УНИВЕРЗИТЕТ У БЕОГРАДУ
МАТЕМАТИЧКИ ФАКУЛТЕТ

Андријана Босиљчић

РАЗВОЈ АПЛИКАЦИЈЕ ЗА ПРОНАЛАЖЕЊЕ МАЈСТОРА УПОТРЕБОМ ПРОГРАМСКОГ ЈЕЗИКА ЈАВА И ТЕХНОЛОГИЈА SPRING BOOT И THYMELEAF

мастер рад

Београд, 2024.

Ментор:

др Иван Чукић, доцент
Универзитет у Београду, Математички факултет

Чланови комисије:

др Саша Малков, ванредни професор
Универзитет у Београду, Математички факултет

др Мирјана Маљковић Ружичић, доцент
Универзитет у Београду, Математички факултет

Датум одбране:

Наслов мастер рада: Развој апликације за проналажење мајстора употребом програмског језика Јава и технологија *Spring Boot* и *Thymeleaf*

Резиме: У овом раду описане су технологије коришћене за развој веб апликације за проналажење мајстора, као и детаљан опис функционалности апликације. За развој *backend* дела апликације коришћен је програмски језик Јава и радни оквир *Spring Boot*, док је за развој *frontend* дела апликације коришћен шаблонски језик *Thymeleaf*. За рад са подацима и базом података коришћене су следеће технологије: *Java Persistence API*, *Hibernate* и *SQL server*. У оквиру рада развијена је апликација која корисницима омогућава да пронађу мајстора за одређени посао. Апликација, кроз алгоритам доделе мајстора, додељује кориснику најближег мајстора доступног у периоду који корисник наведе приликом попуњавања форме.

Кључне речи: Java, Spring, Spring Boot, Spring MVC, Spring Security, Thymeleaf, Hibernate, JSP, SQL server, веб апликација

Садржај

1 Увод **1**

2 Технологије коришћене за развој апликације **2**

2.1 Програмски језик Јава 2
2.2 Радни оквир *Spring* 5
2.3 Радни оквир *Spring Boot* 6
2.4 *Spring MVC* 11
2.5 *Spring Security* 13
2.6 *Thymeleaf* 14
2.7 *JPA* и *Hibernate* 30

3 Апликација за проналажење мајстора **35**

3.1 Серверски део апликације 36
3.2 Клијентски део апликације 47

4 Закључак **59**

Библиографија **60**

Глава 1

Увод

Развој савремених софтверских апликација постао је неопходан у различитим индустријама, захваљујући све већој дигитализацији и потреби за побољшањем пословних процеса. Потражња за услугама мајстора у различитим областима континуирано расте, али проналажење поузданог мајстора може бити изазовно и временски захтевно за многе кориснике. У том контексту, развој веб апликације која омогућава корисницима да брзо и ефикасно пронађу квалификоване мајсторе има изузетан значај. Коришћењем програмског језика **Јава** и следећих технологија: ***Spring Boot***, ***Thymeleaf***, ***JPA (Java Persistence API)***, ***Hibernate*** и ***SQL server***, у оквиру овог рада развијена је апликација која пружа пример како се технолошка решења могу применити на стварне животне сценарије ради побољшања квалитета услуга и олакшавања свакодневних активности корисника.

Структура рада организована је тако да у првом делу даје преглед коришћених технологија за развој апликације, затим у другом делу разматра појединачне функционалности апликације, док се у завршном делу дају предлози за даљи развој система.

Глава 2

Технологије коришћене за развој апликације

Апликација за проналажење мајстора је написана у програмском језику Јава. За њен развој су коришћене технологије *Spring Boot*, *Thymeleaf*, *JPA (Java Persistence API)*, *Hibernate* и *SQL server*.

2.1 Програмски језик Јава

Ова секција даје кратак преглед програмског језика Јава, чиме ће се стећи јасно разумевање основних концепата и карактеристика овог језика.

Историјат

Програмски језик Јава је настао као резултат 15-годишњег рада истраживача Џејмса Гозлинга, Патрика Нотона, Криса Варта, Еда Франка и Мајка Шеридана у великој корпорацији *Sun Microsystems*. Име је добио по чувеној Јава кафи коју су у великим количинама конзумирали креатори језика.

Главни подстицај за развој овог програмског језика била је потреба за програмским језиком који не зависи од платформе и који може да се користи за развој софтвера који би могао да се угради у различите електронске уређаје. Чланови дизајнерског тима Јаве временом су схватили да проблеми са преносивошћу, који се често јављају приликом писања кода за уграђене контролере, постоје и када се код развија за Интернет. Ова спознаја довела је до промене фокуса Јаве са електронксих уређаја на Интернет програмирање.

Поред тога што је поједноставила веб програмирање, Јава је такође решила нека од најтежих питања повезаних са Интернетом, као што су преносивост и безбедност.

На основу свега наведеног за програмски језик Јава се може рећи да за Интернет програмирање представља револуционарну силу која је променила свет. [5]

Особине

Јава је један од најпопуларнијих програмских језика данашњице, познат по својој свестраности и моћним особинама које омогућавају развој различитих врста апликација. У овој секцији биће размотрене кључне особине овог језика које доприносе његовој широкој употреби и успеху у индустрији. Неке од најзначајнијих особина [2] укључују:

- **Једноставност**: Концепти који се користе у Јави су једноставни и лако разумљиви, што олакшава учење језика. Јавина синтакса је једноставнија и сигурнија у поређењу са другим програмским језицима, као што су *C++* и *C*. Поред тога, њене конструкције су дизајниране тако да већина грешака може бити откривена током процеса превођења.

- **Објектна-оријентисаност**: Јава је објектно-оријентисан језик, што омогућава апстракцију података и програмирање своди на прављење објеката који међусобном интеракцијом решавају проблем, олакшавајући програмирање комплексних система.

- **Платформска независност**: Рачунарску платформу чине тип рачунара и оперативни систем. Већина програмских језика зависи од рачунарске платформе на којој се програми преводе и извршавају. Насупрот томе, Јава је независна од рачунарских платформи, што значи да ако програм, написан у овом програмском језику, функционише на једној платформи, он ће радити и на свим осталим платформама на којима постоји Јава преводилац и интерпретер.

- **Дистрибуираност**: Јава је изузетно погодна за мрежно програмирање и развој дистрибуираних апликација. Ове апликације се састоје од више програма који се извршавају на различитим рачунарима повезаним преко мреже. Захваљујући објектно-оријентисаним примитивама које

олакшавају представљање концепата у мрежном програмирању, креирање мрежних апликација у Јави је веома једноставно. Платформска независност, која је већ поменута, додатно поједностављује мрежно програмирање.

- **Интерпретација**: Јава је постала платформски независна захваљујући чињеници да се њен изворни код не преводи директно у машински код рачунара, већ у код нижег нивоа, познат као Јава бајт код, који је независан од платформе. Овај бајт код се извршава на интерпретеру познатом као Јава виртуелна машина. Иако Јава виртуелна машина није стварни рачунар, она делује као симулација праве машине, омогућавајући извршавање бајт кода. За сваку рачунарску платформу постоји прилагођена верзија Јава виртуелне машине, али сви они интерпретирају Јава бајт код на исти начин.

- **Мултитрединг (eng. *Multithreading*)** представља способност програмског језика да истовремено извршава више нити унутар једног програма. Свака нит је независна секвенца извршавања која може да обавља задатке паралелно са другим нитима, што омогућава ефикасније коришћење процесорских ресурса и побољшава перформансе програма. Јава нуди снажну подршку за мултитрединг путем класе `Thread` и интерфејса `Runnable`, који омогућавају програмерима да лако креирају, управљају и синхронизују нити. Захваљујући мултитредингу, Јава апликације могу обављати више задатака истовремено, као што су обрада корисничких захтева, ажурирање корисничког интерфејса, и комуникација са серверима.

- **Робустност**: Јава програми су изузетно робустни из два основна разлога. Први је тај што већина грешака у Јава програмима бива откривена већ током процеса превођења. Захваљујући строгој типизацији језика, све провере типова се извршавају у фази превођења, што омогућава откривање значајног броја грешака и потенцијалних неусклађености типова. Други разлог за робустност Јава програма је тај што Јава онемогућава програмерима коришћење опасних операција, попут аритметике показивача у *C++*. Програмер у Јави нема директан приступ меморији и не манипулише адресама. Уместо тога, Јава виртуелна машина сама управља меморијским простором, а простор који више

није потребан се аутоматски идентификује и ослобађа захваљујући сакупљачу отпадака.

- **Безбедност**: Један од кључних аспеката код мрежних апликација је безбедност. Ако програми размењују поверљиве информације преко мреже, оне морају бити адекватно заштићене. Поред тога, мрежни програми морају бити заштићени од злоупотреба од стране неовлашћених особа или програма, а рачунари од покретања злонамерних аплета. Јава нуди уграђене, ефикасне механизме за безбедност, као што су управљање меморијом и контрола приступа, који помажу у заштити од злонамерних напада.

2.2 Радни оквир *Spring*

Spring је радни оквир отвореног кода заснован на програмском језику Јава. Развио га је Род Џонсон 2003. године. Овај радни оквир пружа разноврсне концепте који подржавају различите аспекте развоја апликација, укључујући веб апликације, рад са базама података, безбедност и микросервисе. Такође се лако интегрише са другим оквирима и библиотекама као што су *Hibernate* за рад са базама података или *Thymeleaf* за рендеровање шаблона. [1]

Кључне карактеристике радног оквира *Spring*:

- **Инверзија контроле (eng. *Inversion of Control*)** је концепт који представља основу овог радног оквира. Омогућава развој апликација тако да се зависностима између објеката управља од стране оквира, а не програмера. То се постиже коришћењем механизма који омогућава управљање зависностима између различитих компоненти апликације на начин који смањује њихову међусобну повезаност. Овај механизам се назива уметање зависности.

- **Аспектно-оријентисано програмирање**: Радни оквир *Spring* омогућава коришћење аспектно-оријентисаног програмирања које омогућава издвајање заједничке логике, као што је пријављивање или безбедност, у посебне аспекте, чиме се смањује дуплирање кода.

2.3 Радни оквир *Spring Boot*

Spring Boot је радни оквир који представља подпројекат радног оквира *Spring* и који омогућава бржи развој Јава апликација. Његова једноставност коришћења и продуктивност у развоју апликација, учинили су га популарним избором за постојеће и нове пројекте. Многе фирме су усвојиле *Spring Boot* како би поједноставиле развој, побољшале одрживост и убрзале испоруку софтверских решења.

Spring Boot је настао као решење за сложеност традиционалних *Spring* конфигурација, пружајући једноставнији и ефикаснији приступ развоју апликација. Његова историја одражава посвећеност побољшању искуства програмера и прилагођавању модерним софтверским праксама, чинећи га кључним алатом у развоју Јава апликација.

Историјат

Радни оквир *Spring* је био популаран избор за Јава програмере због својих свестраних могућности за уметање зависности, аспектно-оријентисано програмирање и развој апликација. Међутим, како је радни оквир *Spring* сазревао, постајало је све сложеније конфигурисати и интегрисати га са различитим компонентама и технологијама. Програмери су често наилазили на изазове у постављању пројеката, управљању зависностима и конфигурисању апликационог контекста који представља централни интерфејс који управља животним циклусом и конфигурацијом компоненти у апликацији. [7]

Као одговор на ове изазове, створен је *Spring Boot* како би се поједноставио развојни процес. Главни циљеви су били:

- **Поједностављење конфигурације**: *Spring Boot* је имао за циљ да смањи потребу за ручном конфигурацијом и писањем шаблонксог кода, омогућавајући програмерима да започну рад на апликацијама са минималним подешавањем.
- **Повећање продуктивности**: Пружајући подразумеване конфигурације и вредности, идеја је била да *Spring Boot* убрза развој и смањи време проведено на задацима постављања и конфигурације.
- **Олакшан рад са микросервисима**: Са успоном архитектуре микросервиса, *Spring Boot* је пружио оквир који може лако подржати развој и

имплементацију микросервиса нудећи уграђене сервере и функционалности спремне за продукцију.

Основне карактеристике

Spring Boot је популарни радни оквир који поједностављује процес креирања Јава апликација. Основне карактеристике [7] овог радног оквира укључују:

- **Аутоматска конфигурација** је једна од најважнијих карактеристика радног оквира *Spring Boot*. Омогућава програмерима да брзо и лако поставе апликацију локално, на облак *(eng.* cloud*)* платформе, сопствене сервере, у *Docker* контејнере, или да користе *Kubernetes* за управљање апликацијама у контејнерима, без потребе за ручним конфигурисањем свих детаља, као што су: подешавање базе података, безбедност, порт на којем апликација ради, кеширање, повезивање са екстерним сервисима. *Spring Boot* анализира присутне зависности у пројекту и аутоматски примењује одговарајуће конфигурације како би апликација функционисала. Аутоматска конфигурација се омогућава коришћењем једне од анотација, `@EnableAutoConfiguration` или `@SpringBootApplication`.

  Пример:

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

  }
  ```

- **Уграђени сервери** играју кључну улогу у једноставности и брзини развоја апликација. За разлику од традиционалних Јава веб апликација које захтевају ручно постављање и конфигурисање спољних сервера као што су *Tomcat*, *Jetty* или *Undertow*, *Spring Boot* омогућава уграђивање сервера директно у апликацију. Након креирања *Spring Boot* апликације

и додавања зависности за веб функционалности, *Spring Boot* аутоматски конфигурише уграђени сервер.

Пример: Уколико се дода зависност за *Tomcat* у `pom.xml` фајл, као у коду испод, *Spring Boot* ће аутоматски укључити *Tomcat* као уграђени сервер. Ово значи да када се апликација покрене као Јава процес, сервер ће бити покренут у истом процесу, и апликација ће бити доступна на дефинисаном порту (подразумевано 8080).

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
    <version>3.2.1</version>
</dependency>
```

- **Подршка за микросервисе** је једна од кључних карактеристика радног оквира *Spring Boot* која је допринела његовој популарности у модерном развоју софтвера. Микросервиси представљају архитектонски стил у којем се велике апликације разбијају на мање, независне сервисе који могу бити развијани, испоручени и скалирани појединачно. *Spring Boot* пружа снажну подршку за развој микросервиса, чинећи овај процес једноставнијим и ефикаснијим. *Spring Boot* омогућава креирање микросервиса са минималном конфигурацијом. Уз анотације као што су `@SpringBootApplication`, `@RestController`, и `@EnableDiscoveryClient`, програмери могу брзо дефинисати и конфигурисати своје микросервисе.

  Пример:

```
@SpringBootApplication
@RestController
public class Mikroservis {
    public static void main(String[] args) {
        SpringApplication.run(Mikroservis.class, args);
    }

    @GetMapping("/hello")
    public String hello() {
        return "Zdravo iz mikroservisa!";
    }
}
```

- ***Spring Boot* стартери** су дизајнирани да поједноставе и убрзају развој апликација пружајући унапред конфигурисане скупове зависности које су неопходне за одређене функционалности у апликацији. На пример, за

креирање веб алпикације довољно је додати `spring-boot-starter-web` у *Maven* или *Gradle* зависности. Овај стартер аутоматски укључује све потребне зависности за развој веб апликације са радним оквиром *Spring MVC*, који је описан у наставку рада.

Пример:

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

Стартери су такође дизајнирани да буду прошириви, што значи да се зависности могу додавати или уклањати по потреби. Стартери обезбеђују стандардан начин конфигурисања апликација унутар *Spring Boot* окружења.

- **Уметање зависности** је процес у којем објекти дефинишу своје зависности искључиво кроз параметре конструктора, параметре метода или особине које се подешавају на инстанци објекта након што је она креирана или враћена из метода. Постоје три типа уметања зависности:

  - **Уметање зависности засновано на конструктору** је процес убацивања зависности неког објекта кроз његов конструкторски аргумент у тренутку његовог инстанцирања. Другим речима, можемо рећи да се зависности обезбеђују као објекти кроз сам конструктор тог објекта.

    Пример:

    ```
    public class Aplikacija {
        private PorukaServis porukaServis;

        // Konstruktor koji ubacuje zavisnost
        public Aplikacija(PorukaServis porukaServis) {
            this.porukaServis = porukaServis;
        }

        public void posaljiPoruku(String poruka,
                                  String primalac
                                  ) {
            // Koriscenje ubrizgane zavisnosti
            this.porukaServis.posalji(poruka,
                                      primalac);
        }
    }
    ```

– **Уметање зависности засновано на сетеру** је врста уметања зависности у објекат путем сетера, односно метода за подешавање вредности. За разлику од уметања зависности заснованом на конструктору, где се зависности убацују преко конструктора, овде се зависности убацују након што је објекат већ креиран.

  Пример:

```
public class Aplikacija {
    private PorukaServis porukaServis;

    // Seter za ubacivanje zavisnosti
    public void setPorukaServis(
            PorukaServis porukaServis
    ) {
        this.porukaServis = porukaServis;
    }

    public void posaljiPoruku(String poruka,
                              String primalac) {
        // Koriscenje ubrizgane zavisnosti
        this.porukaServis.posalji(poruka,
                                  primalac);
    }
}
```

– **Аутоматско уметање зависности** постиже се коришћењем анотације `@Autowired`. Користи се уколико класа има више конструктора, јер се у том случају анотација мора експлицитно навести изнад конструктора који треба да се користи за уметање зависности. Такође постоји аутоматско уметање зависности директно на поље, што се може видети у примеру испод.

  Пример:

```
@Service
public class AdminService {

    // Automatsko ubrizgavanje UserRepository
    // u ovo polje
    @Autowired
    UserRepository userRepository;

    public List<User> getAllUsers ()
    {
        // Koriscenje ubrizgane vrednosti
        return userRepository.findAll();
    }
```

```
}
```

- **Метричка и мониторинг подршка**: *Spring Boot* пружа значајну подршку за праћење перформанси и здравља апликације кроз свој модул `Actuator` који омогућава лако приступање информацијама о раду апликације, као што су метрике перформанси, стање система, конфигурације.

  Пример: Користећи `@Endpoint(id = "metrics")`, може се приступити свим метрикама на `/actuator/metrics` крајњој тачки.

Захваљујући овим особинама, *Spring Boot* омогућава програмерима да се фокусирају на пословну логику апликације, а не на инфраструктурне детаље, што га чини изузетно популарним избором у савременом развоју софтвера.

2.4 *Spring MVC*

Spring MVC (Model-View-Controller) је део радног оквира *Spring* који омогућава развој робусних и скалабилних веб апликација у програмском језику Јава. Овај радни оквир прати познати *MVC* шаблон, који раздваја логику апликације на три основне компоненте: модел (eng. *Model*), приказ (eng. *View*) и контролер (eng. *Controller*). Такво раздвајање омогућава јаснију структуру кода, бољу одрживост и лакше тестирање апликација. [7] Основне карактеристике овог радног оквира су:

- **Анотацијски контролери**: *Spring MVC* користи анотације као што су `@Controller`, `@RequestMapping`, `@GetMapping` и `@PostMapping` за дефинисање контролера и руковање захтевима. Ово омогућава једноставнију и читљивију конфигурацију у поређењу са конфигурацијама базираним на *XML*-у, које представљају метод управљања конфигурацијом апликације, у коме се компоненте конфигуришу *XML* фајловима.

- **Флексибилно мапирање захтева**: *Spring MVC* омогућава да се за сваки метод контролера подесе различити *URL* путеви и *HTTP* методе. Овај приступ пружа прецизну контролу над тим како се обрађују кориснички захтеви и који методи ће се активирати у зависности од врсте захтева.

- **Подршка за више приказа**: *Spring MVC* подржава различите технологије за приказивање, као што су *JSP*, *Thymeleaf*, и *FreeMarker*, омогућавајући развој прилагодљивих корисничких интерфејса.
- **Интеграција са другим *Spring* модулима**: *Spring MVC* се лако интегрише са другим модулима у оквиру радног оквира Spring, као што су Spring Security за аутентикацију и ауторизацију, и Spring Data за приступ бази података.
- **Руковање формама и валидација**: Овај оквир пружа уграђене механизме за обраду података унетих кроз форме, укључујући валидацију улазних података и руковање грешкама.

Архитектура радног оквира *Spring MVC*

Архитектура радног оквира *Spring MVC* је заснована на познатом *Model-View-Controller* шаблону, који дели апликацију на три главна слоја:

- **Модел *(eng.* Model*)*** управља подацима који се приказују корисницима. У радном оквиру *Spring MVC*, модел може бити било која (eng. *POJO (Plain Old Java Object)*) класа која садржи атрибуте и методе за приступ и управљање подацима.
- **Приказ (eng. *View*)** је одговоран за приказивање података кориснику. У радном оквиру *Spring MVC*, приказ је обично шаблонска датотека, на пример *Thymeleaf*, која прима податке из модела и приказује их на одређени начин. Контролер прослеђује податке приказу преко модела.
- **Контролер (eng. *Controller*)** је средишњи део *Spring MVC* апликације који обрађује корисничке захтеве, координише модел и приказ и враћа одговарајући одговор. Контролер прима захтеве од корисника, обрађује их и затим одлучује који подаци треба да се прикажу и у ком приказу.

Ток података у *Spring MVC*

Ток података у овом радном оквиру функционише на следећи начин:

- **Кориснички захтев**: Корисник шаље захтев, на пример преко *URL*-а ка апликацији.

- **Контролер**: Захтев прима одговарајући контролер, који обрађује захтев и интерагује са моделом.
- **Модел**: Контролер позива сервисне слојеве или методе за рад са подацима у моделу.
- **Приказ**: Контролер прослеђује податке моделу, који затим бира одговарајући приказ за приказивање резултата.
- **Одговор**: Приказ враћа *HTML* или други тип садржаја кориснику као одговор на његов захтев.

Овако организована архитектура пружа јасну структуру и раздвајање одговорности, што резултира лакшом одрживошћу и развоју апликација.

2.5 *Spring Security*

Spring Security је свеобухватан оквир унутар радног оквира *Spring* који омогућава примену безбедносних мера у Јава апликацијама. Овај оквир пружа низ функционалности, укључујући аутентификацију и ауторизацију корисника, заштиту од уобичајених безбедносних претњи, као и интеграцију са различитим методама за аутентификацију, као што су LDAP и OAuth2. [7]

Основне карактеристике оквира *Spring Security* су следеће:

- **Аутентификација**: *Spring Security* омогућава проверу идентитета корисника путем различитих метода, укључујући корисничко име и лозинку, токене, и SSO (eng. *Single Sign-On*) механизме. Систем аутентификације се лако интегрише у било коју Јава апликацију помоћу оквира *Spring Security*.
- **Ауторизација**: Након успешне аутентификације, *Spring Security* омогућава контролу приступа различитим деловима апликације на основу улога корисника или других критеријума.
- **Заштита од напада**: *Spring Security* пружа уграђену заштиту од уобичајених веб напада. На пример, *Cross-Site Request Forgery* је врста безбедносног напада на веб апликације, где злонамерни актер покушава да превари корисника да неовлашћено изврши радњу у апликацији у

којој је корисник већ аутентификован. *Spring Security* има уграђену заштиту од оваквих напада тако што је за захтеве који мењају податке неопходан посебан *Cross-Site Request Forgery* токен који се верификује на серверу, што спречава извршење злонамерних захтева.

- **Флексибилност конфигурације**: *Spring Security* се може конфигурисати на више начина, што омогућава програмерима да лако прилагоде безбедносну политику потребама своје апликације. Три главна начина конфигурације су:

  - ***XML* конфигурација** је начин конфигурације, где се користе *XML* датотеке за дефинисање безбедносних правила, као што су захтеви за аутентификацију, приступни нивои и обрада сесија.
  - **Јава конфигурација** је модернији приступ који омогућава конфигурацију безбедности директно у Јава коду помоћу анотација. Овај начин је постао популаран са појавом радног оквира *Spring Boot*, јер омогућава лакшу и читљивију конфигурацију без потребе за додатним *XML* датотекама. На пример, анотација *@EnableWebSecurity* се користи за омогућавање веб безбедности.
  - **Конфигурација путем радног оквира *Spring Boot***: *Spring Boot* пружа подразумевану конфигурацију безбедности која може бити додатно прилагођена помоћу својстава у `application.properties` или *application.yml* датотекама. Овај начин је најједноставнији и најчешће се користи за брзу интеграцију безбедности у *Spring Boot* апликацијама, а програмери могу лако да измене подразумевана подешавања у складу са својим потребама.

2.6 *Thymeleaf*

Thymeleaf је савремени шаблонски језик, који се користи како за веб апликације, тако и за самосталне апликације, и омогућава обраду *HTML*, *XML*, *JavaScript*, *CSS*, па чак и обичног текста. *Thymeleaf* је од самог почетка дизајниран са фокусом на веб стандарде, посебно *HTML5*, што омогућава креирање шаблона који су у потпуности валидни.

Природни шаблони

Природни шаблони представљају кључни део технологије *Thymeleaf*. Овај концепт омогућава коришћење *HTML* или *XML* датотека у њиховом оригиналном облику, без утицаја на дизајн који види крајњи корисник. То значи да програмери могу убризгати логичке елементе у саме шаблоне без нарушавања њихове валидности и функционалности.

Главна предност природних шаблона је у томе што они омогућавају да се исти шаблон користи и као прототип током дизајнирања, као и у коначном развоју. Ово знатно олакшава комуникацију између дизајнерских и развојних тимова, јер дизајнери могу да користе шаблоне у истом облику у коме ће их и развојни тимови користити за развој функционалности. [3]

Основне карактеристике овог концепта укључују:

- **Очување изгледа**: *HTML* код који се користи за изградњу корисничког интерфејса не мора бити модификован до степена где постаје нечитљив или тешко разумљив дизајнерима.
- **Валидни шаблони**: Уколико је потребно, шаблони могу остати у потпуности валидни у смислу стандарда као што је *HTML5*, што је корисно за тестирање и имплементацију.
- **Преглед шаблона без сервера**: *HTML* датотеке могу се приказати у било ком прегледачу без потребе за покретањем серверске стране апликације.

Пример:

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

<head>
  <title>Master Bob - Customer page</title>
</head>
<body>

<div th:if="${message != null}"
     class="alert alert-success">
  [[${message}]]
</div>

<div th:if="${#lists.size(jobs) > 0}" class="column">
  <div class="category">
```

```
    <img class="category-icon"
         th:src="${category.iconUrl}" width="50"
         height="50">
    <h3 th:text="${category.category}"></h3>
  </div>
</div>

<div class="flex-container">
  <div th:each="job, iterStat : ${jobs}"
       class="flex-item">
    <a th:href="@{'/user/customer/job-description/'
                + ${job.id}}" >
    <img th:src="${job.imageUrl}" class="job-image">
    <div class="overlay-text" th:text="${job.name}">
    </div>
    </a>
</div>

</div>

</body>
</html>
```

Овај *HTML* код је потпуно читљив и дизајнер може да га користи у свом радном току, али када се шаблон обради на серверу, динамички се убацују вредности као што су `jobs`, `job.name`, `job.imageUrl`, `category.iconUrl`, `category.category` или се приказују елементи у зависности од логике, као на пример порука успешности из примера.

Природни шаблони тиме обезбеђују лакоћу у развоју и сарадњи између тимова, и чине *Thymeleaf* јединственим решењем за креирање шаблона.

Интеграција са радним оквиром *Spring*

Интеграција шаблонског језика *Thymeleaf* са радним оквиром *Spring* омогућава развој веб апликација које се лако одржавају и проширују, уз подршку за напредне функционалности попут обраде форми, валидације и руковања грешкама. У овом контексту, *Thymeleaf* представља пуну замену за традиционалну технологију *JSP* (eng. *Java Server Pages*) која се користи за креирање динамичких веб страница у програмском језику Јава, уз бројне предности у погледу елегантности и могућности прилагођавања.

У овом поглављу биће детаљно објашњена интеграција шаблонског језика *Thymeleaf* са оквиром *Spring*. Посебна пажња биће посвећена кључним аспек-

тима, као што су конфигурација, руковање обрасцима и начин на који *Thymeleaf* у потпуности замењује традиционалне *JSP* шаблоне у *Spring MVC* апликацијама.

Аутоматска конфигурација шаблонског језика *Thymeleaf* у радном оквиру *Spring Boot*

У поглављу 2.3 описана је аутоматска конфигурација у радном оквиру *Spring Boot*, као и улога стартера у поједностављењу развоја апликација, који омогућавају лако укључивање различитих библиотека и компоненти, а једна од њих је и *Thymeleaf*.

Користећи *Spring Boot Starter Thymeleaf*, развојни процес постаје значајно поједностављен јер *Spring Boot* аутоматски конфигурише све што је потребно за рад са шаблонским језиком *Thymeleaf* додавањем *Maven* зависности у `pom.xml` фајл.

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-thymeleaf</artifactId>
    <version>3.2.1</version>
</dependency>
```

Spring Boot аутоматски конфигурише *ThymeleafViewResolver*, компоненту која решава имена погледа и повезује их са *HTML* фајловима. Такође подешава *ThymeleafTemplateEngine*, који је одговоран за приказивање *HTML* фајлова. Ова конфигурација омогућава да *Spring Boot* аутоматски пронађе и прикаже одговарајући фајл, када се из функције контролера врати име погледа као ниску.

Пример: У коду испод може се видети функција контролера која враћа ниску `homePage`. *Spring Boot* ће аутоматски пронаћи и приказати страницу дефинисану у фајлу `homePage.html`.

```
@RequestMapping(value = {"/", "/user"},
                method = RequestMethod.GET)
    public String homePage(Model model) {
        List<JobCategory> jobCategories =
            adminService.getAllJobCategories();
        model.addAttribute("jobCategories", jobCategories);

        return "homePage";
    }
```

Spring Boot претражује *Thymeleaf* шаблоне у директоријуму `src/main/resources/templates/`. Сви *HTML* фајлови који се налазе на тој локацији могу бити рендеровани као погледи.

Коришћење података у *Thymeleaf* шаблонима

У шаблонском језику *Thymeleaf* подаци из Јава кода се лако користе и приказују у *HTML* шаблонима. Овај механизам омогућава динамичко генерисање садржаја у погледима тако што интегрише податке из контролера са *HTML* страницама. У наставку је описано како се користе подаци у *Thymeleaf* шаблонима, како их проследити из контролера, као и које су могућности за њихову обраду. [3]

- **Прослеђивање података из контролера**: У радном оквиру *Spring*, подаци се у *Thymeleaf* шаблоне прослеђују преко *Model* објекта у контролеру.

  Пример контролера који шаље податке у *HTML* страницу може се видети у коду испод. Метод `deleteServiceRequest` поставља податак `message` у модел.

```
@Controller
@RequestMapping("/user/contractor")
public class ContractorController {

    @Autowired
    ContractorService contractorService;

    @RequestMapping("/service-request/delete/{id}")
    public String deleteServiceRequest(
                    @PathVariable(name = "id")
    Integer serviceRequestId, Model model) {
        contractorService.deleteServiceRequestById
                        (serviceRequestId);
        model.addAttribute("message",
                              "Successfully deleted
                               service request!");

        return getContractorServiceRequests(model);
    }
}
```

- **Коришћење података у *Thymeleaf* шаблонима**: Подаци који су додати у *Model* објекат могу бити директно коришћени у *Thymeleaf* шаблонима помоћу специфичне *Thymeleaf* синтаксе.

  У примеру испод, уколико је податак `message` различит од `null` кориснику ће се приказати вредност тог податка.

```
<!DOCTYPE html>
<html xmlns:th="http://www.thymeleaf.org" >
<head>
    <title>Master Bob - Contractor page</title>
</head>
<body>
    <div th:if="${message != null}"
         class="alert alert-success">
        [[${message}]]
    </div>
</body>
</html>
```

- **Динамичко убацивање текста у *Thymeleaf* шаблоне** омогућено је коришћењем атрибута `th:text`.

  Пример: У коду испод објекат `user` се убацује у поглед преко контролера у радном оквиру *Spring Boot*. Атрибут `th:text` омогућава приказ својстава тог објекта.

```
<td th:text="${user.id}">User ID</td>
```

- **Приказивање листи у *Thymeleaf* шаблонима** се може урадити помоћу `th:each` атрибута, који се користи за итерацију кроз елементе листе.

  Пример: У коду испод може се видети функција контролера која прослеђује листу `usersList` *Thymeleaf* шаблону.

```
@GetMapping("/getAllUsers")
public String getAllUsers (Model model)
{
    List<User> usersList = adminService.getAllUsers();
    model.addAttribute("usersList", usersList);
    return "manageUsers";
}
```

  Начин на који се прослеђена листа `usersList` приказује у шаблону може се видети у коду испод. Користи се `th:each` атрибут за итерацију кроз

елементе листе `usersList`, где је сваки елемент доступан као `user` и његова својства се приказују унутар шаблона.

```
<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:th="http://www.thymeleaf.org">
<head>
  <title>Master Bob - Manage users</title>
</head>
<body>
    <table class="table table-bordered table-striped">
    <thead class="thead-dark"
           style="background-color: #ffcc00;">
    <tr>
      <th style="text-align: center;">User ID</th>
      <th style="text-align: center;">Name</th>
      <th style="text-align: center;">Surname</th>
      <th style="text-align: center;">Email</th>
      <th style="text-align: center;">
        Phone number
      </th>
      <th style="text-align: center;">
        Job Categories
       </th>
      <th style="text-align: center;">Address</th>
      <th style="text-align: center;">Role</th>
      <th style="text-align: center;">Enabled</th>
      <th style="text-align: center;">Delete</th>
    </tr>
    </thead>
    <tbody id="userTableBody">
    <tr th:each="user : ${usersList}">
      <td th:text="${user.id}">User ID</td>
      <td th:text="${user.name}">Name</td>
      <td th:text="${user.surname}">Surname</td>
      <td th:text="${user.username}">Email</td>
      <td th:text="${user.phoneNumber}">
        Phone number
      </td>
      <td>
        <ul>
          <li th:each="category :
                      ${user.jobCategories}"
          th:text="${category.category}"></li>
        </ul>
      </td>
      <td th:text="${user.address}">Address</td>
      <td th:text="${user.role}">Role</td>
      <td>
```

```
            <div class="form-group mb-3 d-flex
                        align-items-center">
              <div style="display: inline-block;">
                <label class="form-label"
                    th:text="${user.enabled ?
                               'Yes' : 'No'}">
                </label>
              </div>
              <div style="display: inline-block;
                          margin-left: 10px;">
              <a class="btn custom-btn"
                 id="editButton"
              th:href="@{'/user/admin/edit/'
                          + ${user.id} + '/' +
              ${user.enabled}}">
              <span
                class="glyphicon glyphicon-pencil">
                    Edit
              </span>
              </a>
              </div>
            </div>

      </td>
      <td style="text-align: center;">
        <a class="btn custom-btn"
        th:href="@{'/user/admin/delete/'
                   + ${user.id}}">
          <i class="fa fa-trash"
             aria-hidden="true">
          </i>
        </a>
      </td>
    </tr>
    </tbody>
  </table>
</body>
</html>
```

- **Контролни изрази у *Thymeleaf* шаблонима**: *Thymeleaf* подржава основне контролне структуре као што су услови `if` и `unless`, које могу да се користе за приказивање садржаја на основу одређених услова.

  Пример: Код испод приказује категорије послова, уколико је величина прослеђене листе `jobs` већа од нуле.

```
<div th:if="${#lists.size(jobs) > 0}" class="column">
  <div class="category">
    <img class="category-icon"
```

```
         th:src="${category.iconUrl}" alt=""
    width="50" height="50">
    <h3 th:text="${category.category}"></h3>
  </div>
</div>
```

- **Укључивање фрагмената у *Thymeleaf* шаблоне**: *Thymeleaf* нуди могућност приказа дела шаблона, који се назива фрагмент, као резултат његовог извршавања. Фрагменти омогућавају лакше одржавање и поновно коришћење кода у *Thymeleaf* шаблонима. Посебно су корисни за стварање компоненти које се могу више пута користити на различитим страницама, као што су хедери, фусноте или форме. У *Spring Boot* пројектима је добра пракса да се фрагменти смештају на локацију `src/main/resources/templates/fragments`. Фрагменти се обележавају *Thymeleaf* атрибутом `th:fragment`, након чега се наводи име фрагмента, као у примеру испод.

  ```
  <div th:fragment="navbar">
      <nav class="navbar navbar-expand-lg
                  navbar-light bg-light">
          <a class="navbar-brand a-img" href="/"">
              <img th:src="@{/images/logo.png}" alt="">
          </a>
      </nav>
  </div>
  ```

  Постоје три начина за укључивање фрагмената у *Thymeleaf* шаблоне:

  - `th:replace` је *Thymeleaf* атрибут који мења елемент на који је примењен садржајем из фрагмента.

    У примеру испод, `<div>` елемент ће бити замењен садржајем фрагмента чији је назив `navbar`.

    ```
    <div th:replace="~{fragments/navbar :: navbar}">
    </div>
    ```

  - `th:insert` је *Thymeleaf* атрибут који убацује садржај фрагмента унутар елемента на који је примењен, без замене самог елемента.

    У примеру испод, *HTML* структура ће задржати `<div>` елемент, али ће унутар њега бити убачен садржај фрагмента.

    ```
    <div th:insert="~{fragments/navbar :: navbar}">
    </div>
    ```

- `th:include` је *Thymeleaf* атрибут који задржава постојећи елемент, али унутар њега убацује садржај фрагмента. То значи да се фрагмент додаје као део елемента уместо да га замени.

  У примеру испод, `<div>` елемент ће остати у *DOM* стаблу, а садржај фрагмента ће бити уметнут унутар тог `<div>` елемента.

  ```
  <div th:include="~{fragments/navbar :: navbar}">
  </div>
  ```

- **Креирање форми у *Thymeleaf* шаблонима** је један од најчешћих случајева употребе овог шаблонског језика у *Spring Boot* апликацијама. *Thymeleaf* пружа снажну подршку за везивање *HTML* образаца са Јава објектима који се користе у контролерима, чиме омогућава једноставну и интуитивну интеракцију између *frontend* и *backend* дела апликације.

  Основни концепти форме у *Thymeleaf* шаблонима:

  - `th:action` и `th:method` су атрибути који се користе за дефинисање акције коју форма треба да изврши и методе којом се врши слање података.
  - `th:field` је атрибут који се користи за повезивање одређеног *HTML* поља са својством објекта.
  - `th:object` је атрибут који се користи за повезивање форме са Јава објектом који се прослеђује из контролера. Сви елементи форме могу користити атрибуте попут `th:field` за директно везивање поља из објекта са *HTML* пољима.

  Пример форме за регистрацију корисника:

  ```
  <!DOCTYPE html>
  <html xmlns:th="http://www.thymeleaf.org">
  <head>
    <meta charset="UTF-8">
    <title>Master Bob - Registration</title>
  </head>
  <body>

      <form method="post" role="form"
            th:action="@{/register}"
            th:object="${user}" >
            <div class="form-group mb-3">
              <label class="form-label" for="firstName">
  ```

```
        First Name
    </label>
    <input
            class="form-control"
            id="firstName"
            name="name"
            placeholder="Enter first name"
            th:field="*{name}"
            type="text"
            required
    />
    <p th:errors = "*{name}"
       class="text-danger"
       th:if="${#fields.hasErrors('name')}">
    </p>
  </div>

  <div class="form-group mb-3">
    <label class="form-label" for="surname">
        Last Name
    </label>
    <input
            class="form-control"
            id="surname"
            name="surname"
            placeholder="Enter last name"
            th:field="*{surname}"
            type="text"
            required
    />
    <p th:errors = "*{surname}"
       class="text-danger"
       th:if="${#fields.hasErrors('surname')}">
    </p>
  </div>

  <div class="form-group mb-3">
    <label class="form-label" for="email">
        Email
    </label>
    <input
            class="form-control"
            id="email"
            name="email"
            placeholder="Enter email address"
            th:field="*{username}"
            type="email"
            required
    />
```

```
  <p th:errors = "*{email}"
     class="text-danger"
     th:if="${#fields.hasErrors('email')}">
  </p>
</div>

<div class="form-group mb-3">
  <label class="form-label"
         for="password">
      Password
  </label>
  <input
          class="form-control"
          id="password"
          name="password"
          placeholder="Enter password"
          th:field="*{password}"
          type="password"
          required
  />
  <p th:errors = "*{password}"
     class="text-danger"
     th:if="${
      #fields.hasErrors('password')
     }">
  </p>
</div>
<div class="form-group mb-3">
  <label class="form-label"
         for="phoneNumber">
      Phone number
  </label>
  <input
          class="form-control"
          id="phoneNumber"
          name="phoneNumber"
          placeholder="Enter phone number"
          th:field="*{phoneNumber}"
          type="text"
  />
  <p th:errors = "*{phoneNumber}"
     class="text-danger"
     th:if="${
      #fields.hasErrors('phoneNumber')
     }">
  </p>
</div>
<div class="form-group mb-3">
  <label>Account type</label>
```

```
    <select class="form-control"
            th:field="*{role}"
    id="role" name="role" required>
      <option value="" selected>
            Select type
      </option>
      <option value="customer">
            Customer
      </option>
      <option value="contractor">
            Contractor
      </option>
    </select>
    <p th:errors = "*{role}"
       class="text-danger"
       th:if="${#fields.hasErrors('role')}">
    </p>
  </div>

  <div class="form-group mb-3"
       id="additionalFields">
    <div class="form-group mb-3">
      <label>Job category</label>
      <select class="form-control"
        th:field="*{jobCategories}"
        id="category"
        name="category" multiple>
        <option value="" selected>
                Select category
        </option>
        <th:block th:each="category :
                   ${jobCategories}">
          <option th:value="${category.id}"
            th:text="${category.category}">
          </option>
        </th:block>
      </select>
      <p th:errors = "*{jobCategories}"
        class="text-danger"
        th:if="${
            #fields.hasErrors('jobCategories')
        }">
      </p>
    </div>

    <div class="form-group mb-3">
      <label class="form-label" for="address">
        Address
      </label>
```

```
        <input
                class="form-control"
                id="address"
                name="address"
                placeholder="Enter address"
                th:field="*{address}"
                type="text"
        />
        <p th:errors = "*{address}"
           class="text-danger"
           th:if="${
              #fields.hasErrors('address')
           }">
        </p>
      </div>

      <div class="form-group mb-3">
        <label class="form-label" for="postCode">
          Post code
        </label>
        <input
                class="form-control"
                id="postCode"
                name="postCode"
                placeholder="Enter post code"
                th:field="*{postCode}"
                type="text"
        />
        <p th:errors = "*{postCode}"
           class="text-danger"
           th:if="
              ${#fields.hasErrors('postCode')}">
        </p>
      </div>
    </div>

    <div class="form-group-button">
      <button class="btn custom-btn"
              type="submit">
          Register
      </button>
      <span>Already registered?
      <button type="button"
              class="btn custom-btn"
      th:onclick=
          "|window.location.href='/login'|">
            Login
      </button>
        </span>
```

```
            </div>
        </form>
</body>
</html>
```

`th:action=„@/register”` дефинише путању на коју ће се подаци послати. У овом случају, подаци ће бити послати на `/register` путању, коју обрађује контролер.

`th:object=„$user”` везује форму за објекат `user`. Сваки елемент у форми који има `th:field` користиће овај објекат као основни модел података.

`th:field=„*name”` повезује *HTML* поље са својствима објекта `user`. На пример, вредност унесена у поље `First Name` биће постављена у `user.name`. Аналогно је и за остала поља форме.

`th:errors = „*email”` приказује све грешке везане за поље `email`. Аналогно је и за остала поља форме.

`#fields.hasErrors('email')` је функција која као параметар прима поље и враћа `boolean` вредност која говори да ли постоје грешке у валидацији за то поље.

Интеграција са радним оквиром *Spring Security*

Интеграција *Spring Security* са шаблонским језиком *Thymeleaf* пружа могућност да се сигурносне функције директно укључе у *HTML* шаблоне. Ово омогућава контролу приступа деловима странице, приказивање информација о кориснику који је пријављен, као и прилагођавање садржаја на основу улога корисника. [4]

- **Почетна конфигурација**: У *Spring Boot* пројекту потребно је додати *Maven* зависности за *Spring Security* и *Thymeleaf*, као у коду испод.

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

<dependency>
    <groupId>org.thymeleaf.extras</groupId>
```

```
    <artifactId>
        thymeleaf-extras-springsecurity6
    </artifactId>
    <version>3.1.1.RELEASE</version>
</dependency>
```

- **Кључни атрибути радног оквира *Spring Security* у *Thymeleaf* шаблонима**:
  - `sec:authorize` је атрибут који контролише приступ деловима странице на основу сигурносних правила.

    Пример:

    ```
    <a class="dropdown-item custom-a"
       th:href="@{/logout}"
    sec:authorize="isAuthenticated()">
        Logout
    </a>

    <div sec:authorize="hasRole('admin')"
         class="users-btn">
        <button type="button"
                class="btn custom-btn"
                th:onclick="|window.location.href=
                '/user/admin/getAllUsers'|">
            Users
        </button>
    </div>
    ```

    `sec:authorize=„isAuthenticated()”` проверава да ли је корисник аутентификован. Уколико јесте, линк за одјављивање из апликације ће бити приказан кориснику, у супротном неће.

    `sec:authorize=„hasRole('admin')”` проверава да ли корисник има админ улогу. Уколико има елемент `<div>` ће бити приказан кориснику, у супротном неће.
  - `sec:authentication` је атрибут који приказује податке о тренутно пријављеном кориснику.

    Пример:

    ```
    <h2>Dobrodosli, <span sec:authentication="name">
                    </span>!</h2>
    ```

    `sec:authentication=„name”` приказује име тренутно пријављеног корисника.

– `sec:authentication=„principal.username”` је атрибут који приказује корисничко име пријављеног корисника из `UserDetails` објекта.

Пример:

```
<h2>Dobrodosli, <span sec:authentication=
                     "principal.username">
                 </span>!</h2>
```

`sec:authentication=„principal.username”` користи податке из објекта `UserDetails` који је конфигурисан у *Spring Security* и приказује корисничко име пријављеног корисника.

2.7 *JPA* и *Hibernate*

Hibernate и *JPA (Java Persistence API)* су кључни алати у развоју Java апликација које захтевају ефикасно управљање подацима у релационим базама података. Ови алати омогућавају програмерима да се фокусирају на пословну логику уместо на сложене *SQL* упите, чиме се повећава продуктивност и смањује ризик од грешака. [6]

JPA

JPA (Java Persistence API) је стандардна спецификација која дефинише начин за управљање перзистентношћу података у Јава апликацијама. Омогућава програмерима да користе Јава објекте уместо *SQL* упита за интеракцију са базама података. Основне карактеристике *JPA* укључују:

- **Објектно-релационо мапирање**: *JPA* мапира Јава класе на табеле у бази података, чиме се поједностављује процес рада са подацима.
- **Једноставно коришћење**: *JPA* омогућава рад са подацима путем објеката, што чини код чистијим и лакшим за одржавање.
- **Подршка за различите имплементације**: *JPA* се може користити са различитим радним оквирима за објектно-релационо мапирање, а *Hibernate* је једна од најпопуларнијих имплементација.

Hibernate

Hibernate је најпознатија имплементација *JPA* спецификације, која пружа додатне функционалности изван стандарда:

- **Напредна објектно-релациона мапирања**: *Hibernate* управља комплексним мапама и релацијама између објеката, олакшавајући рад с вишеструким табелама.
- **Кеширање**: *Hibernate* нуди механизме за кеширање који побољшавају перформансе апликација смањујући број упита ка бази података.
- **Подршка за разне базе података**: *Hibernate* може радити са различитим релационим базама података, чиме се повећава флексибилност апликација.

Основне операције

Основне операције које се могу изводити у алатима *JPA* и *Hibernate*:

- **Креирање ентитета**: У *JPA*, ентитети су кључни објекти који се мапирају на табеле у бази података.

  Пример:

```
@Entity
@Table(name = "users")
public class User {
    @Id
    @GeneratedValue(
        strategy = GenerationType.IDENTITY
    )
    private Integer id;
    private String name;
    private String surname;
    @Column(name = "email")
    private String username;
    @Column(name = "phone_number")
    private String phoneNumber;
    private String password;

    private String address;

    private Double longitude;
```

```
    private Double latitude;

    @ManyToMany
    @JoinTable(
            name = "user_job_category",
            joinColumns = @JoinColumn(
                            name = "user_id"
            ),
            inverseJoinColumns = @JoinColumn(
                        name = "category_id"
            )
    )
    private Set<JobCategory> jobCategories;

    @OneToMany(mappedBy = "customer")
    private List<ServiceRequest>
        customerServiceRequests;

    @OneToMany(mappedBy = "contractor")
    private List<ServiceRequest>
        contractorServiceRequests;

    private String role;

    @Column(name = "post_code")
    private String postCode;

    public List<ServiceRequest>
            getCustomerServiceRequests() {
        return customerServiceRequests;
    }

    @Column(name = "verification_code",
            length = 64)
    private String verificationCode;

    private boolean enabled;
}
```

У коду изнад су употребљене следеће анотације:

- `@Entity` је анотација која означава да је класа `User` *JPA* ентитет, што значи да ће бити мапирана на табелу у бази података.
- `@Table(name = "users")` је анотација којом се дефинише име табеле у бази података на коју ће бити мапиран овај ентитет. У овом случају, назив табеле је `users`.

- `@Id` означава поље `id` као примарни кључ табеле. Свакој инстанци класе `User` ће бити додељен јединствени идентификатор.
- `@GeneratedValue(strategy = GenerationType.IDENTITY)` је анотација која указује да ће идентификатор бити аутоматски генерисан од стране базе података.
- `@Column(name = "email")` је анотација која указује да ће поље `username` бити мапирано на колону `email` у бази података. Аналогно је и за остала поља означена анотацијом `@Column`.
- `@ManyToMany` је анотација која указује на више-према-више релацију између корисника и категорија послова.
- `@JoinTable` је анотација која дефинише спојну табелу која ће се користити за управљање везом. У овом случају, табела ће се звати `user_job_category`, а колоне ће бити `user_id` и `category_id`.
- `@OneToMany(mappedBy = „customer")` је анотација која указује на релацију један-према-више између корисника и захтева за услугу. `mappedBy` указује да је класа `ServiceRequest` одговорна за одржавање везе између ентитета `User` и `ServiceRequest` у бази података.

• **Операције над ентитетима** у контексту *JPA* и *Hibernate* представљају акције које се извршавају над објектима који су мапирани на табеле у бази података. Основне операције над ентитетима које се користе у JPA и Hibernate укључују:

  - ***CRUD* операције** су основне операције над ентитетима:
    * **Креирање (eng. *Create*)** је операција чувања новог ентитета у бази података. Када се нови ентитет креира у апликацији, он се додаје у базу помоћу метода `save()`.
      Пример:
      ```
      public User register(User user) {
          return userRepository.save(user);
      }
      ```
    * **Читање (eng. *Read*)** је операција проналажења постојећег ентитета у бази података. Методе попут `findById()` омогућавају проналажење ентитета на основу његовог идентификатора.
      Пример:

```
public Optional<Job> getJobById (Integer id){
        return jobRepository.findById(id);
}
```

* **Ажурирање (eng. *Update*)** је операција измене постојећег ентитета. Када се атрибути ентитета промене, а затим се ентитет сачува, његова вредност се ажурира у бази података.
  Пример:

```
public String editAccountEnablement
        (Integer id, boolean enabled)
{
    User user = adminRepository.getUser(id);

    user.setEnabled(!enabled);

    adminRepository.save(user);
}
```

* **Брисање (eng. *Delete*)** је операција уклањања ентитета из базе података, користећи методе попут `deleteById()` које бришу ентитет на основу његовог идентификатора.
  Пример:

```
public void deleteUser(Integer id) {
    userRepository.deleteById(id);
}
```

– **Прилагођени упити**: *JPA* и *Hibernate* омогућавају писање прилагођених упита који се користе за извршавање специфичних операција над ентитетима које не покривају основне *CRUD* методе. Ови упити се дефинишу коришћењем анотације `@Query`.

Пример:

```
@Query("SELECT u FROM User u WHERE
       u.verificationCode = ?1")
User findByVerificationCode(String code);
```

Овај прилагођени упит враћа корисника на основу `verificationCode`. Ова врста упита омогућава програмерима да прецизно дефинишу шта желе да преузму из базе података.

Глава 3

Апликација за проналажење мајстора

Master Bob, апликација која је развијана у оквиру овог мастер рада, представља платформу за повезивање корисника са мајсторима из различитих сектора. Главна функција апликације је да корисницима омогући брзо и лако проналажење квалификованих мајстора у њиховој близини, у складу са њиховим индивидуалним потребама. У апликацији постоје три улоге: ***customer***, ***contractor***, ***admin***. Нерегистровани посетиоци имају могућност да виде све послове доступне у апликацији, као и детаљан опис и оквирне цене за сваки посао. Регистровани корисници који имају улогу ***customer*** могу да одаберу посао који треба да им се одради и да креирају захтев за ту услугу у ком наводе термин који им одговара, након чега им алгоритам описан у наставку рада, додељује најближег мајстора који је доступан у том периоду. Такође имају могућност да виде, мењају и бришу креиране захтеве за услугу. Корисници који имају улогу ***contractor*** имају могућност да забележе период када су заузети, у случају да су у том периоду на одмору или имају заказан неки посао и да виде, мењају и бришу захтеве за услугу за коју су задужени. Корисници који имају улогу ***admin*** имају могућност да управљају пословима доступним у апликацији и корисницима апликације. Сви регистровани корисници, независно од улоге коју имају у апликацији, могу мењати своје податке на профилу. Апликација се састоји од серверског и клијентског дела, који комуницирају путем *REST API*. За дизајн апликације коришћен је образац *MVC (Model-View-Controller)*. Подаци се чувају у релационој бази података и за те потребе је коришћен *SQL Server*. Како би се обезбедила ефикасност у кому-

никацији између мајстора и купца, након сваке промене у захтеву корисника шаље се мејл нотификације помоћу веб сервиса *mailtrap.io*. За одређивање тачне локације мајстора и купца, како би се купцу доделио најближи мајстор, користи се веб сервис *Nominatim API*. Апликација је дизајнирана да буде једноставна за употребу, са једноставним и функционалним интерфејсом који омогућава лаку навигацију, чак и за кориснике који немају много техничког искуства.

3.1 Серверски део апликације

Серверски део апликације развијен је са циљем да подржи ефикасно руковање корисничким захтевима, управљање подацима, као и имплементацију пословне логике која омогућава корисницима заказивање услуга мајстора.

Апликација је организована на начин који омогућава једноставну комуникацију између корисника и система путем ***REST API***, који је задужен за процесирање свих захтева са клијентског дела апликације.

Функционалности серверског дела апликације

Функционалности серверског дела апликације обухватају обраду корисничких захтева, управљање базом података, аутентификацију и ауторизацију корисника, доделу мајстора на основу локације и доступности, као и интеграцију са екстерним сервисима за слање нотификација и довлачења података о географској ширини и дужини локације.

Аутентификација и ауторизација корисника

Аутентификација и ауторизација у оквиру апликације реализоване су коришћењем оквира *Spring Security*. Аутентификација корисника омогућава проверавање њиховог идентитета на основу података за пријаву, док ауторизација одређује којим ресурсима и функционалностима корисник може да приступи на основу своје улоге.

Аутентификација је имплементирана преко сервиса `LoginService`, који управља корисничким подацима и верификацијом уноса за пријаву. Када корисник покуша да се пријави, *Spring Security* прелази у режим аутентификације и шаље корисничке податке `authenticationProvider`, који користи

сервис `LoginService` да би преузео корисничке информације из базе података, а затим их пореди са подацима које је корисник унео током пријаве. Ако су кориснички подаци тачни, `authenticationProvider` враћа објекат `Authentication` који представља аутентификованог корисника. У супротном, аутентификација ће бити одбијена.

За сигурност лозинки користи се ***BCrypt*** алгоритам за енкрипцију, који обезбеђује висок степен сигурности. Лозинка коју корисник уноси током пријаве се упоређује са енкриптованом лозинком која се налази у бази података. Класа ***BCryptPasswordEncoder*** се користи за упоређивање лозинки на сигуран начин.

```
@Autowired
LoginService loginService;

@Bean
public UserDetailsService userDetailsService() {
    return loginService;
}

@Bean
public AuthenticationProvider authenticationProvider()
{
    DaoAuthenticationProvider authenticationProvider =
        new DaoAuthenticationProvider();
    authenticationProvider.setUserDetailsService
        (userDetailsService());
    authenticationProvider.setPasswordEncoder
                           (passwordEncoder());
    return authenticationProvider;
}

@Bean
public BCryptPasswordEncoder passwordEncoder() {
    return new BCryptPasswordEncoder();
}
```

За ауторизацију корисника коришћена је улога корисника коју систем додељује приликом регистрације. Постоје три основне улоге: ***customer***, ***contractor*** и ***admin***, где свака од њих има специфичан ниво приступа и могућности у апликацији. Улогу ***customer*** имају корисници који траже мајстора, улогу ***contractor*** имају корисници који су мајстори, тј. извођачи радова, а улогу ***admin*** има корисник који управља корисницима и пословима у апликацији. Путем *Spring Security* конфигурације, у класи `WebSecurityConfig`, дефинисане су путање којима свака улога може приступити коришћењем `aut-`

`horizeHttpRequests` функције.

На пример, путање које се односе на мајсторе `/contractor/` доступне су само корисницима са улогом `contractor`, док ***admin*** има приступ само административним рутама `/admin/`. Путање `/register`, `/login`, `/category/`, `/job-description/` су доступне свим корисницима, без потребе за аутентификацијом. Путање `/user/` захтевају да корисници буду аутентификовани. Сви захтеви који почињу са `/user/` морају бити одобрени само корисницима који су се пријавили.

`http.formLogin(AbstractAuthenticationFilterConfigurer::permitAll)` подразумева да је форма за пријављивање доступна свима.

```
@Bean
public SecurityFilterChain securityFilterChain
(HttpSecurity http) throws Exception {
    http
        .csrf(AbstractHttpConfigurer::disable)
        .authorizeHttpRequests((requests) -> requests
            .requestMatchers("/", "/register",
            "/login", "/category/**",
            "/job-description/**").permitAll()
            .requestMatchers("/user/**")
                .authenticated()
            .requestMatchers("/customer/**")
                .hasRole("customer")
            .requestMatchers("/contractor/**")
                .hasRole("contractor")
            .requestMatchers("/admin/**")
                .hasRole("admin")
            .anyRequest().authenticated()
        ).formLogin(
            AbstractAuthenticationFilterConfigurer
                ::permitAll
         )
         .formLogin((form) -> form
            .loginPage("/login")
            .usernameParameter("email")
            .defaultSuccessUrl("/user")
            .permitAll())
        .logout((logout) -> logout.permitAll());

    return http.build();
}
```

Рад са базом података

За потребе развоја апликације коришћен је ***SQL Server*** као систем за управљање релационим базама података. У `application.properties` фајлу је потребно дефинисати параметре за повезивање са базом података.

```
spring.datasource.url=jdbc:sqlserver://DESKTOP-
U70VHA6\\SQLEXPRESS;database=MasterBob;
spring.datasource.username=usrMasterBob
spring.datasource.password=masterWork12
spring.datasource.driverClassName=com.microsoft.
sqlserver.jdbc.SQLServerDriver

spring.jpa.hibernate.ddl-auto=update
spring.jpa.show-sql=true
spring.jpa.properties.hibernate.dialect=
    org.hibernate.dialect.SQLServer2012Dialect
```

За управљање ентитетима и подацима коришћени су алати ***JPA*** (***Java Persistence API***) и ***Hibernate***. У апликацији су дефинисани следећи ентитети:

- User представља корисника алликације.

```
@Entity
@Table(name = "users")
public class User {
    @Id
    @GeneratedValue(
        strategy = GenerationType.IDENTITY
    )
    private Integer id;
    private String name;
    private String surname;
    @Column(name = "email")
    private String username;
    @Column(name = "phone_number")
    private String phoneNumber;
    private String password;

    private String address;

    private Double longitude;

    private Double latitude;

    @ManyToMany
    @JoinTable(
        name = "user_job_category",
```

```
        joinColumns =
            @JoinColumn(name = "user_id"),
        inverseJoinColumns =
            @JoinColumn(name = "category_id")
    )
    private Set<JobCategory> jobCategories;

    @OneToMany(mappedBy = "customer")
    private List<ServiceRequest>
        customerServiceRequests;

    @OneToMany(mappedBy = "contractor")
    private List<ServiceRequest>
        contractorServiceRequests;
    private String role;

    @Column(name = "post_code")
    private String postCode;

    @Column(name = "verification_code",
            length = 64)
    private String verificationCode;

    private boolean enabled;
}
```

- JobCategory дефинише категорије послова, што омогућава њихово груписање по врсти услуге.

```
@Entity
@Table(name = "job_category")
public class JobCategory {
    @Id
    @GeneratedValue(
        strategy = GenerationType.IDENTITY
    )
    private Integer id;

    private String category;
    @Column(name ="icon_url")
    private String iconUrl;

    @OneToMany(mappedBy = "category")
    private List<Job> jobs;

    @ManyToMany(mappedBy = "jobCategories")
    private Set<User> users;
}
```

- Job представља послове који су доступни у апликацији, при чему је сваки посао сврстан у неку категорију.

```
@Entity
@Table(name = "job")
public class Job {

    @Id
    @GeneratedValue(
        strategy = GenerationType.IDENTITY
    )
    private Integer id;

    private String name;

    private Integer duration;

    @Column(name = "image_url")
    private String imageUrl;
    @ManyToOne
    @JoinColumn(name = "category_id")
    private JobCategory category;

    @OneToMany(mappedBy = "job")
    private List<ServiceRequest> serviceRequests;
}
```

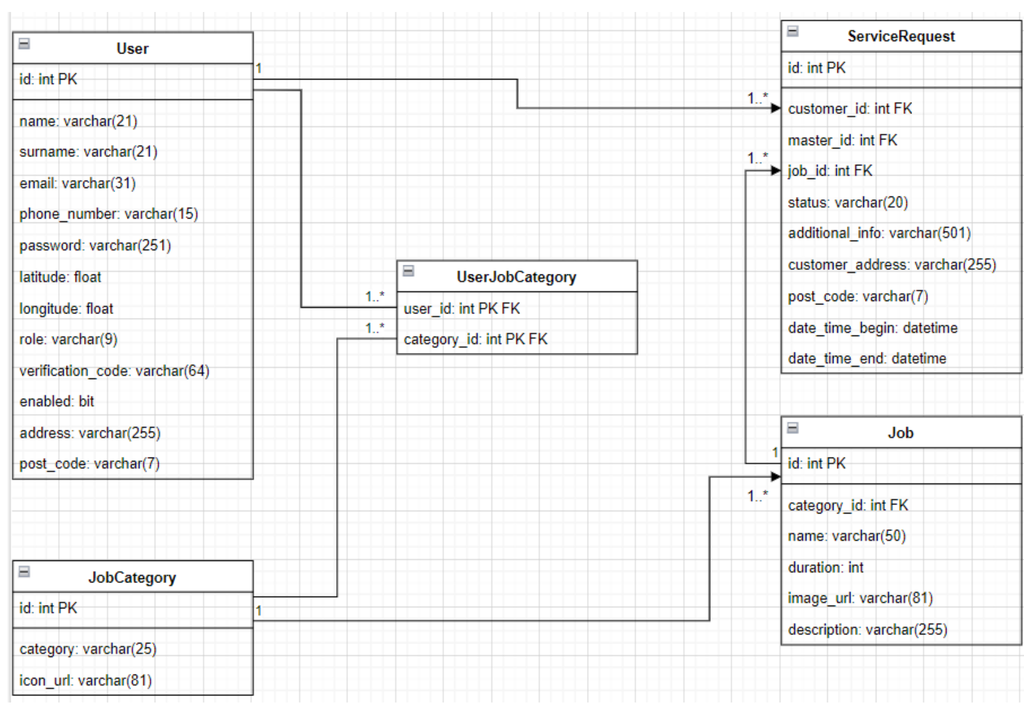
- ServiceRequest представља захтев за услугу мајстора који креирају корисници са улогом ***customer***.

```
@Entity
@Table(name = "service_request")
public class ServiceRequest {

    @Id
    @GeneratedValue(
        strategy = GenerationType.IDENTITY
    )
    private Integer id;

    @ManyToOne
    @JoinColumn(name = "customer_id")
    private User customer;

    @ManyToOne
    @JoinColumn(name = "contractor_id")
    private User contractor;

    @ManyToOne
```

```
    @JoinColumn(name = "job_id")
    private Job job;

    @Enumerated(EnumType.STRING)
    @Column(name = "status")
    private ServiceStatus serviceStatus;

    @Column(name = "date_time_begin")
    private Timestamp dateTimeBegin;

    private String dateTimeBeginString;

    @Column(name = "date_time_end")
    private Timestamp dateTimeEnd;

    private String dateTimeEndString;
    @Column(name = "post_code")
    private String postCode;
    @Column(name = "additional_info")
    private String additionalInfo;

    @Column(name = "customer_address")
    private String customerAddress;
}
```

Између ентитета `ServiceRequest` и `User` постоји веза `@ManyToOne`, што значи да један корисник може креирати више захтева за услугу мајстора. Иста веза постоји и између ентитета `ServiceRequest` и `Job`, што значи да више захтева за услугу мајстора могу бити повезани са једним истим послом и између ентитета `Job` и `JobCategory`, што значи да више послова може припадати једној истој категорији. Између ентитета `User` и `JobCategory` постоји веза `@ManyToMany`, што значи да један мајстор може да буде задужен за више категорија послова, и једна категорија посла може бити везана за више мајстора. Дијаграм који приказује релације између ентитета може се видети на слици 3.1.

За управљање ентитетима у бази података и апстракцију између података и бизнис логике коришћени су репозиторијуми. Они користе ***CRUD*** операције за креирање, читање, ажурирање и брисање података из базе, а ***JPA*** омогућава аутоматску имплементацију ових операција без потребе за писањем *SQL* упита. Такође је могуће дефинисати прилагођен упит коришћењем анотације `@Query`.



Слика 3.1: Дијаграм који приказује релације између ентитета

```
public interface UserRepository extends
                    JpaRepository<User, Integer> {

    Optional<User> findByUsername (String username);

    @Query("SELECT u FROM User u
            WHERE u.verificationCode = ?1")
    User findByVerificationCode(String code);
}
```

Објашњење кода изнад: `UserRepository` проширује `JpaRepository<User, Integer>`, што значи да наслеђује основне *CRUD* операције за ентитет `User`. Функција `findByUsername` тражи корисника у бази података на основу његовог корисничког имена. *JPA* аутоматски генерише упит заснован на називу методе, што значи да ће тражити запис где је `username` једнако датом параметру. Функција `findByVerificationCode` користи прилагођену `@Query` анотацију да би дефинисала прилагођен упит.

Обрада захтева преко *REST API*

REST API пружа основу за комуникацију између серверског и клијентског дела апликације. Све кључне операције у апликацији обављају се преко *HTTP*

захтева.

- `/user/admin/getAllUsers` - GET захтев за дохватање свих корисника из базе података
- `/user/admin/getAllJobs` - GET захтев за дохватање свих послова из базе података
- `/user/admin/delete/id` - POST захтев за брисање корисника на основу id-ја
- `/user/admin/job/delete/id` - POST захтев за брисање посла на основу id-ја
- `/user/admin/delete-category/id` - *POST* захтев за брисање категорије посла на основу *id*-ја
- `/user/admin/edit/id/enabled` - POST захтев за ажурирање доступности налога корисника на основу id-ја
- `/user/admin/job/edit/id` - *POST* захтев за ажурирање посла на основу *id*-ја
- `/user/admin/new-job` - *POST* захтев за креирање новог посла
- `/user/admin/new-job-category` - *POST* захтев за креирање нове категорије посла
- `/user/customer/category/id` - *GET* захтев за дохватање категорије посла на основу *id*-ја
- `/user/customer/job-description/id` - *GET* захтев за дохватање описа посла на основу *id*-ја
- `/user/customer/job/id` - *GET* захтев за дохватање посла на основу *id*-ја
- `/user/customer/service-request` - *POST* захтев за креирање захтева за услугу мајстора
- `/user/customer/service-request` - *GET* захтев за дохватање захтева за услугу мајстора који је тренутно улоговани корисник креирао

- `/user/customer/service-request/delete/id` - *POST* захтев за брисање захтева за услугу мајстора на основу *id*-ја
- `/user/customer/service-request/edit/id` - *GET* захтев за дохватање странице на којој се налази форма за ажурирање захтева за услугу мајстора
- `/user/customer/service-request/save/edit/id` - *POST* захтев за ажурирање захтева за услугу мајстора на основу *id*-ја
- `/change-profile-info` - *GET* захтев за дохватање странице на којој се приказује форма за ажурирање података о кориснику
- `/login` - *GET* захтев за дохватање странице на којој се приказује форма за пријављивање
- `/logout` - *POST* захтев за одјављивање из апликације
- `/change-profile-info/id` - *POST* захтев за ажурарање података о кориснику на основу *id*-ја
- `/user/contractor/service-request` - *GET* захтев за дохватање захтева за које је задужен тренутно улоговани мајстор
- `/user/contractor/service-request/delete/id` - *POST* захтев за брисање захтева за услугу мајстора на основу *id*-ја
- `/user/contractor/service-request/edit/id` - *GET* захтев за дохватање странице на којој се приказује форма за ажурирање захтева за услугу мајстора на основу *id*-ја
- `/user/contractor/service-request/save/edit/id` - *POST* захтев за чување ажурираних информација о захтеву за услугу мајстора
- `/user/contractor/book-a-date` - *GET* захтев за дохватање странице на којој се приказује форма за бележење датума и времена када је корисник који има улогу *contractor* заузет
- `/user/contractor/book-a-date` - *POST* захтев за бележење датума и времена када је корисник који има улогу *contractor* заузет

- `/register` - *GET* захтев за дохватање странице на којој се приказује форма за регистрацију
- `/register` - *POST* захтев за креирање новог корисника

Интеграција са екстерним сервисима

Апликација користи екстерне сервисе за слање нотификација путем мејла и прецизно одређивање локације купца и мајстора.

- **Слање нотификација**: Сваки пут када дође до промене статуса захтева, апликација шаље обавештење кориснику путем мејла. За те потребе коришћен је веб сервис ***mailtrap.io***, који функционише као **SMTP (eng. *Simple Mail Transfer Protocol*)** сервер који омогућава слање и пријем електронске поште преко интернета.
- ***Nominatim API*** је веб сервис који служи за геокодирање, тј. претварање текстуалних адреса у географске координате. Такође може обављати и реверзно геокодирање, тј. претварање географских координата у читљиву адресу. Овај веб сервис се у апликацији користи за добијање иформација о географској ширини и дужини на основу адресе и поштанског броја, за потребе рачунања удаљености мајстора и купца.

Алгоритам доделе мајстора

Алгоритам за доделу мајстора састоји се из следећих корака:

- Из базе података се дохватају сви мајстори задужени за посао наведен у захтеву креираном од стране корисника са улогом ***customer***.
- Проверава се ко је од тих мајстора доступан у периоду који је наведен у захтеву.
- Уколико ниједан мајстор није доступан, онда се сви мајстори задужени за посао наведен у захтеву сматрају доступним, али се кориснику који је креирао захтев приказује порука да мајстор може променити термин.
- Листа доступних мајстора се филтрира тако што се тражи мајстор најближи кориснику који је креирао захтев. Најближи мајстор се тражи

на основу географске ширине и дужине добијене помоћу јавног веб сервиса. ***Nominatim API*** на основу адресе и поштанског броја. Удаљеност купца и мајстора рачуна се помоћу ***Haversine*** формуле, која се користи за израчунавање удаљености између две тачке на површини сфере, узимајући у обзир закривљеност Земље.

```
private double calculateDistance (
    double customerLatitude,
    double customerLongitude,
    double contractorLatitude,
    double contractorLongitude)
{
    double latDistance = Math.toRadians(
            customerLatitude - contractorLatitude
    );
    double lonDistance = Math.toRadians(
            customerLongitude - contractorLongitude
    );
    double a = Math.sin(latDistance / 2) *
               Math.sin(latDistance / 2)
               + Math.cos(Math.toRadians
                   (customerLatitude)
                )
               * Math.cos(Math.toRadians
                   (contractorLatitude)
                )
               * Math.sin(lonDistance / 2)
               * Math.sin(lonDistance / 2);
    double c = 2 * Math.atan2(Math.sqrt(a),
                   Math.sqrt(1 - a));

    // RADIUS_ZEMLJE = 6371
    return 6371 * c;
}
```

- Купцу се за одабрану услугу додељује најближи мајстор

3.2 Клијентски део апликације

У оквиру овог поглавља биће описан клијентски део апликације, који представља интерфејс са којим корисници директно комуницирају. Основна улога клијентског дела је да омогући кориснику лаку и интуитивну навигацију кроз функционалности апликације, као и интеракцију са сервером путем различитих захтева. За развој клијентског дела коришћен је ***Thymeleaf***, као

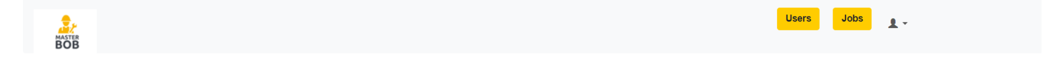
шаблонкси језик, који обезбеђује динамичко генерисање *HTML* страница на основу података са сервера. У наставку ће бити објашњен кориснички интерфејс апликације.

Функционалности клијентског дела апликације

Функционалности клијентског дела апликације играју кључну улогу у омогућавању корисницима да лако приступе и користе различите услуге. Ове функционалности обухватају регистрацију и пријаву корисника, интуитивно прегледање и одабир жељених услуга, као и преглед и ажурирање налога и креираних захтева.

Навигациони мени

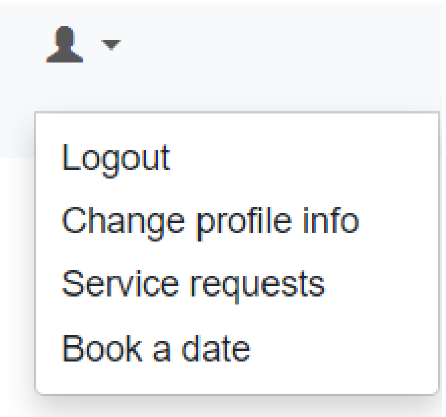
Навигациони мени (слика 3.2) се састоји од слике која представља лого апликације и од падајућег менија профила (слика 3.3) у ком корисник који није улогован или регистрован може одабрати опцију за пријаву или регистрацију, док корисник који је пријављен има следеће опције: да се одјави из апликације, да оде на страницу за ажурирање личних података или на страницу на којој се приказују захтеви за услугу мајстора. Корисник који има улогу ***customer*** у падајућем менију профила има и опцију да забележи датум и време када је заузет. Додатно, корисник који има улогу ***admin*** у навигационом менију може видети и дугме за преусмеравање на страницу за управљање корисницима или на страницу за управљање пословима.



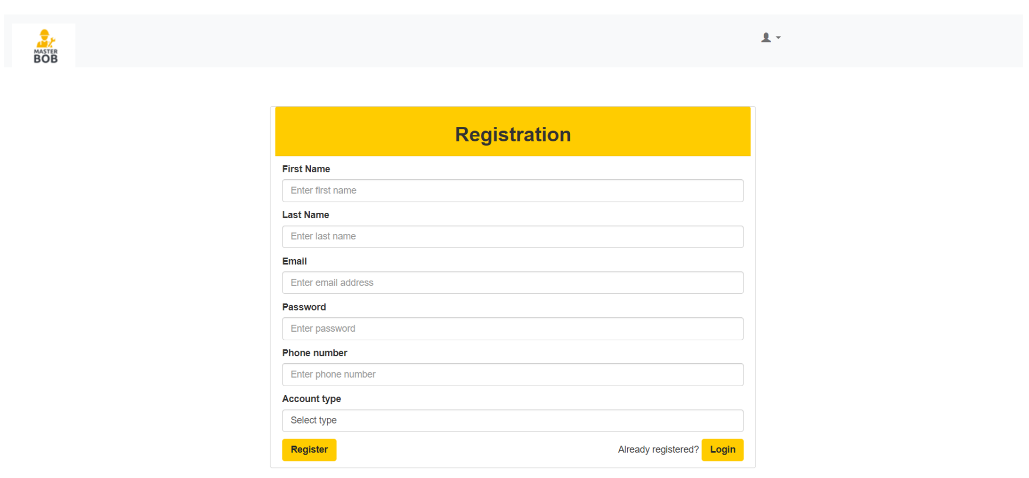
Слика 3.2: Навигациони мени

Страница за регистрацију корисника

Страница за регистрацију (слика 3.4) омогућава корисницима да креирају налог уносом личних података попут имена, презимена, мејл адресе, лозинке, броја телефона. Поред основних информација, корисници имају могућност да изаберу тип налога и унесу додатне податке као што су адреса, поштански број и категорија посла, што је неопходно за кориснике са улогом ***contractor***.



Слика 3.3: Профилни падајући мени



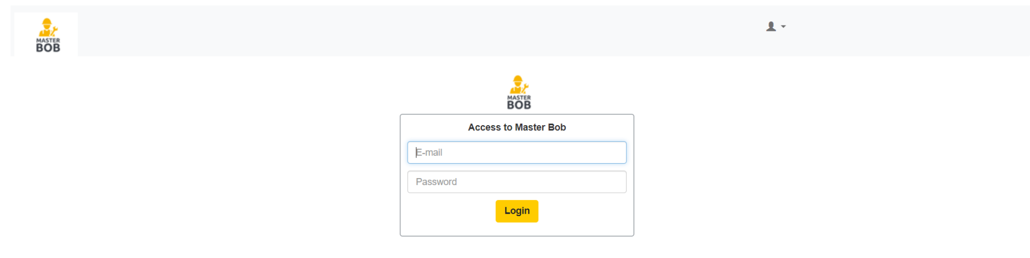
Слика 3.4: Страница за регистрацију корисника

Страница за пријаву корисника

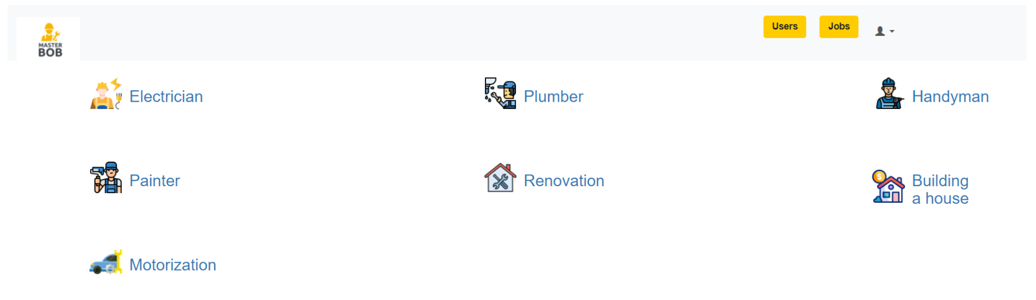
Страница за пријаву (слика 3.5) омогућава корисницима да се упуте на свој налог уношењем корисничког имена и лозинке. Овде корисници могу приступити свим функцијама и информацијама на свом налогу, у зависности од улоге коју имају у апликацији.

Почетна страница

На почетној страници (слика 3.6) су излистане категорије послова доступне у апликацији. Додатно, корисници који имају улогу ***admin*** могу управљати корисничким налозима и пословима у апликацији.



Слика 3.5: Страница за пријаву корисника



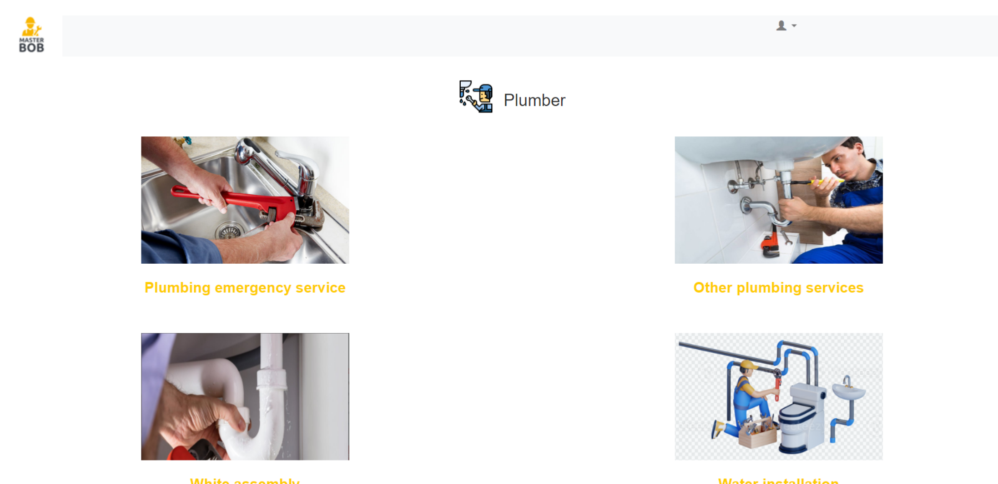
Слика 3.6: Почетна страница

Страница са пословима изабране категорије

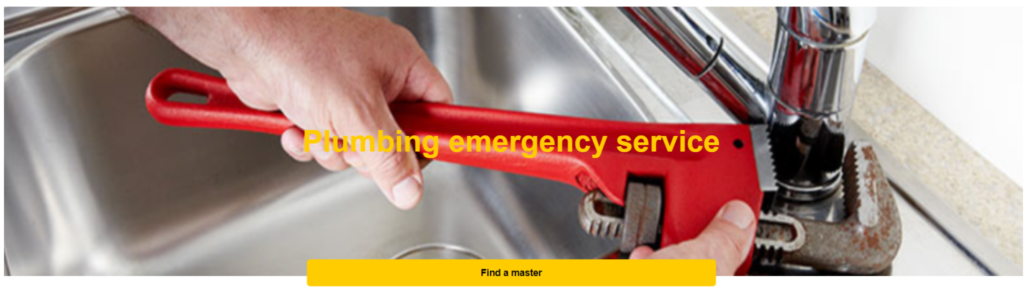
На страници са пословима изабране категорије (слика 3.7) налазе се излистани послови из те категорије који су доступни у апликацији. Кликом на одређен посао приказује се страница са детаљним описом тог посла.

Страница са описом изабраног посла

На страници са описом изабраног посла (слика 3.8) може се наћи детаљан опис тог посла, као и оквирне цене. Кликом на дугме ***Find a contractor*** приказује се страница која садржи форму за креирање услуге за одабрани посао.



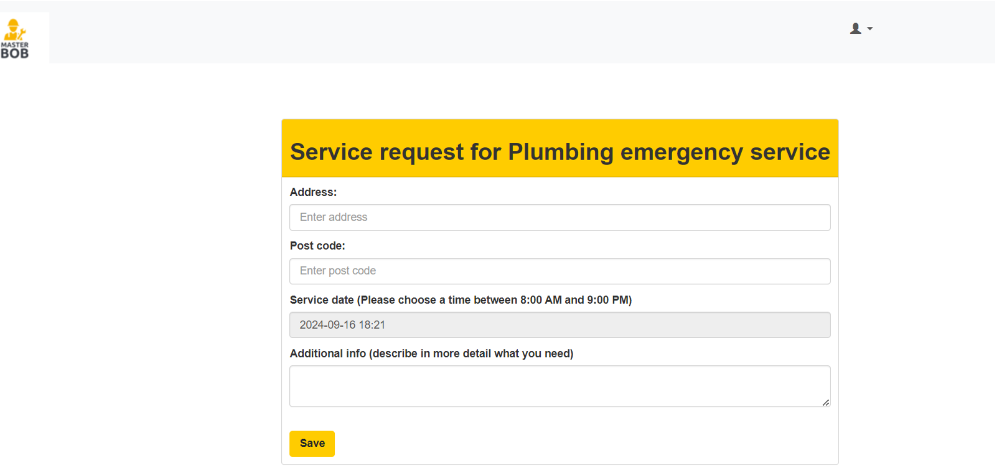
Слика 3.7: Страница са пословима изабране категорије



Слика 3.8: Страница са описом изабраног посла

Страница за креирање захтева

На страници за креирање захтева (слика 3.9) налази се форма коју корисник треба да попуни информацијама као што су адреса, поштански број, као и датум када жели да се посао обави. Опционо, корисник може навести додатне информације везано за изабрани посао. Кликом на дугме ***Save*** креира се захтев за услугу мајстора.



Слика 3.9: Страница за креирање захтева

Страница за бележење заузетости мајстора

На страници за бележење заузетости мајстора (слика 3.10) налази се форма коју корисник, који има улогу ***contractor***, треба да попуни како би забележио датуме када је заузет. Од обавезних поља у форми, корисник треба да попуни датуме од кад до кад је заузет, а опционо може навести посао који треба да одради, адресу на којој посао треба да буде одрађен, као и додатне информације везане за посао. Кликом на дугме ***Save*** бележи се заузетост мајстора.

Страница за приказ захтева

Уколико корисник има улогу ***customer***, на страници за приказ захтева (слика 3.11) излиставају се сви захтеви које је корисник креирао. Уколико корисник има улогу ***contractor***, на страници за приказ захтева 3.11 излиставају се сви захтеви за које је задужен тај корисник, тј. мајстор. У апликацији постоје три статуса захтева: ***PENDING***, ***CONFIRMED*** и ***FINISHED***. Сваки од корисника може да обрише и ажурира захтев, уколико захтев није у статусу ***FINISHED***.

Страница за ажурирање захтева

Сваки корисник апликације може ажурирати захтев за услугу мајстора, с тим што корисник који има улогу ***customer*** може ажурирати датум и време

Book a date

Job

Electrical installations

Address:

Enter address

Service date begin

2024-10-06 19:05

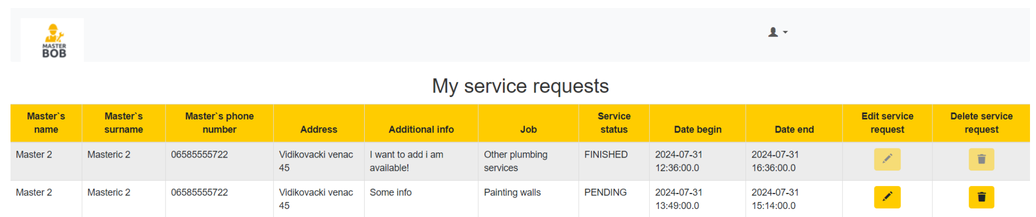
Service date end

2024-10-06 19:05

Additional info (describe in more detail what you need)

Save

Слика 3.10: Страница за бележење заузетости мајстора



MASTER BOB

My service requests

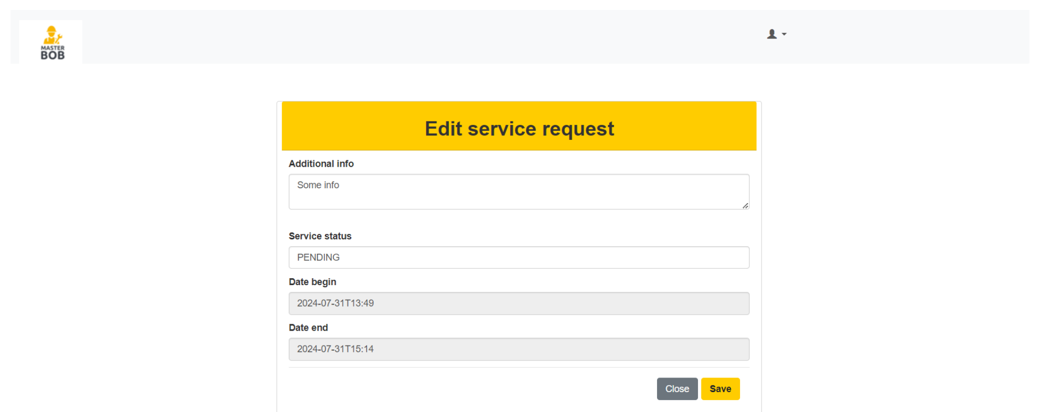
Master`s name	Master`s surname	Master`s phone number	Address	Additional info	Job	Service status	Date begin	Date end	Edit service request	Delete service request
Master 2	Masteric 2	06585555722	Vidikovacki venac 45	I want to add i am available!	Other plumbing services	FINISHED	2024-07-31 12:36:00.0	2024-07-31 16:36:00.0		
Master 2	Masteric 2	06585555722	Vidikovacki venac 45	Some info	Painting walls	PENDING	2024-07-31 13:49:00.0	2024-07-31 15:14:00.0		

Слика 3.11: Страница за приказ захтева

када жели да се обави посао и додатне информације, а корисник који има улогу ***contractor*** може ажурирати и статус захтева и датум и време када ће тражени посао бити обављен. На слици 3.12 је приказана страница за ажурирање захтева уколико је у апликацији пријављен корисник који има улогу ***contractor***.

Страница за ажурирање личних података на налогу

Сваки корисник апликације може ажурирати своје податке на налогу. Корисник који има улогу ***customer*** или ***admin*** може ажурирати име, презиме,



Слика 3.12: Страница за ажурирање захтева

мејл адресу, број мобилног телефона и лозинку, док корисник који има улогу ***contractor*** може ажурирати и категорију посла за који је задужен, адресу и поштански број. На слици 3.13 се може видети страница за ажурирање података на налогу која се приказује кориснику који има улогу ***customer*** или ***admin***.

Страница за управљање корисницима

Страница за управљање корисницима (слика 3.14) доступна је кориснику који има улогу ***admin***, кликом на дугме ***Users*** из навигационог менија. На овој страници доступне су информације о свим регистрованим корисницима апликације. Админи могу да бришу налог корисника или да га омогуће/онемогуће.

Страница за управљање пословима

Страница за управљање пословима (слика 3.15) доступна је кориснику који има улогу ***admin***, кликом на дугме ***Jobs*** из навигационог менија. На овој страници се могу видети информације о свим пословима доступним у апликацији. Админи могу да ажурирају податке о пословима, да их бришу, додају нове послове, категорије послова, као и да виде категорије послова доступне у апликацији.

Andrijana

First Name

Andrijana

Last Name

Bosiljcic

Email

andrijanab2000@gmail.com

Old password

Enter old password

New password

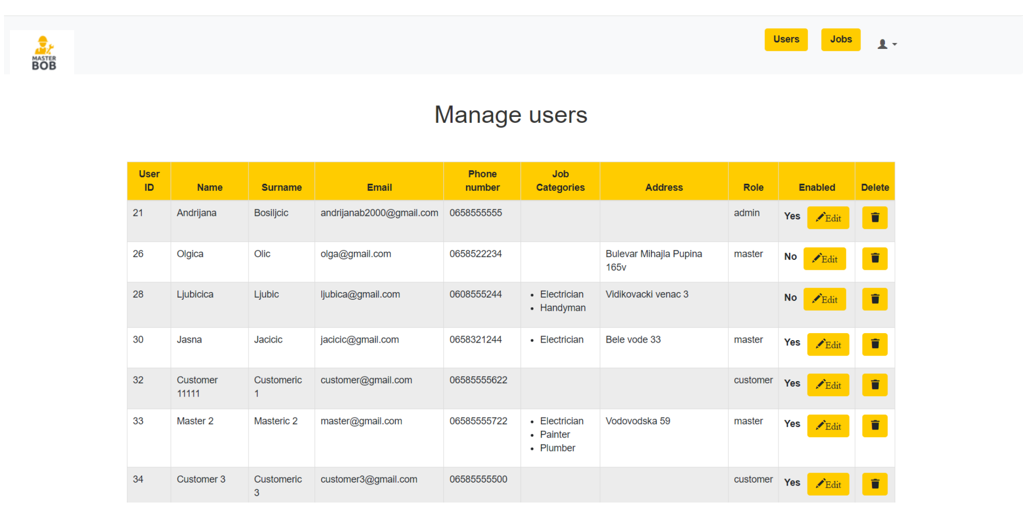
Enter new password

Phone number

0658555555

Save

Слика 3.13: Страница за ажурирање личних података на налогу

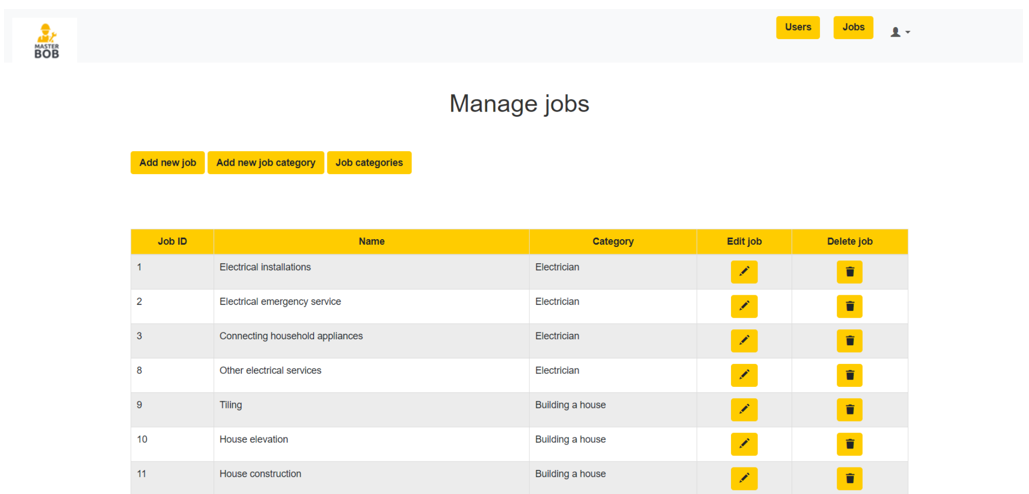


User ID	Name	Surname	Email	Phone number	Job Categories	Address	Role	Enabled	Delete
21	Andrijana	Bosiljcic	andrijanab2000@gmail.com	0658555555			admin	Yes Edit	
26	Olgica	Olic	olga@gmail.com	0658522234		Bulevar Mihajla Pupina 165v	master	No Edit	
28	Ljubicica	Ljubic	ljubica@gmail.com	0608555244	• Electrician • Handyman	Vidikovacki venac 3		No Edit	
30	Jasna	Jacicic	jacicic@gmail.com	0658321244	• Electrician	Bele vode 33	master	Yes Edit	
32	Customer 11111	Customeric 1	customer@gmail.com	06585555622			customer	Yes Edit	
33	Master 2	Masteric 2	master@gmail.com	06585555722	• Electrician • Painter • Plumber	Vodovodska 59	master	Yes Edit	
34	Customer 3	Customeric 3	customer3@gmail.com	06585555500			customer	Yes Edit	

Слика 3.14: Страница за управљање корисницима

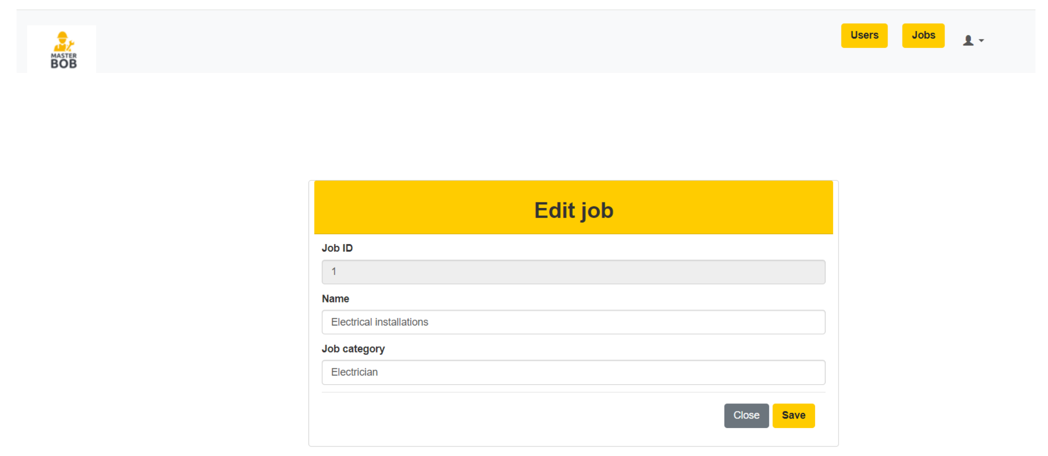
Страница за ажурирање података о послу

Страница за ажурирање података о послу (слика 3.16) доступна је кориснику који има улогу ***admin***, кликом на иконицу пенкала у колони ***Edit job***



Слика 3.15: Страница за управљање пословима

табеле која се налази на страници за управљање пословима. Админи могу да ажурирају име посла, као и категорију којој тај посао припада.



Слика 3.16: Страница за ажурирање података о послу

Приказ категорија послова

Кликом на дугме ***Job categories***, које се налази на страници за управљање пословима, приказује се табела (слика 3.17) у којој се налазе излистане категорије послова доступне у апликацији. Админи имају могућност брисања категорија послова.

Category ID	Category	Delete category
1	Electrician	
2	Plumber	
3	Handyman	
4	Painter	
5	Renovation	
6	Building a house	
7	Motorization	

Слика 3.17: Приказ категорија послова

Искачући прозор за креирање новог посла

Кликом на дугме ***Add new job***, које се налази на страници за управљање пословима, админу се појављује искачући прозор (слика 3.18) на ком се налази форма за креирање новог посла. Потребно је да се форма попуни следећим подацима: име посла, категорија којој посао припада, трајање посла у минутима, путања до слике посла. Кликом на дугме ***Save*** креираће се нови посао.

Искачући прозор за креирање нове категорије посла

Кликом на дугме ***Add new job category***, које се налази на страници за управљање пословима, админу се појављује искачући прозор (слика 3.19) на ком се налази форма за креирање нове категорије посла. Потребно је да се форма попуни следећим подацима: име категорије, путања до слике посла. Кликом на дугме ***Save*** креираће се нова категорија.

Add new job

Name

Enter job name

Job category

Select category

Duration

Enter job duration

Image url

Enter image url

Close Save

Слика 3.18: Искачући прозор за креирање новог посла

Add new job category

Category name

Enter category name

Icon url

Enter icon url

Close Save

Слика 3.19: Искачући прозор за креирање нове категорије посла

Глава 4

Закључак

У овом раду је развијена веб апликација која корисницима омогућава брзо проналажење мајстора у различитим секторима, прилагођено њиховим индивидуалним потребама, унапређује корисничко искуство и доприноси ефикаснијем повезивању корисника са квалификованим мајсторима у њиховој близини. Рад је показао како се могу имплементирати сложени системи за управљање информацијама коришћењем савремених алата и технологија. Иако су почетни циљеви апликације успешно остварени, постоји простор за даљи развој и унапређење, нарочито у домену корисничког интерфејса и додатних функционалности. Будуће надоградње могу подразумевати проширење функционалности апликације у складу са новим потребама корисника. Неке од планираних будућих функционалности укључују: давање рецензија мајсторима, њихов преглед, давање одређених погодности верним корисницима апликације, награђивање мајстора са најбољим рецензијама на месечном нивоу.

Библиографија

[1] Chris Schaefer Clarence Ho Iuliana Cosmina, Rob Harrop. *Pro Spring 5 An In-Depth Guide to the Spring Framework and Its Tools*. Apress, fifth edition, 2017.

[2] Milos Radovanović Mirjana Ivanović, Zoran Budimac and Dejan Mitrović. *Objektno-orijentisano programiranje i programski jezik Java*. Sigra Star d.o.o. Beograd, 2016.

[3] Thymeleaf project team. Using Thymeleaf, 2023. on-line at: `https://www.thymeleaf.org/doc/tutorials/3.1/usingthymeleaf.pdf`, accessed: 15.09.2024.

[4] José Miguel Samper. Thymeleaf + Spring Security integration basics. on-line at: `https://www.thymeleaf.org/doc/articles/springsecurity.html`, accessed: 15.09.2024.

[5] Herbert Schildt. *Java*. McGraw-Hill Education, ninth edition, 2014.

[6] Catalin Tudose. *Java Persistence with Spring Data and Hibernate*. Manning, 2023.

[7] Greg L. Turnquist. *Learning Spring Boot 3.0*. Packt Publishing Ltd., 35 Livery Street, third edition, 2022.

Биографија аутора

Андријана Босиљчић је рођена 05. августа 2000. године у Београду. Основну школу и природно-математички смер гимназије завршила је у Београду. Након завршене гимназије, уписала је Математички факултет у Београду на смеру Информатика. Основне студије је завршила у року, након чега је уписала мастер студије на Математичком факултету у Београду на смеру Информатика. Након успешно завршене праксе у NLB Komercijalnoj banci у јулу 2023. године, запослила се у истој фирми као Јуниор софтвер инжењер, где и даље ради.