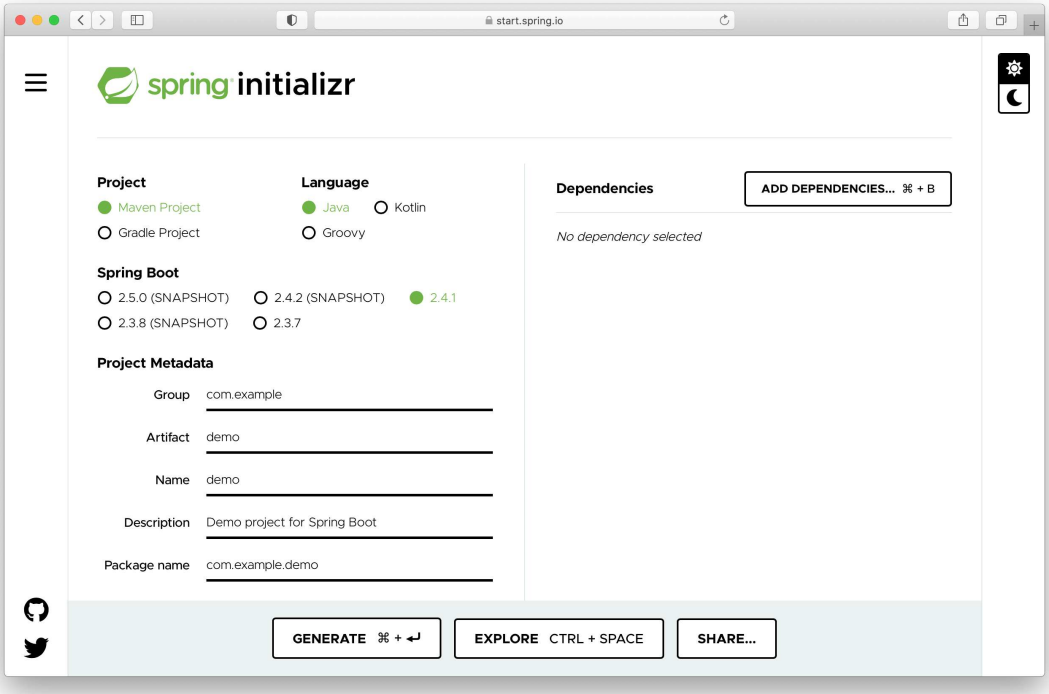
Spring Initializr može se koristiti na nekoliko načina:

- kroz web sučelje
- putem Spring Tool Suite
- putem IntelliJ IDEA
- korištenjem CLI Spring Boot

Korištenje Spring Initializr web sučelja

Spring Initializr web sučelje dostupno je na `http://start.spring.io`. Izgled web sučelja prikazan je na slici 3.1.

Prve tri stvari koje je potrebno ispuniti u obrascu su odabir Maven ili Gradle *build* alata, programskog jezika i verzije Spring Boot-a. Zatim je potrebno navesti metapodatke



Slika 3.1: Spring Initializr web sučelje

projekta: grupu, artefakt projekta, naziv, opis, naziv paketa, vrsta pakiranja (JAR ili WAR) i verziju Jave. Ovi se metapodaci koriste za popunjavanje Maven datoteke *pom.xml* ili Gradle datoteke *build.gradle*.

Klikom na ”Add Dependencies” pojavljuje se lista dostupnih ovisnosti koju je moguće pretraživati. Odabirom potrebnih ovisnosti u projekt se dodaju odgovarajuće *starter* ovisnosti.

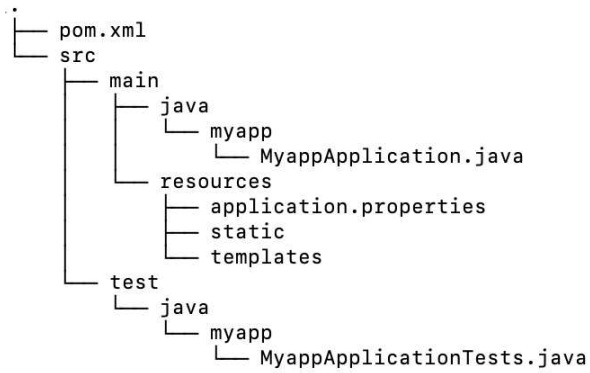
Klikom na ”Generate” Spring Initializr generira projekt. Projekt postaje dostupan kao *zip* datoteka s nazivom koji odgovara postavljenom artefaktu. Sadržaj *zip* datoteke se razlikuje ovisno o odabranim vrijednostima, u svakom slučaju datoteka će sadržavati kostur projekta koji je spreman za razvijanje aplikacije u Spring Boot-u. Dobivenu datoteku moguće je otvoriti u nekom od integriranih razvojnih okruženja npr. IntelliJ IDEA ili Spring Tool Suite.

Jednostavna web aplikacija

Za razvoj Spring Boot aplikacije potrebno je [2]:

- Java SE Development Kit (JDK) 8 ili noviji
- Gradle 4+ ili Maven 3.2+
- uređivač teksta ili integrirano razvojno okruženje (IDE)

Pomoću Spring Initializr kreiran je Maven projekt s artefaktom projekta "myapp", nazivom paketa "myapp" i ovisnostima Spring Web i Thymeleaf. Generirana *zip* datoteka "myapp.zip" sadrži strukturu projekta koja je prikazana na slici 3.2.



Slika 3.2: Struktura projekta generiranog pomoću Spring Initializr web sučelja

Struktura projekta slijedi tipičnu Maven ili Gradle strukturu. Glavni kod aplikacije smješten je u *src/main/java*, resursi su smješteni u *src/main/resources*, a testovi se nalaze u *src/test/java*. Generirani projekt sadrži nekoliko direktorija i datoteke:

- *pom.xml* - Maven *build* specifikacija, u slučaju Gradle projekta *build.gradle*.
- *MyappApplication.java* - Klasa s *main()* metodom za pokretanje aplikacije.
- *MyappApplicationTests.java* - Prazna JUnit test klasa koja učitava Spring aplikacijski kontekst pomoću automatske konfiguracije.
- *application.properties* - Prazna svojstvena datoteka.
- *static* - Prazan direktorij u kojem su obično smještene JavaScript datoteke, CSS datoteke, slike itd.

- *templates* - Prazan direktorij za spremanje pogleda.

Generirana *pom.xml* datoteka sadrži ovisnosti Thymeleaf *starter*, web *starter* i test *starter*:

```
...
  <dependencies>
    <dependency>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-starter-thymeleaf</artifactId>
    </dependency>
    <dependency>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-starter-web</artifactId>
    </dependency>

    <dependency>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-starter-test</artifactId>
      <scope>test</scope>
    </dependency>
  </dependencies>

...
```

Pokretanje aplikacije

Klasa *MyappApplication* u svojoj *main* metodi poziva *SpringApplication.run()* za pokretanje aplikacije (Kod 3.3).

```
package myapp;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication
public class MyappApplication {

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

}
```

Kod 3.3: Klasa MyappApplication

Klasa *MyappApplication* predstavlja i središnju konfiguracijsku klasu. Iako je automatska konfiguracija smanjila količinu konfiguracije, ipak je potrebna minimalna konfiguracija koja omogućuje korištenje automatske konfiguracije. *@SpringBootApplication* kombinira tri Spring-ove anotacije:

- *@Configuration* - Označava klasu kao konfiguracijsku klasu.
- *@ComponentScan* - Omogućuje skeniranje komponenti koje se registriraju kao *bean*-ovi u aplikacijskom kontekstu.
- *@EnableAutoConfiguration* - Omogućuje Spring Boot automatsku konfiguraciju.

Iako nije napisan dodatni kod unutar aplikacije, moguće je napravit *build* aplikacije i pokrenuti je u komandnoj liniji:

```
$ mvn spring-boot:run
```

Alternativno, moguće je napraviti *build* projekta s Maven-om i pokrenuti ga s *java* naredbom:

```
$ mvn clean package
...
$ java -jar build/libs/myapp-0.0.1-SNAPSHOT.jar
```

Aplikacija bi se trebala pokrenuti na Tomcat poslužitelju koji sluša na portu 8080.

Stvaranje upravljača

```
package myapp;

import org.springframework.stereotype.Controller;
import org.springframework.ui.Model;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.PostMapping;
import org.springframework.web.bind.annotation.RequestParam;

@Controller
public class HelloController {

    @GetMapping("/")
    public String home() {
        return "home";
    }
```

```
    @PostMapping("/")
    public String hello(@RequestParam(name = "name", defaultValue =
                        "World") String name, Model model) {
        model.addAttribute("name", name);
        return "hello";
    }
}
```

Kod 3.4: Upravljač *HelloController*

Klasa *HelloController* sadrži dvije metode, *home* i *hello*. Metodi *home* se pristupa putem GET zahtjeva koji je povezan s putanjom "/". Metoda je zadužena za generiranje početne stranice aplikacije i vraća logičko ime pogleda "home". Drugoj metodi *hello* pristupa se putem POST zahtjeva koji je povezan s putanjom "/" koja može sadržavati i parametar upita "name". U slučaju da ga ne sadrži tada se za vrijednost varijable *name* uzima vrijednost "World". Metoda postavlja vrijednost "name" atributa modela i vraća logičko ime pogleda "hello".

Stvaranje pogleda

Potrebno je napraviti Thymeleaf predloške pogleda koje će odgovarati pojedinim logičkim imenima. Pošto *home* metoda vraća *String* "home" potrebno je implementirati pogled unutar datoteke *home.html* koja mora biti smještena u direktoriju *src/main/resources/templates*.

Kod 3.5 prikazuje sadržaj datoteke *home.html*. Web stranica sadrži jednostavnu formu u kojoj je potrebno unijeti ime i gumb "Submit". Klikom na gumb "Submit" sadržaj forme se šalje POST metodom na "/".

```
<!DOCTYPE html>
<html xmlns:th="http://www.thymeleaf.org">
<head>
    <meta http-equiv="Content-Type" content="text/html";
          charset="UTF-8">
    <title>Home</title>
</head>
<body>
    <form th:action="@{/}" method="post">
        <h1>Dobro dosli!</h1>
        <label>Ime:</label>
        <input type="text" th:value="${name}" name="name">
        <input type="submit" />
    </form>
</body>
</html>
```

Kod 3.5: Pogled *home.html*

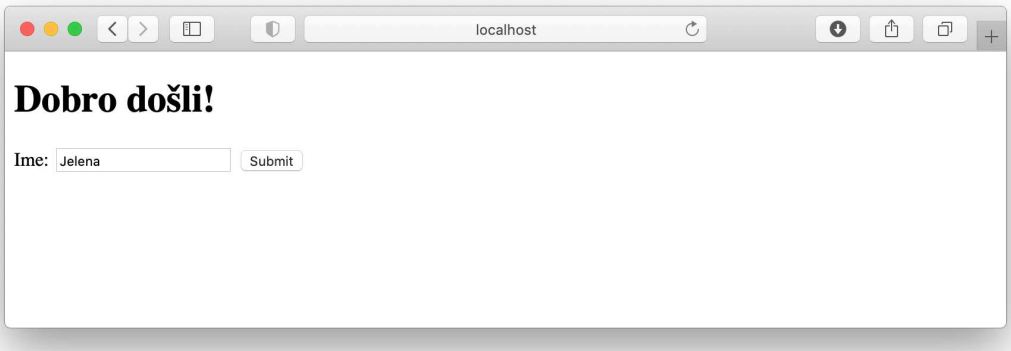
Potrebna je još implementacija pogleda koji odgovara logičkom imenu ”hello”. U ovom slučaju potrebno je dohvatiti vrijednost atributa ”name” i prikazati ga (Kod 3.6).

```
<!DOCTYPE html>
<html xmlns:th="http://www.thymeleaf.org">
<head>
    <meta http-equiv="Content-Type" content="text/html";
        charset="UTF-8">
    <title>Hello</title>
</head>
<body>
    <h3 th:text="'Bok, ' + ${name} + '!'"/>
</body>
</html>
```

Kod 3.6: Pogled *hello.html*

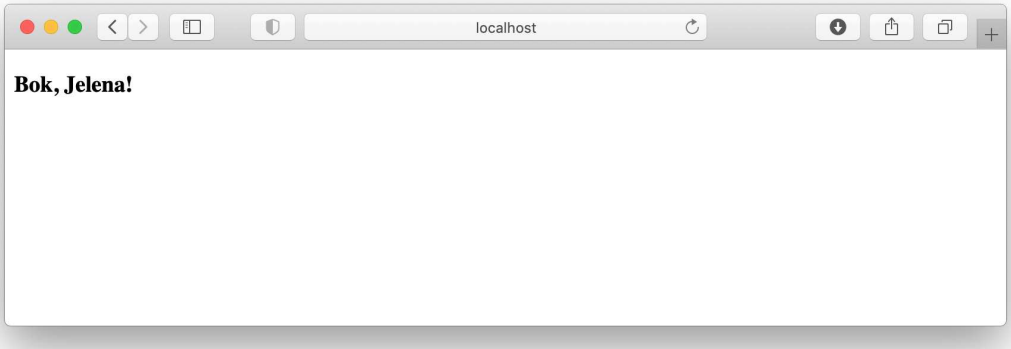
Aplikacija

Kada se aplikacija pokrene, na http://localhost:8080/ dostupna je početna stranica koja prikazuje formu za unos imena (Slika 3.3).



Slika 3.3: Početna stranica aplikacije

Unosom imena i klikom na gumb ”Submit” prikazuje se stranica s pozdravnom porukom (Slika 3.4).



Slika 3.4: Pozdravna stranica aplikacije

3.5 Objava (eng. *deploy*) Spring Boot aplikacije

Pokretanje aplikacije pomoću IDE-a ili Maven-a omogućuju lokalni razvoj i pokretanje koje nije spremno za produkciju. Kreiranjem izvršnih JAR datoteka ili tradicionalnih Java WAR datoteka omogućuje se postavljanje aplikacije u produkcijsko okruženje. Izbor WAR ili JAR datoteke ovisi o tome planira li se aplikacija objaviti na tradicionalnom Java poslužitelju ili na platformi u oblaku [11]:

- Objava na Java poslužitelju - ukoliko se aplikacija objavljuje na Tomcat-u, WebSphere-u, WebLogic-u ili nekom drugom tradicionalnom Java poslužitelju, potrebno je kreirati WAR datoteku.
- Objava na platformi u oblaku - ukoliko se aplikacija objavljuje na Amazon Web Services-u, Google Cloud-u, Heroku-u ili nekoj drugoj platformi u oblaku, JAR datoteka je najbolji izbor iako su podržane i WAR datoteke (JAR format datoteke je jednostavniji od WAR formata koji je napravljen za objavu na poslužiteljima).

Objava na Java poslužitelju

Za objavu aplikacije na Java poslužitelju potrebno je kreirati WAR datoteku. Pokretanje Spring Boot aplikacije iz WAR datoteke omogućuje *SpringBootServletInitializr*. *SpringBootServletInitializr* je posebna implementacija Spring-ovog *WebApplicationInitializer* koja koristi Spring Boot.

Potrebno je unutar aplikacije stvoriti klasu koja proširuje *SpringBootServletInitializer* i nadjačava njenu metodu *configure*. Unutar *configure* poziva se metoda *sources* koja registrira Spring-ove konfiguracijske klase.

```
import org.springframework.boot.builder.SpringApplicationBuilder;
import org.springframework.boot.web.servlet.support.
   SpringBootServletInitializer;

public class ServletInitializer extends SpringBootServletInitializer {

  @Override
  protected SpringApplicationBuilder configure(SpringApplicationBuilder
                                application) {
    return application.sources(MyappApplication.class);
  }

}
```

Još preostaje reći alatu za izgradnju aplikacije da koristi WAR pakiranje. U slučaju Maven projekta dovoljno je promijeniti *packaging* u *pom.xml* datoteci:

```
<packaging>war</packaging>
```

Sada je moguće stvoriti WAR datoteku pomoću naredbe:

```
mvn package
```

Ukoliko je izgradnja bila uspješna, WAR datoteka bit će dostupna u direktoriju *target*.

U slučaju da je odabrana WAR opcija pakiranja prilikom stvaranja aplikacija pomoću Spring Initializr-a, ništa od navedenoga nije potrebno. Initializr osigurava da generirani projekt sadržava klasu koja proširuje *SpringBootServletInitializer*, a *pom.xml* će biti prilagođena za stvaranje WAR datoteke [11].

Objava na Heroku platformi

Heroku je platforma kao usluga (eng. *Platform as a Service* - PaaS). Razvojni programeri koriste Heroku za objavu, upravljanje i skaliranje modernih aplikacije. Platforma nudi fleksibilnu upotrebu i jednostavan put izlaska aplikacije na tržište.

Jedna je od prvih platformi u oblaku osnovana je 2007. godine. Tada je podržavala samo programski jezik Ruby, danas podržava još programske jezike Java, Node.js, Scala, Clojure, Python, PHP i Go.

Jedan od načina objave koju nudi Heroku je pomoću integracije s alatom za upravljanje verzijama Git-om. Heroku nudi mogućnost povezivanja s hosting platformom za

upravljanje verzijama kao što je GitHub i pruža jednostavnu objava koda koji se nalazi na GitHub-u. Kada je GitHub integracija konfigurirana na Heroku platformi, Heroku može automatski graditi i objavljivati promjene koje se događaju u GitHub repozitoriju.

Priprema aplikacije za objavu

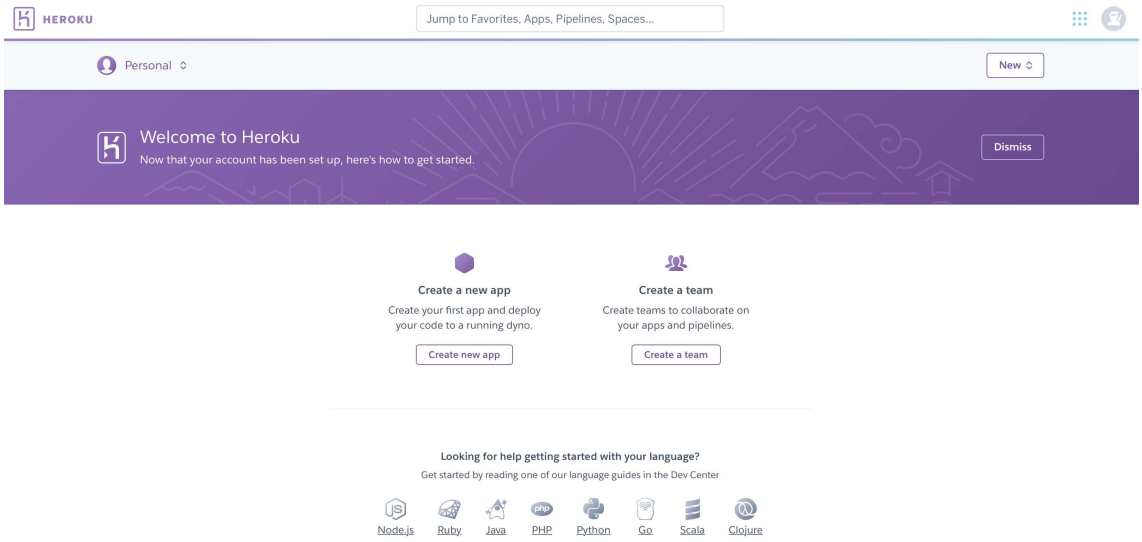
Jedina modifikacija unutar Spring Boot projekta prije objave na Heroku je kreiranje datoteke *system.properties* u korijenu projekta. Ona mora sadržavati informaciju o verziji Jave koja se koristi u aplikaciji:

```
java.runtime.version=11
```

Prije same objave potrebno je konačnu verziju aplikacije staviti na GitHub repozitorij.

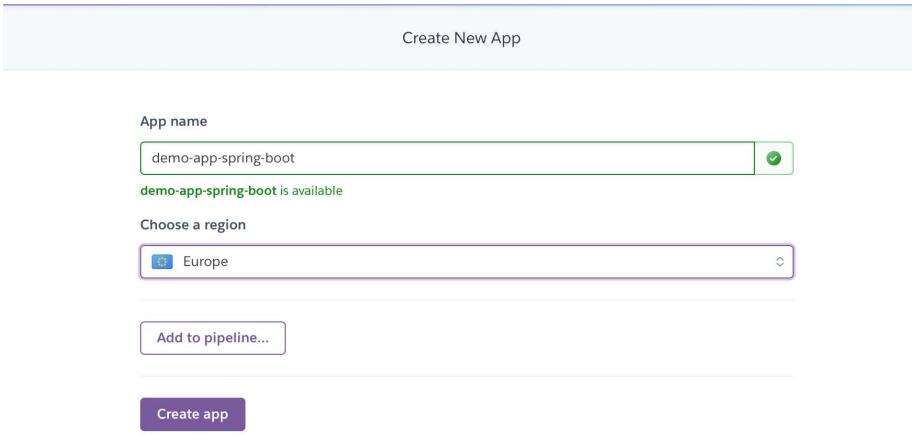
Objava

Prvo je potrebno registrirati se na Heroku web stranici `https://signup.heroku.com/`. Nakon uspješne registracije i prijave prikazuje se web sučelje slično onome na slici 3.5.

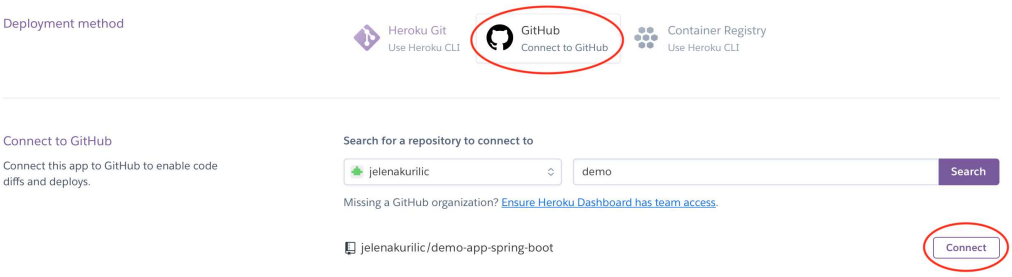


Slika 3.5: Heroku web sučelje

Potrebno je kreirati novu aplikaciju unutar sučelja klikom na ”New” pa ”Create new app”. Nakon unosa jedinstvenog imena aplikacije klikom na gumb ”Create app” kreira se nova aplikacija (Slika 3.6). Sljedeći korak je odabrati način objave npr. GitHub integracija. Potrebno je povezati aplikaciju s GitHub računom i pronaći repozitorij koji se želi objaviti.



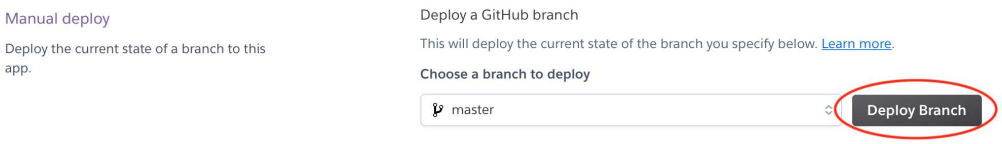
Slika 3.6: Heroku stvaranje nove aplikacije



Slika 3.7: Heroku GitHub integracija

Klikom na ”Connect” pored pronađenog repozitorija ostvaruje se povezivanje s odabranim repozitorijem (Slika 3.7) [3].

Još preostaje odabrati odgovarajuću GitHub granu repozitorija koju se želi objaviti i kliknuti na gumb ”Deploy Branch” (Slika 3.8). Nakon uspješnog *build*-a aplikacija je dostupna na odabranoj domeni, u ovom slučaju na https://demo-app-spring-boot.herokuapp.com.



Slika 3.8: Heroku objava aplikacije

Heroku također nudi i integraciju s raznim bazama podataka. Ukoliko aplikacija koristi PostgreSQL bazu podataka potrebno je unutar kreirane Heroku aplikacije instalirati dodatak Heroku Postgres.

Poglavlje 4

Aplikacija

Web aplikacija za prijavu na različite edukativne događaje poput radionica, ljetnih škola, tečajeva razvijena je pomoću Spring Boot-a okvira, koristeći IntelliJ IDEA razvojno okruženje. Aplikacija prati obrazac Model-View-Controller, a za razvijanje pogleda korišten je Thymeleaf. Svi podaci se spremaju u PostgreSQL bazu podataka. *Build* alat Maven koristi se za automatiziranu izgradnju softvera. Konačna verzija aplikacije objavljena je na Heroku platformi i dostupna je na `https://apply-app-spring-boot.herokuapp.com/`.

4.1 Opis aplikacije

Web aplikacija za prijavu na različite događaje namijenjena je svima koji se žele prijaviti i sudjelovati na edukativnim događajima te onima koji žele organizirati događaje, pratiti i pregledavati pristigle prijave.

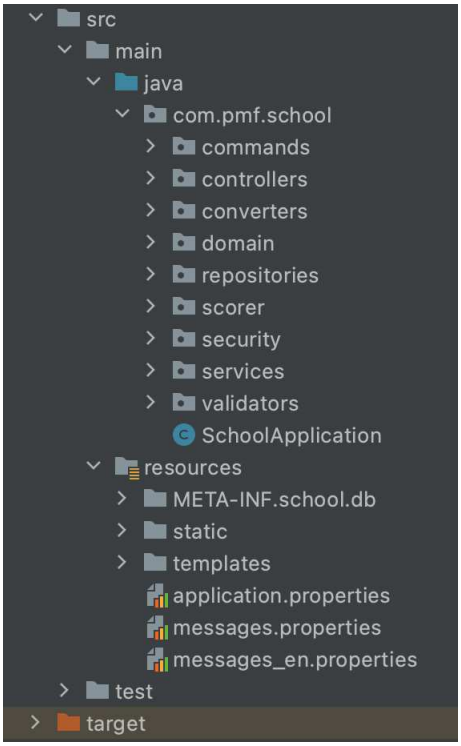
Organizatori mogu kreirati događaj koji žele organizirati ispunjavanjem forme u kojoj navode naziv, opis, početak događaja, kraj događaja, maksimalni broj sudionika, početak prijava i kraj prijava. Nakon stvaranja događaja organizator može dodati i poželjne vještine za sudjelovanje. Svakoj navedenoj vještini organizator pridružuje bodove koji će se uzimati u obzir prilikom bodovanja prijava. Organizator unutar aplikacije može vidjeti sve kreirane događaje, svaki od događaja može uređivati i pregledavati pristigle prijave. Dodatno, za svaku prijavu može ostaviti svoj komentar i dati određeni broj bodova. Nakon završetka prijava aplikacija omogućuje organizatoru automatski odabir sudionika na temelju ostvarenih bodova. Ukupan broj bodova se računa na temelju vještina i razine iskustva koje je naveo korisnik u prijavi te bodova koje je dao organizator.

Svaki korisnik koji se želi prijaviti na edukativne događaje mora biti registriran. Prili-

kom registracije korisnik unosi adresu e-pošte, ime, prezime, broj telefona, kratki životopis, lozinku i ponovljenu lozinku. Nakon uspješne registracije korisnik se može prijaviti u aplikaciju. Unutar aplikacije može vidjeti sve događaje čije su prijave u tijeku i na svaki od njih se može prijaviti. Također, korisnik može vidjeti sve svoje prijave i njihov status (prijava u tijeku, prihvaćena ili odbijena). Korisniku se nudi i mogućnost pregleda i uređivanja profila.

4.2 Struktura aplikacije

Aplikacija prati obrazac Model-View-Controller, dok se poslovna logika aplikacije obrađuje u odgovarajućim servisima. Struktura direktorija aplikacije prikazana je na slici 4.1.



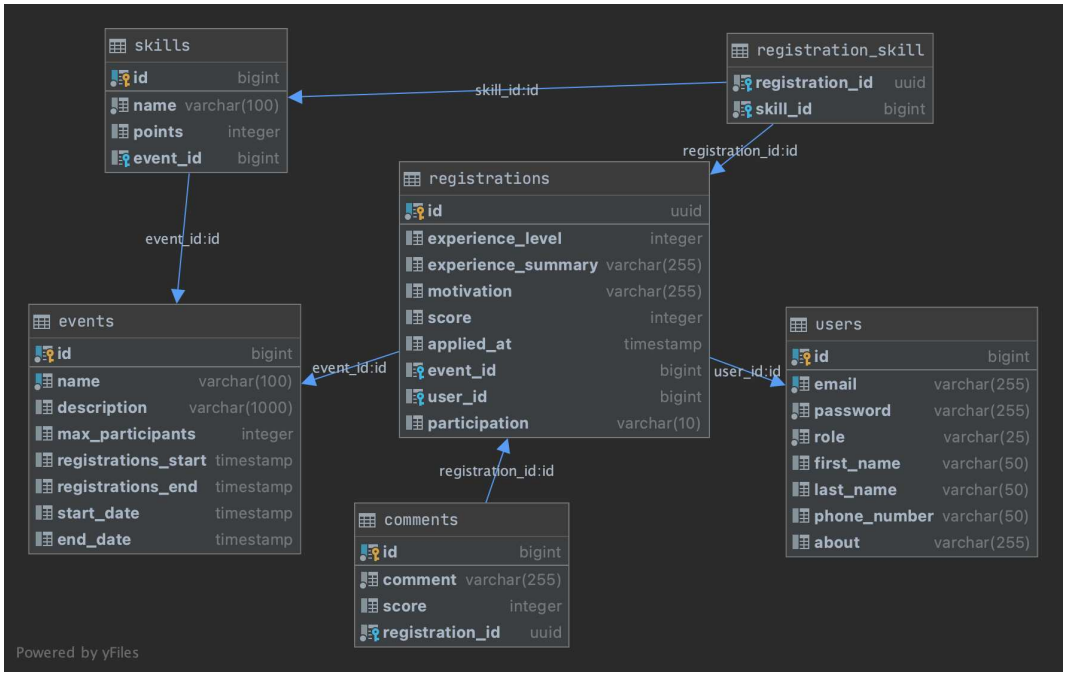
Slika 4.1: Struktura direktorija aplikacije

Modeli aplikacije nalaze se u direktoriju *domain*. Unutar direktorija *commands* nalaze se objekti koji se šalju pogledu i forme koje pogled šalje. *converters* sadrži klase za mapiranje modela u *command* objekte i mapiranje formi u modele. Unutar *repositories* nalaze se repozitoriji koji proširuju *JpaRepository*. Unutar *service* nalaze se servisi u kojima je

implementirana poslovna logika aplikacije. Upravljači aplikacije smješteni su u *controllers*. Svaka forma koja dolazi od strane pogleda provjerava se, klase zadužene za provjeru pojedine forme smještene su unutar direktorija *validators*. Unutar aplikacije implementirana je i sigurnost, sve postavke sigurnosti nalaze se u direktoriju *security*. Pogledi su smješteni u direktoriju */resources/templates*.

Modeli

Modeli aplikaciji nalaze se u paketu *com.pmf.school.domain*. Modeli su: *User*, *Registration*, *Comment*, *Event* i *Skill*. Na slici 4.2 prikazana je struktura baze podataka i odnosi između objekata baze podataka dobivenih na temelju definiranih modela.



Slika 4.2: Struktura baze podataka i odnosi između objekata baze podataka

Model *User* predstavlja korisnika aplikacije. Svaki od korisnika može imati ulogu ROLE_USER ili ROLE_ADMIN. Ovisno o ulozi korisnik ima različita prava unutra aplikacije. *User* još sadrži atribute adresa e-pošte, lozinka, ime, prezime, broj telefona i kratak životopis.

Model *Registration* predstavlja pojedinu prijavu koju je napravio korisnik. Prijava sadrži razinu iskustva čija vrijednost može biti između jedan (bez iskustva) i pet (vrlo is-

kusan), kratak opis iskustva i motivaciju korisnika. Također *Registration* sadrži i ostvarene bodove te datum kada je prijava kreirana.

Model *Comment* predstavlja komentar koji organizator, odnosno korisnik s ulogom administratora, može ostaviti na pojedinoj prijavi. Model sadrži tekst koji je ostavljen u komentaru i dodijeljene bodove.

Model *Skill* predstavlja vještinu koju je moguće dodati na pojedini događaj. Model sadrži atribute ime i bodove.

Model *Event* predstavlja pojedini događaj. Svaki od događaja sadrži naziv, opis, maksimalni broj sudionika, vrijeme početka i kraja te vrijeme početka i kraja prijava. Model *Event* implementiran je na sljedeći način:

```
@Data
@Entity(name = "EVENTS")
public class Event {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    @Column(name = "ID")
    private Long id;

    @Column(name = "NAME", unique = true)
    private String name;

    @Column(name = "DESCRIPTION")
    private String description;

    @Column(name = "MAX_PARTICIPANTS")
    private Integer maxParticipants;

    @Column(name = "REGISTRATIONS_START")
    private LocalDateTime registrationsStart;

    @Column(name = "REGISTRATIONS_END")
    private LocalDateTime registrationsEnd;

    @Column(name = "START_DATE")
    private LocalDateTime startDate;

    @Column(name = "END_DATE")
    private LocalDateTime endDate;

    @OneToMany(mappedBy = "event")
    private List<Registration> registrations;
```

```
    @OneToMany(mappedBy = "event")
    private List<Skill> skills;
}
```

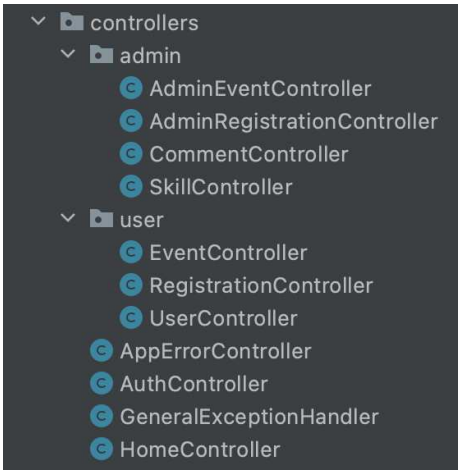
Anotacija *@Data* generira ponavljajući kod karakterističan za sve Java objekte: *getters*, *setters*, konstruktor i odgovarajuće *toString*, *equals* te *hashCode* implementacije. Ova anotacija dolazi iz biblioteke Lombok čiju ovisnost je potrebno uključiti u *pom.xml* datoteci:

```
<dependency>
  <groupId>org.projectlombok</groupId>
  <artifactId>lombok</artifactId>
  <optional>true</optional>
</dependency>
```

@Entity označava da se radi od JPA enititetu te da je on mapiran s relacijskom tablicom EVENTS. Unutar anotacije *@Column* naveden je naziv atributa entiteta. *@OneToMany* označava da je prisutna *one-to-many* relacijska veza između *Event*-a i *Registration*-a te *Event*-a i *Skill*-a.

Upravljači

Upravljači unutar aplikacije smješteni su u paketu *com.pmf.school.controllers*. Struktura direktorija *controllers* prikazana je na slici 4.3. Unutar paketa nalaze se poddirektoriji *admin* i *user*.



Slika 4.3: Struktura direktorija *controllers*

AuthController koristi se za prijavu i registraciju korisnika. *AppErrorController* brine o prikazivanju odgovarajućeg pogleda u slučaju pojave greške. *HomeController* koristi se za

vraćanje početne stranice uspješno prijavljenog korisnika. *HomeController* implementiran je na sljedeći način:

```
@Controller
public class HomeController {


    @GetMapping( value = {"/home", ""})
    public String defaultAfterLogin(HttpServletRequest request) {
        return request.isUserInRole("ROLE_ADMIN") ?
                "redirect:/event/all" : "home";
    }
}
```

Ukoliko se radi o uspješno prijavljenom korisniku s ulogom ROLE_ADMIN tada se zahtjev preusmjerava na krajnju točku koja vraća sve događaje, inače se vraća pogled "home".

Unutar drektorija *admin* nalaze se upravljači kojima pravo pristupa ima administrator, a to su *AdminEventController*, *AdminRegistrationController*, *CommentController* i *SkillController*.

AdminEventController koristi se za dohvaćanje svih događaja i pojedinog događaja na temelju *id*-a te sadrži krajnje točke za kreiranje novog događaja, uređivanje postojećeg i prihvaćanje najbolje rangiranih prijava na pojedinom događaju.

AdminRegistrationController koristi se za dohvaćanje svih prijava određenog događaja i pojedine prijave na temelju *id*-a. Implementacija *AdminRegistrationController*-a izgleda ovako:

```
@Controller
@PreAuthorize("hasRole('ROLE_ADMIN')")
@RequestMapping("/registration")
public class AdminRegistrationController {

    private RegistrationService registrationService;

    @Autowired
    public AdminRegistrationController(RegistrationService rs) {
        this.registrationService = rs;
    }

    @GetMapping("/all/{eventId}")
    public String getAllRegistrations(@PathVariable Long eventId,
                                      Model model) {

        List<RegistrationCommand> registrations = registrationService
```

```
                        .getAllRegistrationsByEventIdSorted(eventId);

        model.addAttribute("registrations", registrations);
        return "registration/admin-registrations";
    }

    @GetMapping("/one/{registrationId}")
    public String getRegistration(@PathVariable UUID registrationId,
                                  Model model) {

        RegistrationCommand registration = registrationService
                            .getRegistrationById(registrationId);

        model.addAttribute("registration", registration);
        model.addAttribute("commentForm", new CommentForm());
        return "registration/registration";
    }

}
```

Prijave koje se vraćaju u metodi *getAllRegistrations* sortirane su na temelju bodova. Metoda *getRegistration* pogledu vraća prijavu i praznu formu za stvaranje komentara.

SkillController koristi se za spremanje vještina za pojedini događaj.

Unutar direktorija *user* nalaze se upravljači kojima pravo pristupa ima korisnik, a to su *EventController*, *RegistrationController* i *UserController*.

EventController koristi se za dohvaćanje svih događaja čije su prijave trenutno u tijeku. *EventController* implementiran je na sljedeći način:

```
@Controller
@PreAuthorize("hasRole('ROLE_USER')")
@RequestMapping("/event")
public class EventController {
    private EventService eventService;

    @Autowired
    public EventController(EventService eventService) {
        this.eventService = eventService;
    }

    @GetMapping("/current/all")
    public String getCurrentAllEvents(Model model) {
        List<String> myRegistrations = registrationService
                                .getMyRegistrations(principal.getName())
        List<EventCommand> events = eventService.getAllCurrentEvents();
```

```
        model.addAttribute("events", events);
        model.addAttribute("myRegistrations", myRegistrations);
        return "event/current-events";
    }

}
```

RegistrationController se brine o dohvaćanju svih prijava prijavljenoga korisnika. Također sadrži i krajnju točku za slanje prijave na događaj. *UserController* sadrži krajnje točke za dohvaćanje i uređivanje profila.

Sigurnost

Sigurnost unutar aplikacije implementirana je koristeći Spring Security. Za prijavljenog korisnika potrebno je spremati njegove informacije, SpringSecurity mora autentificirati korisnika iz baze. U tu svrhu napravljena je implementacija sučelja *UserDetailsService*:

```
@Service
public class UserRepositoryUserDetailsService implements
   UserDetailsService {
    private UserRepository userRepository;

    @Autowired
    public UserRepositoryUserDetailsService(UserRepository ur) {
        this.userRepository = ur;
    }

    @Override
    public UserDetails loadUserByUsername(String s) throw
                                UsernameNotFoundException {

        User user = userRepository.getByEmail(s);
        if (user != null) {
            return user;
        }
        throw new UsernameNotFoundException("User not found.");
    }
}
```

U ovoj klasi se nadjačava metoda *loadUserByUsername* koja nalazi odgovarajućeg korisnika u bazi koristeći *UserRepository*. Prilikom autentifikacije korisnika umjesto *username*-a koristi se *email*.

Također je prilagođena i konfiguracija za presretanje zahtjeva na sljedeći način:

```
protected void configure(HttpSecurity http) throws Exception {
```

```
    http
        .csrf()
        .disable()
        .authorizeRequests()
        .antMatchers("/login/**", "/register","/**/*.js", "/**/*.css").
   permitAll()
        .anyRequest().authenticated()
        .and()
        .formLogin()
        .loginPage("/login")
        .usernameParameter("email").passwordParameter("password")
        .permitAll()
        .defaultSuccessUrl("/home")
        .failureUrl("/login-error")
        .and()
        .logout()
        .logoutSuccessUrl("/login");
}
```

Svaki od zahtjeva koji dolazi prema aplikaciji, osim onih koji su povezani s */login* i */register*, moraju biti autentificirani. Mogućnost prijave u aplikaciju povezana je s krajnjom točkom */login*, u slučaju uspješne prijave korisnik će biti preusmjeren na */home*, a u slučaju neuspjele prijave bit će preusmjeren na */login-error*. Krajnja točka za odjavu iz aplikacije je prema zadanim postavkama */logout*, nakon uspješne odjave korisnik je preusmjeren na */login*.

Internacionalizacija

Aplikacija je dvojezična, odnosno nudi korisničko sučelje na hrvatskom i engleskom jeziku. Moguće je promijeniti jezik u bilo kojem dijelu aplikacije. Zadani jezik je hrvatski, ukoliko korisnik odluči promijeniti jezik informacija o novo odabranom jeziku se sprema u kolačiću (eng. *cookie*).

Da bi aplikacija znala koji jezik se trenutno koristi potrebno je dodati *bean LocaleResolver*:

```
@Bean
public LocaleResolver localeResolver() {
  CookieLocaleResolver localeResolver = new CookieLocaleResolver();
  localeResolver.setDefaultLocale(new Locale("hr", "HR"));
  return localeResolver;
}
```

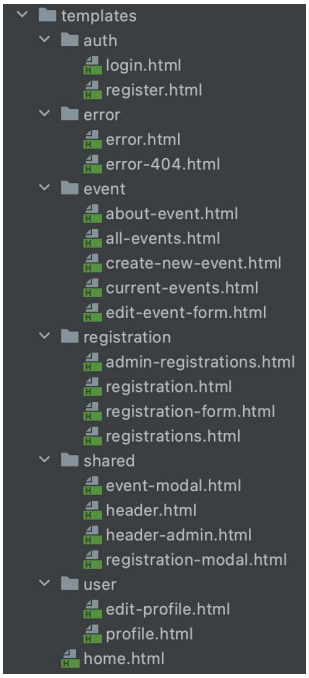
LocaleResolver određuje koji se jezik koristi na temelju kolačića. Nadalje, potrebno je dodati presretač zahtjeva koji će promijeniti jezik na temelju vrijednosti parametra *lang*:

```
@Bean
public LocaleChangeInterceptor localeChangeInterceptor() {
  LocaleChangeInterceptor lci = new LocaleChangeInterceptor();
  lci.setParamName("lang");
  return lci;
}
...
public void addInterceptors(InterceptorRegistry registry) {
  registry.addInterceptor(localeChangeInterceptor());
}
```

Još je potrebno definirati datoteke *message* koje će sadržavati ključeve i vrijednosti internacionalizacije. Datoteke *message* moraju biti smještene u direktoriju *src/main/resources*. Datoteka za zadani jezik ima naziv *messages.properties*, a datoteke za druge jezike imaju nazive *messages_XX.properties*, gdje je XX kod jezika.

Pogledi

Pogledi su smješteni unutar direktorija */resources/templates*. Struktura organizacije pogleda prikazana je na slici 4.4.

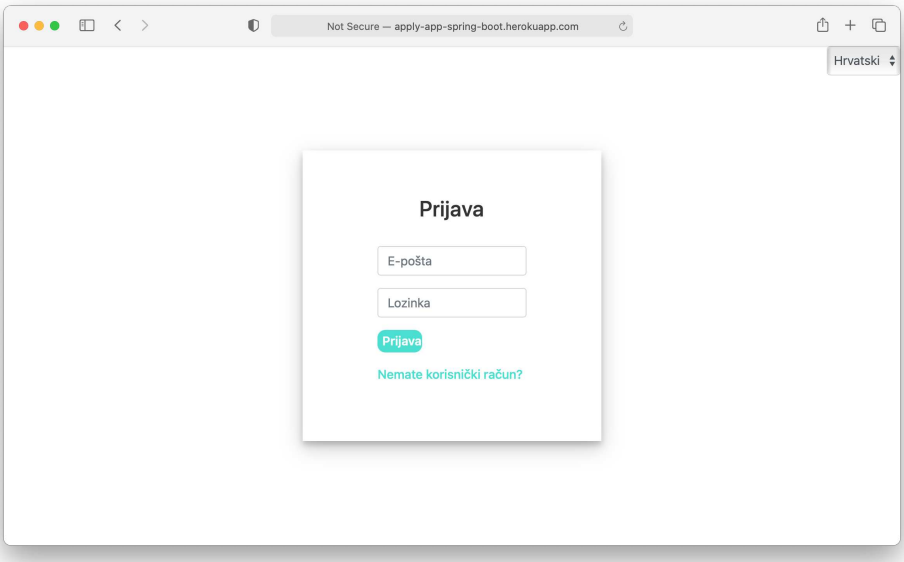


Slika 4.4: Pogledi

Unutar poddirektorija *auth* nalaze se pogledi povezani s autentificiranjem korisnika, *error* sadrži poglede koji se prikazuju u slučaju greške. Poddirektoriji *event* i *registration* sadrže poglede koji se odnose na događaje, odnosno na prijave, dok direktorij *shared* sadrži ponavljajuće dijelove pogleda kao što je zaglavlje web stranice. Unutar direktorija *user* smješteni su pogledi za prikazivanje i uređivanje korisnikovog profila.

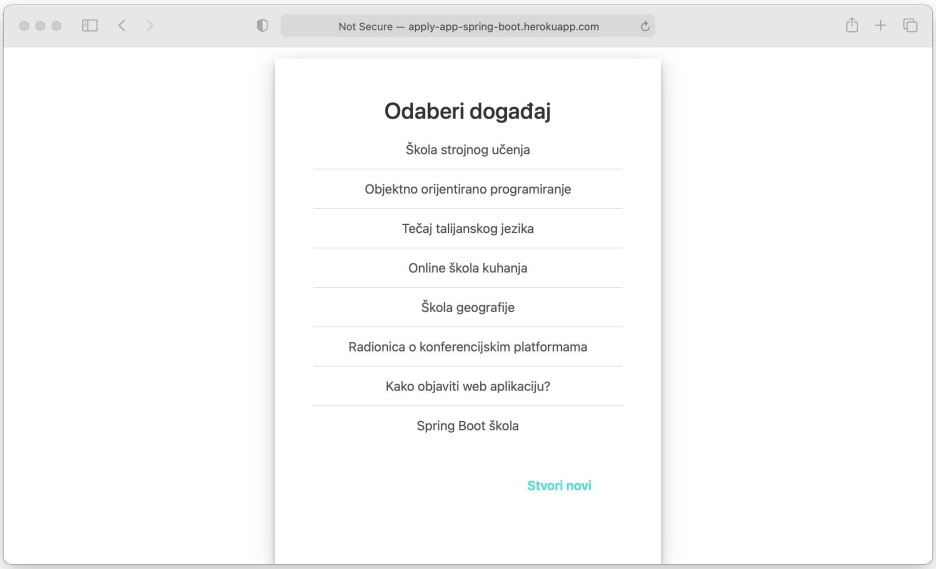
Za stilsko oblikovanje web stranice korišten je CSS okvir Bootstrap. Dodatne CSS i JavaScript datoteke smještene su unutar */resources/static*.

Na slici 4.5 prikazan je izgled prijave u aplikaciju. Potrebno je unijeti adresu e-pošte i lozinku, a klikom na "Nemate korisnički račun?" moguće je kreirati novi račun. Nakon uspješne prijave, u slučaju da je autentificirani korisnik administrator, otvara se prozor u kojem je moguće izabrati događaj koji se želi pregledati (Slika 4.6). Odabirom željenog događaja otvara se pregled i mogućnost uređivanja događaja (Slika 4.7). Klikom na *Registrations* u glavnom izborniku moguće je pregledati prijave za odabrani događaj (Slika 4.8).

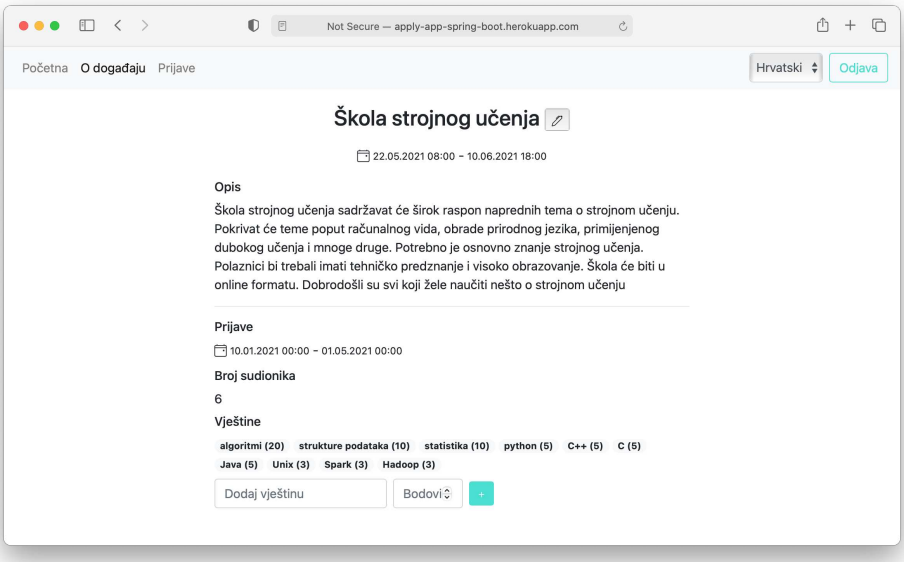


Slika 4.5: Prijava u aplikaciju

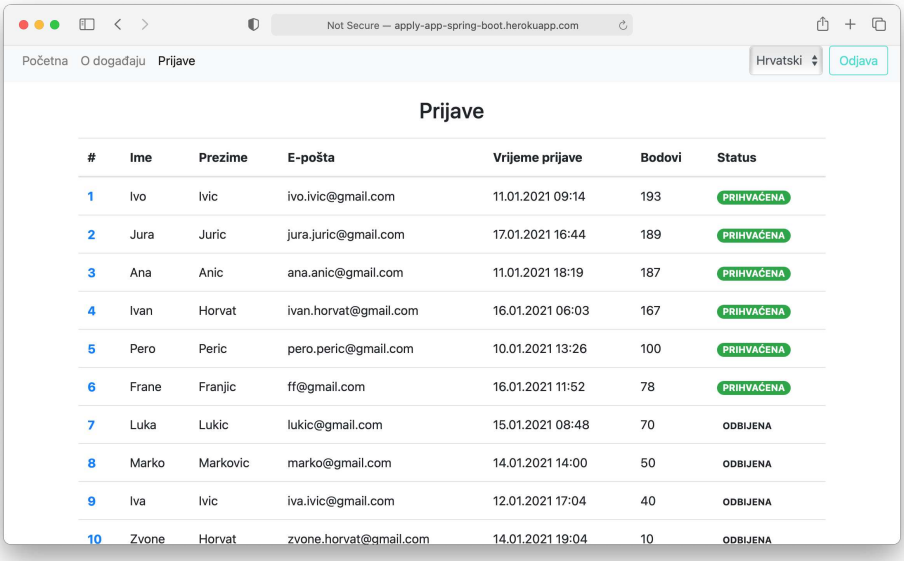
Ukoliko se nakon prijave radi o korisniku tada se otvara početna stranica s pozdravnom porukom (Slika 4.9). U glavnom izborniku pod "Događaji" dostupan je pregled svih trenutnih događaja i prijava na njih (slika 4.10). Klikom na "Moje prijave" korisnik može pregledavati sve svoje prijave (slika 4.11).



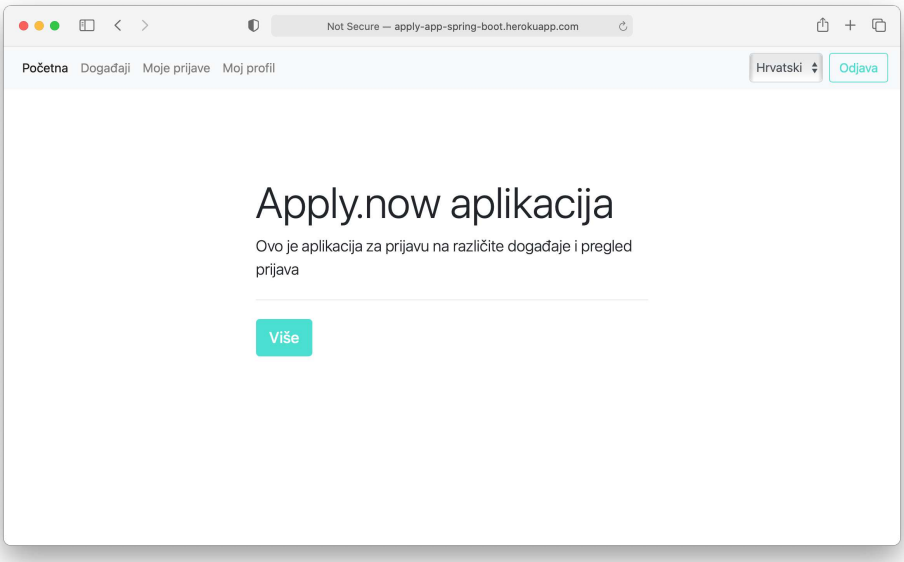
Slika 4.6: Odabir događaja



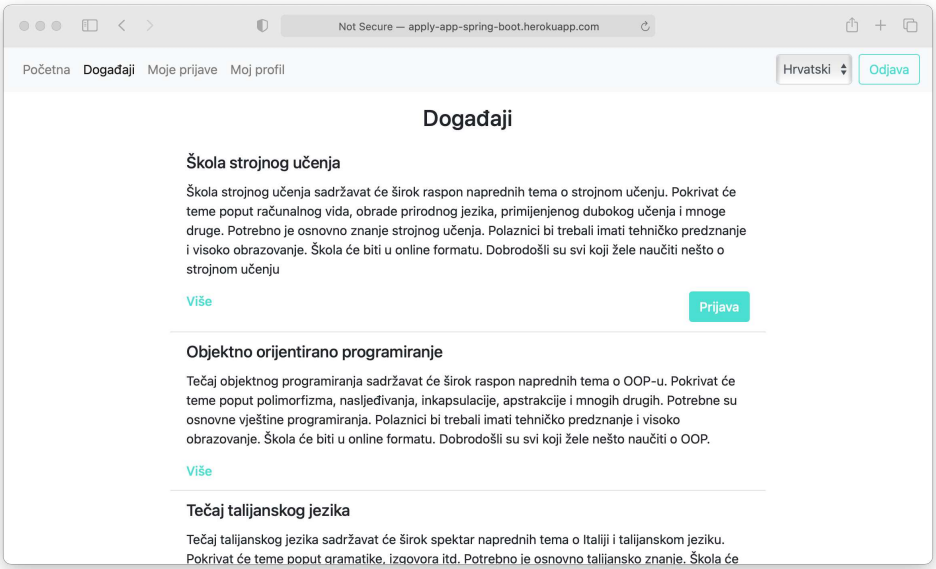
Slika 4.7: Pregled i uređivanje događaja



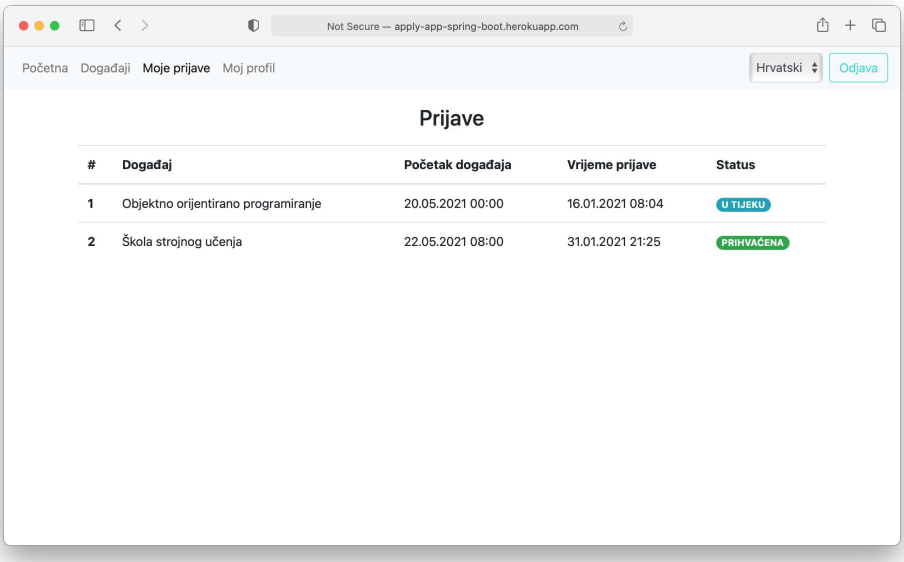
Slika 4.8: Pregled prijava za događaj



Slika 4.9: Početna stranica



Slika 4.10: Pregled događaja dostupnih za prijavu



Slika 4.11: Prijave prijavljenog korisnika

Bibliografija

[1] *Application Framework*, https://www.techopedia.com/definition/6005/application-framework, 06.10.2020.

[2] *Building an Application with Spring Boot*, https://spring.io/guides/gs/spring-boot/, 24.01.2021.

[3] *GitHub Integration (Heroku GitHub Deploys)*, https://devcenter.heroku.com/articles/github-integration, 24.01.2021.

[4] Ben Alex, Luke Taylor et al., *Spring Security Reference - Java Configuration*, https://docs.spring.io/spring-security/site/docs/5.4.x/reference/html5/#jc, 20.12.2020.

[5] Oliver Gierke, Thomas Darimont et al., *Spring Data JPA Reference - Query Creation*, https://docs.spring.io/spring-data/jpa/docs/2.5.x/reference/html/#repositories.query-methods.query-creation, 20.12.2020.

[6] ———, *Spring Data JPA Reference - Using @Query*, https://docs.spring.io/spring-data/jpa/docs/2.5.x/reference/html/#jpa.query-methods.at-query, 20.12.2020.

[7] Rod Johnson, Juergen Hoeller et al., *Core Technologies*, https://docs.spring.io/spring-framework/docs/5.3.x/reference/html/core.html, 20.12.2020.

[8] ———, *Framework Modules*, https://docs.spring.io/spring-framework/docs/5.0.0.RC3/spring-framework-reference/overview.html, 17.12.2020.

[9] Michael Nash, *Java Frameworks and Components: Accelerate Your Web Application Development*, Cambridge University Press, Cambridge, 2003.

[10] Craig Walls, *Spring in Action*, Manning Publications, Shelter Island, 2015.

[11] ———, *Spring Boot in Action*, Manning Publications, Shelter Island, 2016.

Sažetak

Spring okvir omogućava lagan i jednostavan razvoj modernih aplikacija u Javi. Nudi brojne značajke koje su organizirane u module. Središnji modul je Spring spremnik koji je zadužen za stvaranje, povezivanje, konfiguriranje i upravljanje životnim ciklusom pojedinog objekta (Spring *bean*-a). Za razvoj web aplikacija važan je Spring MVC modul koji kombinira sve prednosti Model-View-Controller obrasca i Spring okvira. Osim Spring modula postoje i Spring projekti koji nude sve značajke za određenu funkcionalnost. Za razvoj web aplikacija posebno su važni projekti Spring Data i Spring Security. Spring Data olakšava komunikaciju s različitim relacijskim i NoSql bazama podataka, dok Spring Security nudi cjelovito sigurnosno rješenje aplikacije. Osim toga, Spring Boot okvir donosi novi način razvoja Spring aplikacija. Na primjer, automatska konfiguracija smanjuje pisanje ponavljajućeg koda konfiguracije karakteristične za Spring-ove aplikacije, Spring Boot *starter* ovisnosti omogućuju razvojnom programeru fokusiranje na funkcionalnosti aplikacije, dok aktuator daje uvid u pozadinu rada pokrenute Spring Boot aplikacije.

Summary

Spring framework offers an easy and simple way of developing modern Java applications. It contains many features organised in modules. Core module is a Spring Container which creates, wires, configures and manages the life cycle of objects (Spring beans). Also of note, module for developing web applications is Spring MVC module, combining advantages of Model-View-Controller pattern and Spring framework. In addition to the Spring module, Spring projects are offered to provide all features for a particular functionality. Spring Data and Spring Security are thus significant projects for web development, Spring Data facilitates communication with various relational and NoSql databases, whereas Spring Security offers a complete security solution. Furthermore Spring Boot offers a new way of developing Spring applications. For example, auto-configuration reduces boilerplate configuration that is characteristic for Spring applications. Spring Boot starter dependencies also enable developer to focus on the functionality of the application, whereas actuator gives an inside look of running application.

Životopis

Rođena sam 31.10.1996. godine u Zadru. Pohađala sam osnovnu školu Jurja Dalmatinca, a zatim opću gimnaziju u Pagu. 2015. godine upisala sam preddiplomski studij Matematika na Prirodoslovno-matematičkom fakultetu u Zagrebu. Nakon završetka 2018. godine upisala sam diplomski studij Računarstvo i matematika na istom fakultetu. Tijekom prve godine diplomskog studija sudjelovala sam na ljetnoj školi za izradu web aplikacija firme Agency04. Od tada sam tamo zaposlena kao razvojni programer.