Izrada aplikacije za evidenciju korištenja dana godišnjih odmora i plaćenih dopusta

Ceković, Ivan

VISOKO UČILIŠTE ALGEBRA

ZAVRŠNI RAD

IZRADA APLIKACIJE ZA EVIDENCIJU KORIŠTENJA DANA GODIŠNJIH ODMORA I PLAĆENIH DOPUSTA

Ivan Ceković

Zagreb, siječanj 2018.

Student vlastoručno potpisuje Završni rad na prvoj stranici ispred Predgovora s datumom i oznakom mjesta završetka rada te naznakom:

„Pod punom odgovornošću pismeno potvrđujem da je ovo moj autorski rad čiji niti jedan dio nije nastao kopiranjem ili plagiranjem tuđeg sadržaja. Prilikom izrade rada koristio sam tuđe materijale navedene u popisu literature, ali nisam kopirao niti jedan njihov dio, osim citata za koje sam naveo autora i izvor, te ih jasno označio znakovima navodnika. U slučaju da se u bilo kojem trenutku dokaže suprotno, spreman sam snositi sve posljedice uključivo i poništenje javne isprave stečene dijelom i na temelju ovoga rada “.

U Zagrebu, datum.

Ime Prezime

Predgovor

Ova stranica treba sadržavati izjavu ili zahvalu kandidata….

Prilikom uvezivanja rada, Umjesto ove stranice ne zaboravite umetnuti original potvrde o prihvaćanju teme završnog rada kojeg ste preuzeli u studentskoj referadi

Sažetak

Spring je programski okvir otvorenog koda koji se koristi za izradu poslovnih Java web aplikacija. Sastoji se od više različitih projekata od kojih je najzanimljiviji Spring Boot jer sadrži sve ostale projekte, od kojih su najpoznatiji Spring Security, Spring Data i Spring MVC. Za pristup podacima iz baze podataka koristi se JDBC, JPA i Hibernate. Navedene tehnologije se kombiniraju kako bi se dobila optimalna brzina izvođenja aplikacije i razumljiviji kod. Za prikaz obrađenih podataka najčešće se koristi Thymeleaf predložak zbog jednostavnijeg procesiranja od strane servlet kontejnera i činjenice da je Thymeleaf čisti HTML, za razliku od JSP-a.

Cilj ovog završnog rada je detaljnije objasniti Spring programski okvir, njegove karakteristike i primjenu u modernim poslovnim Java web aplikacijama.

Ključne riječi: Spring, Spring Boot, Maven repozitorij, JDBC, JPA, Hibernate, Thymeleaf

Abstract

Spring is an open-source framework used for Java enterprise applications development. It consists of multiple different projects, of which Spring Boot is the most interesting because it contains all other projects, of which Spring Security, Spring Dana and Spring MVC. are the most well-known. For accessing database data, JDBC, JPA and Hibernate technologies are used. These technologies are often combined so better performance and more understandable code would be achieved. When viewing processed data, Thymeleaf template is the most used because of its easier processing by servlet container and because Thymeleaf is pure HTML, unlike JSP.

The goal of this final work is to give more detailed explanation about Spring framework characteristics and its use in modern day business Java web applications.

Key words: Spring, Spring Boot, Maven repository, JDBC, JPA, Hibernate, Thymeleaf

Sadržaj

1. Uvod

Spring programski okvir je programski okvir otvorenog koda za izradu web aplikacija u Java programskom jeziku. Temeljna karakteristika ove tehnologije je postojanje više zasebnih, odvojenih projekata koji se mogu kombinirati ovisno o zahtjevima aplikacije. Spring programski okvir je tehnologija koja omogućava razvoj poslovnih aplikacija bilo koje vrste i namjene. Jedna od tih vrsta su i aplikacije za evidenciju korištenja godišnjih odmora i plaćenih dopusta.

Aplikacije za evidenciju korištenja godišnjih odmora i plaćenih dopusta se koriste u svim tvrtkama kako bi se na jednostavan način vodile evidencije o korištenju godišnjih odmora i plaćenih dopusta zaposlenika tvrtke. Te se aplikacije sastoje od više različitih poslovnih domena, zbog čega se korištenje Spring programskog okvira nameće kao logični odabir.

Cilj rada detaljno prikazati karakteristike Spring Boot programskog okvira, kao i drugih tehnologija korištenih uz njega (Hibernate objektno relacijsko mapiranje) i primjenu tih tehnologija u praksi izradom aplikacije za evidenciju korištenja dana godišnjeg odmora i plaćenog dopusta.

Literatura koja se bavi ovom tematikom se sastoji od knjiga navedenih u popisu literature i službene dokumentacije na stranicama Spring programskog okvira.

2. Spring programski okvir

Spring programski okvir se prvi put spominje 2002. godine u knjizi *Expert One-on-One: J2EE Design and Development* autora Roda Johnsona i bio je zamišljen kao tehnologija za rješavanje problema složenosti izrade aplikacija korištenjem tadašnjih Java tehnologija, kao što je EJB (engl. *Enterprise JavaBeans*). (Walls, 2015) Spring je donio novi pristup u vidu *dependency injection*-a (u daljnjem tekstu DI), i aspektno orijentiranog programiranja, što je aplikacijama omogućilo posjedovanje EJB svojstava (odvajanje poslovne logike od prezentacije, skalabilnost, integritet podataka,...) korištenjem običnih Java objekata/klasa (engl. *Plain Old Java Object*, POJO). DI i POJO objekti su temeljni principi na kojima radi Spring.

POJO objekti su obične Java klase koje ne nasljeđuju druge Java klase i ne implementiraju Java sučelja, a najčešće se koriste kod pohranjivanja podataka. Primjer POJO-a je klasa `Grad`:

```
public class Grad
{
	private int id_grad;
	private String naziv;
}
```

Ako se povezuju sa bazom podataka, POJO objekti mogu imati određene anotacije, primjerice `@Entity`, ali o tome više riječi u nastavku rada.

DI označava međudjelovanje više povezanih klasa za obavljanje neke poslovne logike. Svaki objekt ima referencu na objekt s kojim je povezan, što može dovesti do čvrsto povezanog (engl. *highly coupled*) koda kojeg je teško održavati i testirati. Takav primjer je slijedeća klasa (Walls, 2015):

```
public class DamselRescuingKnight implements Knight
{
	private RescueDamselQuest quest;
	public DamselRescuingKnight()
	{
		this.quest = new RescueDamselQuest();
	}
}
```

Masno otisnuta linija koda prikazuje slučaj nekorištenja DI-a. `RescueDamselQuest` kreira instancu klase u konstruktoru, što dovodi do čvrste veze sa `DamselRescuingKnight` klasom. To nije dobra praksa jer se jedna klasa ne može testirati neovisno o drugoj, što povećava mogućnosti grešaka u aplikaciji.

Slijedeći primjer prikazuje klasu sa koja koristi DI (Walls, 2015):

```
public class BraveKnight implements Knight
{
      private Quest;
      public BraveKnight(Quest quest)
      {
            this.quest = quest;
      }
}
```

U ovom slučaju `BraveKnight` ne kreira instancu klase `Quest` nego ju dobiva kao argument (masno otisnuto). Još jedna bitna stvar je da je klasa `Quest` napisana kao sučelje, što znači da `BraveKnight` može imati bilo koju implementaciju klase `Quest`, tj. nije čvrsto vezan.

Spring je projekt otvorenog koji se počeo razvijati 2003. godine i do danas je prošao mnoge promjene i verzije. Spring programski okvir je podijeljen na module koji se mogu kombinirati ovisno o potrebama aplikacije (Spring Framework Overview, 2018). Najpoznatiji su Spring Boot, Spring Security, Spring Data, Spring Cloud i Spring Web. Upravo je Spring Boot najzanimljiviji jer obuhvaća sve ostale projekte i danas se najviše koristi za izradu aplikacija zbog njihove relativno brze inicijalizacije i razvoja, kao i automatske konfiguracije potrebnih datoteka i dohvaćanja potrebnih datoteka i aplikacija koje nisu dio programskog okvira (engl. *third party*) , primjerice Maven repozitorij.

2.1. Spring Boot

Spring Boot je projekt Spring programskog okvira koji sadrži sve ostale projekte. To znači da se preko njega može pristupiti svim ostalim projektima koji se žele koristiti u aplikaciji. Spring Boot projekt se kreira pomoću Spring Initializr-a. Spring Initializr je aplikacija koja generira strukturu Spring Boot projekta (ne generira nikakav kod same aplikacije, nego samo njenu strukturu). Postoji više načina pomoću kojih se može koristiti Spring Initializr (Walls,

2016): preko Spring Boot web sučelja, korištenjem razvojnih alata (Spring Tool Suite, Eclipse...) ili korištenjem Spring Boot CLI-a (sam postupak kreiranja aplikacije se neće objašnjavati jer je isti kao kreiranje bilo kojeg drugog projekta). Nakon kreiranja aplikacije kreiraju se dvije datoteke: pom.xml i klasa koja služi za pokretanje aplikacije. Ta klasa je glavna konfiguracijska klasa u aplikaciji i povezuje ostale module aplikacije. Ona je dovoljna za pokretanje Spring Boot aplikacije. Izgled klase se može vidjeti u slijedećem primjeru:

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

Slika 1: Glavna konfiguracijska klasa

Anotacija `@SpringBootApplication` omogućava skeniranje i povezivanje svih komponenti i klasa korištenih u projektu i korištenje automatske konfiguracije. Umjesto ove anotacije se mogu koristiti `@Configuration`, `@ComponentScan` i `@EnableAutoConfiguration`, ali se zbog jednostavnosti obično koristi `@SpringBootApplication`.

Pom.xml datoteka je slijedeća vrlo bitna datoteka u Spring Boot projektu koja služi za pohranu podataka o modulima i jar datotekama koje se koriste u projektu, kao i neke postavke aplikacije. To olakšava izradu aplikacije jer se jar datoteke ne moraju tražiti ručno, nego se dohvaćaju preko Maven repozitorija. Slijedeći primjer prikazuje pohranu jedne jar datoteke:

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

Gornji kod sa Maven repozitorija dohvaća jar potreban za kreiranje Web MVC aplikacija. `groupId` označava organizaciju ili projekt u kojem se jar nalazi (Spring Boot ima više jar datoteka koje imaju svoju funkciju), `artifactId` označava naziv jar datoteke, bez

verzije. Neke jar datoteke mogu imati atribut `version` posebno, iako nije obavezan kod svih.

Na prethodno opisani način se dohvaćaju i konfiguriraju jar datoteke za sve Spring Boot module (Spring Security, Spring Data, ...).

Prethodno opisane datoteke su osnovne konfiguracijske datoteke za sve Spring Boot aplikacije. Međutim, postoji još jedna datoteka koja se također kreira u projektu, ali inicijalno ne sadrži nikakve podatke. Ta klasa se zove `application.config` i služi za pohranu specifičnih konfiguracijskih podataka, najčešće za povezivanje na bazu podataka, konfiguraciju za slanje e-mail poruka i slično.

Primjer konfiguracije za povezivanje na bazu podataka:

```
spring.datasource.url=
jdbc:sqlserver://localhost:1433;databaseName=GodisnjiOdmorPla
ceniDopust
spring.datasource.username= sa
spring.datasource.password= SQL
spring.jpa.database-
platform=org.hibernate.dialect.SQLServerDialect
```

2.2. Povezivanje s bazom podataka

Relacijske baze podataka su temelj svih poslovnih aplikacija i same aplikacije bez njih ne bi imale smisla. U Spring programskog okviru postoje dva načina pomoću kojih se može ostvariti veza sa bazom podataka: korištenjem Spring JDBC-a, objektno-relacijskog mapiranja (Hibernate) ili korištenjem Spring Java Persistence API-a (JPA).

2.2.1. Spring JDBC

JDBC(engl. *Java Database Connectivity*) je najosnovniji način povezivanja aplikacije sa bazom podataka i vrlo se često koristi zbog dobrih karakteristika kao što su rad s podacima na nižoj razini (engl. *low level*), što znači rad sa podacima na razini stupaca i redaka. Također je moguće detaljnije praćenje i poboljšavanje performansi aplikacije.

Za povezivanje aplikacije sa bazom potrebno je kreirati izvor podataka (engl. *data source*). Spring daje na izbor tri klase kojima se može kreirati izvor podataka (Walls, 2015):

- `DriverManagerDataSource` – vraća novu vezu na bazu svaki put kada je zatražena. Ova klasa ne sprema veze u *cache* (engl. *connection pooling*).
- `SimpleDriverDataSource` – ista svojstva kao i prethodna, s tom razlikom da ova klasa radi izravno sa JDBC upravljačkim programom zbog izbjegavanja potencijalnih problema s učitavanjem klase.
- `SingleConnectionDataSource` – vraća istu vezu svaki put kada je zatražena. Radi na principu spremanja veze u *cache* (engl. *connection pooling*), iako nije takav tip veze (nije *pooled data source*).

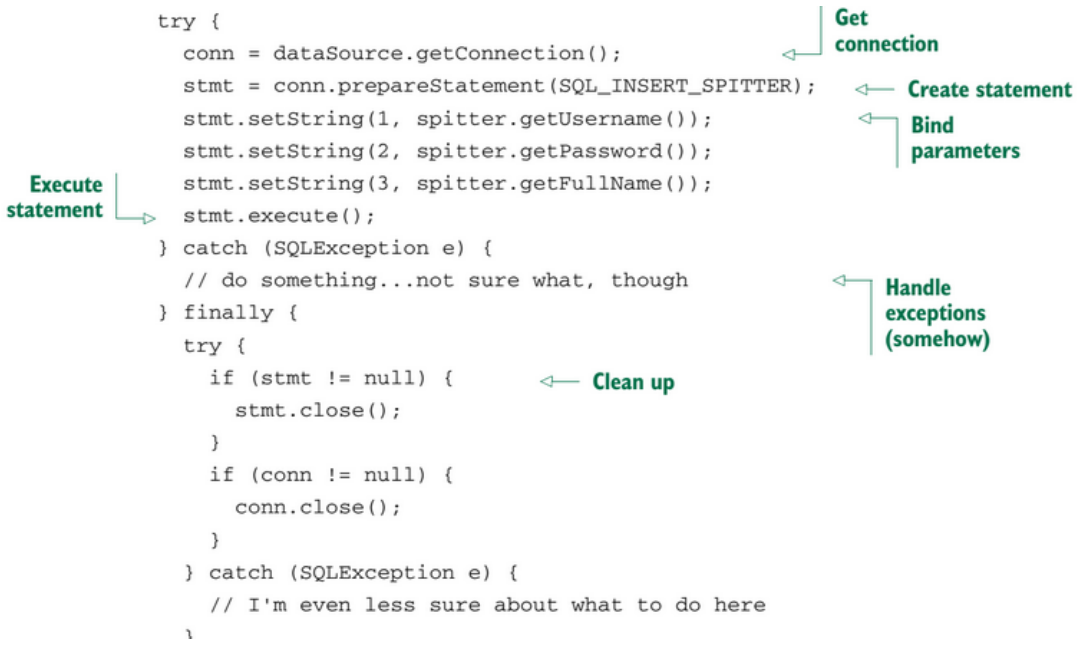
Sva tri izvora podataka se kreiraju na isti način. Slijedeći primjer prikazuje kreiranje `DriverManagerDataSource` izvora podataka:

```
@Bean
public DataSource dataSource()
{
      DriverManagerDataSource ds = new
      DriverManagerDataSource();
      ds.setDriverClassName("com.microsoft.sqlserver.jdbc.SQL
      ServerDriver");
      ds.setUrl("jdbc:sqlserver://localhost:1433;databaseName
      =GodisnjiOdmorPlaceniDopust");
      ds.setUsername("sa");
      ds.setPassword("SQL");
      return ds;
}
```

Ove tri klase nisu pogodne za rad u produkciji jer `SingleConnectionDataSource` sadrži samo jednu vezu na bazu, pa ne može raditi u višenitnim aplikacijama, a klase `DriverManagerDataSource` i `SimpleDriverDataSource` utječu na performanse jer se svaki put kreira nova veza na bazu. Zbog toga je ove klase preporučljivo koristiti u testnim okruženjima, a ne u produkciji.

Uz svoje dobre strane, JDBC ima i svoje strane, zbog kojeg se danas većinom koriste JPA tehnologije, primjerice Hibernate.

Najveći problem JDBC-a je postojanje puno linija redundantnog koda za obavljanje jednostavnih i sličnih operacija, primjerice umetanje i ažuriranje retka u bazi. Slijedeći primjer prikazuje slučaj sa umetanjem retka (Walls, 2015):



Slika 2: Umetanje retka korištenjem JDBC-a

Za uređivanje retka cijeli kod izgleda isto, osim što se umjesto insert piše `update`. Zapravo samo jedna linija koda radi promjene na bazi, i to `stmt.execute()`; Ostatak koda služi za rukovanje vezom na bazu i kreiranje izvora podataka, što kod složenih aplikacija dovodi do velike količine redundantnog koda.

U slijedećem poglavlju će se govoriti o spajanju na bazu putem JPA-a i objektno-relacijskog mapiranja, tehnologijama koje olakšavaju povezivanje aplikacije i baze.

2.2.2. Spring Java Persistence API

Java Persistence API (JPA) dolazi iz EJB tehnologije i temelji se na POJO objektima; također posjeduje svojstva Hibernate i Java Data Objects tehnologija. Da bi se mogao koristiti JPA u aplikaciji, potrebno je konfigurirati *entity manager factory* u aplikaciji. Postoje dva tipa *entity manager*-a (Walls, 2015): **Application-managed** i **Container-managed**.

Application-managed kreira *entity manager* onda kada ga aplikacija eksplicitno traži. U tom slučaju ona je odgovorna za otvaranje i zatvaranje *entity manager*-a i rukovanje transakcijama. Najprikladnije ga je koristiti kod samostalnih aplikacija koje ne koriste Java EE kontejner, primjerice desktop aplikacije.

Container-managed *entity manager*-i su kreirani i upravljani od strane Java EE kontejnera i to preko JNDI-a, aplikacija s njima nema doticaja. Koristi se kod povezivanja web aplikacija sa bazom podataka i češće se koristi.

Postavke za izvor podataka container-managed JPA-a se mogu kreirati u posebnoj klasi, što izgleda ovako:

```
@Bean
public LocalContainerEntityManagerFactoryBean entityManagerFactory(
        DataSource dataSource, JpaVendorAdapter jpaVendorAdapter) {
  LocalContainerEntityManagerFactoryBean emfb =
      new LocalContainerEntityManagerFactoryBean();
  emfb.setDataSource(dataSource);
  emfb.setJpaVendorAdapter(jpaVendorAdapter);
  return emfb;
}
```

Slika 3: Postavke za container-managed entity manager

Argument tipa `JpaVendorAdapter` služi za odabir određene JPA implementacije. Moguće opcije su: (Walls, 2015)

- `EclipseLinkJpaVendorAdapter`
- `HibernateJpaVendorAdapter`
- `OpenJpaVendorAdapter`
- `TopLinkJpaVendorAdapter` (zastarjelo u verziji Spring 3.1)

Dohvaćanje i rukovanje s podacima iz baze se obavlja pomoću klase `EntityManager`. Primjer umetanja podataka se može vidjeti u slijedećem kodu:

```
@Repository
@Transactional
public class RepozitorijRezervacija implements IRepozitorijRezervacija
{
    @PersistenceContext
    private EntityManager entityManager;

    public void rezervirajSobu(Rezervacija rezervacija)
    {
        entityManager.persist(rezervacija);
    }
```

Slika 4: Umetanje retka preko entity manager-a

Vidi se da je, za razliku od JDBC-a, gdje je potrebno puno dodatnog koda za otvaranje veze na bazu, izvođenja naredbe za umetanje podataka, zatvaranje veze i hvatanje eventualnih

grešaka, ovdje dovoljna samo jedna: `entityManager.persist(rezervacija)`. U pozadini se obavljaju sve naredbe koje bi se u slučaju JDBC-a , morale ručno pisati. Još jedna novost su anotacije `@Repository`, `@Transactional` i `@PersistenceContext`. `@Repository` služi za automatsko skeniranje svake klase koja ima tu anotaciju (tu funkciju imaju i sve druge anotacije), ali ta anotacija također mora označavati i klasu koja je repozitorij, odnosno da služi za pristup i rad sa bazom podataka. Nadalje, `@Transactional` određuje da su sve metode kojima se pristupa bazi (`persist`, `merge`, `find`...) u kontekstu transakcije, što znači da programer ne mora ručno otvarati i zatvarati veze na bazu (iako može), `EntityManager` će to obaviti umjesto njega. Na kraju, `@PersistenceContext` definira izvor podataka koji se koristi u aplikaciji (bazu podataka, tablice...) i grupira ih u jednu cjelinu.

Korištenje Entity managera je brz i jednostavan način za rad s podacima. Međutim, nakon nekog vremena i takav kod postane redundantan. Za primjer možemo uzeti istu liniju koda, `entityManager.persist(rezervacija)`. U svim većim aplikacijama umetanje podataka (koji nisu istog tipa ni iste poslovne domene) u bazu se radi više puta, što znači da sami kod ostaje isti, a mijenja se argument koji se predaje metodi. Spring JPA nudi rješenje u vidu kreiranja sučelja koje nasljeđuje klasu `JpaRepository`, što je prikazano u slijedećem kodu:

```
public interface ZaposlenikRepozitorij extends
JpaRepository<Zaposlenik, Long>
{
      Zaposlenik dohvatiZaposlenika(String ime);
}
```

Ovo je potpuno dovoljno. Implementacija repozitorija nije potrebna, Spring će sam implementirati repozitorij kada bude potrebno. Bitna karakteristika sučelja `JpaRepository` je da već sadrži 18 metoda za transakcije koje se često obavljaju nad podacima, no moguće je kreiranje i vlastitih metoda, ako je potrebno, primjerice metoda `dohvatiZaposlenika(String ime)`. Metoda se kreira isto kao i kod ostalih sučelja i, kao niti ostatak metoda iz repozitorija, ne trebaju implementaciju. Samo definiranje metode je dovoljno za implementaciju.

Korištenje Java Persistence API-a omogućava dohvat podataka iz baze za bilo koje parametre (sve podatke, prema imenu korisnika, prema identifikatoru korisnika...), međutim takav način ne podržava složenije upite, primjerice dohvat podataka iz više tablica i njihovo grupiranje, agregatne funkcije i slično. Objektno relacijsko mapiranje (Hibernate) je ono što je potrebno kako bi s mogli kreirati složeniji upiti.

2.2.3. Hibernate

Hibernate je programski okvir otvorenog koda koji služi za povezivanje aplikacije i baze podataka i rad sa njenim podacima (objektno relacijsko mapiranje). Osim toga, omogućava i složenije transakcije nad podacima, primjerice pohranjivanje podataka u privremeni spremnik (engl. *caching*), sortiranje, grupiranje, filtriranje podataka i slično. Temelj Hibernate programskog okvira su POJO objekti.

Kao i JPA, i Hibernate ima bolje karakteristike u usporedbi sa JDBC-om, primjerice korištenje manje resursa kod rada sa podacima (nije potrebno voditi brigu o spajanju na bazu, otvaranju i zatvaranju veza i slično, Hibernate sam upravlja tim stvarima), omogućavanje konfiguracije i povezivanja POJO objekata na isti način kojim su povezani u bazi (vezama 1:N, N:N, N:1, 1:1), što omogućuje kaskadno dohvaćanje podataka iz drugih tablica. Jezik kojim se dohvaćaju podaci iz baze i koji se koristi u Hibernate svijetu (HQL, engl. *Hibernate Query Language*) ima vrlo sličnu sintaksu kao SQL i Entity Framework iz .NET svijeta, što znači da privikavanje na ovu tehnologiju traje vrlo kratko, ako je programer upoznat s tim tehnologijama. Zbog velikog broja dobrih karakteristika Hibernate se vrlo često koristi, kako sam, tako i u kombinaciji sa JPA-om.

Za korištenje u Spring aplikaciji, Hibernate je najjednostavnije dohvatiti sa Maven repozitorija, kao i sve ostale jar datoteke:

```
<dependency>
    <groupId>org.hibernate</groupId>
    <artifactId>hibernate-core</artifactId>
    <version>5.2.12.Final</version>
</dependency>
```

Anotacije

Da bi bilo moguće raditi sa podacima iz baze, potrebno je odrediti koja tablica pripada kojoj klasi. Zbog toga je potrebno mapirati tablice na klase (POJO objekte). Temeljne anotacije potrebne za mapiranje klasa na tablice su `@Entity`, `@Id` i `@GeneratedValue`. `@Entity` definira klasu kao entitet, što znači da ta klasa predstavlja istu tablicu u bazi. `@Id` označava stupac koji je primarni ključ u bazi, a `@GeneratedValue` označava tip kreiranja i čuvanja primarnih ključeva. Postoje četiri tipa kreiranja i čuvanja primarnih ključeva (Ottinger et al, 2016):

- `AUTO` – Hibernate sam odlučuje koji tip upotrijebiti, ovisno o postavkama baze za kreiranje primarnih ključeva
- `IDENTITY` – baza podataka je odgovorna za generiranje primarnih ključeva (u ovom slučaju se generirani ključevi se samo dohvaćaju iz baze)
- `SEQUENCE` – koristi se ako baze podržavaju `SEQUENCE` tip stupaca (tip koji se može koristiti za pohranjivanje primarnih ključeva, sličan je tipu `IDENTITY`, jedina je razlika u tome što `SEQUENCE` nije vezan za točno određenu tablicu i zato ga mogu koristiti više tablica odjednom). Primjerice:

```
@Id
@SequenceGenerator(name="seq1",sequenceName="HIB_SEQ")
@GeneratedValue(strategy=SEQUENCE,generator="seq1")
int id;
```

Slika 5: Generiranje primarnog ključa Sequence generatorom

- `TABLE` – ovaj tip generira posebnu tablicu za čuvanje primarnih ključeva. Koristi se slično kao i SEQUENCE, ali je za razliku od njega prenosiv između različitih tipova baza jer koristi standardne tablice za pohranjivanje ključeva. Primjer korištenja prikazuje sljedeći kod:

```
@Id
@TableGenerator(name="tablegen",
                table="ID_TABLE",
                pkColumnName="ID",
                valueColumnName="NEXT_ID")
@GeneratedValue(strategy=TABLE,generator="tablegen")
int id;
```

Slika 6: Generiranje primarnog ključa Table generatorom

Atribut `ime` je obavezan, a ostali su opcionalni. Svi opcionalni atributi su prikazani u nastavku:

  - `allocationSize` – koristi se za rukovanje primarnim ključevima kod poboljšavanja performansi
  - `catalog` – određuje katalog u kojem se nalazi određena tablica
  - `initialValue` – određuje vrijednost stupca primarnog ključa

- `pkColumnName` – određuje naziv stupca primarnog ključa
- `pkColumnValue` – određuje informacije o generiranju primarnog ključa
- `schema` – određuje shemu u kojoj se tablica nalazi
- `table` – određuje tablicu sa primarnim ključevima
- `uniqueConstraints` – određuje ograničenja za kreiranje tablica
- `valueColumnName` – omogućuje stupcu koji sadrži informacije o generiranju primarnog ključa da identificira entitet na kojeg se primarni ključ odnosi

Anotacija koja se može koristiti, ali nije obavezna (svojstva koja ova anotacija podešava se najčešće podese u bazi, tako da ih nije potrebno podesiti u aplikaciji) je anotacija `@Column`. Ona se koristi za definiranje detalja/atributa tablice koja se mapira. Atributi mogu biti `name` (određuje ime stupca tablice u bazi), `length` (određuje duljinu podataka stupca, najveća vrijednost je 255, najčešće se koristi kod String tipova podataka), `nullable` (određuje da vrijednost podatka ne smije biti `null`, inicijalno smije), `unique` (označava da se podatak u stupcu ne smije ponavljati).

Ostali rjeđe korišteni atributi su `table` (koristi se kada je entitet na kojeg se atribut odnosi mapiran kroz više sekundarnih tablica), `insertable` (inicijalno je `true`, a ako se postavi na `false`, polje na koje se atribut odnosi će biti izostavljeno od naredbe `insert`), `updatable` (isto kao `insetable`, osim što je ovdje naredba `update`), `columnDefinition` (koristi se kod kreiranja retka u bazi i može poprimiti vrijednost prikladnog DDL fragmenta. Trebao bi se koristiti samo kod kreiranja sheme baze jer inače može biti problema kod portabilnosti aplikacije u vezi dijalekata baze), `precision` (definira preciznost decimalne znamenke kod stupaca decimalnih vrijednosti, broj znamenki u broju. Ako je vrijednost stupca cijeli broj, atribut se zanemaruje.), `scale` (definira broj mjesta nakon decimalne točke. Ako je vrijednost stupca cijeli broj, atribut se zanemaruje). `@Column` se ne mora koristiti, ali onda elementi klase i nazivi stupaca u bazi moraju biti istih imena.

Primjer entiteta sa anotacijama :

```
@Entity
public class OrganizacijskaJedinica
{
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private int id_organizacijska_jedinica;

    @Column(name="naziv")
    private String naziv;
```

Slika 7: Entitet sa anotacijama

Anotacija `@Column` u ovom slučaju nije jer se element entiteta zove isto kao stupac u tablici, ovdje je samo za prikaz.

Asocijacije

Do sada je bilo riječi o kreiranju klasa i mapiranju tablica na njih. Međutim, to nije dovoljno jer su tablice u bazi podataka međusobno povezane. To znači da se i klase moraju međusobno povezati u isti odnos kao i tablice da bi se moglo rukovati podacima. Asocijacije su ono što je za to potrebno.

Postoje četiri vrste asocijacija za povezivanje entiteta sa tablicama (Ottinger et al, 2016):

- `@OneToOne` – može se prikazati na različite načine. U najjednostavnijem slučaju, atributi obje klase se nalaze u istoj tablici, primjerice:

ID	Username	Email
1	dcminter	dcminter@example.com
2	jlinwood	jlinwood@example.com
3	tjkitchen	tjkitchen@example.com

Slika 8: Atributi obje klase u istoj tablici

U drugom slučaju, entiteti se mogu nalaziti u zasebnim tablicama sa identičnim primarnim ključevima ili ključem referenciranim od jednog identiteta u drugi, što prikazuje slijedeća slika.

Table 5-6. *The User Table*

ID	Username
1	Dcminter
2	Jlinwood
3	Tjkitchen

Table 5-7. *The Email Table*

ID	Username
1	dcminter@example.com
2	jlinwood@example.com
3	tjkitchen@example.com

Slika 9: Svaka klasa u svojoj tablici

Također se može u jednoj tablici čuvati primarni ključ na drugu tablicu i u praksi se tako najčešće radi, što se vidi na slijedećoj slici:

ID	Email	UserID (Unique)
34	dcminter@example.com	1
35	jlinwood@example.com	2
36	tjkitchen@example.com	3

Slika 10: Primarni ključ u istoj tablici

Ova se asocijacija najčešće koristi zbog jednostavnosti i jer ju je moguće brzo promijeniti u bilo koju drugu asocijaciju micanjem `Unique` ograničenja.

- `@ManyToOne` i `@OneToMany` – ove dvije asocijacije su iste, ovisno s koje strane se gleda. Mogu se prikazati korištenjem stranog ključa bez dodatnih ograničenja ili korištenjem vezivne tablice. Vezivna tablica sadrži primarne ključeve od svih povezanih tablica, koji se u vezivnoj tablici prikazuju kao strani ključevi. Primjer takve tablice se može vidjeti na slijedećoj slici.

UserID	EmailID
1	1
1	2
2	3
2	4

Slika 11: Vezivna tablica

Stupci `UserID` i `EmailID` su primarni ključevi tablica `User` i `Email`. Jedna tablica mora imati ograničenje `Unique`, inače ovo postaje veza `@ManyToMany`.

Primjer `@ManyToOne` asocijacije se može vidjeti u slijedećem kodu:

```
@ManyToOne
@JoinColumn(name = "organizacijska_jedinica_id")
private OrganizacijskaJedinica organizacijska_jedinica;
```

Primjer `@OneToMany` asocijacije se može vidjeti u slijedećem kodu:

```
@OneToMany(mappedBy = "organizacijska_jedinica")
private List<Zaposlenik> zaposlenici;
```

U slučaju `@OneToMany` asocijacije postoji atribut `mappedBy` koji traži tablicu sa imenom koje je definirano u atributu i dohvaća sve podatke za određeni primarni ključ (u ovom slučaju jedna organizacijska jedinica može imati više zaposlenika).

- `@ManyToMany` – ova je asocijacija ista kao i prethodne dvije, jedino nema ograničenje `Unique`. Uz dodavanje primarnog ključa vezivnoj tablici, ona postaje potpuna *many to many* veza.

Transakcije nad podacima

Za obavljanje transakcija nad podacima u Hibernate programskom okviru se koristi HQL (engl. *Hibernate Query Language*). HQL je objektno orijentirani jezik upita sa sličnom sintaksom kao i SQL, jedina je razlika što, za razliku od SQL-a koji radi sa tablicama i stupcima, HQL radi sa klasama mapiranim na tablice i njenim atributima. Osim jednostavnijih upita nad podacima kao što stu CRUD operacije, HQL omogućava i složenije, primjerice grupiranje više tablica, korištenje agregatnih funkcija i definiranje kriterija za specifične uvjete transakcija nad podacima. HQL je moguće kombinirati sa JPA-om, zbog jednostavnosti (primjerice, za umetanje retka u bazu je jednostavnije napisati

`entityManager.persist()` nego pisati cijeli HQL insert upit) što omogućava korištenje najboljih karakteristika obje tehnologije.

Slijedeći kod prikazuje dohvaćanje podataka iz baze uz ulazni parametar:

```
@SuppressWarnings("unchecked")
public List<Zaposlenik> dohvatiZaposlenika(String korisnickoIme)
{
    Query query = entityManager.createQuery("from Zaposlenik where korisnicko_ime = :korisnickoIme");
    query.setParameter("korisnickoIme", korisnickoIme);
    return query.getResultList();
}
```

Slika 12: Dohvaćanje podataka iz baze

Vidljivo je da je sintaksa upita vrlo slična onome iz SQL svijeta, uz male razlike. Kao prvo, naredba `select` nije potrebna (iako se može napisati, ali nije obavezna). Druga stvar je da HQL zapravo ne dohvaća podatke direktno iz tablice u bazi, nego klase koja je mapirana na tablicu (`Zaposlenik` je ime klase, a `korisnicko_ime` je naziv atributa klase i stupca u tablici, istog su imena zbog manje vjerojatnosti pogreške). To je vrlo bitno zapamtiti jer ako bi slučajno u upitu napisali naziv tablice koji je različit od naziva klase, upit bi bacio grešku. Također, parametri nakon `where` klauzule nisu nazivi stupca u bazi nego atributa klase. Zbog toga je dobra praksa nazivati tablice i stupce/klase i atribute istim imenima zbog izbjegavanja grešaka u pisanju. Anotacija `@SurpressWarnings` nije toliko bitna, ako se makne neće biti greške. Ona se u ovom slučaju tiče naredbe `query.getResultList()`, kod pretvaranja tipa `ResultList` u drugi tip. Još jedna stvar u vezi ovog koda je da upit vraća jednog zaposlenika, a cijela metoda je tipa liste, kao i argument kojeg vraća. Takva je specifikacija Hibernate tehnologije, svi podaci koji se dohvaćaju iz baze (čak i ako je u pitanju samo jedan redak) se vraćaju kao lista. Ako se vraća samo jedan redak, on se dohvaća kao nulti element liste: `dohvatiZaposlenika(korisnickoIme).get(0);`. Što se tiče insert naredbe, ona je malo drukčija on one iz SQL-a. U HQL-u podaci se ne mogu umetati direktno kroz entitete, nego se umeću kroz selektirani entitet, što izgleda ovako:

```
Query query=session.createQuery("insert into purged_users(id, name, status) "+
    "select id, name, status from users where status=:status");
query.setString("status", "purged");
int rowsCopied=query.executeUpdate();
```

Slika 13: Umetanje podataka

Rezultat naredbe `insert` (kao i `update` i `delete`) je tipa `int`, ako upit prođe bez greške. S obzirom da ova HQL naredba može raditi samo sa podacima koji su dohvaćeni sa `select` naredbom, korištenje HQL insert naredbe može biti ograničavajuće. To je razlog zbog kojeg se često HQL zamjenjuje JPA-om, kao i zbog potencijalno velike duljine upita, o čemu će biti riječi kasnije. Još jedan problem HQL-a je nepostojanje provjere sintakse upita, što predstavlja veliki problem ako se pišu veliki upiti. U slučaju greške potrebno je puno vremena da se ona pronađe.

`Update` i `delete` upiti su iste kao i kod SQL-a:

```
Query query=session.createQuery("update Person set creditscore=:creditscore where
name=:name");
query.setInteger("creditscore", 612);
query.setString("name", "John Q. Public");
int modifications=query.executeUpdate();
```

Slika 14:Update upit

```
Query query=session.createQuery("delete from Person where accountstatus=:status");
query.setString("status", "purged");
int rowsDeleted=query.executeUpdate();
```

Slika 15: Delete upit

Imenovani upiti

Hibernate također omogućava kreiranje imenovanih upita, što zapravo predstavlja upite kreirane na razini klase. Svaka klasa može imati više imenovanih upita, koji se sastoje od naziva upita i samog upita. Slijedeća slika prikazuje dva imenovana upita.

```
@NamedQueries({
        @NamedQuery(name = "supplier.findAll", query = "from Supplier s"),
        @NamedQuery(name = "supplier.findByName",
                query = "from Supplier s where s.name=:name"),
})
```

Slika 16: Imenovani upiti na jednoj klasi

Izvođenje imenovanih upita je isto kao i izvođenje običnih:

```
Query = session.getNamedQuery("supplier.findAll");
List<Supplier> suppliers = query.list();
```

Jedina razlika između običnih i imenovanih upita je postojanje više koda kod imenovanih, jer se osim izvođenja moraju kreirati anotacije i upiti na klasama. Odabir vrste upita ovisi o programeru.

Svi upiti obrađeni do sada nisu išli dalje od dohvaćanja podataka iz jedne tablice. Međutim, u praksi se najčešće podaci dohvaćaju iz više tablica odjednom. Za to je potrebno koristiti spajanje entiteta (asocijacije) , agregatne funkcije i kriterije.

Asocijacije (spajanje entiteta)

Asocijacije[1] se koriste za dohvaćanje podataka iz više povezanih tablica, isto kao u SQL bazama podataka. Moguća su pet tipa asocijacija: `inner join`, `cross join`, `left outer join`, `right outer join`, `full outer join`. Kod korištenja `cross join`-a dovoljno je u `from` klauzuli navesti klase iz kojih se dohvaćaju podaci (`from Product p, Supplier s`), a za ostale se prvo navede prvi entitet, alias za njega, tip spajanja, drugi entitet, alias za njega (Ottinger et al, 2016). Primjer spajanja entiteta prikazuje slijedeći kod:

```
select s.name, p.name, p.price from Product p inner join
p.supplier as s
```

U upitu koji sadrži asocijacije potrebno je kreirati aliase za entitete. To je pogotovo važno ako se radi upit koji ima više tablica koji završava na prvoj tablici (kružni upit, obično se takvi upiti koriste za organizaciju stablaste strukture podataka). Hibernate ne vraća rezultat tipa `Object` nego polje `Object` tipa. Za pristup podacima entiteta potrebno je proći po elementima polja.

Agregatne funkcije

Kao i SQL, HQL podržava korištenje agregatnih funkcija. Funkcije su gotovo iste, jedina je razlika što se u HQL-u primjenjuju na atribute mapiranih entiteta. Funkcije koje HQL

[1] Asocijacije su već prije spomenute, ali se u ovom kontekstu koriste za spajanje entiteta (inner, outer join,…)

podržava su `avg`, `count`, `max`, `min`, `sum`. Nazivi funkcija se ne moraju posebno objašnjavati. Primjer funkcije `count` izgleda ovako:

```
select count(distinct product.supplier.name) from Product
```

Kriterijski upiti

Kriterijski upiti su još jedan način rada sa podacima u Hibernate okruženju. U svom radu koriste model informacije koja se želi pronaći. Primjerice, ako se želi pronaći studenta imena Pero, kreira se instanca klase Student, atribut ime se postavi u Pero i ta se instanca koristi u upitu. Ovakav način zadavanja upita omogućuje kreiranje ugniježđenih izraza, uklanja vjerojatnost greške u sintaksi upita koja postoji u HQL-u, i korištenje upita prema primjeru sa studentom. Kriterijski upiti su vrlo moćan način zadavanja upita, ali imaju jednu manu: za složenije upite mogu postati nerazumljivi.

Kriterijski upiti koriste određene koncepte za kreiranje modela objekata koji se žele pronaći: klasu `CriteriaBuilder`, tipizirani `CriteriaQuery` (tipizirani označava tip klase vraćenog objekta), korijenski element za kreiranje upita koji sadrži kriterije prema kojima su objekti odabrani, set uvjeta prema kojima se dohvaćaju objekti i metamodel koji predstavlja tražene tipove klasa za jednostavnije referenciranje (Ottinger et al, 2016).

Slijedeći kod je primjer najjednostavnijeg kriterijskog upita, bez određenih uvjeta i metamodela:

```
CriteriaBuilder builder = em.getCriteriaBuilder();
CriteriaQuery<Product> criteria = builder.createQuery(Product.class)
Root<Product> root = criteria.from(Product.class);
criteria.select(root);

assertEquals(em.createQuery(criteria).getResultList().size(), 7);
```

Slika 17: Primjer najjednostavnijeg kriterijskog upita

Ovaj kod dohvaća sve podatke iz entiteta Proizvod. Operativna linija koda upita je `criteria.select(root);`, koja definira tip upita (u ovom slučaju tipa `Product`). Dodavanje kriterija za dohvat podataka se najlakše dodaje pomoću metamodela, što predstavlja način dohvata atributa klase. Primjer upita sa kriterijem koji dohvaća sve proizvode koji nemaju naziv „Mouse” predstavlja slijedeća slika.

```
CriteriaBuilder builder = em.getCriteriaBuilder();
CriteriaQuery<Product> criteria = builder.createQuery(Product.class);

Metamodel m = em.getMetamodel();
EntityType<Product> product = m.entity(Product.class);
Root<Product> root = criteria.from(product);
criteria.select(root);
criteria.where(
        builder.notEqual(
                root.get(Product_.description),
                builder.parameter(String.class, "description")
        )
);

criteria.select(root);

assertEquals(em.createQuery(criteria)
        .setParameter("description", "Mouse")
        .getResultList().size(), 5);
```

Slika 18: Upit sa kriterijem

Za dohvat podataka koji nemaju određenog atributa se koristi metoda `notEqual()` i u njoj se definira atribut klase koji se provjerava, a sama vrijednost parametra se kasnije predaje kao argument metode `assertEquals.setParameter()`. Na ovaj se način mogu definirati kriteriji za dohvat podataka počinju ili završavaju sa određenim znakovima, atributi se mogu pretvarati u mala ili velika slova, moguće je koristiti sortiranja, agregatne funkcije, i slično.

Kriterijski upiti su dobar način za programsko kreiranje upita, gdje je mogućnost sintaksne pogreške vrlo mala. Međutim, za složenije upite kod može biti vrlo nepregledan i otkrivanje pogreške može dugo trajati. Zbog toga se obično koristi kombinacija JPA i HQL upita, a kriterijski upiti se koriste rjeđe, za specifične upite.

2.3. Prikaz podataka

Dohvaćanje i obrada podataka čini osnovu poslovne aplikacije. Međutim, bez prikaza rezultata obrađenih podataka sama aplikacija ne bi imala smisla. U nastavku rada će se pojasniti prikazivanje obrađenih podataka korištenjem Thymeleaf predloška i zašto je Thymeleaf preferiraniji predložak od JSP-a.

2.3.1. Thymeleaf

Thymeleaf je moderan server-side model predložaka za prikaz podataka na web stranicama i aplikacijama na samostalnim okruženjima. Za razliku od JSP-a (koji je zapravo druga verzija apstrakcije Java servleta), Thymeleaf je čisti HTML (uz dodatne karakteristike) sa istim karakteristikama JSP-a (dohvaćanje podataka sa kontrolera, korištenje petlji, uvjetnih funkcija, ...).

Da bi se Thymeleaf mogao koristiti u web aplikaciji, potrebno ga je referencirati. To se obavlja preko Maven repozitorija, kao i za svaki drugi modul:

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

Također, ako se u razvoju aplikacije koristi Spring Tool Suite ili Eclipse, potrebno je skinuti Thymeleaf Eclipse *plugin* sa Interneta (postupak je objašnjen na http://www.thymeleaf.org/eclipse-plugin-update-site/). Thymeleaf datoteke se kreiraju kao obične HTML datoteke.

Slijedeća slika prikazuje jednostavnu HTML stranicu kreiranu u Thymeleaf predlošku:

```
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:th="http://www.thymeleaf.org">        <-- Declare Thymeleaf namespace
  <head>
    <title>Spittr</title>
    <link rel="stylesheet"
          type="text/css"
          th:href="@{/resources/style.css}"></link>  <-- th:href link to stylesheet
  </head>
  <body>
    <h1>Welcome to Spittr</h1>

    <a th:href="@{/spittles}">Spittles</a> |        <-- th:href links to pages
    <a th:href="@{/spitter/register}">Register</a>
  </body>
</html>
```

Slika 19: Jednostavna stranica u Thymeleaf predlošku

U ovom slučaju, jedina razlika između običnog HTML koda i Thymeleaf-a je `th:href` atribut, koji generira obični `href` atribut, a taj atribut sadržava vrijednost koja je dinamički generirana u vrijeme prikazivanja stranice. Tako radi svaki atribut u Thymeleaf predlošku: svaki dinamički atribut zrcali statički HTML atribut kako bi prikazao dinamičku vrijednost (Walls, 2015). Još jedna novost je izraz `@{/spittles}` koji se koristi za dohvaćane relativnih URL-ova u različitim kontekstima na istom serveru U slučaju JSP-a, link bi bio `<c:url></c:url>`. Razlog zbog kojeg se koristi Thymeleaf umjesto JSP-a je činjenica da je Thymeleaf predložak tipa „Ono-što-se-vidi-je-ono-što-se-dobije”, što znači da nije potrebno dodatno procesiranje koda da bi se prikazali elementi. Primjerice, ako JSP datoteka sadrži

```
<%@taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core"
%>,
```

da nema servlet kontejnera koji prevodi web datoteke, ta bi se linija prikazala i na web stranici u pretraživaču, jer JSP zahtijeva dodatno procesiranje kako bi se prikazali samo HTML elementi, dok sa Thymeleaf predloškom to nije slučaj.

Što se tiče dohvata i prikaza podataka sa kontrolera, najprije je podatke koje želimo prikazati potrebno staviti u `request, session` ili bilo koji drugi kontekst, a potom pomoću ekspresijskog jezika dohvatiti i ispisati.

Primjer dodavanja liste u sesiju izgleda ovako:

```
request.getSession().setAttribute("zahtjevi",
zaposlenik.getZahtjevi());
```

Dohvaćanje i prikaz elemenata liste izgleda ovako:

```
<th:block th:each="zahtjev : ${session.zahtjevi}">
    <tr>
        <td th:text="${zahtjev.tip}"></td>
        <td th:text="${zahtjev.od_datuma}"></td>
        <td th:text="${zahtjev.do_datuma}"></td>
        <td th:text="${zahtjev.broj_radnih_dana}"></td>
        <td th:text="${zahtjev.status_zahtjeva.status}"></td>
    </tr>
</th:block>
```

Slika 20: Prikaz podataka u Thymeleaf predlošku

Element `th:block` služi kao for petlja za prolazak po elementima liste koja se dohvaća iz sesije. Za svaki zahtjev se preko `th:text` atributa dohvaćaju i prikazuju njegovi atributi.

2.4. Osiguravanje web aplikacije

U svakoj poslovnoj aplikaciji postoje informacije koje su povjerljive, primjerice korisničko ime, lozinka, broj računa, i slično. Zbog toga ih je potrebno zaštititi od neovlaštenog pristupa i zloupotrebe.

Spring Security programski okvir je projekt koji pruža sveobuhvatnu zaštitu web aplikacija, autentikaciju i autorizaciju web zahtjeva i pozivanja metoda, kao i enkodiranje podataka.

Prvotno se Spring Security zvao Acegi Security koji je bio vrlo moćan programski okvir, ali je zahtijevao veliku količinu XML konfiguracijskog koda (vrlo često i nekoliko stotina linija koda) (Walls, 2015).

U verziji 2.0 Spring programskog okvira Acegi je postao Spring Security. Ova verzija je donijela novi XML imenski prostor i konfiguracijske anotacije, što je značajno smanjilo broj linija koda od nekoliko stotina na dvadesetak. Sadašnja verzija (3.2) omogućava osiguravanje web zahtjeva, kao i njihovo filtriranje (primjerice, svaki zaposlenik vidi svoj dio aplikacije, ovisno o roli), korištenje *proxy*-a za objekte, i slično.

Spring Security 3.2 se sastoji od 11 modula prikazanih na slijedećoj slici.

ACL	Provides support for domain object security through access control lists (ACLs).
Aspects	A small module providing support for AspectJ-based aspects instead of standard Spring AOP when using Spring Security annotations.
CAS Client	Support for single sign-on authentication using Jasig's Central Authentication Service (CAS).
Configuration	Contains support for configuring Spring Security with XML and Java. (Java configuration support introduced in Spring Security 3.2.)
Core	Provides the essential Spring Security library.
Cryptography	Provides support for encryption and password encoding.
LDAP	Provides support for LDAP-based authentication.
OpenID	Contains support for centralized authentication with OpenID.
Remoting	Provides integration with Spring Remoting.
Tag Library	Spring Security's JSP tag library.
Web	Provides Spring Security's filter-based web security support.

Slika 21: Spring Security moduli

Najčešće korišteni moduli su Core, Configuration i za web aplikacije Cryptography (ako postoji potreba za enkripcijom podataka) i Web moduli.

2.4.1. Konfiguracija

Verzija 3.2 je potpuno ukinula konfiguraciju preko XML-a, što znači da se koriste Java konfiguracijske klase. Slijedeća slika prikazuje najjednostavniji primjer konfiguracijske klase:

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

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

Slika 22: Konfiguracijska klasa za Spring Security

Anotacija `@EnableWebSecurity` omogućava korištenje web sigurnosti u aplikaciji. Konfiguracijska klasa mora implementirati `WebSecurityConfigurer` sučelje ili (zbog jednostavnosti) nasljeđivati klasu `WebSecurityConfigurerAdapter`.

U slučaju nasljeđivanja klase `WebSecurityConfigurerAdapter` nije potrebno implementirati ni prepisati (engl. *override*) metode. Prethodna slika prikazuje sve što je dovoljno za temeljnu konfiguraciju Spring sigurnosti.

U slučaju detaljnije konfiguracije sigurnosti, koristi se prepisivanje metode `configure()` iz klase `WebSecurityConfigurerAdapter`. Metoda može imati tri različita oblika, ovisno o postavkama koje se žele konfigurirati:

- `configure(WebSecurity)` – konfiguracija lanca filtriranja
- `configure(HttpSecurity)` – konfiguracija osiguravanja zahtjeva
- `configure(AuthenticationManagerBuilder)` – konfiguracija postavki za autentikaciju i autorizaciju korisnika (postavke se tiču rola)

Primjer konfiguracije osiguravanja zahtjeva prikazuje slijedeći kod:

```
protected void configure(HttpSecurity http) throws Exception {
  http
    .authorizeRequests()
      .anyRequest().authenticated()
      .and()
    .formLogin().and()
    .httpBasic();
}
```

Slika 23: Konfiguracija osiguravanja zahtjeva

Ovaj primjer prikazuje način osiguravanja HTTP zahtjeva i neke moguće postavke za autentikaciju. Metode `authorizeRequests()`, `anyRequest()` i `authenticated()` definiraju autentikaciju svih HTTP zahtjeva prema aplikaciji. Metode `formLogin()` i `httpBasic()` služe za definiranje autentikacije preko predefinirane forme za prijavu (što znači da nije potrebno kreirati vlastitu). Ova verzija metode `configure()` ne omogućava autentikaciju korisnika u aplikaciju, samo se zahtjevi autenticiraju. Za autentikaciju korisnika je potrebna verzija sa atributom `AuthenticationManagerBuilder`.

Postoje dva pristupa za definiranje postavki korisničke autentikacije: baza u memoriji i klasična baza podataka.

Pristup sa bazom u memoriji se vrlo rijetko koristi jer sve poslovne web aplikacije svoje podatke pohranjuju u klasične baze podataka na diskovima ili serverima. Temeljna metoda korištena u konfiguraciji za memorijsku bazu je metoda `inMemoryAuthentication()`. Slijedeća slika prikazuje cjelokupni konfiguracijski kod za dva korisnika baze:

```
@Configuration
@EnableWebMvcSecurity
public class SecurityConfig extends WebSecurityConfigurerAdapter {

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

}
```

Slika 24: Konfiguracija memorijske baze za dva korisnika

Metoda `withUser()` dodaje novog zaposlenika u bazu sa određenim korisničkim imenom `UserDetailsManagerConfigurer.UserDetailsBuilder`, koja sadrži metode za detaljniju konfiguraciju korisnika, uključujući metodu password(), koja definira lozinku korisnika. Također se mogu dodijeliti role korisnicima, što je vidljivo u primjeru. U daljnjem tekstu su navedene ostale metode za detaljniju konfiguraciju korisnika baze.

- `accountExpired(boolean)` – je li račun istekao ili ne
- `accountLocked(boolean)` – je li račun zaključan ili ne
- `and()` - koristi se za ulančavanje konfiguracija
- `authorities(GrantedAuthority...)` - definira jedno ili više ovlaštenja korisnika
- `authorities(List<? extends GrantedAuthority>)` - definira jedno ili više ovlaštenja korisnika
- `authorities(String...)` - definira jedno ili više ovlaštenja korisnika
- `credentialsExpired(boolean)` – definira jesu li postavke korisnika istekle ili ne
- `disabled(boolean)` – definira je li korisnički račun istekao ili ne

Baze u memoriji mogu biti vrlo korisne kao testne baze kod razvoja aplikacije i testiranja, ali nisu dobre u produkciji. Za to je potrebno konfigurirati klasičnu bazu podataka.

Za konfiguraciju klasične baze podataka potrebna je metoda `jdbcAuthentication()`. Primjer korištenja u konfiguraciji (minimalan kod potreban za konfiguraciju) prikazan je na slijedećoj slici.

```
@Autowired
DataSource dataSource;

@Override
protected void configure(AuthenticationManagerBuilder auth)
                                                    throws Exception {
  auth
    .jdbcAuthentication()
      .dataSource(dataSource);
}
```

Slika 25: Minimalan kod za konfiguraciju klasične baze podataka

Jedino što se mora dodatno konfigurirati za minimalan kod je izvor podataka koji će se koristiti u aplikaciji.

Što se tiče sheme baze podataka, ako se posebno ne specificira, Spring Security ima predefinirane postavke koje se tiču sheme i dizajna baze podataka. Međutim, u stvarnim bazama podataka to gotovo nikad nije slučaj. Zbog toga je potrebno kreirati upite ovisno o shemi baze, što prikazuje slijedeći primjer:

```
@Override
protected void configure(AuthenticationManagerBuilder auth) throws Exception
{
    auth
        .jdbcAuthentication()
        .dataSource(dataSource)
        .usersByUsernameQuery("select username, password, true " +"from Spitter where username=?")
        .authoritiesByUsernameQuery("select username, 'ROLE_USER' from Spitter where username=?");
}
```

Slika 26: Upit za specifičnu shemu baze

Lozinke korisnika su još jedan tip podataka koji se pohranjuju u bazu. Njihov problem je velika mogućnost probijanja ako su pohranjene kao obični tekst. A ako se lozinke enkriptiraju u bazi, korisnik se neće moći prijaviti jer se lozinka upisana kao običan tekst u aplikaciji neće poklapati sa lozinkom u bazi. Za to se u postavkama poziva metoda za enkriptiranje lozinke:

```
@Override
protected void configure(AuthenticationManagerBuilder auth)
                                                    throws Exception {
  auth
    .jdbcAuthentication()
      .dataSource(dataSource)
      .usersByUsernameQuery(
        "select username, password, true " +
        "from Spitter where username=?")
      .authoritiesByUsernameQuery(
        "select username, 'ROLE_USER' from Spitter where username=?")
      .passwordEncoder(new StandardPasswordEncoder("53cr3t"));
}
```

Slika 27: Enkripcija upisane lozinke

Metoda `passwordEncoder()` prihvaća bilo koju implementaciju sučelja `PasswordEncoder` (moguće implementacije su `BCryptPasswordEncoder`, `NoOpPasswordEncoder` i `StandardPasswordEncoder`) (Walls, 2015). Također je moguće kreirati i vlastitu implementaciju, ako navedene tri ne zadovoljavaju potrebe aplikacije. `PasswordEncoder` sučelje ima samo dvije metode: `String encode(CharSequence rawPassword);` i `boolean matches(CharSequence rawPassword, String encodedPassword);`.

Bitno je shvatiti da se lozinka u bazi nikad ne dekodira, nego se lozinka upisana od korisnika kodira istim algoritmom kao i ona u bazi i s njom se uspoređuje. Za uspoređivanje se koristi metoda `matches()`.

3. Aplikacija

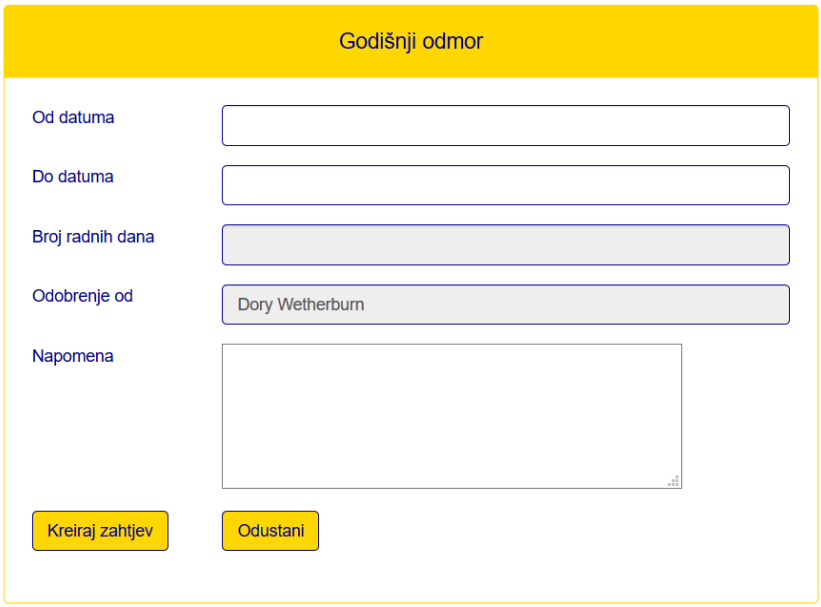
3.1. Opis aplikacije

Aplikacija izrađena za potrebe ovog rada je aplikacija za evidenciju korištenja dana godišnjih odmora i plaćenih dopusta zaposlenika neke firme. Aplikacija se sastoji od dva osnovna dijela: modula za kreiranje zahtjeva za godišnje odmore i plaćene dopuste i rezervacije hotelske sobe u odabranom gradu. U aplikaciji je također implementirana mogućnost kreiranja Excel izvješća (samo rukovoditelji odjela imaju tu mogućnost). Aplikaciji se pristupa preko web forme za prijavu upisom korisničkog imena i lozinke, što je prikazano na slijedećoj slici.

Slika 28: Prijava u aplikaciju

Ako zaposlenik nema korisničko ime i lozinku, klikne na gumb za registraciju (koji se nalazi na istoj formi za prijavu) i aplikacija ga preusmjeri na formu za registraciju. Nakon uspješne registracije prikaže mu se njegov profil.

Nakon uspješne prijave, prikaže se profil zaposlenika gdje isti može kreirati zahtjeve za godišnje odmore, plaćene dopuste i pregledavati već kreirane zahtjeve. Ako je prijavljeni zaposlenik rukovoditelj odjela, njemu je prikazan popis zahtjeva svih zaposlenika koje može odobriti ili odbiti klikom na određeni gumb. Klikom na gumb za novi zahtjev za godišnji odmor ili plaćeni dopusti prikaže se forma za unos podataka. Svaki zahtjev ima svoju formu, što prikazuju slike 29 i 30:



Slika 29: Forma za kreiranje zahtjeva za godišnji odmor

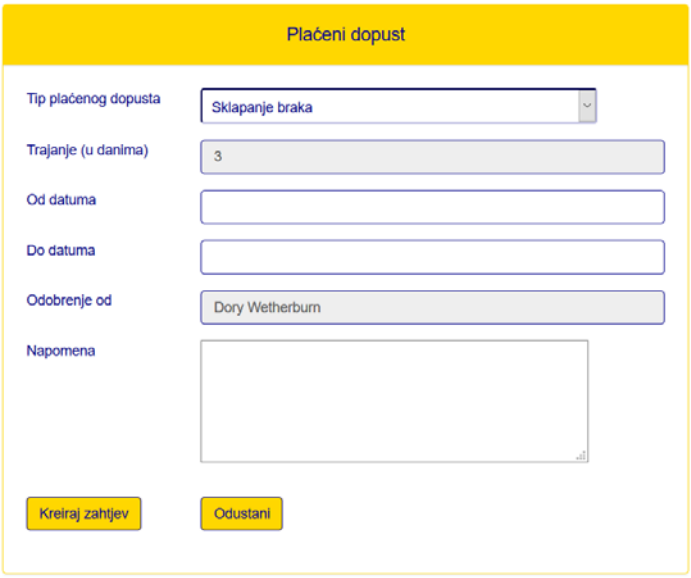
Podaci koji se unose su datum početka i datum završetka godišnjeg odmora (datumi se unose preko jQuery datepicker kalendara). Za izračun i prikaz broja radnih dana se koristi jQuery. Što se tiče odobravanja zahtjeva, definiran je jedan zaposlenik u bazi koji rukovodi svim zahtjevima. Datumi početka i završetka godišnjeg odmora su obavezni, pa se u slučaju neunošenja jednog ili oba datuma prikaže poruka.

Kreirani zahtjev za godišnji odmor smije imati minimalno trajanje 20, a maksimalno 30 dana. Ako se odabere raspon datuma koji ima više, zaposleniku se prikaže poruka da odabere drugi raspon datuma.

Ako su svi uvjeti forme zadovoljeni, klikom na gumb „Kreiraj zahtjev“ novi zahtjev se kreira i prikazuje se na profilu zaposlenika, što prikazuje slika 30.

Slika 30: Popis kreiranih zahtjeva

Forma za kreiranje plaćeni dopusta radi na istom principu, ali se neka polja za unos razlikuju, što je vidljivo na slici 31:



Slika 31: Forma za kreiranje plaćenih dopusta

Vidljivo je da zaposlenik može birati više tipova plaćenog dopusta, od kojih svaki ima različito trajanje:

- Sklapanje braka – 3 dana
- Porod supruge – 2 dana
- Smrt člana uže obitelji – 4 dana
- Smrt roditelja supružnika – 2 dana
- Selidba u drugo mjesto – 3 dana
- Selidba u istom mjestu – 2 dana
- Elementarna nepogoda u domaćinstvu radnika – 3 dana
- Teža bolest člana uže obitelji izvan mjesta stanovanja – 2
- Dobrovoljno davanje krvi – 1 dan
- Za stipendiste za pripremu doktorske disertacije – 7 dana

Ovisno o odabiru tipa plaćenog dopusta, mijenja se i iznos dana na formi. Ako zaposlenik odabere raspon datuma koji ima više ili manje dana od definiranog, prikaže se poruka za odabirom drugog raspona. I ovdje su svi podaci obavezni osim napomene. Ako su svi uvjeti forme zadovoljeni, kreira se zahtjev i prikaže se na profilu.

Klikom na gumb za izradu izvješća otvara se nova forma gdje se mogu kreirati izvješća o godišnjim odmorima, plaćenim dopustima i naknadama za korištenje godišnjim odmora (regresa). Kreirana izvješća se preuzimaju sa aplikacije klikom na link, što prikazuje slijedeća slika.

Slika 32. Kreiranje i prikaz izvješća

Klikom na određeni gumb kreira se izvješće koje se pohrani u mapi na računalu, te se preko aplikacije dohvati i prikaže putanje kreiranog izvješća. Klikom na njegov naziv izvješće se preuzme iz aplikacije. (Napomena: desno od plaćenih dopusta se nalazi gumb za kreiranje izvješća regresa godišnjih odmora koje radi na istom principu, ali su prikazana samo dva gumba jer bi uz treći slika bila premala).

Gumb za rezervaciju sobe se nalazi u zaglavlju, uz osobne podatke zaposlenika. Klikom na gumb aplikacija se preusmjerava na formu sa popisom gradova i njima pripadajućih hotela. Ovisno o odabranom gradu prikazuju se hoteli u kojima je moguće rezervirati sobu, što se može vidjeti na slijedećoj slici.

Slika 33: Popis hotela za odabrani grad

Odabirom željenog hotela aplikacija se preusmjerava na formu za rezervaciju gdje se upisuju datumi prijave i odjave iz hotela. Ti podaci su obavezni i ako se ne upiše jedan od njih, korisniku se prikaže poruka. Zaposlenik, odabrani grad i naziv hotela se ne moraju posebno upisivati jer su dohvaćeni iz prethodnih koraka rezervacije. Klikom na gumb obavlja se rezervacija sobe. Na istoj je formi moguće vidjeti prethodno kreirane rezervacije.

Slike 34, 35 i 36 to prikazuju.

Slika 34: Podaci rezervacije dohvaćeni u prethodnim koracima

Slika 35: Forma za upis datuma prijave i odjave iz hotela

Slika 36: Popis svih rezervacija

3.2. Baza podataka

Za izradu baze podataka korišten je Microsoft SQL Server 2016. Baza podataka kreirana za završni rad se zove GodisnjiOdmorPlaceniDopust.mdf. i sastoji se od slijedećih tablica:

- **dijete** – sadrži stupac za primarni ključ (id_dijete), starost djeteta i strani ključ na tablicu zaposlenik (zaposlenik_id). Starost djeteta je potrebna za definiranje dodatnih dana godišnjeg odmora zaposlenika.
- **grad** – sastoji se od stupca za primarni ključ (id_grad) i naziva grada.
- **hotel** – sastoji se od stupca identifikatora (id_hotel), naziva, broja zvjezdica, detaljnog opisa hotela i strano ključa na tablicu grad.
- **organizacijska_jedinica** – sadrži stupac za primarni ključ (id_organizacijska_jedinica) i naziv organizacijske jedinice.
- **placeni_dopust** – tablica sadrži tipove plaćenih dopusta. Sastoji se od stupca primarnog ključa (id_placeni_dopust), tipa i trajanja plaćenih dopusta.
- **rezervacija** – sadrži podatke o kreiranim rezervacijama. Sadrži stupac za primarne ključeve (id_rezervacija), datume prijave i odjave i strane ključeve na tablice **zaposlenik i hotel**
- **rola** – sastoji se od stupca primarnog ključa (id_rola) i naziva role.
- **status_zahtjeva** – sadrži statuse kreiranih zahtjeva (Zaprimljen, Odobren Odbijen). Sastoji se od stupca primarnog ključa (id_status_zahtjeva) i vrijednosti statusa.
- **zahtjev** – sadrži podatke o kreiranom zahtjevu za godišnji odmor ili plaćeni dopust. Sastoji se od stupca primarnog ključa (id_zahtjev), datuma početka i završetka odmora/dopusta, broja radnih dana, imena i prezimena osobe koja odobrava zahtjev, napomene i stranih ključeva na tablice **zaposlenik**, **placeni_dopust** i **status_zahtjeva**.
- **zaposlenik** – sadrži podatke zaposlenika. Sastoji se od stupca primarnog ključa (id_zaposlenik), imena, prezimena, e-mail adrese, korisničkog imena, lozinke, matičnog broja, datuma zaposlenja, stupca za tjelesno oštećenje ili invalidnost (moguće vrijednosti su NE, Tjelesno oštećenje, Invalidnost), godine staža, broja djece i stranih ključeva na tablice **rola** i **organizacijska_jedinica**

3.3. Poslovna logika

3.3.1. Konfiguracija

Da bi aplikacija mogla raditi sa podacima, potrebno ju je povezati na bazu. U Spring programskom okviru povezivanje aplikacije s bazom se obavlja preko konfiguracijske datoteke `application.config`. Slijedeći kod je potreban za povezivanje aplikacije i baze:

```
spring.datasource.url=
jdbc:sqlserver://localhost:1433;databaseName=GodisnjiOdmorPla
ceniDopust
spring.datasource.username= sa
spring.datasource.password= SQL
spring.jpa.database-
platform=org.hibernate.dialect.SQLServerDialect
```

Kod je vrlo jednostavan, potreban je URL do baze na sa kojom se želi raditi, korisničko ime i lozinka koji se spajamo na bazu i platforma baze. Platforma baze označava tip relacijskog sustava upravljanja bazom podataka (RDBMS, engl. *Relational Database Management System*); to je potrebno da bi Hibernate mogao komunicirati s bazom. Hibernate podržava sve poznatije RDBMS sustave: SQL Server, Oracle, PostgreSQL, MySQL, DB2.

Također je konfigurirano slanje e-mail poruka, što predstavlja slijedeći kod:

```
spring.mail.host: smtp.gmail.com
spring.mail.port: 465
spring.mail.username: dorywetherburn@gmail.com
spring.mail.password: dwetherburn123
spring.mail.properties.mail.smtp.auth: true
spring.mail.properties.mail.smtp.starttls.enable: true
spring.mail.properties.mail.smtp.starttls.required: true
spring.mail.properties.mail.smtp.ssl.enable = true
spring.mail.test-connection=true
```

U aplikaciji je konfiguriran Gmail servis, iako se može konfigurirati bilo koji. Da bi se e-mail poruke mogle uspješno slati, potrebno je konfigurirati host preko kojeg se šalje, port i korisničko ime i lozinka na koje se šalju poruke (poruke se šalju rukovoditelju koji odobrava ili odbija zahtjeve). Ostatak koda se odnosi na sigurnost konekcije i prijave.

Svaki prijavljeni korisnik u aplikaciji se pohranjuje u sesiju da bi mogao biti vidljiv u dometu cijele aplikacije. Tip pohrane i trajanje sesije se također podešava u application.config datoteci, što prikazuje slijedeći kod:

```
spring.session.store-type=HASH_MAP
server.session.timeout=1800
```

Zbog jednostavnosti odabrana je pohrana u `HashMap` (druge moguće opcije su JDBC, MongoDB, Redis i Hazelcast). Trajanje sesije je definirano u sekundama.

3.3.2. Rad s podacima

U web aplikaciji rad s podacima obavljaju kontroleri. Oni izvršavaju repozitorijske metode za obradu podataka i obrađene podatke šalju prezentacijskom sloju na ispis. Kontrolerske metode moraju imati `@GetMapping` ili `@PostMapping` anotacije, ovisno o funkciji koju obavljaju. `@GetMapping` se koristi kada se web stranica treba samo renderirati, a u slučaju prihvaćanja i rada s podacima iz druge forme i renderiranja nove stranice, metoda se piše dva puta (prvi put sa `@GetMapping`, a drugi sa `@PostMapping` anotacijom). Slijedeći kodovi prikazuju kako se to radi u praksi.

```
<form id="home" th:action="@{/profil-zaposlenika}" method="post">
    <table class="table table-borderless table-responsive">
        <tr>
            <td>Korisničko ime</td>
            <td>
                <input type="text" class="form-control" id="korisnickoIme" name="korisnickoIme" autofocus="autofocus"/>
            </td>
            <td th:text="${kriviPodaci}"></td>
        </tr>
        <tr>
            <td>Lozinka</td>
            <td>
                <input type="password" class="form-control" name="lozinka" id="lozinka"/>
            </td>
        </tr>
        <tr>
            <td></td>
            <td>
                <input type="submit" id="btnPrijava" class="btn btn-default btnPrijava" value="Prijavite se"/>
            </td>
        </tr>
    </table>
</form>
```

Slika 37: Primjer forme koja šalje upisane podatke

Za primjer je uzeta forma za prijavu u aplikaciju. Forma prihvaća korisničko ime i lozinku koje šalje u kontrolersku metodu `profil-zaposlenika` gdje se upisani podaci uspoređuju sa onima dohvaćenim iz baze i ako se podudaraju, aplikacija se preusmjerava na profil korisnika, što se vidi na slikama 38 i 39.

```
@GetMapping(value = "/profil-zaposlenika")
public String profilZaposlenika(Model model)
{
    List<PlaceniDopust> tipoviPlacenogDopusta = repozitorijGlavnaAplikacija.dohvatiTipovePlacenihDopusta();
    model.addAttribute("tipoviPlacenogDopusta", tipoviPlacenogDopusta);
    return "profilZaposlenika";
}
```

Slika 39: Prikaz profila zaposlenika - GET metoda

Ova metoda mora biti tipa GET i POST. Najprije se implementira metoda tipa GET koja iz repozitorija dohvaća podatke, tj. tipove plaćenih dopusta koji su potrebni kod kreiranja zahtjeva za plaćene dopuste. Tipovi plaćenih dopusta se spremaju u model iz kojeg se na stranici dohvaćaju i ispisuju. U slučaju samog ispisa podataka, GET metoda je dovoljna. Međutim, kako se radi sa podacima primljenim sa druge forme, potrebna je i POST metoda, što prikazuje slijedeća slika:

```
@PostMapping(value = "/profil-zaposlenika")
public String profilZaposlenika(Model model, HttpServletRequest request)
{
    List<Zahtjev> sviZahtjevi = repozitorijGlavnaAplikacija.dohvatiSveZahtjeve();
    List<PlaceniDopust> tipoviPlacenogDopusta = repozitorijGlavnaAplikacija.dohvatiTipovePlacenihDopusta();
    String korisnickoIme = request.getParameter("korisnickoIme");
    String lozinka = request.getParameter("lozinka");
    BCryptPasswordEncoder encoder = new BCryptPasswordEncoder(12);
    Zaposlenik zaposlenik = repozitorijGlavnaAplikacija.dohvatiZaposlenika(korisnickoIme).get(0);

    if(zaposlenik.getKorisnicko_ime().equals(korisnickoIme) && encoder.matches(lozinka, zaposlenik.getLozinka()))
    {
        request.getSession().setAttribute("zaposlenik", zaposlenik);
        model.addAttribute("zahtjevi", zaposlenik.getZahtjevi());
        model.addAttribute("sviZahtjevi", sviZahtjevi);
        model.addAttribute("tipoviPlacenogDopusta", tipoviPlacenogDopusta);
        return "profilZaposlenika";
    }

    model.addAttribute("kriviPodaci","Korisničko ime ili lozinka nisu ispravni");
    return "home";
}
```

Slika 38: Prikaz profila zaposlenika - POST metoda

Iz baze se najprije dohvaćaju svi potrebni podaci koji će se prikazati na profilu korisnika (svi kreirani zahtjevi se dohvaćaju u slučaju prijave osobe koja obavlja odobravanje i obijanje zahtjeva). Potom se preko `request` parametra dohvaćaju korisničko ime i lozinka upisani na formi za prijavu. Instanca klase `BCryptPasswordEncoder` služi za uspoređivanje upisanih korisničkih podataka i onih iz baze. Iz baze se dohvaća zaposlenik koji ima upisano korisničko ime i kreira se instanca klase `Zaposlenik`. Uspoređuju se korisnički podaci sa podacima iz baze i ako se podudaraju, u sesiju se dodaje kreirani zaposlenik, u model njegovi kreirani zahtjevi i tipovi plaćenog dopusta potrebni za kreiranje zahtjeva i renderira se pogled `profilZaposlenika`. Ako se podaci ne podudaraju, ispisuje se poruka da su podaci krivi i ne preusmjerava se na profil, nego se opet prikaže stranica za prijavu.

Nakon uspješne prijave, korisnik može kreirati zahtjeve za godišnje odmore i plaćene dopuste. Kreiranje zahtjeva radi na istom principu kao i prijava: preko formi u koje se upisuju podaci, nakon čega se aplikacija preusmjerava na kontrolerske metode koje primaju upisane podatke i upisuju ih u bazu. Osim unosa u bazu, također šalju i e-mail poruke rukovoditelju koji odobrava ili odbija zahtjeve. Repozitorijske metode za upis zahtjeva i slanje e-mail poruka rukovoditelju su prikazane slikama 40 i 41.

```
public void dodajZahtjev(Zahtjev zahtjev, Zaposlenik zaposlenik)
{
    StatusZahtjeva statusZahtjeva = new StatusZahtjeva();
    statusZahtjeva.setId_status_zahtjeva(1);
    statusZahtjeva.setStatus("Zaprimljen");

    zahtjev.setZaposlenik(zaposlenik);
    zahtjev.setStatus_zahtjeva(statusZahtjeva);
    entityManager.persist(zahtjev);
}
```

Slika 40: Kreiranje zahtjeva - repozitorij

Ova metoda prima dva parametra: zahtjev kreiran u kontrolerskoj metodi od podataka dohvaćenih sa forme i zaposlenika prijavljenog u aplikaciji. Instanca klase `Zahtjev` je već kreirana u kontroleru sa upisanim podacima, međutim je tablica **zahtjev** u bazi povezana sa tablicama **status_zahtjeva** i **zaposlenik**, pa je zato zahtjevu potrebno dodati i status zahtjeva i zaposlenika na kojeg se zahtjev odnosi. Nakon toga zahtjev se može spremiti u bazu.

U kontroleru se uz odmah nakon kreiranja zahtjeva šalje e-mail poruka rukovoditelju da je zahtjev kreiran. Repozitorijske metoda koja kreira i šalje e-mail poruku prikazana je na slici 41:

```
public void posaljiMailRukovoditelju(Zahtjev zahtjev, Zaposlenik zaposlenik)
{
    SimpleMailMessage message = new SimpleMailMessage();
    message.setTo("dorywetherburn@gmail.com");
    message.setFrom(zaposlenik.getEmail());
    message.setSubject(zahtjev.getTip());
    message.setText("Zaposlenik: " + zaposlenik.getIme() + " " + zaposlenik.getPrezime() +
            "\nTip zahtjeva: " + zahtjev.getTip() + "\nOd datuma: " +
    zahtjev.getOd_datuma() + "\nDo datuma: " + zahtjev.getDo_datuma());
    mailSender.send(message);
}
```

Slika 41: Slanje e-mail poruke

Ova metoda također kao argument prima kreirani zahtjev i zaposlenika na kojeg se zahtjev odnosi. Potrebno je kreirati e-mail adresu primatelja, pošiljatelja, naslov i tekst poruke (šalju se podaci o zahtjevu i zaposleniku koji ga je kreirao).

Kreirani zahtjevi se mogu odobriti ili odbiti. Tu funkciju ima samo jedna osoba, i ako se ona prijavi u aplikaciju, na njenom profilu se prikaže popis svih zahtjeva. U slučaju prijave bilo koje druge osobe, taj dio aplikacije je skriven. Prikaz svih zahtjeva je prikazan na slijedećoj slici:

```
<th:block th:each="zahtjev : ${sviZahtjevi}">
    <tr>
        <td th:text="${zahtjev.tip}"></td>
        <td th:text="${zahtjev.od_datuma}"></td>
        <td th:text="${zahtjev.do_datuma}"></td>
        <td th:text="${zahtjev.broj_radnih_dana}"></td>
        <td th:text="${zahtjev.zaposlenik.ime + ' ' + zahtjev.zaposlenik.prezime}"></td>
        <td th:text="${zahtjev.status_zahtjeva.status}"></td>
        <td>
            <form th:action="@{/odobri-zahtjev}" method="post">
                <input type="hidden" id="idZahtjev" name="idZahtjev" th:value="${zahtjev.id_zahtjev}" />
                <input type="submit" class="btn btn-default" value="Odobri zahtjev"/>
            </form>

        </td>
        <td>
            <form th:action="@{/odbij-zahtjev}" method="post">
                <input type="hidden" class="btn btn-default" id="idZahtjev" name="idZahtjev" th:value="${zahtjev.id_zahtjev}" />
                <input type="submit" class="btn btn-default btn-odbij-zahtjev" value="Odbij zahtjev"/>
            </form>
        </td>
    </tr>
</th:block>
```

Slika 42: Prikaz svih zahtjeva na profilu

Ideja je da se iz kontrolera dohvate svi zahtjevi i preko ekspresijskog jezika u Thymeleaf predlošku prolazi po listi zahtjeva i prikaže iste na stranici. Atribut `th:each` služi kao petlja za prolaz. Za svaki zahtjev u listi kreiraju se forme za odobravanje ili odbijanje koje prima identifikator zahtjeva i preusmjeravaju na kontrolersku metodu koja obavlja odobravanje ili odbijanje. Identifikator zahtjeva je definiran kao *hidden* polje jer se ne treba prikazivati na stranici i šalje se kao parametar u metodu. Kontrolerska metoda za odobravanje zahtjeva prikazana je na slijedećoj slici:

```
@PostMapping(value = "/odobri-zahtjev")
public String odobriZahtjev(HttpServletRequest request, Model model)
{
    int idZahtjev = Integer.parseInt(request.getParameter("idZahtjev"));
    repozitorijGlavnaAplikacija.odobriZahtjev(idZahtjev);
    List<Zahtjev> sviZahtjevi = repozitorijGlavnaAplikacija.dohvatiSveZahtjeve();
    model.addAttribute("sviZahtjevi", sviZahtjevi);
    return "profilZaposlenika";
}
```

Slika 43: Odobravanje zahtjeva - kontrolerska metoda

Dohvaća se identifikator odabranog zahtjeva, poziva se repozitorijska metoda koja radi stvarno odobravanje, dohvaća se ažurirani popis zahtjeva iz baze i isti se prikazuje na stranici. Implementacija repozitorijske metode prikazana je na slici 44.

```
public void odobriZahtjev(int idZahtjev)
{
    Zahtjev zahtjev = entityManager.find(Zahtjev.class, idZahtjev);
    zahtjev.setStatus_zahtjeva(entityManager.getReference(StatusZahtjeva.class, 2));
    entityManager.merge(zahtjev);
}
```

Slika 44: Odobravanje zahtjeva - repozitorijska metoda

Najprije se iz baze dohvati odabrani zahtjev preko njegovog identifikatora. Zatim se njegov status postavi na **Odobren** (dohvati se referenca na klasu koja je mapirana na tablicu **status_zahtjeva** i identifikator statusa se postavi na 2 – identifikator odobrenog statusa u bazi). Metoda `merge(zahtjev)` služi za ažuriranje zahtjeva. Metode za odbijanje zahtjeva rade na potpuno istom principu, jedino se u repozitorijskoj metodi identifikator statusa postavlja na 3.

Kreirana izvješća se spremaju u lokalnu mapu GitHub repozitorija, a također se prikazuju i u aplikaciji kao linkovi, preko kojih ih je moguće preuzeti. Izvješća se kreiraju kao Excel datoteke.

Da bi se mogla kreirati Excel izvješća, potrebno je sa Maven repozitorija preuzeti Apache POI *library* i referencirati ga preko pom.xml datoteke. Kod koji to omogućava je slijedeći:

```
<dependency>
      <groupId>org.apache.poi</groupId>
      <artifactId>poi</artifactId>
      <version>3.17</version>
</dependency>
```

Za kreiranje izvješća potrebno je kreirati radnu knjigu i naslovne ćelije (naslovne ćelije nisu obavezne, ali ih ima svaki dokument):

```
HSSFWorkbook workbook = new HSSFWorkbook();
HSSFSheet sheet = workbook.createSheet("Iznos regresa za godišnji odmor");

Row redakNasloviStupaca = sheet.createRow(0);
redakNasloviStupaca.createCell(0).setCellValue("Organizacijska jedinica");
redakNasloviStupaca.createCell(1).setCellValue("Zaposlenik");
redakNasloviStupaca.createCell(2).setCellValue("Maticni broj zaposlenika");
redakNasloviStupaca.createCell(3).setCellValue("Iznos regresa za godišnji odmor");
```

Slika 45: Kreiranje radne knjige i naslovnih ćelija

Nakon toga je potrebno dohvatiti podatke i dodati ih dokumentu. Podaci korišteni u izradi izvješća se nalaze u različitim tablicama povezanim stranim ključevima, pa se u aplikaciji entiteti pridruženi tim tablicama spajaju preko for petlji kako bi se dohvatili željeni podaci. Slijedeća slika je dio koda repozitorijske metode koja kreira izvješća:

```
int i = 1;
List<OrganizacijskaJedinica> organizacijskeJedinice = dohvatiOrganizacijskeJedinice();

for(OrganizacijskaJedinica organizacijskaJedinica : organizacijskeJedinice)
{
    for(Zaposlenik zaposlenik : organizacijskaJedinica.getZaposlenici())
    {
        for(Zahtjev zahtjev : zaposlenik.getZahtjevi())
        {
            Row redak = sheet.createRow(i);
            redak.createCell(0).setCellValue(organizacijskaJedinica.getNaziv());
            redak.createCell(1).setCellValue(zaposlenik.getIme() +" " + zaposlenik.getPrezime());
            redak.createCell(2).setCellValue(zaposlenik.getMaticni_broj());
            redak.createCell(3).setCellValue(zahtjev.getBroj_radnih_dana() * 3000);
            i++;
        }
    }
}
```

Slika 46: Kaskadno dohvaćanje podataka

Kontrolersku metodu nema potrebe pisati jer je ista kao i ostale, samo poziva repozitorijsku metodu. Podaci se dohvaćaju kaskadno, prema shemi baze podataka, i u izvješću se kreiraju međusobno povezani redci. Da bi ovakav način dohvata podataka bio moguć, veza između entiteta i tablica mora biti 1:M (jedna organizacijska jedinica može imati više zaposlenika, jedan zaposlenik može kreirati više zahtjeva i slično).

Na kraju je potrebno spremiti dokument na disk i zatvoriti radnu knjigu u aplikaciji. Slijedeća slika to prikazuje:

```
try
{
    workbook.write(new File("C:/Users/Ivan/Documents/GitHub/Izvjesca/IznosiRegresa.xls"));
    workbook.close();
}

catch (IOException e)
{
    e.printStackTrace();
}
```

Slika 47: Spremanje i zatvaranje dokumenta

Slijedeći korak je prikaz kreiranih izvješća na formi u obliku linkova za preuzimanje. Primjer takvog linka prikazuje slijedeći kod:

```
<a class="linkIzvjesce" th:text="${izvjesceGodisnjiOdmori}"
href="preuzmi-izvjesce-godisnjih-odmora">
```

Vrijednost atributa `th:text` na formi prikazuje naziv kreiranog dokumenta. Ta je vrijednost atributa dohvaćena preko kontrolerske metode koja dohvaća kreirani dokument iz mape na računalu. Implementacija metode se može vidjeti na slijedećoj slici:

```
private void prikaziIzvjesceGodisnjihOdmora(Model model)
{
    File folder = new File("C:/Users/Ivan/Documents/GitHub/Izvjesca");
    String[] datoteke = folder.list();
    String izvjesceGodisnjiOdmori = null;

    for(String datoteka : datoteke)
    {
        if(datoteka.equals("GodisnjiOdmori.xls"))
        {
            izvjesceGodisnjiOdmori = datoteka;
        }
    }

    model.addAttribute("izvjesceGodisnjiOdmori", izvjesceGodisnjiOdmori);
}
```

Slika 48: Kontrolerska metoda za dohvat i prikaz izvješća

Metoda dohvaća polje svih dokumenata u mapi, iterira po njemu i ako pronađe željeni (u ovom slučaju izvješće godišnjih odmora), postavi ga u model gdje ga Thymeleaf forma može pronaći i ispisati.

Da bi bilo moguće preuzeti datoteku sa forme, ta ista forma mora sadržavati link koji sadržava putanju do mjesta na kojem je datoteka pohranjena. Atribut `href` u linku u kodu na početku stranice vodi do metode koja implementira funkciju preuzimanja datoteke, što se može vidjeti na slijedećoj slici:

```java
@GetMapping(value = "/preuzmi-izvjesce-godisnjih-odmora")
public void preuzmiIzvjesceGodisnjihOdmora(HttpServletRequest request, HttpServletResponse response)
{
    File izvjesce =  new File("C:/Users/Ivan/Documents/GitHub/Izvjesca/GodisnjiOdmori.xls");

    try
    {
        InputStream inputStream = new BufferedInputStream(new FileInputStream(izvjesce));
        String mimeType = URLConnection.guessContentTypeFromStream(inputStream);

        if(mimeType == null)
        {
            mimeType = "application/octet-stream";
        }

        response.setContentType(mimeType);
        response.setContentLength((int) izvjesce.length());
        response.setHeader("Content-Disposition", String.format("attachment; filename=\"%s\"", izvjesce.getName()));
        FileCopyUtils.copy(inputStream, response.getOutputStream());
    }
    catch (IOException e)
    {
        e.printStackTrace();
    }
}
```

Slika 49: Funkcija za preuzimanje generiranog izvješća

Metoda `response.setHeader();` postavlja željenu datoteku u zaglavlje stranice kao prilog (engl. *attachment*), a preuzimanje datoteke obavlja metoda `response.setHeader()`.

Metode izrade, prikaza i preuzimanja izvješća je isti i za preostala izvješća, nije ih potrebno dodatno pojašnjavati.

Pristup modulu za rezervaciju hotelske sob odvija se klikom na odgovarajući gumb na profilu zaposlenika. Time se aplikacije preusmjerava na stranicu sa popisom gradova i njima pripadajućih hotela (poslovna logika za ovaj modul odrađena je posebnim kontrolerom).

Kontrolerska metoda `homeRezervacija()` iz baze dohvaća popise gradova i njima pripadajućih hotela i prikazuje njih i stranicu, što se može vidjeti na slijedećoj slici:

```java
@GetMapping(value = "/home-rezervacija")
public String homeRezervacija(Model model)
{
    List<Grad> gradovi = repozitorijRezervacija.dohvatiGradove();
    List<Hotel> hoteli = repozitorijRezervacija.dohvatiHotele("Crikvenica");
    model.addAttribute("gradovi", gradovi);
    model.addAttribute("hoteli", hoteli);
    model.addAttribute("odabraniGrad", "Crikvenica");

    return "homeRezervacija";
}
```

Slika 50: Kontrolerska metoda za prikaz gradova i hotela

Stranica osvježuje prikaz hotela, ovisno o odabranom gradu. Za to je zaslužna posebna metoda `hoteli()` koja prikazuje popis gradova, hotela i u zaglavlju forme naziv odabranog grada. To je također kontrolerska metoda kojoj se pristupa preko veze na stranici:

```
<form th:action="@{/hoteli}" method="post">
      <input type="hidden" id="odabraniGrad"
      name="odabraniGrad" th:value="${grad.naziv}" />
      <input type="submit" class="btn btn-default btnGradovi"
      th:value="${grad.naziv}"/>
</form>
```

Ova se forma nalazi unutar `<th:block></th:block>` elementa koji dohvaća popis gradova i za svaki grad u `<form></form>` preko metode `hoteli()` prikazuje potrebne podatke. Implementacija te metode se može vidjeti u nastavku.

```
@PostMapping(value = "/hoteli")
public String hoteli(Model model, HttpServletRequest request)
{
    String odabraniGrad = request.getParameter("odabraniGrad");
    List<Grad> gradovi = repozitorijRezervacija.dohvatiGradove();
    List<Hotel> hoteli = repozitorijRezervacija.dohvatiHotele(odabraniGrad);
    model.addAttribute("odabraniGrad", odabraniGrad);
    model.addAttribute("gradovi", gradovi);
    model.addAttribute("hoteli", hoteli);

    return "homeRezervacija";
}
```

Slika 51: Implementacija metode `hoteli()`

Iz forme se preko *hidden* polja dohvaća identifikator odabranog grada (u ovom slučaju to je naziv grada - `th:value="${grad.naziv}"`) i njime se iz repozitorija dohvaćaju svi hoteli koji mu pripadaju. U model se spremaju popis gradova, hotela i odabrani grad i isti se prikazuju na stranici.

Nakon odabira željenog hotela, aplikacija se preusmjerava na formu za rezervaciju. Forma je implementirana kao i svaka druga forma za unos podataka u aplikaciju: traži se upis datuma prijave i odjave iz hotela (datumi su obavezni, a podaci o zaposleniku, gradu i hotelu se dohvaćaju sa prethodne stranice) i nakon njihovog unosa zahtjev se kreira. Njega je moguće vidjeti na stranici.

Implementacija metode za dohvat podataka sa prethodne stranice može se vidjeti na slijedećoj slici.

```
@PostMapping(value = "/rezervacija")
public String rezervacija(Model model, HttpServletRequest request)
{
    Zaposlenik zaposlenik = (Zaposlenik) request.getSession().getAttribute("zaposlenik");
    String nazivHotela = request.getParameter("nazivHotela");
    Hotel odabraniHotel = repozitorijRezervacija.dohvatiHotel(nazivHotela);
    List<Rezervacija> rezervacije = repozitorijRezervacija.dohvatiRezervacije(zaposlenik);

    model.addAttribute("rezervacije", rezervacije);
    model.addAttribute("odabraniHotel", odabraniHotel);
    request.getSession().setAttribute("odabraniHotel", odabraniHotel);

    return "rezervacija";
}
```

Slika 52: Metoda za dohvat podataka sa prethodne stranice

Potrebni podaci su ime i prezime zaposlenika, grad i hotel u kojem se želi rezervirati soba. Grad se dohvaća preko odabranog hotela jer su ti entiteti u tablicama međusobno povezani, što se radi ovako:

```
<td th:text="${session.odabraniHotel.grad.naziv}"></td>
```

Metoda za kreiranje rezervacije je ista kao i svaka druga. Njena implementacija je vidljiva na slijedećoj slici:

```
@PostMapping(value = "/rezerviraj")
public String rezerviraj(Model model, HttpServletRequest request)
{
    Zaposlenik zaposlenik = (Zaposlenik) request.getSession().getAttribute("zaposlenik");
    Hotel hotel = (Hotel) request.getSession().getAttribute("odabraniHotel");
    String datumPrijave = request.getParameter("datum_prijave");
    String datumOdjave = request.getParameter("datum_odjave");


    Rezervacija rezervacija = new Rezervacija();
    rezervacija.setDatum_prijave(datumPrijave);
    rezervacija.setDatum_odjave(datumOdjave);
    rezervacija.setZaposlenik(zaposlenik);
    rezervacija.setHotel(hotel);
    repozitorijRezervacija.rezervirajSobu(rezervacija);

    List<Rezervacija> rezervacije = repozitorijRezervacija.dohvatiRezervacije(zaposlenik);
    model.addAttribute("rezervacije", rezervacije);

    return "rezervacija";
}
```

Slika 53: Metoda za kreiranje rezervacije

Osim što kreira rezervaciju, metoda također dohvaća kreirane rezervacije iz baze i prikazuje ih na stranici.

Zaključak

U završnom radu prikazana je primjena Spring i Hibernate tehnologija u izradi poslovnih web aplikacija, točnije aplikacije za evidenciju korištenja dana godišnjih odmora i plaćenih dopusta zaposlenika neke tvrtke. Za izradu poslovne logike se koristio Spring Boot pomoću kojeg se pristupalo ostalim potrebnim modulima (Spring Security, Spring Data, Spring MVC, ...). Za pristup i obradu podataka su se kombinirale JPA i Hibernate HQL tehnologije, ovisno o potrebi, kako bi se dobile optimalne performanse aplikacije i što bolja razumljivost koda. Na klijentskoj strani obrađeni podaci su se prikazivali korištenjem Thymeleaf predloška.

Cilj rada je bio detaljno objašnjenje tehnologija koje se koriste u razvoju Java web aplikacija i njihova praktična primjena na stvarnoj aplikaciji. Što se tiče same složenosti tehnologija korištenih u radu, ideje i principi rada su isti kao kod .NET tehnologija, međutim postoji nekoliko koncepata koji se razlikuju, primjerice Maven repozitorij, podjela poslovnih domena u različite Spring projekte,… Koncept koji se vrlo razlikuje od .NET svijeta su konfiguracijske datoteke. Ovisno o tipu projekta koji se radi, konfiguracijske datoteke mogu biti relativno složene i u početku rada može biti potrebno puno vremena dok se shvati njihov koncept i dok aplikacija ne počne raditi bez grešaka, zbog čega se programer u početku izrade aplikacije može osjećati neproduktivno. Međutim, nakon shvaćanja koncepata konfiguracijskih datoteka i logike rada samog programskog okvira, razvoj aplikacija postaje vrlo jednostavan i brz.

Spring programski okvir, kao i svaka druga tehnologija, ima mjesta za poboljšanje, primjerice pojednostavljivanje konfiguracije aplikacija u određenim slučajevima, međutim puno je više dobrih karakteristika, na primjer, podjela programskog okvira u različite module prema poslovnoj domeni (što je praktično jer je moguće koristiti samo one module koji su aplikaciji potrebni), kao i njihovo dohvaćanje i referenciranje u aplikaciji korištenjem Maven repozitorija (preko kojega se također mogu dohvaćati i tehnologije koje nisu dio programskog okvira), mogućnost korištenja tehnologija za objektno relacijsko mapiranje za jednostavnije povezivanje i rad aplikacije i baze podataka, kao i mogućnost brzog razvoja aplikacija korištenjem Spring Boot modula. Zbog navedenih dobrih karakteristika, rekao bih da je Spring programski okvir tehnologija vrijedna učenja i primjene u praksi.

Popis kratica

EJB	*Enterprise JavaBeans*	Enterprise Java Bean-ovi
DI	*Dependency Injection*	umetanje ovisnosti
POJO	*Plain Old Java Object*	obični Java objekt
HQL	*Hibernate Query Language*	Hibernate jezik upita
JPA	*Java Persistence API*	Java API za rad s podacima

Popis slika

Literatura

[1] WALLS, C. *Spring in Action, Fourth Edition*. Shelter Island, New York: Manning Publications Co., 2015

[2] WALLS, C. *Spring Boot in Action*. Shelter Island, New York: Manning Publications Co., 2016.

[3] OTTINGER, J. B., Linwood, J., Minter, D. *Beginning Hibernate: For Hibernate 5*. New York: Apress Inc., 2016

[4] SPRING DOCUMENTATION, Spring Framework Overview, https://docs.spring.io/spring/docs/current/spring-framework-reference/overview.html, siječanj. 2018.