Unos i registracija korisnika pomoću Spring framework API-ja

Cecelja, Marko

SVEUČILIŠTE JOSIPA JURJA STORSSMAYERA U OSIJEKU
FAKULTET ELEKTROTEHNIKE, RAČUNARSTVA I INFORMACIJSKIH TEHNOLOGIJA

Sveučilišni studij

UNOS I REGISTRACIJA KORISNIKA POMOĆU SPRING FRAMEWORK API-JA

Završni rad

Marko Cecelja

Osijek, 2020.

FERIT
FAKULTET ELEKTROTEHNIKE, RAČUNARSTVA I INFORMACIJSKIH TEHNOLOGIJA OSIJEK

Obrazac Z1P - Obrazac za ocjenu završnog rada na preddiplomskom sveučilišnom studiju

Osijek, 08.07.2020.

Odboru za završne i diplomske ispite

Prijedlog ocjene završnog rada na preddiplomskom sveučilišnom studiju

Ime i prezime studenta:	Marko Cecelja
Studij, smjer:	Preddiplomski sveučilišni studij Računarstvo
Mat. br. studenta, godina upisa:	R4044, 24.09.2019.
OIB studenta:	61140051417
Mentor:	Izv. prof. dr. sc. Ivica Lukić
Sumentor:	
Sumentor iz tvrtke:	
Naslov završnog rada:	Unos i registracija korisnika pomoću Spring framework API-ja
Znanstvena grana rada:	**Programsko inženjerstvo (zn. polje računarstvo)**
Predložena ocjena završnog rada:	Izvrstan (5)
Kratko obrazloženje ocjene prema Kriterijima za ocjenjivanje završnih i diplomskih radova:	Primjena znanja stečenih na fakultetu: 3 bod/boda Postignuti rezultati u odnosu na složenost zadatka: 3 bod/boda Jasnoća pismenog izražavanja: 3 bod/boda Razina samostalnosti: 3 razina
Datum prijedloga ocjene mentora:	08.07.2020.
Datum potvrde ocjene Odbora:	15.07.2020.

Potpis mentora za predaju konačne verzije rada u Studentsku službu pri završetku studija:	Potpis:
	Datum:

FERIT
FAKULTET ELEKTROTEHNIKE, RAČUNARSTVA
I INFORMACIJSKIH TEHNOLOGIJA OSIJEK

IZJAVA O ORIGINALNOSTI RADA

Osijek, 19.07.2020.

Ime i prezime studenta:	Marko Cecelja
Studij:	Preddiplomski sveučilišni studij Računarstvo
Mat. br. studenta, godina upisa:	R4044, 24.09.2019.
Turnitin podudaranje [%]:	9

Ovom izjavom izjavljujem da je rad pod nazivom**: Unos i registracija korisnika pomoću Spring framework API-ja**

izrađen pod vodstvom mentora Izv. prof. dr. sc. Ivica Lukić

i sumentora

moj vlastiti rad i prema mom najboljem znanju ne sadrži prethodno objavljene ili neobjavljene pisane materijale drugih osoba, osim onih koji su izričito priznati navođenjem literature i drugih izvora informacija.
Izjavljujem da je intelektualni sadržaj navedenog rada proizvod mog vlastitog rada, osim u onom dijelu za koji mi je bila potrebna pomoć mentora, sumentora i drugih osoba, a što je izričito navedeno u radu.

Potpis studenta:

SADRŽAJ

1. UVOD

U ovome radu prikazuje se postupak izrade aplikacijskog programskog sučelja (engl. *Application programming interface*, API) za unos i registraciju korisnika u neki sustav. Svrha ovog API-ja je kompleksni proces registracije velikog broj korisnika svesti na jednostavan unos preko excel datoteke. Cijeli sustav izrađen je pomoću Java programskog jezika i Spring razvojnog okvira (engl. framework). Biti će prikazan proces izrade objektne baze podataka, njeno mapiranje pomoću hibernate alata i kreiranje testnih podataka. Definirat će se uloga servisa pri obavljanju transakcija i upotreba objekta za prijenos podataka (engl. *Dana transfer object*, DTO). Nadalje, opisati će se način upravljanja iznimkama koje mogu nastati za vrijeme izvođenja neke transakcije. Definirati će se HTTP zahtjevi te način na koji ih pisati. Kako na određeni zahtjev dobiti odgovarajući odgovor te njegova serializacija u JSON objekte. Prikazati će se koraci koje je potrebno poduzeti u svrhu sigurnosti sustava te ograničavanja korisnika prilikom rada sa sustavom. Na kraju, opisati će se proces testiranja, te dati primjer dvije konkretne vrste testiranja.

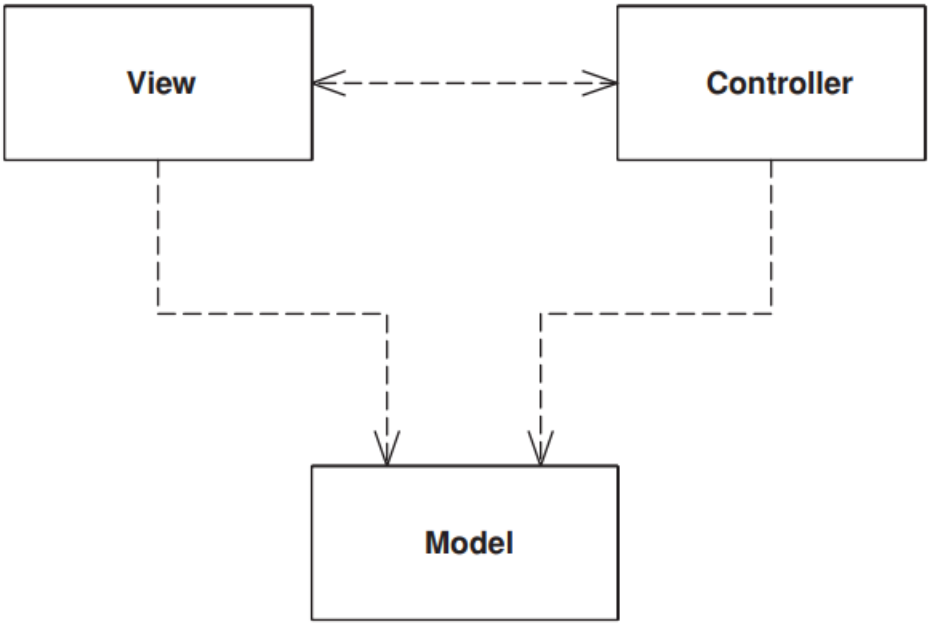
1.1. Zadatak završnog rada

Potrebno je napraviti sustav koji će omogućiti dodavanje excel datoteke s korisnicima koji trebaju biti uneseni u sustav. Iz datoteke, korisnici se pomoću HTTP zahtjeva šalju servisu koji kao odgovor vraća popis ispravnih korisnika, te popis neispravnih uz obrazloženje zašto ne mogu biti stvoreni. Na kraju se popis ispravnih korisnika prosljeđuje zasebnom API-ju koji vrši njihovu registraciju u sustav.

2. PROGRAMSKA ARHITEKTURA I ALATI

2.1. MVC arhitektura

Model-Pogled-Upravitelj (eng. *Model-View-Controller*, MVC) je oblikovni obrazac koji se najčešće koristi pri razvoju web aplikacija. Razlog tome leži u dobro organiziranoj komunikaciji između klijenta i servera što ovakva podjela donosi. Započeo je kao razvojni okvir kojeg je razvio Trygve Reenskaug za Smalltalk platformu 1970. godine [1, str. 330].



Slika 2.1. *Grafički prikaz MVC obrasca [1, str. 330]*

Model se može definirati kao nevidljivi objekt koji sadrži sve informacije i pravila ponašanja neke domene [1, str. 330]. Domena, u ovome slučaju, predstavlja relaciju u bazi podataka stoga su u modelu definirani svi njezini atributi i veze koje sadržava s drugim relacijama.

Pogled predstavlja sve što sadržava korisničko sučelje (engl. *User interface*, UI), od najobičnijih oznaka pa sve do različitih alata s kojima korisnik ima interakciju. Važno je naglasiti kako ono predstavlja samo prikaz informacija o modelu, sve promijene u tim informacija obavlja treća grana ove strukture [1, str. 330]. Budući da se web aplikacije dijele na klijentsku i serversku stranu te uzimajući u obzir da se zadatak rada obavlja na serverskoj strani, dok se pogled realizira na klijentskoj, on neće biti daljnje obrađivana u ovome radu.

Kako je već ranije spomenuto, upravitelj je zadužen za obavljanje svih promjena na nekom modelu. Predstavlja poslovnu logiku sustava baziranog na ovoj arhitekturi. U ovome radu upravitelj sačinjava skup srodnih API-ja koji na određeni HTTP zahtjev pozivaju određeni servis koji obavljaju neku transakciju, te vraćaju odgovor koji pogled prikazuje.

Kod ove arhitekture najvažnije je razdvojiti pogled od modela. Ukoliko se radi o vrlo jednostavnim sustavima, u kojima model ne posjeduje vlastito ponašanje, ne mora doći do razdvajanja. To je ujedno i jedini takav slučaj. Čim model posjeduje neku logiku iza sebe potrebno ih je razdvojiti [1, str. 332]. Razdvajanje pogleda i upravitelja je manje važno no ukoliko se projekt dijeli na *front end* i *back end*, kao što je slučaj u ovome radu, dobra praksa ih je razdvojiti.

2.2. Java programski jezik

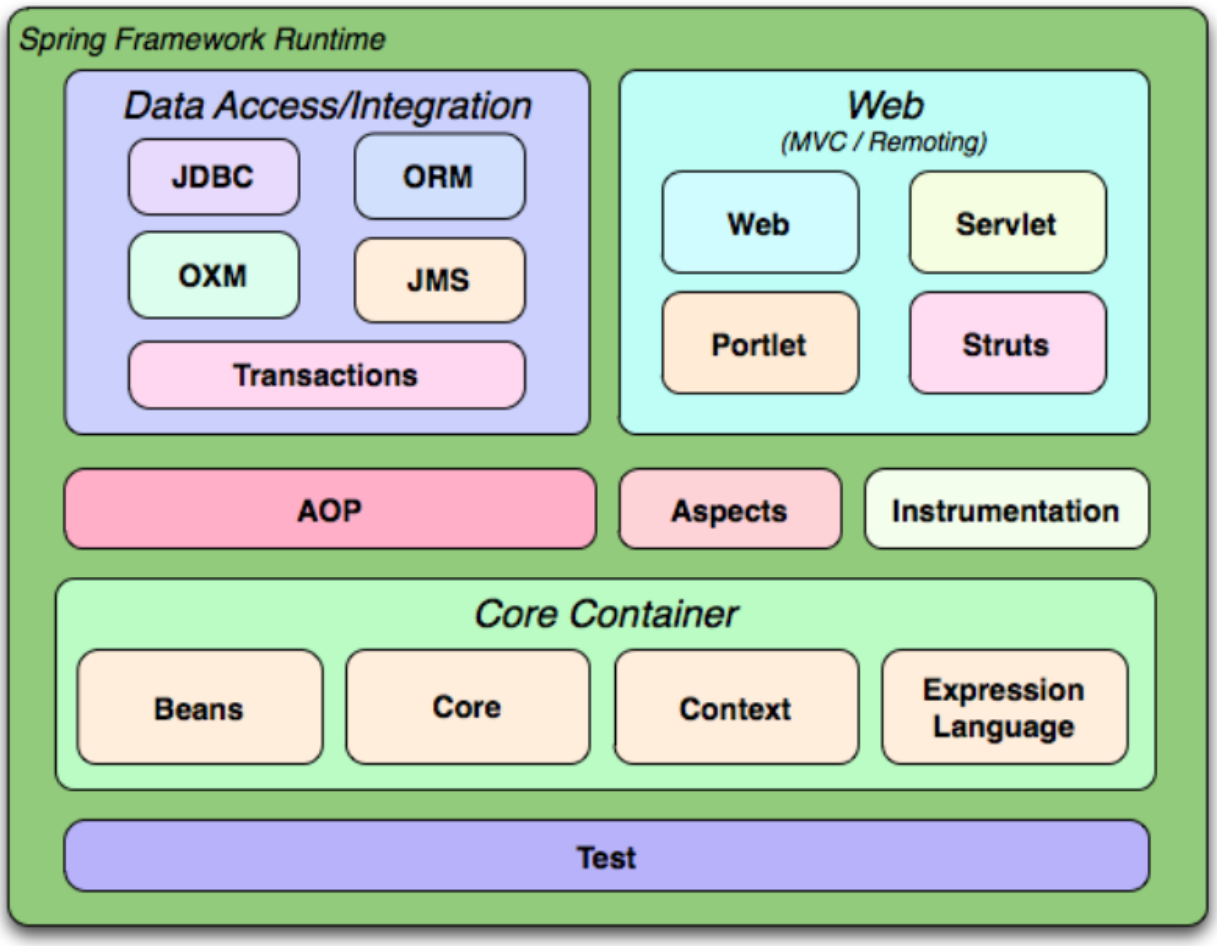
Java je objektno orijentirani programski jezik, što znači da se programi grade od klasa iz kojih se zatim mogu kreirati njihove instance, odnosno objekti. Klasu se može promatrati kao nacrt nekog objekta. Ona sadrži sve informacije o njemu, kao što su dijelovi od kojih je izgrađen te koje operacije se nad tim objektom mogu izvršavati. Ono što je specifično za ovaj programski jezik su anotacije. Prema [2] anotacija se može definirati kao posebni oblik sučelja koji se označuje @ oznakom prije ključne riječi *interface*. Njihova svrha je olakšati pisanje koda, što se može vidjeti upotrebom razvojnog okvira kod kojeg su one učestalije, kao što je Spring. Projekt Lombok, o kojem će biti više govora u nastavku, također uvodi veliki broj anotacija koje su znatno skratile pisanje određenih dijelova koda.

2.2.1. Projekt Lombok

Projekt Lombok je biblioteka za Java programski jezik kojem je glavna svrha smanjiti pisanje koda. To je postignuto na način da komponente, koje se pojavljuju u velikoj većini klasa kao što su metode za postavljanje i dohvaćanje te konstruktori, uvode anotacijama. Prednost toga se najbolje može vidjeti kod klasa koje predstavljaju relaciju u bazi podataka. Ukoliko ta relacija ima veliki broj atributa dosta vremena bi se potrošilo na pisanje pojedinih metoda za postavljanje i dohvaćanje za svaki taj atribut. Upotrebom Lombok anotacije oni se kreiraju automatski te se ne mora voditi računa o tome da se pri dodavanju novog atributa stvore njegove pripadajuće metode. Također, jedna korisna anotacija koju Lombok uvodi je i implementacija oblikovnog obrasca graditelj u neku klasu. Pomoću ove anotacije, objekt te klase može se kreirati korak po korak i samo s potrebnim atributima.

2.3. Spring razvojni okvir

Prema [3, str. 2], Spring je platforma koja pruža široku infrastrukturnu podršku za razvoj Java aplikacija. Omogućuje izradu aplikacije upotrebom jednostavnih Java objekta (engl. *Plain old Java objects,* POJO) i obavljanje operacija nad tim objektima upotrebom servisa.



Slika 2.2. *Građa Spring razvojnog okvira [3, str. 3]*

Prema slici možemo vidjeti kako se Spring razvojni okvir sastoji od rutine za pohranjivanje i dohvaćanje podataka iz baze te rutine za obavljanje transakcija nad tim podacima. Web dio sadrži implementaciju MVC obrasca. Jezgra pruža neizostavni dio ovog razvojnog okvira koji se temelji na *Inversion of Controll* i *Dependency Injection* [3, str. 3]. Ova dva mehanizma su usko povezani. Klasa nikada ne bi trebale ovisiti o konkretnoj implementaciji druge klase jer u tom slučaju promjena jedne može povlačiti promjenu druge. Trebala bi ovisiti o apstrakcije jer se one rijetko mijenjaju. *Dependency Injection* razdvaja

proces stvaranja od uporabe koji se prebacuje na posebni mehanizam, *Inversion of Controll*. Objekti, koje ovaj mehanizam stvara, nazivaju se *Beans*.

3. MODELI, SERVISI I UPRAVITELJI

Kako je već ranije spomenuto, model predstavlja bazu podataka. U njemu se definiraju domene koje će predstavljati pojedine relacije unutar baze podataka te repozitoriji koji će pohranjivati podatke i vršiti upite nad bazom.

Servisi su klase namijenjene za obavljanje operacija nad objektima. Sastoje se od sučelja u kojem su definirane metode servisa, te od njegove implementacije. Kako bi se označilo da je neka klasa servis, potrebno je iznad njenoga imena upotrijebiti *Service* anotaciju.

Upravitelj je klasa označena *Controller* anotacijom unutar koje su definirani API-ji koji pozivaju servise.

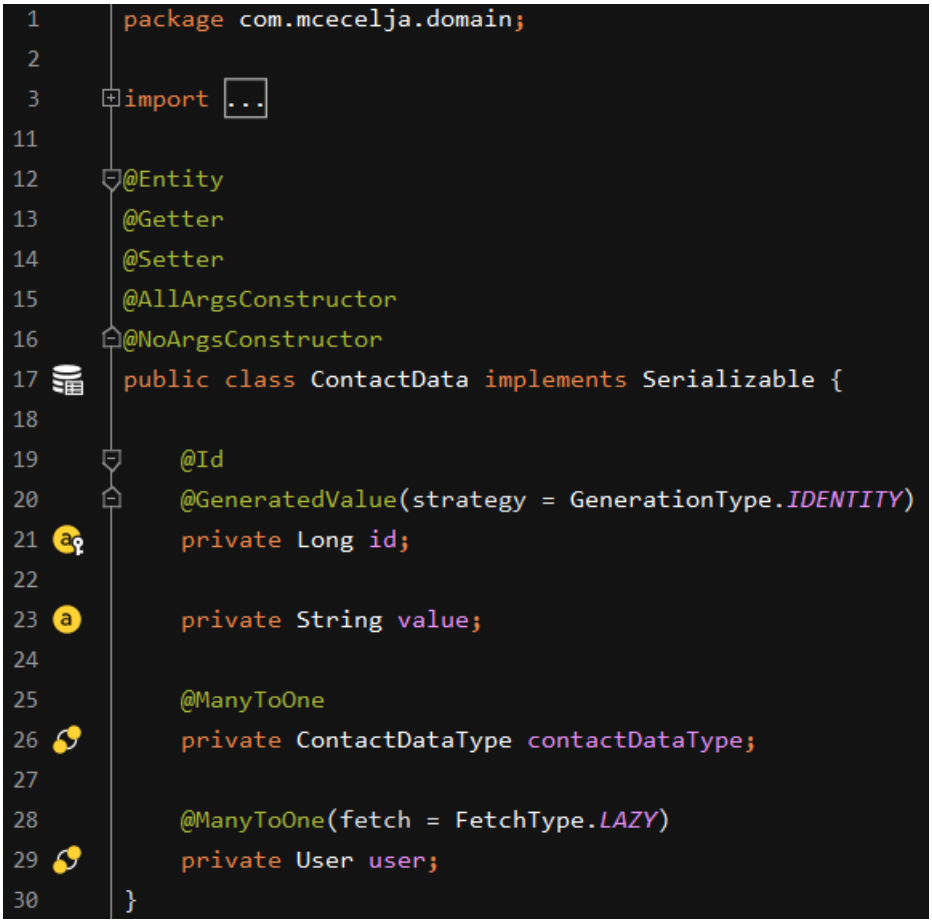
3.1.Domenske klase i hibernate

Svaka domenska klasa označava se anotacijom *Entity* iznad njenog imena. Ova klasa sadrži samo atribute koji predstavljaju stupce u našoj tablici te metode za njihovo postavljanje i dohvaćanje što objekte takvih klasa čini POJO objektima. Ono što mora sadržavati svaka klasa označena s *Entity* je podatak koji predstavlja primarni ključ. On se označuje anotacijom *Id*. Poželjno je definirati strategija generiranja ovakvog podatka. Ostali atributi mogu biti primitivnog tipa ili reference nekih drugih *Entity* klasa. Ukoliko se radi o referencama potrebno je definirati vrstu veze između te dvije klase.

Veza je podatak koji definira ovisnost između dva objekta. Ona može biti jedan naprema jedan (engl. *One-To-One*), jedan naprema više (engl. *One-To-Many*), više naprema jedan (*Many-To-One*) i više naprema više (engl. *Many-To-Many*). Veza jedan naprema jedan govori da objekt jedne klase može ovisiti isključivo o jednom objektu druge klase. Više naprema jedan i više naprema više govore kako objekt jedne klase ovisi o više objekata druge klase. Često dolaze u paru, a veza jedan naprema više mora postojati nad kolekcijom. Ukoliko se radi o relaciji više naprema više potrebno je pomoću *JoinTable* anotacije definirati veznu tablicu koja će sadržavati primarne ključeve obje relacije. Još jedna opcija je definirati vezni entitet na koji se s jedne strane entitet povezuje relacijom više naprema jedan, a uzvraća se vezom jedan naprema više što na kraju rezultira vezom više naprema više. Ovim pristupom, anotacija *JoinTable* nije potrebna ali treba se opreznu pristupiti prilikom mapiranja.

Unutar ovih anotacija moguće je definirati i ostala ponašanja u bazi. *Cascade* pruža logiku promijene objekata, odnosno kako se mijenja jedan objekt promjenom drugoga. *Fetch* definira vrstu dohvaćanja iz baze. Ukoliko se radi o lijenom dohvaćanju (engl. *Lazy*), parametri objekta se dohvaćaju isključivo kada se nad njima obavlja neka transakcija, dok se kod

nestrpljivog (engl. *Eager*) dohvaćaju uvijek što u nekim slučajevima može prouzročiti usporavanje sustava. *MappedBy* služi za mapiranje dvije relacije, odnosno dodavanje stranog ključa. Postavljanjem *orphanRemoval* na *true* prilikom brisanja roditeljskog objekta, biti će obrisan i njegov dijete objekt.



```
1  package com.mcecelja.domain;
2
3  import ...
11
12 @Entity
13 @Getter
14 @Setter
15 @AllArgsConstructor
16 @NoArgsConstructor
17 public class ContactData implements Serializable {
18
19     @Id
20     @GeneratedValue(strategy = GenerationType.IDENTITY)
21     private Long id;
22
23     private String value;
24
25     @ManyToOne
26     private ContactDataType contactDataType;
27
28     @ManyToOne(fetch = FetchType.LAZY)
29     private User user;
30 }
```

Slika 3.1. *Primjer domenske klase*

Na slici 3.1. prikazana je jedna domenska klasa iz ovoga rada. Implementacija *Serializable* sučelja omogućila je lakše mapiranje ovih objekata i njihovo pohranjivanje u bazu podataka. Ono što se može primijetiti je da objekti drugih domenskih klasa imaju definiranu vezu između njih. Tako na primjer jedan kontakt podatak može pripadati samo jednom korisniku, dok jedan korisnik može imati više kontakt podataka što je definirano više naprema jedan anotacijom na strani kontakt podatka. Osim ove, u ovome radu definirana su još neke relacije. *Address* sadrži podatke o adresi korisnika. Podaci za autentifikaciju nalaze se unutar *Login* relacije. Definirano je i nekoliko šifrarnika nad kojima se ne mogu vršiti transakcije nego služe samo za detaljniji opis objekta. Prema tome, šifrarnik *Role* definira koju ulogu korisnik ima u sustavu, *ContactDataType* pruža uvid u tip kontakt podatka (E-mail ili

telefonski broj), a *Subject* definira predmete. Relacija *User* sadrži sve ove podatke i definira korisnike sustava.

Prema [3, str. 95] Hibernate je alat zadužen za mapiranje Java klasa u SQL relacije. Prolaskom kroz projekt, detektirati će sve klase označene s *Entity* i na temelju njih stvoriti relacije u bazi podataka. Ukoliko želimo te relacije popuniti testnim podacima, može se stvoriti SQL (engl. *Structured Query Language*) datoteke unutar mape za resurse u koju se onda piše naredbe za unos podataka. Alat će detektirati ovu vrstu datoteke te popuniti relacije podacima. Baza podataka koju stvara ovaj alat naziva se H2 baza te joj se može pristupit preko sučelja za pisanje komadi ili preko grafičkog sučelja uključi li ga se u datoteci s postavkama.

SELECT * FROM CONTACT_DATA;

ID	VALUE	CONTACT_DATA_TYPE_ID	USER_ID
1	ikovac@mail.com	1	1
2	0996352147	2	1
3	0915321478	2	2

(3 rows, 8 ms)

Slika 3.2. *Primjer relacije nakon mapiranja domenske klase i popunjavanja podacima*

3.2. Repozitorij

Repozitoriji predstavlja skladište podataka. To je programsko sučelje koje nasljeđuje jedno od Spring sučelja u kojima se već nalazi definirana logika skladištenja. Dovoljno je prilikom nasljeđivanja navesti objekt koji će se skladištiti te tip njegovog primarnog ključa preko kojega će se dohvaćati. U njemu se nalaze metode zadužene za obavljanje CRUD (engl. *Create-Read-Update-Delete*) operacija. Standardizirano je da metode koje služe za dohvaćanje objekta, koje su ujedno i najčešće pisane metode unutar repozitorija, započinju riječju *get* ili *find* te upisivanjem te riječi razvojno okruženje je u mogućnosti nadopuniti ostatak fraze koja će implementirati traženu logiku. Ukoliko je željena operacija nad objektom kompliciranija, sučelje nije u mogućnosti pružiti gotovu implementaciju ponašanja. Kod takvih slučajeva potrebno je pomoću *Query* anotacije napisati SQL koji će se izvršiti traženi upit nad bazom podataka. Upite je moguće pisati i u Java programskom jeziku ali je tada potrebno pružiti kompletnu implementaciju sučelja koja će definirati ponašanje svih metoda navedenih u tome sučelju.

```java
package com.mcecelja.repositories.users;

import com.mcecelja.domain.User;
import org.springframework.data.repository.CrudRepository;

public interface UserRepository extends CrudRepository<User, Long> {

    User findUserById(Long userId);
}
```

Slika 3.3. Primjer repozitorija

3.3. Objekti za prijenos podataka i mapiranje

Objekti za prijenos podataka (engl. *Data Transfer Object*, DTO) predstavljaju jednostavni spremnik podataka koji služi za njihov prijenos između slojeva aplikacije. Oni su kopija stvarnog objekta koja može sadržavati sve ili samo neke njegove atribute [4, str. 109]. Ako korisnik želi stvoriti novi zapis u bazi, nije dobra praksa omogućiti da se to radi direktno jer se nikad ne zna što će biti poslano. Zbog toga korisnik će upravitelju, u tijelu zahtjeva, poslati objekt s podacima o onome što želi stvoriti, a upravitelj će zatim taj objekt proslijediti servisu koji će izvršiti sve potrebne provjere nad njim i ukoliko je sve u redu stvorit zapis u bazi. Nakon toga, servis ponovno stvara kopiju objekta i šalje ga upravitelju kao odgovor u obliku objekta za prijenos podataka. Kako bi sve od navedenog bilo moguće, potrebno je razviti dobro logiku stvaranja ovakvih objekata za što služi mapiranje.

```java
package com.mcecelja.common.dto.user;

import ...

@Getter
@Setter
@NoArgsConstructor
@AllArgsConstructor
public class CreateUserDTO extends UserDTO {

    @NotNull
    @NotEmpty
    @NotBlank
    private String username;

    @NotEmpty
    @NotNull
    @NotBlank
    private String password;

}
```

Slika 3.4. Primjer DTO klase

Na slici 3.4. može se vidjeti primjer DTO klase za stvaranje korisnika u projektiranom sustavu. Može se primijetit kako su unutar ove klase definirana i neka ograničenja kao što su da poslano korisničko ime ne smije sadržavati *null* vrijednost niti biti prazno. Ova DTO klasa nasljeđuje drugu takvu klasu, a samim time i sve njezine atribute.

Sučelje koje sadrži metode za mapiranje stvarnog objekta u njegovu DTO reprezentaciju i obrnuto naziva se *Mapper*. Ukoliko se atributi objekta podudaraju po imenu, sučelje će ih automatski povezati. Ako se razlikuju, moguće je pomoću anotacija navesti što se želi mapirati, a kod kompliciranije logike mapiranja moguće je unutar anotacije navesti ime metode koja će obavljati mapiranje toga atributa te je u tome slučaju potrebno pružiti i njezinu implementaciju. Implementaciju samih metoda za mapiranje nije potrebno pružati direktno. Prilikom pokretanja projekta ona će se stvoriti sama iščitavajući sadržaj anotacija.

```
1   package com.mcecelja.common.mappers;
2
3   import ...
11
12  @Mapper(componentModel = "spring",
13          unmappedTargetPolicy = ReportingPolicy.IGNORE, imports = Collectors.class)
14  public interface FileResourceMapper {
15
16      @Mappings({
17              @Mapping(target = "name", source = "filename"),
18              @Mapping(target = "mimeType", source = "mimeType"),
19              @Mapping(target = "size", source = "contentLength")
20      })
21      FileResourceDTO fileResourceToFileResourceDTO(FileResource entity);
22  }
```

Slika 3.5. *Primjer sučelja za mapiranje*

3.4. Upravljanje iznimkama

Kako je već i ranije spomenuto, nemoguće je predvidjeti što će korisnik poslati u zahtjevu, stoga, kako ne bi došlo do „pucanja“ programa, potrebno je osmisliti dobar sustav upravljanja iznimkama. U tu svrhu, unutar ovog projekta, kreiran je *enum* unutar kojeg su popisane očekivana neželjena ponašanja do kojih može doći. Zatim je kreirana klasa koja kao atribut sadrži taj *enum* i koja je zadužena za rukovanje iznimkama. Provjere podataka obavljaju se u servisima, te ako se ustanovi nepravilnost „baca“ se odgovarajuća iznimka što zaustavlja daljnje izvođenje transakcije te ne dolazi do nepredviđenog ponašanja. Prije nego se transakcija zaustavi poželjno je podatke o iznimci zapisati unutar log datoteke kako bi ostala zabilježena

i kako bi se mogla dalje analizirati, ukoliko je to potrebno. Iznimka se serijalizira u JSON format, te se vraća u odgovoru na zahtjev. Ukoliko se radi o servisima koju obavljaju neku transakciju nad bazom potrebno ih je označiti s anotacijom *Transactional* unutar koje se može postaviti poništavanje transakcije ako dođe do iznimke. Na taj način osigurava se stabilnost baze podataka.

3.5. Aplikacijsko programsko sučelje

Generalno govoreći, aplikacijsko programsko sučelje služi za pružanje skupa podataka i funkcija u svrhu olakšavanja interakcije između računalnih programa i njihovu razmjenu [5, str. 5]. Ova razmjena podataka obavlja se preko HTTP zahtjeva. Razlikujemo četiri glavna zahtjeva, a to su GET (za dohvaćanje), POST (za pohranu), PUT (za izmjenu) i DELETE (za brisanje). U tijelu zahtjeva nalaze se podaci napisani u jednom od jezika za njihovu serializaciju. Pomoću API-ja ti podaci se šalju nekom sustavu koji nakon njihove obrade vraća odgovor na zahtjev koji također sadržava serijalizirani oblik podataka. Za izradu ovoga rada bilo je potrebno definirati nekoliko takvih API-ja unutar različitih upravitelja.

```java
package com.mcecelja.rest;

import ...

@RestController
@RequestMapping("/api/users")
public class UserController {


    private final UserParserService userParserService;

    private final UserService userService;

    public UserController(UserParserService userParserService, UserService userService) {
        this.userParserService = userParserService;
        this.userService = userService;
    }

    @PostMapping("/parse")
    public ResponseEntity<ResponseMessage<UserFileParseResponseDTO>> getAllValidUsersFromFile
            (@Valid @RequestBody FileResourceDTO fileResourceDTO) throws UsersException {
        return ResponseEntity.ok(new ResponseMessage<>(userParserService.parseUsersFromFile(fileResourceDTO)));
    }

    @PostMapping("/import")
    public ResponseEntity<ResponseMessage<List<UserDTO>>> createImportedUsers
            (@Valid @RequestBody List<CreateUserDTO> createUsersDTO) throws UsersException {
        return ResponseEntity.ok(new ResponseMessage<>(userService.createImportedUsers(createUsersDTO)));
    }
}
```

Slika 3.6. *Primjer upravitelja unutar kojega su definirani API-ji*

Na slici 3.6. prikazan je upravitelj koji sadrži API za unos korisnika u sustav i jedan za njihovu registraciju. Unutar *RequestMapping* anotacije, koja se piše iznad naziva klase, nalazi se osnovni URL API-ja. S *PostMapping* anotacijom označeno je da se radi o POST zahtjevu te se unutar nje nalazi ostatak URL-a koji se nadovezuje na osnovni. Zahtjev za unos korisnika u svojem tijelu ima podatke o excel datoteci koja se šalje preko zasebnog zahtjeva. *Valid* anotacija služi kako bi se izvršila provjera ograničenja koja su postavljena u objektu za prijenos podataka. Izvršava se poziv servisa koji će izvršiti preostale provjere nad podacima i vratiti listu korisnika koji se mogu stvoriti i onih koji ne mogu uz obrazloženje. Podaci koji moraju biti prisutni su korisničko ime, lozinka, ime, prezime i kontakt podatak, e-mail ili telefonski broj. Podaci o adresi se također prikupljaju ali nisu obavezni. Ostale nepravilnosti koje mogu rezultirati s nemogućnošću stvaranja korisnika su:

1. Prekratka lozinka, duljina lozinke mora biti minimalno 8 znakova,
2. Željeno korisničko ime se već koristi,
3. Ukoliko je prisutan barem jedan podatak o adresi i svi ostali moraju biti,
4. Format e-mail adrese nije odgovarajući.

Lista korisnika koji mogu biti stvoreni zatim se šalje servisu za upravljanje korisnicima koji izvršava njihovu registraciju. Važno je napomenuti kako samo administrator može koristiti ovaj API, ukoliko korisnik s nekom drugom ulogom pokuša poslati zahtjev vratiti će se poruka o neovlaštenom zahtjevu.

4. SIGURNOST I TESTIRANJE

Sigurnost i testiranje predstavljaju jedne od neizostavnih čimbenika svake aplikacije. Autentifikacija i autorizacija, uz ovlasti rukovanja bazom podataka, su jedni od najvažnije aspekata sigurnosti. Autentifikacija predstavlja provjeru vjerodostojnosti jednoga korisnika koji se prijavljuje u neki sustav, dok autorizacija služi za provjeru ima li neki korisnik, koji je prošao autentifikaciju, ovlasti za obavljanje određenih akcija [6, str. 15 – 16]. U ovome radu za autentifikaciju korisnika korišten je *JSON Web Token* (JWT), a autorizacija se obavlja pomoću uloga u sustavu i bacanja iznimke o neovlaštenom zahtjevu. Radi dodatne sigurnosti, lozinka koju korisnik unosi za prijavu u sustav ne pohranjuje se u direktnom obliku u bazu podataka nego se kriptira *BCrypt* metodom čiji se *Bean* definira u konfiguracijskoj klasi.

```
60      @Bean
61      public BCryptPasswordEncoder encoder() {
62          return new BCryptPasswordEncoder();
63      }
```

Slika 4.1. *Bean za BCrypt*

Testiranje Pruža konkretan uvid u greške koje su nastalo tijekom razvoja, te njihovo efektivno uklanjanja. Postoji nekoliko vrsta testiranja, a najvažniji oblici su:

1. Jedinično (engl. *Unit*) testiranje
2. Integracijsko testiranje
3. *End-to-End* testiranje

Kod jediničnog testiranja, pojedine komponente sustava izoliraju se od okoline i njihove ovisnosti. Omogućuje pravovremeno detektiranje i uklanjanje pogrešaka prije nego se nađu u produkcijskoj okolini.

Integracijsko testiranje služi za testiranje suradnji pojedinih komponenti unutar sustava. Upravo zbog toga je pogodan za testiranje Spring API-ja te će biti detaljnije objašnjen u nastavku.

End-to-End se odnosi na testiranje ponašanja čitavog sustava i svih njegovih komponenti i podsustava. Dobar alat za ovu vrstu testiranja je Postman jer se unutar njega mogu definirati razni API-ji i simulirati pravo ponašanje sustava.

4.1. JSON WEB Token

JSON Web Token je sredstvo za predstavljanje zahtjeva koji se prenosi između dvije strane. JWT zahtjevi su kodirani kao JSON objekti te su digitalno potpisani korištenjem *JSON Web Signature* (JWS), a ukoliko se želi može ih se i kriptirati upotrebom *JSON Web Encryption* (JWE) [7].

Kako bi se JWT mogao koristiti u Spring aplikaciji potrebno je unutar konfiguracijske datoteke definirati tajnu koji koristi, vrijeme isteka, gdje ga se postavlja, te po želji i ostala svojstva. Ove postavke se učitaju u objekt klase korištenjem *ConfigurationProperties* anotacije.

```
8   @Configuration
9   @ConfigurationProperties("jwt")
10  @Getter
11  @Setter
12  public class JwtProperties {
13
14      String secret;
15
16      long expirationInMs;
17
18      String header;
19
20      String bearer;
21
22      String sid;
23  }
24
```

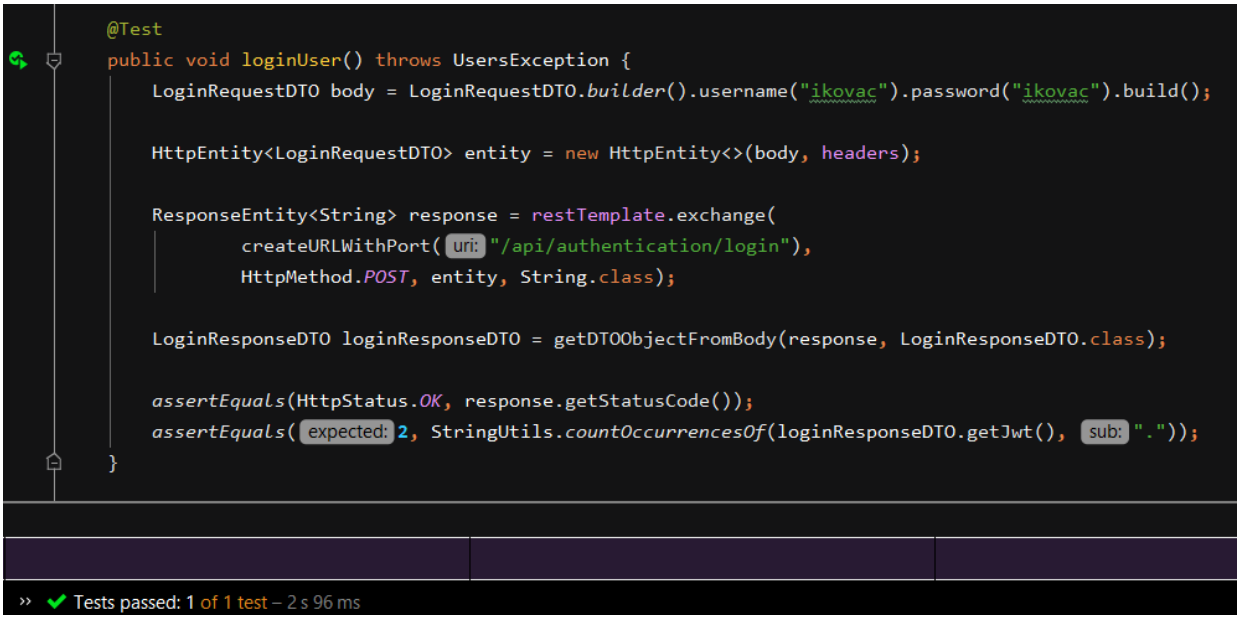
Slika 4.2. *JWT klasa unutar koje se nalaze svojstva*

Taj objekt se koristi unutar pomoćne klase koje služi za provjeru valjanosti tokena, izvlačenje tokena iz zaglavlja zahtjeva i slično. Potrebno je definirati klasu u kojoj će se nalaziti metoda za generiranje novog tokena. Kada korisnik pošalje zahtjev za prijavu u sustav, njegovi podaci se provjeravaju i ukoliko su ispravni stvara se token s kojim taj korisnik rukuje dok se ne odjavi iz sustava ili dok ne istekne, a ako podaci nisu ispravni vraća se poruka o neispravnim podacima. Ukoliko token istekne na svaki sljedeći zahtjev baca se poruka o istekloj sesiji te se je potrebno ponovno prijaviti u sustav. Upotreba tokena je vrlo korisna jer se preko njega točno zna koji korisnik šalje zahtjeve te se pomoću toga mogu definirati razne ovlasti za različite korisnike.

4.2. Integracijsko testiranje

Kako je već ranije spomenutu, integracijsko testiranje je vrlo korisno kada se želi provjeriti ponašanje određenog API-ja, njegove suradnje s bazom podataka, te vrijednosti koje vraća kao odgovor na različite zahtjeve. Postoje razni načini na koje se može implementirati ova vrsta testiranja u projekt, a u nastavku će biti objašnjeno kako je to realizirano u ovome radu.

Za početak, kreirana je jedna konfiguracijska datoteka unutar koje se mogu unositi svojstva koja su važna za testiranje. U ovome radu jedino takvo svojstvo bilo je lokacija za pohranjivanje datoteka. Zatim je bilo potrebno stvoriti jednu apstraktnu klasu koja služi za implementaciju tih svojstava, definiranje često korištenih metoda, stvaranje zaglavlja zahtjeva i slično. Ovu klasu zatim nasljeđuje sve konkretne klase koje služe za testiranje pojedinih dijelova sustava. Provedeno je testiranje tri različite stavke, a to su: autentifikacija korisnika, unos datoteke i registriranje korisnika. Test predstavlja običnu metodu, bez povratnog tipa, označenu s *Test* anotacijom. Ukoliko se unutar testa obavljaju radnje koje mogu utjecati na bazu podataka, tada ga je potrebno označiti s anotacijom *DirtiesContext* kako se po njegovom završetku promijene ne bi trajno pohranile u bazu podataka.



Slika 4.3. *Primjer metode za integracijsko testiranje*

Za autentifikaciju korisnika bilo je potrebno definirati tijelo zahtjeva unutar kojeg se unosi korisničko ime i lozinka korisnika kojeg se prijavljuje u sustav. Zatim se kreira odgovor na zahtjev na način da se unese URL i tip zahtjeva te što se očekuje kao odgovor. U ovome slučaju je to znakovni niz pošto se odgovor serijalizira u JSON. Nakon toga, potrebno je dobiti

objekt iz ovakvog odgovora. To se postiže pomoću jedne metode koja vrši deserijalizaciju i vraća objekt. Za kraj, potrebno je napisati dio koda koji provjerava jesu li dobivene vrijednosti jednake očekivanima te pokrenuti test. Jednaki postupak se provodi i za ostale stavke sustava.

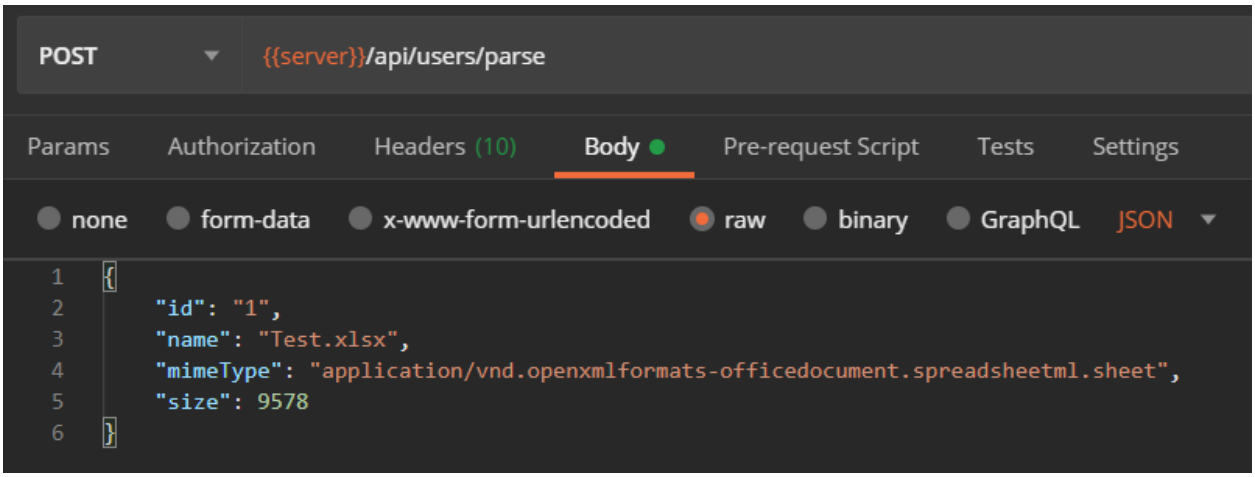
Potrebno je testirati i ponašanje sustava kod zahtjeva koji uzrokuju neku pogrešku. Ovakvi testovi se uglavnom sastoje od kreiranja zahtjeva za koji se očekuje pojavljivanje iznimke. Nakon što se dobije odgovor, provjerava se je li dobivena iznimka jednaka očekivanoj.

4.3. Postman

Ukoliko je potrebno provjeriti komunikaciju pojedinih dijelova sustava jednih s drugima potrebno je odgovor na jedan zahtjev poslati kao drugi zahtjev. Ovo se najjednostavnije može postići pomoću alata za testiranje API-ja, Postman.

Postman nudi razne mogućnosti, a jedna od njih je kreiranje grupa. Korisno je sve API-je koji su vezani za jedan projekt smjestiti unutar jedne grupe. U postavkama grupe moguće je navesti njezin opis, vrstu autorizacije, predefinirane skripte koje se pokreću prije svakog zahtjeva u grupi, testove koji se izvode nakon svakog zahtjeva iz grupe, te varijable. Unošenjem tokena nekoga korisnika pod opcijom autorizacije, taj korisnik se gleda kao da je prijavljen u sustav te izvodi sve zahtjeve te kolekcije. Korisno je pod varijable postaviti lokalni URL servera kako se on ne bi morao pisati za svaki zahtjev.

Nakon stvaranja određenog zahtjeva, potrebno je odabrati njegov tip te unijeti njegov URL. Ukoliko zahtjev sadrži nekakve parametre, oni se mogu pisati u samom URL-u ili pod opcijom parametri. Moguće je definirati autorizaciju za neki specifičan zahtjev ili postaviti opciju da se ona naslijedi od grupe u kojoj se nalazi. Pod opcijom zaglavlja se konfigurira zaglavlje zahtjeva. Tu je moguće definirati razne stvari specifične za neki zahtjev.



Slika 4.4. *Tijelo zahtjeva*

Ukoliko zahtjev sadrži tijelo, tada se u *Body* kartici definira kojeg je tipa (datoteka, tekstualni, graf…), te ako je tekstualni, u kojem jeziku je pisan (JSON, JavaScript, Html, XML ili običan tekst). Moguće je postaviti i predefinirane skripte i testove koji su specifični za taj zahtjev. Nakon što su ispunjeni svi navedeni dijelovi po želji, potrebno je pritisnuti gumb pošalji te se zahtjev šalje aplikaciji.

Za testiranje API-ja ovog sustava bilo je potrebno za početak obaviti kreirati autorizacijski token korisnika pomoću zahtjeva za prijavu u sustav, te njegovo postavljanje u grupu zahtjeva. Nakon toga, kreiran je zahtjev za unos datoteke u sustav. Datoteka se nalazi u tijelu zahtjeva, a kao odgovor se vraća apstraktni objekt koji predstavlja tu datoteku. Takav objekt se zatim šalje API-ju koji obavlja provjeru datoteke te iz nje izvlači korisnike za stvaranje. Na kraju se popis korisnika, koji mogu biti stvoreni, šalju unutar API-ja koji obavlja njihovu registraciju.

Slika 4.5. *Odgovor na zahtjev*

5. ZAKLJUČAK

Kroz ovaj rad pokazano je kako se na jednostavan način, upotrebom Spring razvojnog okvira, mogu razvijati aplikacijska programska sučelja nekih kompleksnih sustava i to upotrebom objektno orijentirane paradigme. Sve što se treba napraviti je definirati model baze podataka, repozitorije za skladištenje podataka, servise koji će obavljati određeno operacije na podacima te upravitelje koji će pozivati servise i vraćati neki rezultat.

Spring omogućuje razvoj objektno orijentiranih baza podataka pa samim time nije potrebno ili je potrebno vrlo osnovno znanje SQL jezika. Dovoljno je definirati POJO klase koje predstavljaju relacije i prepustiti Hibernate-u da obavi ostatak posla.

Unos velikog broja korisnika u nekim velikim sustavima može biti vrlo komplicirano. Kroz rad je pokazano kako se taj proces može olakšati omogućujući da se veliki broj korisnika unese preko vanjske datoteke u par klikova. Ovaj rad prikazuje jednu konkretnu implementaciju takvih sustavi ali ga se također može napraviti apstraktnim. Na taj način bilo bi ga moguće izvesti u biblioteku i koristiti u bilo kojim sustavima. Također se može proširiti dodavanjem *front end* kako bi sustav postao korisniku jednostavniji za korištenje.

LITERATURA

[1] M. Fowler, Patterns of Enterprise Application Architecture, Addison-Wesley Professional, 2002.

[2] K. Arnold, J. Gosling, D. Holmes, The Java™ Programming Language, Addison-Wesley Professional, 2002.

[3] R. Johnson, J. Hoeller, K. Donald, C. Sampaleanu, R. Harrop, T. Risberg, A. Arendsen, D. Davison, D. Kopylenko, M. Pollack, T. Templier, E. Vervaet, Portia Tung, B. Hale, A. Colyer, J. Lewis, C. Leau, M. Fisher, S. Brannen, R. Laddad, A. Poutsma, C. Beams, T. Abedrabbo, A. Clement, D. Syer, O.Gierke, R. Stoyanchev, P. Webb, Spring Framework Reference Documentation

[4] A. Hooshangi, Reducing Development Time Using Automated Data Transfer Object, International Journal of Science and Engineering Investigations, Vol. 1, str. 109 – 112, Veljača 2012.

[5] M. Masse, REST API Design Rulebook, O'Reilly Media, 2011.

[6] P. Mularien, Spring Security 3, Packt Publishing, 2010.

[7] M. B. Jones, The Emerging JSON-Based Identity Protocol Suite, Travanj 2011.

SAŽETAK

Sustav razvijen radi pojednostavljivanja administratorima unos velikog broja korisnika u neku organizaciju. Potrebno je posjedovati excel datoteku s korisnicima i njihovim osobnim podacima te ih se samo u par koraka može registrirati u bazu podataka. Cijeli sustav je razvijen pomoću Java programskog jezika i Spring razvojnog okvira. Temelji se na REST API-jima koji na temelju HTTP zahtjeva obavljaju određene operacije. Poduzete su određene mjere sigurnosti kako bi se rad sa sustavom omogućio samo korisnicima koji imaju ulogu administratora. Kako bi sustav radio stabilno i bez prekida implementirano je rukovanje iznimkama. Radi dodatne sigurnosti, korisnici nikad ne rade s konkretnim objektima baze nego s objektima za prijenos podataka. Za svaki dio sustava napisani su integracijski testovi kako bi se provjerila njegova ispravnost, te kako se ponaša u određenim situacijama.

Ključne riječi: HTTP, integracijsko testiranje, objekti za prijenos podataka, REST API, Spring razvojni okvir

ABSTRACT

USER ENTRY AND REGISTRATION WITH SPRING FRAMEWORK API

A system developed for a simple registration process and categorizing many users into an organization. It is necessary to possess an excel file with users and their associated personal data, which will allow their registration in just a few steps. The whole system was developed using Java programming language and Spring framework. It is based on REST API which uses HTTP requests for executing certain operations. Necessary measurements have been taken to ensure a high level of security. Only users with the administrator role can use this system. To ensure the system stability, a high level of exception handling has been provided. For the sake of extra security, users never work with direct database objects. Instead, they work with data transfer objects, which ensures that nothing irregular will be stored to the database. For each part of this system, integration tests have been written to ensure proper behavior. Furthermore, some tests have been written to check how the system responds in unexpected situations.

Keywords: data transfer objects, HTTP, integration testing, REST API, Spring framework

ŽIVOTOPIS

Marko Cecelja rođen je 05.07.1998. u Našicama. Pohađao je osnovnu školu Matije Petra Katančića u Valpovu. Nakon osnovne škole upisao je Elektrotehničku i prometnu školu u Osijeku, smjer tehničar za računalstvo. Za vrijeme srednjoškolskog obrazovanja, odrađuje dvotjednu praksu u sklopu Erasmus+ projekta u tvrtki iPhone Fix UK. Sudjelovao je na 4. međužupanijskoj izložbi inovacija, Ivanić Grad gdje osvaja zlatnu plaketu za android aplikaciju. Za istu aplikaciju osvaja zlatno odličje na 10. nacionalnoj izložbi u Matuljima, te zlatnu medalju na Inovi u Osijeku. Nakon završetka srednje škole, upisuje Fakultet elektrotehnike, računarstva I informacijskih tehnologija u Osijeku, preddiplomski sveučilišni smjer računarstvo. Za vrijeme studiranja kreće raditi za tvrtku Lamaro Digital d.o.o. kao *back end* developer.