



Funded by the
Erasmus+ Programme
of the European Union



This project has been funded with support from the European Commission. This publication [communication] reflects the views only of the author, and the Commission cannot be held responsible for any use which may be made of the information contained therein.



KI203 - JAVA 6: NAPREDNO PROGRAMIRANJE U JAVI

Java Persistence API

Lekcija 04

PRIRUČNIK ZA STUDENTE

KI203 - JAVA 6: NAPREDNO PROGRAMIRANJE U JAVI

Lekcija 04

JAVA PERSISTENCE API

- ✓ Java Persistence API
- ✓ Poglavlje 1: Kreiranje JPA entiteta
- ✓ Poglavlje 2: DAO (Data Access Objects) klase
- ✓ Poglavlje 3: Automatsko generisanje JPA entiteta
- ✓ Poglavlje 4: Obeleženi upiti i JPQL
- ✓ Poglavlje 5: Validacija zrna
- ✓ Poglavlje 6: Relacije entiteta
- ✓ Poglavlje 7: Kreiranje JSF aplikacija iz JPA entiteta
- ✓ Poglavlje 8: Domaći zadatak
- ✓ Zaključak

Copyright © 2017 – UNIVERZITET METROPOLITAN, Beograd. Sva prava zadržana. Bez prethodne pismene dozvole od strane Univerziteta METROPOLITAN zabranjena je reprodukcija, transfer, distribucija ili memorisanje nekog dela ili čitavih sadržaja ovog dokumenta., kopiranjem, snimanjem, elektronskim putem, skeniranjem ili na bilo koji drugi način.

Copyright © 2017 BELGRADE METROPOLITAN UNIVERSITY. All rights reserved. No part of this publication may be reproduced, stored in a retrieval system or transmitted in any form or by any means, electronic, mechanical, photocopying, recording, scanning or otherwise, without the prior written permission of Belgrade Metropolitan University.

▼ Uvod

UVOD

Lekcija će se fokusirati na rad sa bazama podataka primenom Java Persistence API (JPA).

Java Persistence API (JPA) predstavlja API za objektno - relaciono mapiranje (**object - relational mapping** - ORM). ORM alati pomažu prilikom automatizovanja mapiranja Java objekata u tabele relacionih baza podataka. Prve verzije J2EE koristile su entitetska zrna kao standardni ORM pristup. Entitetska zrna (**entity beans**) su uvek pokušavala da drže sinhronizovanim podatke koji se čuvaju u memoriji sa podacima baza podataka. Iako je ovo odlična ideja, u praksi je pokazala ozbiljne nedostatke koji su se reflektovali kroz loše performanse aplikacija.

U međuvremenu, razvijeno je nekoliko ORM API - ja, sa ciljem prevazilaženja ograničenja koja se dovode u vezu sa primenom entitetskih zrna. Između ostalih, najpoznatiji su: **Hibernate**, **iBatis**, **Cayenne** i **TopLink**.

Sa Java EE 5 dolazi do deprecije entitetskih zrna u korist JPA. JPA je integrisala ideje nekoliko ORM alata i postavila ih kao standard. Upravo tim alatima će se baviti ova lekcija.

Lekcija će se fokusirati na sledeće teme:

- Kreiranje JPA entiteta;
- Interakcija sa JPA entitetima primenom klase **EntityManager**;
- Generisanje JPA entiteta iz postojećih šema baza podataka;
- Korišćene JPA upita i **Java Persistence Query Language (JPQL)**;
- Kreiranje JSF aplikacije sa JPA entitetima.

Savladavanjem ove lekcije student će razumeti ORM pristup i ovladati primenom ORM alata u radu sa bazama podataka.

▼ Poglavlje 1

Kreiranje JPA entiteta

PRIMER 1 - KREIRANJE PRVOG JPA ENTITETA

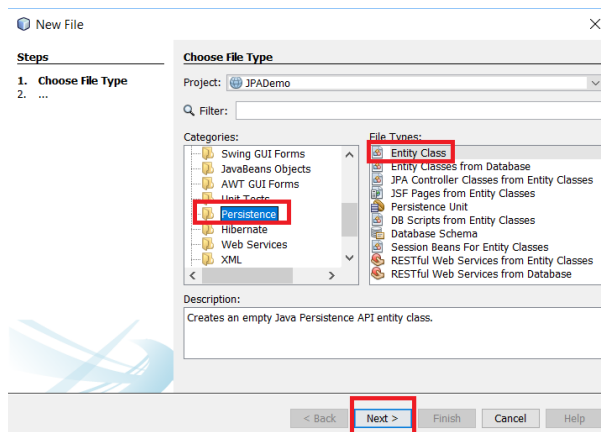
Lekcija započinje izlaganje sa demonstracijom kreiranja prvog JPA entiteta.

JPA entiteti su Java klase čija se polja čuvaju (perzistiraju) u bazama podataka pomoću JPA API. Navedene Java klase predstavljaju POJO (**Plain Old Java Objects**) objekte i kao takvi ne moraju da nasleđuju bilo kakvu roditeljsku klasu ili da implementiraju bilo kakav specifični interfejs. **Java klasa koja se odnosi na JPA entitet je obeležena anotacijom @Entity.**

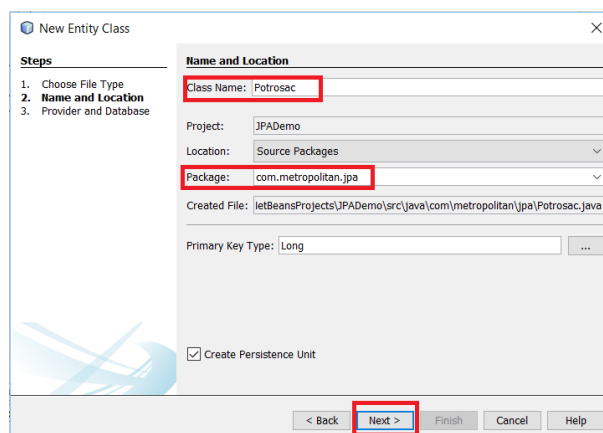
Lekcija će nastaviti praksu, uvedenu u prethodnim izlaganjima, da svaku analizu bazira na adekvatnom primeru. Takav slučaj će biti i ovde. Neophodno je kreirati novi veb projekat, primenom JavaServer Faces okvira, u okviru kojeg će biti implementiran prvi JPA entitet. Primenom razvojnog okruženja, neka to za potrebe demonstracije bude NetBeans IDE, biće kreiran projekat veb aplikacije pod nazivom JPADemo. Takođe, biće korišćen aplikativni server GlassFish.

U kreiranom projektu, sada je moguće kreirati prvu JPA klasu. U meniju File, bira se opcija New File. Otvara se dobro poznat prozor (videti sliku broj 1) u kojem se bira kategorija datoteke koja se kreira i odgovarajući tip datoteke. **Kao kategorija će biti izabrano Persistence, a tip datoteke će biti Entity Class.** Klikom na dugme Next otvara se prozor (videti sliku broj 2) u kojem se definiše naziv klase koja se kreira i paket kojem će klasa pripadati.

Klikom na Dugme Next, novog prozora, prelazi se na novi prozor **Provider and Database** o kojem će više diskusije biti u narednom izlaganju.



Slika 1.1.1 Izbor kategorije i tipa datoteke



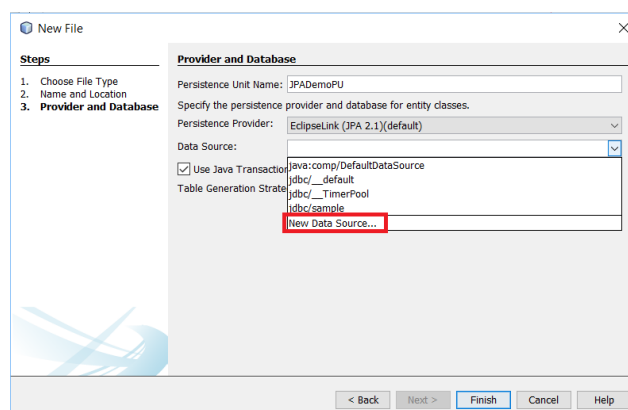
Slika 1.1.2 Definisanje naziva i paketa JPA klase

PRIMER 1 - PROVIDER AND DATABASE PODEŠAVANJE - PROVAJDER

EclipseLink je podrazumevana JPA implementacija za GlassFish aplikativni server.

Projekat koji koristi JPA zahteva jedinicu perzistencije (persistence unit). Ona je definisana u fajlu pod nazivom persistence.xml. Razvojno okruženje NetBeans je automatski detektovao da ovaj fajl ne postoji pa je na prethodnoj slici čekirao opciju Create Persistence UNIT.

Klikom na dugme **Next**, prikazanog prozora u prethodnom izlaganju, prelazi se na novi prozor **Provider and Database** koji je prikazan sledećom slikom.



Slika 1.1.3 Provider and Database podešavanje

Provider and Database čarobnjak, prikazan slikom, predložiće naziv za jedinicu perzistencije koji u velikoj većini slučajeva može da bude prihvaćen.

NetBeans IDE podržava nekoliko JPA implementacija, kao što su: **EclipseLink**, **TopLink Essentials**, **Hibernate**, **KODO** i **OpenJPA**. EclipseLink je podrazumevana JPA implementacija za

GlassFish aplikativni server, pa će zbog toga, u ovom slučaju, biti izabrana kao provajder perzistencije (**Persistence Provider**), a to je pokazano prethodnom slikom.

Još jedan bitan zadatak je neophodno obaviti pre bilo kakve interakcije sa bazom podataka. Neophodno je podesiti tip baze podataka i izvora podataka na aplikativnom serveru.

PRIMER 1 - PROVIDER AND DATABASE PODEŠAVANJE - TIP BAZE PODATAKA

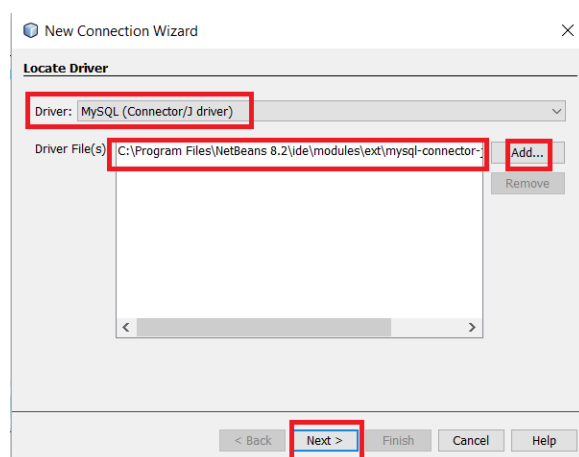
Konekcije u connection pool-u nikad nisu zatvorene.

Veoma bitan zadatak je obezbeđivanje informacija kojima je omogućeno konektovanje na bazu podataka, poput: naziva servera, porta, kredencijala i slično. Ove informacije su poznate pod nazivom **connection pool**. Prednost korišćenja ovakvog pristupa u radu sa bazama podataka, u odnosu na direktnu primenu JDBC, jeste u tome što konekcije u connection pool - u nikad nisu zatvorene i jednostavno se koriste u aplikaciji kada za njima postoji potreba. Ovim se značajno povećavaju performanse aplikacije budući da se izbegavaju zahtevne akcije, po pitanju performansi, otvaranja i zatvaranja konekcija.

Izvor podataka dozvoljava dobijanje informacija iz **connection pool** - a i poziv vlastite metode **getConnection()** za dobijanje konekcije sa bazom podataka iz **connection pool** - a. Ovde nije potrebno direktno obezbeđivanje reference na izvor podataka, JPA će to učiniti automatski.

Sada je neophodno vratiti se na prethodnu sliku. Za kreiranje novog izvora podataka neophodno je kliknuti na opciju **New Data Source** koja je istaknuta crvenom bojom u **Data Source ComboBox** kontroli.

Nakon izvršene navedene akcije, otvara se prozor pod nazivom **Locate Driver** (videti sledeću sliku) gde se bira tip drajvera za bazu podataka i odgovarajući JAR fajl (ukoliko nije ponuđen klikom na Add neophodno ga je pronaći i dodati).



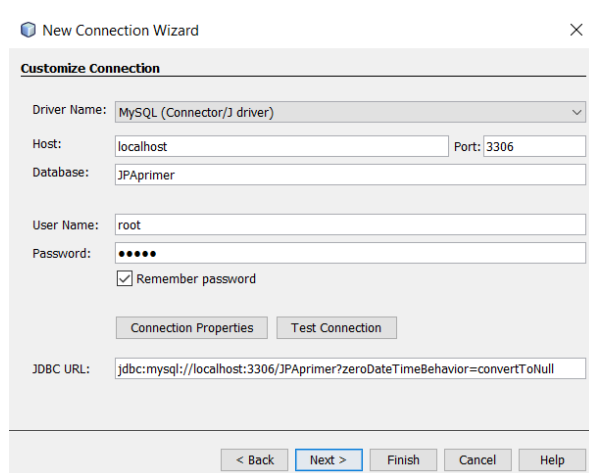
Slika 1.1.4 Izbor tipa i fajla drajvera baze podataka

Sa slike je moguće primetiti da je izbor pao na MySQL tip baze podataka i njemu odgovarajući drajver. Klikom na dugme Next otvara se novi prozor, pod nazivom **Customize Connection** čija analiza sledi u narednom izlaganju.

PRIMER 1 - PROVIDER AND DATABASE PODEŠAVANJE - DODATNA PODEŠAVANJA

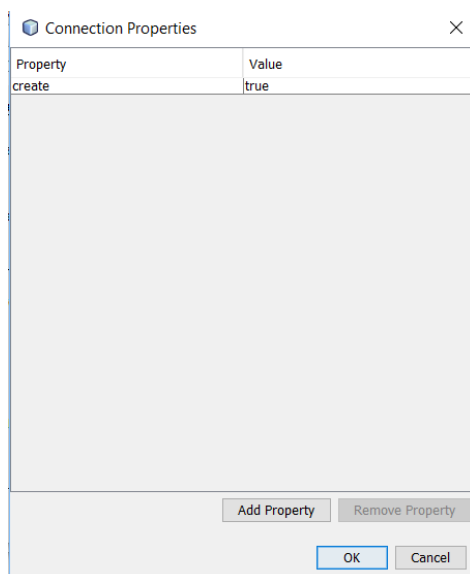
Sledi izbor Host - a, Port - a i baze podataka iz koje će podaci biti korišćeni.

Kao što je istaknuto u prethodnom izlaganju, dalja analiza započinje kod prozora **Customize Connections** koji je prezentovan sledećom slikom.



Slika 1.1.5 Customize Connections prozor

Ovde je još jednom istaknut tip dražvera baze podataka, nakon čega **sledi izbor Host - a, Port - a i baze podataka iz koje će podaci biti korišćeni**. Ovi podaci su bili automatski kreirani, osim naziva baze podataka kojeg korisnik sam određuje (**JPAPrimer**). Nakon toga unose se podaci, **User Name** i **Password**, neophodni za pristup bazi podataka (ukoliko postoje). Ukoliko baza podataka sa navedenim imenom, u ovom trenutku ne postoji, moguće je izvršiti njeno kreiranje klikom na opciju **Connection Properties** i podešavanjem opcije **create** na vrednost **true** (sledeća slika).



Slika 1.1.6 Connection Properties prozor

Nakon obavljanja ove akcije, moguće je još pogledati i polje za unos teksta koje ukazuje na JDBC URL. Nakon unosa imena baze podataka, odgovarajući String je automatski upisan u ovo polje.

Na kraju, biće neophodno uneti i JNDI ime, na primer jdbc/jpademo, čime će ova podešavanja biti završena.

PRIMER 1 - PRVI JPA ENTITET - NASTAVAK

Nakon izvršenih podešavanja moguće je nastaviti kreiranja prvog JPA entiteta.

Nakon izvršenih podešavanja moguće je nastaviti kreiranja prvog JPA entiteta na način koji je već prikazan slikama 1 i 2. Klasa poseduje naziv `Potrosac.java` i pripada paketu kome je dodeljen naziv `com.metropolitan.jpa` (pogledati sliku 2). Sadržaj datoteke je priložen sledećim inicijalnim listingom:

```
package com.ensode.jpaweb;
import java.io.Serializable;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;
@Entity
public class Potrosac implements Serializable {
    private static final long serialVersionUID = 1L;
    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private Long id;
    public Long getId() {
        return id;
    }
    public void setId(Long id) {
        this.id = id;
    }
    //ostale metode (equals(), hashCode(), toString())
    izostavljene su zbog preglednosi
}
```

Ono što je prvo moguće uočiti, budući da se radi o entitetskoj klasi, jeste da je ona automatski obeležena anotacijom **@Entity**. To praktično znači da njoj direktno odgovara tabela sa identičnim nazivom iz kreirane baze podataka `JPAPrimer`.

Sledećom anotacijom **@id** je označeno koja polja u JPA entitetu čine primarni ključ tj, jednoznačni identifikator entiteta. Ne postoje dva entiteta sa identičnim primarnim ključevima. Postojanje navedenih anotacija je minimum koji klasa mora da ispunjava da bi mogla da bude smatrana entitetskom klasom. JPA dozvoljava automatsko generisanje primarnih ključeva. U tu svrhu je moguće koristiti anotaciju **@GeneratedValue**. Ovom

anotacijom se definiše strategija kojom se generiše primarni ključ. U velikom broju slučajeva vrednost ove anotacije **GenerationType.AUTO** funkcioniše kako treba pa se uglavnom i koristi (kao i u slučaju tekućeg primera). Moguće vrednosti za anotaciju **@GeneratedValue** date su sledećom tabelom.

Strategija generisanja primarnog ključa	Opis
<code>GenerationType.AUTO</code>	Provajder perzistencije će automatski izabrati strategiju generisanja primarnog ključa. Ovo je podrazumevana vrednost, ukoliko nije navedena.
<code>GenerationType.IDENTITY</code>	"Identity" kolona u tabeli baze podataka koja je mapirana sa JPA entitetom, mora biti upotrebljena za generisanje vrednosti primarnog ključa.
<code>GenerationType.SEQUENCE</code>	Sekvenca baze podataka se koristi za generisanje vrednosti primarnog ključa.
<code>GenerationType.TABLE</code>	Tabela baze podataka se koristi za generisanje vrednosti primarnog ključa.

Slika 1.1.7 Moguće vrednosti za anotaciju **@GeneratedValue**

PRIMER 1 - DODAVANJE PERZISTENTNIH POLJA U ENTITET

Automatski generisan kod klase entiteta se proširuje odgovarajućim poljima i metodama.

U nastavku, neophodno je u JPA entitet, odnosno odgovarajuću klasu, dodati polja koja će služiti da se u odgovarajućim kolonama, tabele koja odgovara JPA entitetu, čuvaju konkretne vrednosti u bazi podataka. Budući da su polja kreirana primenom modifikatora **private**, neophodno je u kod dodati i odgovarajuće **setter** i **getter** javne metode za manipulisanje ovim poljima.

Za konkretan primer, odmah ispod polja **id**, koje je automatski generisano, biće dodatak dva **String** polja **firstName** i **lastName** za ime i prezime kreiranih potrošača, respektivno. Za ova dva polja dodaju se i odgovarajuće **setter** i **getter** javne metode.

Kao rezultat javlja se klasa `Potrosac.java` čiji je modifikovan kod priložen u celosti sledećim listingom:

```
package com.metropolitan.jpa;

import java.io.Serializable;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;

/**
 *
 * @author Vladimir Milicevic
 */
@Entity
```

[illegible]

[illegible]

▼ 1.1 Zadaci za samostalni rad

ZADATAK 1 - POSTAVKA

Kreira se veb aplikacija nad bazom podataka.

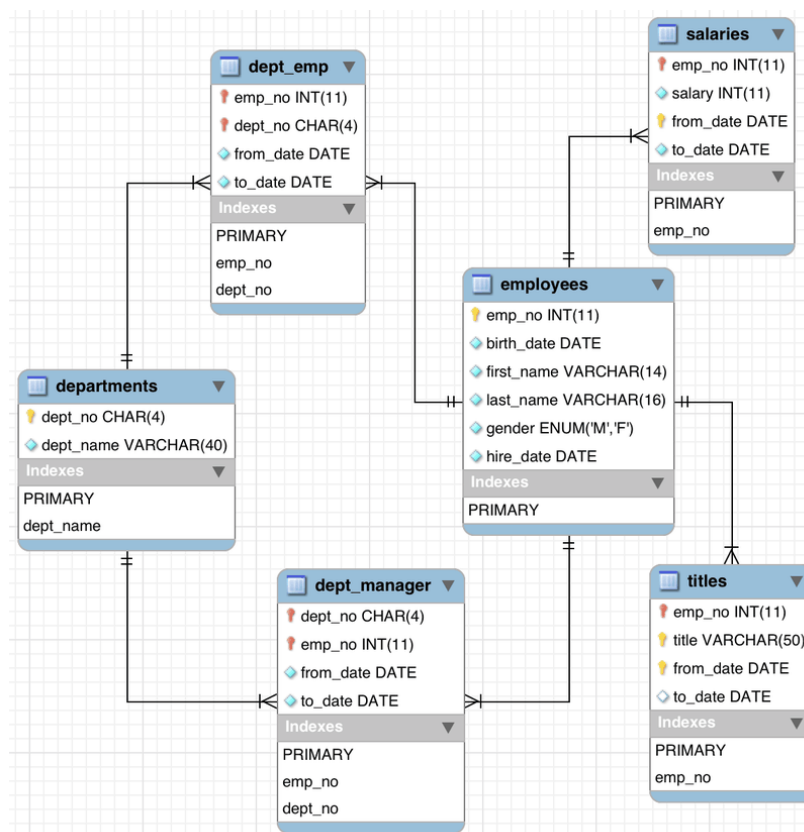
Nad šemom baze podataka **company**, koja je data sledećom slikom, kreirati veb aplikaciju prateći sledeće korake:

1. Kreirati veb projekat pod nazivom JPA-vezba7;
2. Kreirati MySQL bazu podataka pod nazivom company;
3. Izvršiti u kreiranom projektu povezivanje na datu bazu, a zatim izvršiti fajl sa priloženim SQL upitima za kreiranje baze podataka;
4. Nad tabelama baze podataka kreirati JPA entitete;

```
CREATE TABLE employees (
    emp_no      INT                NOT NULL,
    birth_date   DATE              NOT NULL,
    first_name   VARCHAR(14)       NOT NULL,
    last_name    VARCHAR(16)       NOT NULL,
    gender       ENUM ('M', 'F')  NOT NULL,
    hire_date    DATE              NOT NULL,
    PRIMARY KEY (emp_no)
);

CREATE TABLE departments (
    dept_no      CHAR(4)           NOT NULL,
    dept_name    VARCHAR(40)       NOT NULL,
    PRIMARY KEY (dept_no),
    UNIQUE KEY (dept_name)
);
```

```
CREATE TABLE dept_manager (  
    dept_no      CHAR(4)          NOT NULL,  
    emp_no       INT              NOT NULL,  
    from_date    DATE             NOT NULL,  
    to_date      DATE             NOT NULL,  
    KEY          (emp_no),  
    KEY          (dept_no),  
    FOREIGN KEY (emp_no) REFERENCES employees (emp_no) ON DELETE CASCADE,  
    FOREIGN KEY (dept_no) REFERENCES departments (dept_no) ON DELETE CASCADE,  
    PRIMARY KEY (emp_no,dept_no)  
);  
  
CREATE TABLE dept_emp (  
    emp_no       INT              NOT NULL,  
    dept_no      CHAR(4)          NOT NULL,  
    from_date    DATE             NOT NULL,  
    to_date      DATE             NOT NULL,  
    KEY          (emp_no),  
    KEY          (dept_no),  
    FOREIGN KEY (emp_no) REFERENCES employees (emp_no) ON DELETE CASCADE,  
    FOREIGN KEY (dept_no) REFERENCES departments (dept_no) ON DELETE CASCADE,  
    PRIMARY KEY (emp_no,dept_no)  
);  
  
CREATE TABLE titles (  
    emp_no       INT              NOT NULL,  
    title        VARCHAR(50)      NOT NULL,  
    from_date    DATE             NOT NULL,  
    to_date      DATE,  
    KEY          (emp_no),  
    FOREIGN KEY (emp_no) REFERENCES employees (emp_no) ON DELETE CASCADE,  
    PRIMARY KEY (emp_no,title, from_date)  
);  
  
CREATE TABLE salaries (  
    emp_no       INT              NOT NULL,  
    salary       INT              NOT NULL,  
    from_date    DATE             NOT NULL,  
    to_date      DATE             NOT NULL,  
    KEY          (emp_no),  
    FOREIGN KEY (emp_no) REFERENCES employees (emp_no) ON DELETE CASCADE,  
    PRIMARY KEY (emp_no, from_date)  
);
```

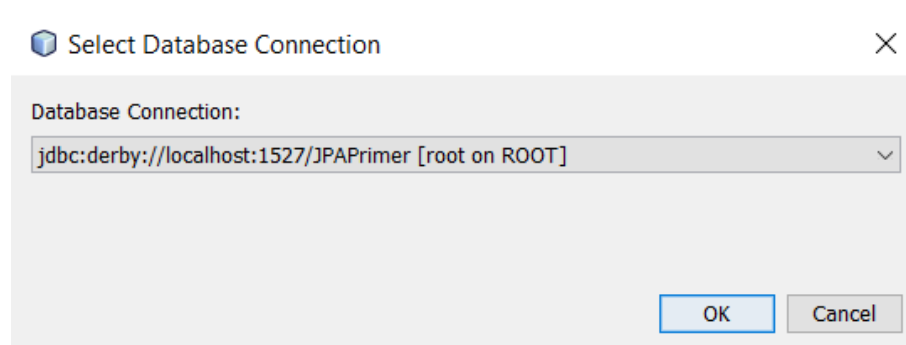


Slika 1.2.1 Šema baze podataka company

ZADATAK 1 - IZVRŠAVANJE UPITA I KREIRANJE TABELA BAZE PODATAKA

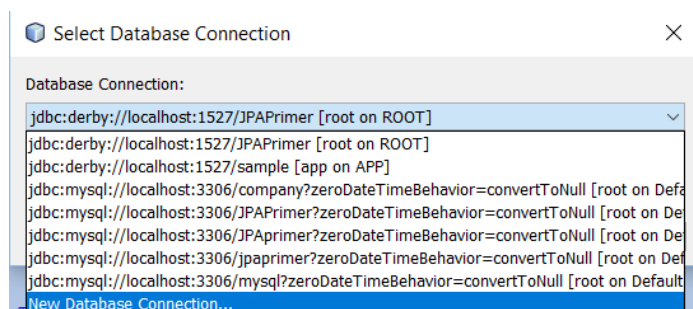
U razvojnom okruženju je neophodno izvršiti datoteku sa SQL upitima.

U projektu je kreiran folder SQL i u njega je ubačen fajl koji je dobio naziv `create_populate_tables.sql` i koji sadrži kod sa prethodne sekcije. Klikom na dugme Run SQL otvara se prozor za izbor DB konekcije:



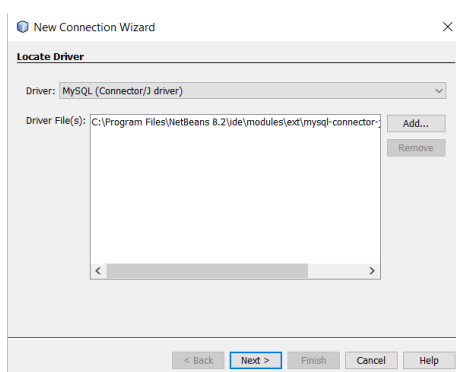
Slika 1.2.2 Izbor DB konekcije

U padajućem meniju sa slike, neophodno je izvršiti povezivanje na bazu podataka `company`.



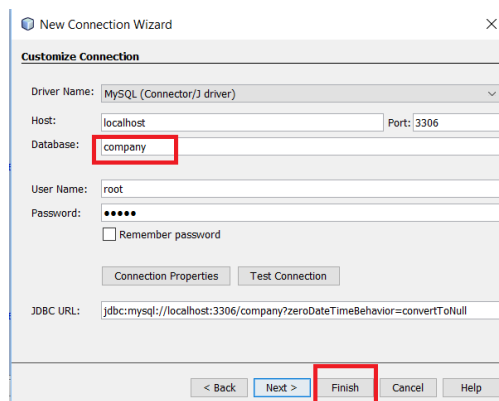
Slika 1.2.3 New DB konekcija

Bira se MySQL drajver.



Slika 1.2.4 Izbor tipa baze podataka

Unosi se naziv baze i izvršava se lista upita.



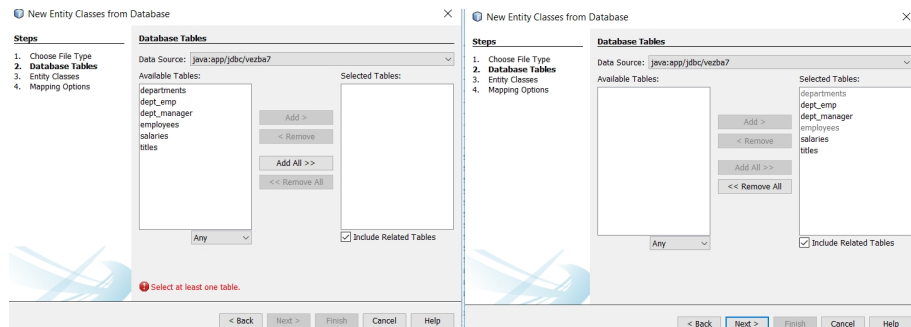
Slika 1.2.5 Povezivanje na željenu bazu podataka.

ZADATAK 1 - KREIRANJE JPA ENTITETA

Sledi kreiranje JPA entiteta nad tabelama baze podataka.

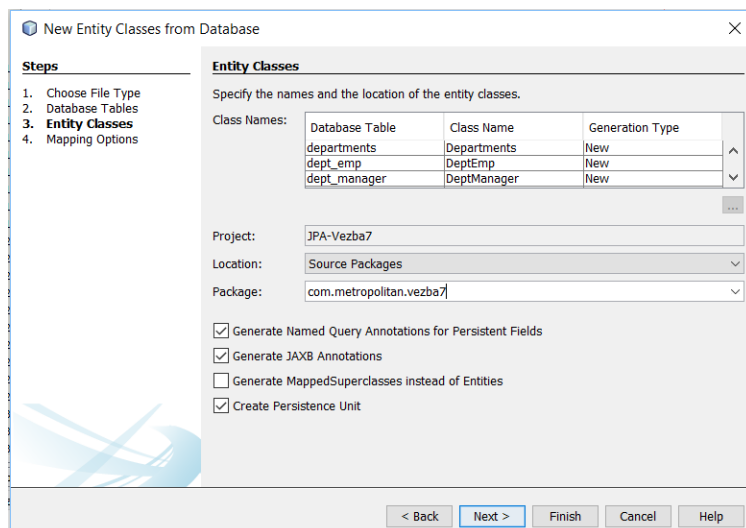
Kreirane su tabele baze podataka. U novom projektu će biti izabrana opcija **File | New**, a zatim se kreira datoteka iz kategorije **Persistence** i tipa **Entity Classes from Database** (o svemu ovome je već bilo govora).

U prozoru **New Entity Classes from Database** biraju se tabele od kojih je neophodno kreirati JPA entitete (sledeća slika).



Slika 1.2.6 Izbor tabela za kreiranje JPA entiteta

Nakon izabranih tabela klikom na **Next** otvara se prozor u kojem se navodi naziv paketa kojem će kreirane klase pripadati.



Slika 1.2.7 Definisanje paketa i kraj kreiranja JPA klasa

▼ Poglavlje 2

DAO (Data Access Objects) klase

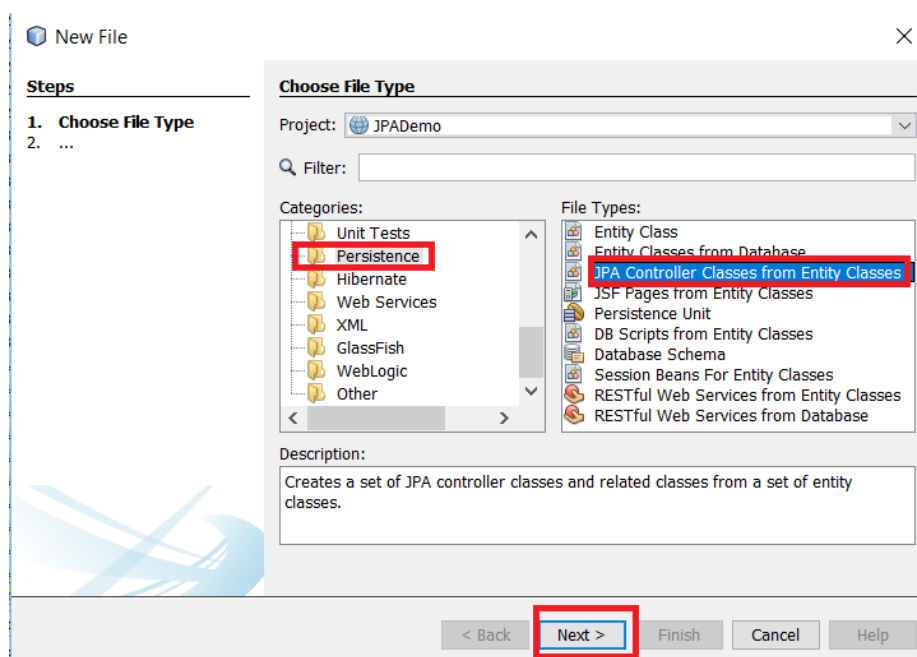
PRIMER 2 - KREIRANJE DAO KLASE

Dobra praksa je korišćenje šablona dizajna za pisanje koda koji se obraća bazi podataka.

Prilikom razvoja savremenih aplikacija koje podrazumevaju rad sa bazama podataka, **veoma dobru praksu predstavlja korišćenje Data Access Objects (DAO) šablona dizajna za pisanje koda koji se obraća bazi podataka**. DAO šabloni dizajna čuvaju sve funkcionalnosti pristupa bazi podataka unutar DAO klase. Na ovaj način se postiže jasno razdvajanje modula programa pomoću kojeg ostali slojevi aplikacije, poput logike korisničkog interfejsa ili poslovne logike, ne zavise od logike perzistencije.

Razvojno okruženje NetBeans IDE poseduje sposobnost direktnog kreiranja JPA klase kontrolera iz postojećih entiteta. Upravo u narednom izlaganju sledi kreiranje jedne ovakve klase i proširivanje započetog primera. Po dobro poznatom scenariju, koristeći razvojno okruženje, bira se opcija **File | New** i otvara se poznati prozor za izbor kategorije i tipa aplikacije. U ovom slučaju kao kategorija se bira **Persistence** category, dok će **JPA Controller Classes from Entity Classes** predstavljati tip datoteke koja se kreira.

Sledećom slikom je prikazan početak, objašnjen u prethodnom izlaganju, kreiranja DAO klase.

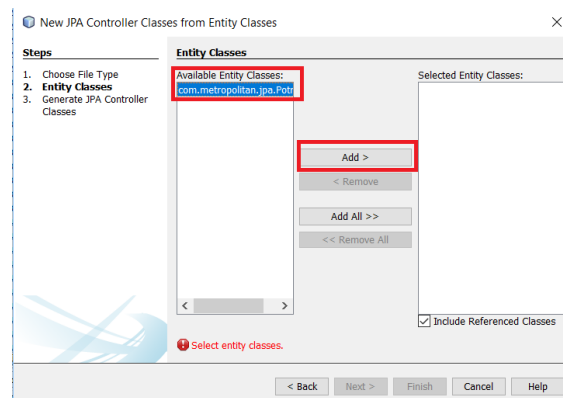


Slika 2.1.1 New File prozor za kreiranje DAO klase

PRIMER 2 - KREIRANJE DAO KLASE - IZBOR ENTITETA

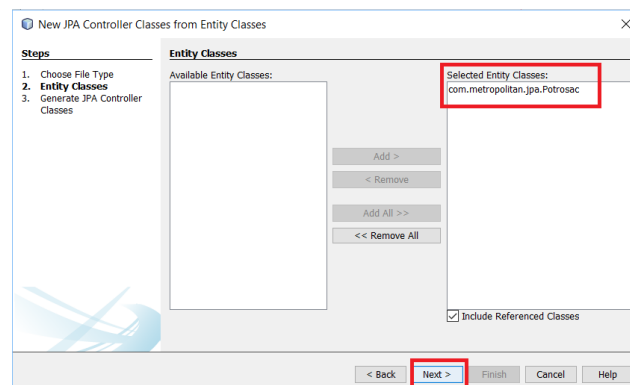
Moguće je kreiranje DAO klase direktno iz entiteta.

Kao što je istaknuto u prethodnom izlaganju, razvojno okruženje NetBeans IDE poseduje sposobnost direktnog kreiranja JPA klase kontrolera iz postojećih entiteta. Kada u prozoru, sa slike 1, usledi akcija korisnika na dugme **Next**, otvara se čarobnjak za izbor entiteta koji će poslužiti kao osnov za kreiranje JPA klase kontrolera. Navedeno je prikazano sledećom slikom.



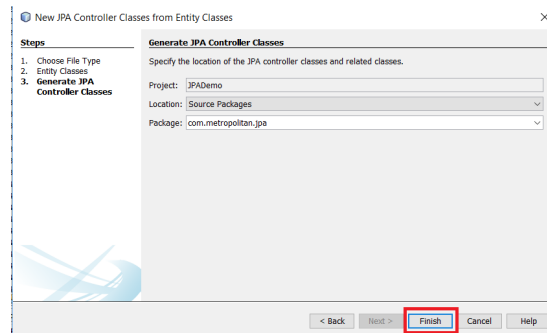
Slika 2.1.2 Izbor klase entiteta

Označavanjem klase u levom delu čarobnjaka, i klikom na dugme Add, u sredini, vrši se izbor entitetske klase pomoću koje će biti formirana kontroler klasa. Novo stanje u čarobnjaku je prikazano sledećom slikom.



Slika 2.1.3 Izabrana klasa entiteta

U nastavku, klikom na dugme **Next**, vrši se potvrđivanje ovog izbora i otvara prozor kojim se postupak kreiranja JPA klase kontrolera završava (klikom na **Finish**). Navedeno je prikazano sledećom slikom.



Slika 2.1.4 Poslednji korak u kreiranju JPA klase kontrolera

PRIMER 2 - DAO KLASA - LISTING SA OBJAŠNENJEM

Sledi prikaz i objašnjenje kreiranog kontrolera.

Kao što može da se nasluti, iz prethodnog izlaganja, razvojno okruženje je na osnovu instrukcija dobijenih kroz navigaciju u prethodno prikazanom čarobnjaku, generisalo kod kontroler klase `PotrosacJpaController.java`, na osnovu entitetske klase `Potrosac.java`. Sledi listing koda klase:

```
package com.metropolitan.jpa;

import com.metropolitan.jpa.exceptions.NonexistentEntityException;
import com.metropolitan.jpa.exceptions.RollbackFailureException;
import java.io.Serializable;
import java.util.List;
import javax.persistence.EntityManager;
import javax.persistence.EntityManagerFactory;
import javax.persistence.Query;
import javax.persistence.EntityNotFoundException;
import javax.persistence.criteria.CriteriaQuery;
import javax.persistence.criteria.Root;
import javax.transaction.UserTransaction;

/**
 *
 * @author Vladimir Milicevic
 */
public class PotrosacJpaController implements Serializable {

    public PotrosacJpaController(UserTransaction utx,
        EntityManagerFactory emf) {
        this.utx = utx;
        this.emf = emf;
    }
    private UserTransaction utx = null;
    private EntityManagerFactory emf = null;

    public EntityManager getEntityManager() {
        return emf.createEntityManager();
    }
}
```

```

    }

    public void create(Potrosac customer) throws
        RollbackFailureException, Exception {
        EntityManager em = null;
        try {
            utx.begin();
            em = getEntityManager();
            em.persist(customer);
            utx.commit();
        } catch (Exception ex) {
            try {
                utx.rollback();
            } catch (Exception re) {
                throw new RollbackFailureException(
                    "An error occurred attempting to roll back the transaction.", re );
            }
        }
        throw ex;
    } finally {
        if (em != null) {
            em.close();
        }
    }
}

    public void edit(Potrosac customer) throws
        NonexistentEntityException, RollbackFailureException, Exception {
        EntityManager em = null;
        try {
            utx.begin();
            em = getEntityManager();
            customer = em.merge(customer);
            utx.commit();
        } catch (Exception ex) {
            try {
                utx.rollback();
            } catch (Exception re) {
                throw new RollbackFailureException(
                    "An error occurred attempting to roll back the transaction.", re);
            }
        }
        String msg = ex.getLocalizedMessage();
        if (msg == null || msg.length() == 0) {
            Long id = customer.getId();
            if (findCustomer(id) == null) {
                throw new NonexistentEntityException(
                    "The customer with id " + id
                    + " no longer exists.");
            }
        }
        throw ex;
    } finally {
        if (em != null) {
            em.close();
        }
    }
}

```

```

        }
    }
}

public void destroy(Long id) throws NonexistentEntityException,
    RollbackFailureException, Exception {
    EntityManager em = null;
    try {
        utx.begin();
        em = getEntityManager();
        Potrosac customer;
        try {
            customer = em.getReference(Potrosac.class, id);
            customer.getId();
        } catch (EntityNotFoundException enfe) {
            throw new NonexistentEntityException(
                "The customer with id " + id
                + " no longer exists.", enfe);
        }
        em.remove(customer);
        utx.commit();
    } catch (Exception ex) {
        try {
            utx.rollback();
        } catch (Exception re) {
            throw new RollbackFailureException(
                "An error occurred attempting to roll back the transaction.", re);
        }
    }
    throw ex;
    } finally {
        if (em != null) {
            em.close();
        }
    }
}

public List<Potrosac> findCustomerEntities() {
    return findCustomerEntities(true, -1, -1);
}

public List<Potrosac> findCustomerEntities(int maxResults,
    int firstResult) {
    return findCustomerEntities(false, maxResults, firstResult);
}

private List<Potrosac> findCustomerEntities(boolean all, int maxResults,
    int firstResult) {
    EntityManager em = getEntityManager();
    try {
        CriteriaQuery cq = em.getCriteriaBuilder().createQuery();
        cq.select(cq.from(Potrosac.class));
        Query q = em.createQuery(cq);
        if (!all) {

```

```

        q.setMaxResults(maxResults);
        q.setFirstResult(firstResult);
    }
    return q.getResultList();
} finally {
    em.close();
}
}

public Potrosac findCustomer(Long id) {
    EntityManager em = getEntityManager();
    try {
        return em.find(Potrosac.class, id);
    } finally {
        em.close();
    }
}

public int getCustomerCount() {
    EntityManager em = getEntityManager();
    try {
        CriteriaQuery cq = em.getCriteriaBuilder().createQuery();
        Root<Potrosac> rt = cq.from(Potrosac.class);
        cq.select(em.getCriteriaBuilder().count(rt));
        Query q = em.createQuery(cq);
        return ((Long) q.getSingleResult()).intValue();
    } finally {
        em.close();
    }
}
}

```

Kao što je moguće primetiti kreirane su CRUD metode za JPA entitete. Metoda koja kreira nove entitete naziva se **create()** i uzima, kao jedini argument, instancu JPA entiteta. Metoda jednostavno poziva metodu **persist()** za **EntityManager** objekat čime se brine o čuvanju podataka entiteta u bazi podataka.

CRUD operacija **read** realizovana je pomoću nekoliko operacija. Metoda **findCustomer()** preuzima primarni ključ JPA entiteta, koji bi trebalo da bude vraćen, kao jedini argument i poziva **find()** metodu objekta **EntityManager** za preuzimanje podataka iz baze podataka i vraćanje instance JPA klase. Generisano je i nekoliko preklopljenih **findCustomerEntities()** metoda pomoću kojih je moguće vratiti više od jednog entiteta iz baze podataka. Jedna od njih ima sledeći oblik:

```

private List<Customer> findCustomerEntities(boolean all, int maxResults, int
firstResult)

```

Prvi parametar je **Boolean** tipa i ukazuje da li se žele preuzeti sve vrednosti iz baze podataka. Drugi parametar određuje najveći broj rezultata koje je moguće preuzeti, a poslednji parametar ukazuje na rezultat koji će prvi biti preuzet.

Metodom `edit()` se vrši ažuriranje postojećih entiteta, pozivom metode `merge()`, objekta `EntityManager`, koja preuzima kao parametar JPA entitet kojim ažurira bazu podataka.

Konačno, entitet se briše metodom `destroy()`.

2.1 Zadaci za samostalni rad

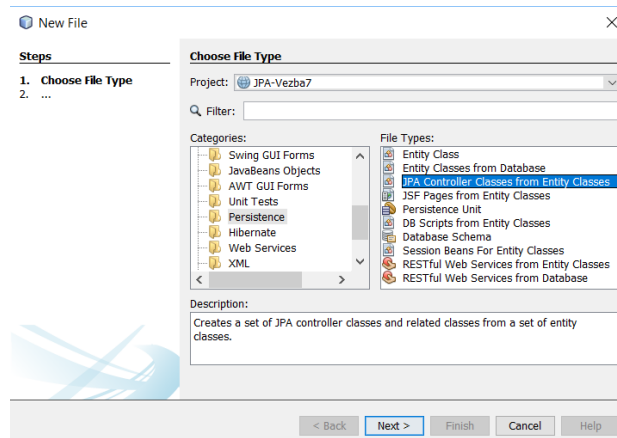
ZADATAK 2 - KREIRANJE KONTOLERA JPA ENTITETA

Dao klase za upravljanje podacima su sledeće koje se kreiraju.

Zadatak 2:

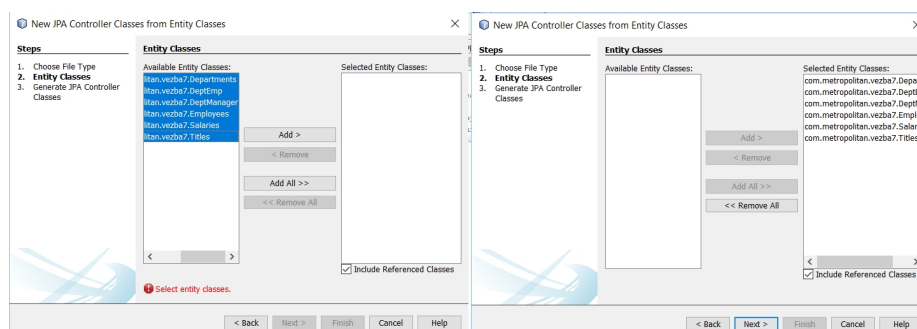
Nad JPA entitetima generisanim u Zadatku 1 kreirati odgovarajuće kontrolere.

Po dobro poznatom scenariju, koristeći razvojno okruženje, bira se opcija `File | New` i otvara se poznati prozor za izbor kategorije i tipa aplikacije. U ovom slučaju kao kategorija se bira `Persistence category`, dok će `JPA Controller Classes from Entity Classes` predstavljati tip datoteke koja se kreira.



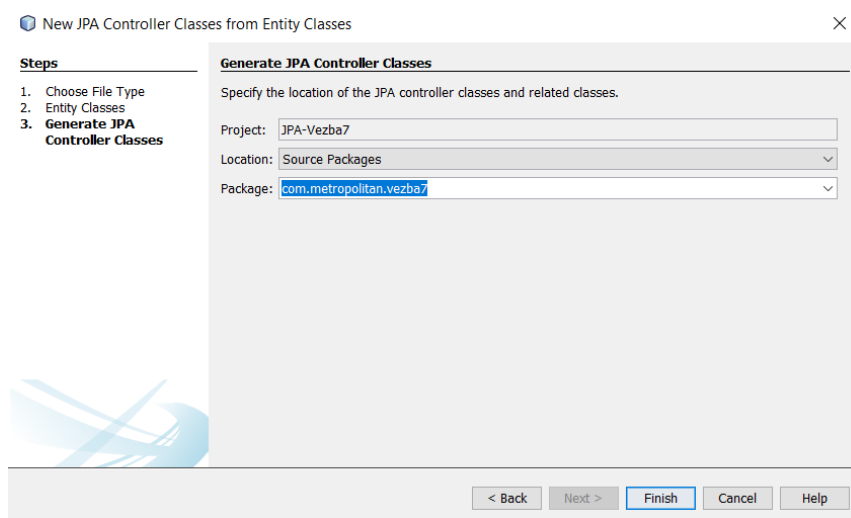
Slika 2.2.1 Izbor za kreiranje JPA kontroler klasa

Dalje se otvara prozor za dodavanje JPA klasa entiteta za koje se kreiraju kontroleri.



Slika 2.2.2 Izbor JPA klasa entiteta

Na kraju je neophodno izabrati paket kojem će pripadati kreirane klase.



▼ Poglavlje 3

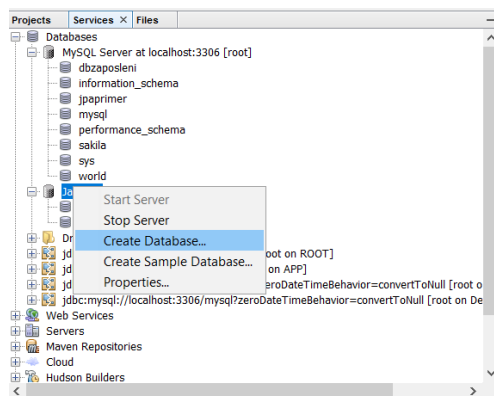
Automatsko generisanje JPA entiteta

KREIRANJE I / ILI PRISTUP BAZI PODATAKA

Savremeni alati dozvoljavaju generisanje JPA entiteta iz već postojećih šema baza podataka.

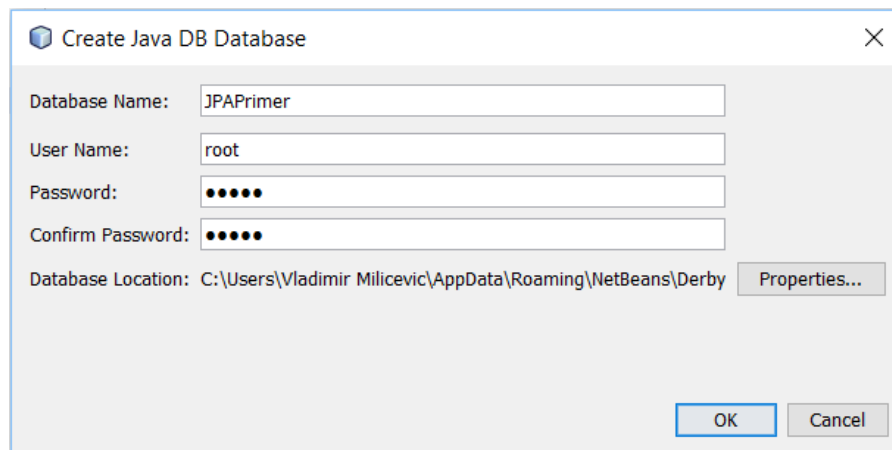
Kada se rade aplikacije nad bazama podataka, u velikom broju slučajeva, koriste se baze podataka koje su već kreirane od strane administratora baze podataka. Savremeni alati dozvoljavaju generisanje JPA entiteta iz već postojećih šema baza podataka i nataj način olakšavaju rad i smanjuju napor programera prilikom rada na kreiranju datoteka neke veb aplikacije.

U ovom delu lekcije, oslanjajući se na konkretne primere, biće analiziran i demonstriran način kreiranja JPA entiteta iz postojeće baze podataka, koja je uvedena i korišćena u primeru iz prethodnog izlaganja. Da bi ovo bilo obavljeno neophodno je otvoriti, primenom razvojnog okruženja, postojeći projekat i u tabu Services izvršiti proširenje kreirane baze podataka. Ovde se desnim klikom bira opcija JAVA DB, a zatim i Create DB (sledeća slika).



Slika 3.1.1 Kreiranje nove baze podataka

Sada je neophodno uneti informacije o bazi podataka u otvorenom čarobnjaku Create Java DB Database, kao na sledećoj slici.



Slika 3.1.2 Podaci o bazi podataka

Sada je sve spremno za izvršavanje SQL skriptova kojima će biti kreirane tabele baze podataka.

Ova lekcija sadrži video materijal. Ukoliko želite da pogledate ovaj video morate da otvorite LAMS lekciju.

PRIMER 3 - KREIRANJE NOVIH TABELA U POSTOJEĆOJ BAZI PODATAKA

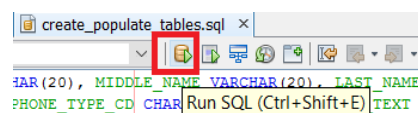
Izvršavanje SQL upita biće kreirane tabele u bazi podataka.

Nad postojećom bazom podataka JPAPrimer, moguće je sada izvršiti brojne SQL skriptove od kojih će većina, u početku, da se odnosi na kreiranje tabela baze podataka. Da bi posao bio olakšan dodatno, preuzet je fajl pod nazivom `create_populate_tables.sql` kao dodatni materijal uz preporučeni udžbenik (videti literaturu pod 2) koji je snabdeven SQL upitima za kreiranje i popunjavanje baze podataka. Sledi deo koda, koji se odnosi na kreiranje tabela:

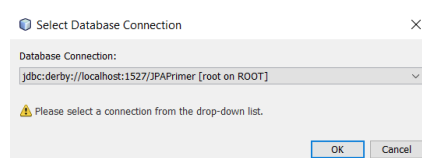
```
CREATE TABLE CUSTOMER (CUSTOMER_ID INT PRIMARY KEY, FIRST_NAME VARCHAR(20),
MIDDLE_NAME VARCHAR(20), LAST_NAME VARCHAR(20), EMAIL VARCHAR(30));
CREATE TABLE TELEPHONE_TYPE(TELEPHONE_TYPE_ID INT PRIMARY KEY, TELEPHONE_TYPE_CD
CHAR(1), TELEPHONE_TYPE_TEXT VARCHAR(20));
CREATE TABLE TELEPHONE (TELEPHONE_ID INT PRIMARY KEY, TELEPHONE_TYPE_ID INT
REFERENCES TELEPHONE_TYPE, CUSTOMER_ID INT REFERENCES CUSTOMER, TELEPHONE_NUMBER
CHAR(12));
CREATE TABLE ADDRESS_TYPE (ADDRESS_TYPE_ID INT PRIMARY KEY, ADDRESS_TYPE_CODE
CHAR(1), ADDRESS_TYPE_TEXT VARCHAR(20));
CREATE TABLE US_STATE(US_STATE_ID INT PRIMARY KEY, US_STATE_CD CHAR(2) NOT NULL,
US_STATE_NM VARCHAR(30) NOT NULL);
CREATE TABLE "APP"."US_CITY"(US_CITY_ID int PRIMARY KEY NOT NULL,US_CITY_NM
varchar(30),ZIP char(5),US_STATE_ID int);
ALTER TABLE US_CITY ADD CONSTRAINT SQL080413034555231 FOREIGN KEY (US_STATE_ID)
REFERENCES US_STATE(US_STATE_ID) ON DELETE NO ACTION ON UPDATE NO ACTION;
CREATE INDEX SQL080413034555231 ON US_CITY(US_STATE_ID);
CREATE UNIQUE INDEX SQL080413034555230 ON US_CITY(US_CITY_ID);
```

```
CREATE TABLE ADDRESS (ADDRESS_ID INT PRIMARY KEY, ADDRESS_TYPE_ID INT REFERENCES
ADDRESS_TYPE, CUSTOMER_ID INT REFERENCES CUSTOMER,
ADDR_LINE_1 VARCHAR(100), ADDR_LINE_2 VARCHAR(100), CITY VARCHAR (100), US_STATE_ID
INT REFERENCES US_STATE, ZIP CHAR(5));
CREATE TABLE CUSTOMER_ORDER(CUSTOMER_ORDER_ID INT PRIMARY KEY, CUSTOMER_ID INT
REFERENCES CUSTOMER, ORDER_NUMBER VARCHAR (10), ORDER_DESCRIPTION VARCHAR(200));
CREATE TABLE ITEM(ITEM_ID INT PRIMARY KEY, ITEM_NUMBER VARCHAR(10), ITEM_SHORT_DESC
VARCHAR(100), ITEM_LONG_DESC VARCHAR(500));
CREATE TABLE ORDER_ITEM(CUSTOMER_ORDER_ID INT REFERENCES CUSTOMER_ORDER, ITEM_ID
INT REFERENCES ITEM);
CREATE TABLE APP_USER(APP_USER_ID INT PRIMARY KEY, USER_NAME VARCHAR(10), PASSWORD
VARCHAR (15));
CREATE TABLE USER_ROLE(ROLE_ID INT PRIMARY KEY, ROLE_NAME VARCHAR(10));
CREATE TABLE APP_USER_ROLE(APP_USER_ID INT REFERENCES APP_USER, USER_ROLE_ID INT
REFERENCES USER_ROLE);
CREATE TABLE "APP"."SEQUENCE"
(
    SEQ_NAME varchar(50),
    SEQ_COUNT decimal(15)
)
```

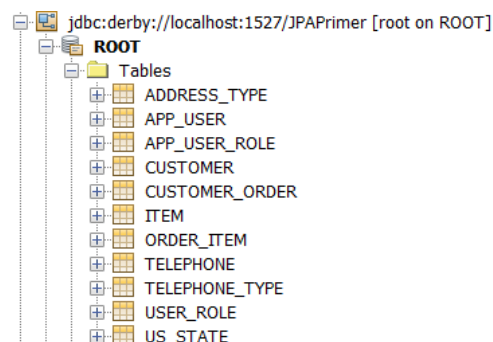
U projektu se jednostavno dolazi do ovog fajla. Bira se **File | Open File**, a zatim se datoteka pronalazi na disku. Kada je ova datoteka, sa SQL upitima, učitana u projekat, primenom razvojnog okruženja je neophodno je izvršiti je klikom na ikonicu **Run SQL** kao što je prikazano na Slici 3. Otvara se prozor za izbor baze podataka nad kojom će upit biti izvršen (Slika 4).



Slika 3.1.3 Izvršavanje SQL skripta



Slika 3.1.4 Izbor baze podataka za izvršenje SQL skripta



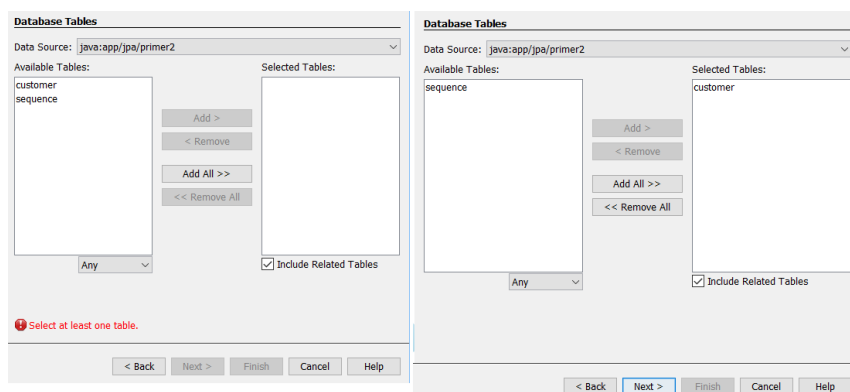
Slika 3.1.5 Tabele su kreirane i popunjene

PRIMER 3 - KREIRANJE JPA ENTITETA IZ POSTOJEĆE ŠEME BAZE PODATAKA

Iz tabela baze potaka je moguće automatski generisati JPA entitete.

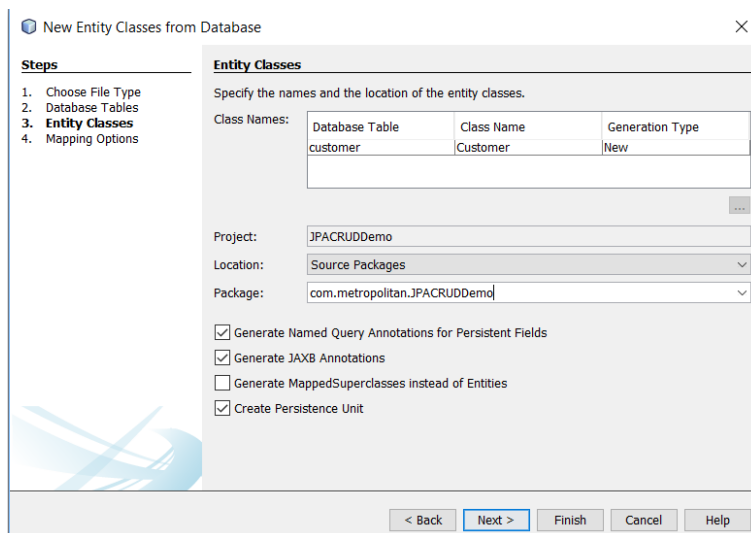
Za početak će biti, na dobro poznat način kreiran veb projekat pod nazivom, **JPACRUDDemo**. U novom projektu će biti izabrana opcija **File | New**, a zatim se kreira datoteka iz kategorije **Persistence** i tipa **Entity Classes from Database** (o svemu ovome je već bilo govora).

U prozoru **New Entity Classes from Database** biraju se tabele od kojih je neophodno kreirati JPA entitete (sledeća slika).

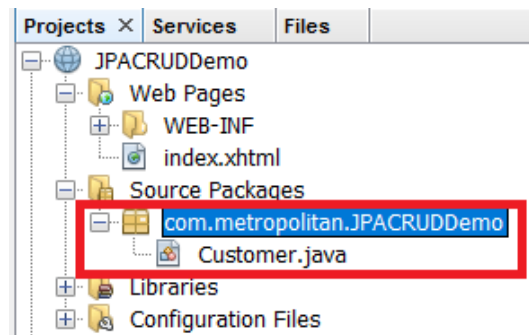


Slika 3.1.6 Izbor tabele baze podataka za kreiranje JPA entiteta

Za potrebe primera, biće izabrana samo jedna tabela **customer** markiranjem i klikom na dugme **Add** (vidi sliku gore). Klikom na dugme **Next**, ide se na naredni prozor u kojem se nalaze sve informacije u vezi sa klasom entiteta koja se kreira, uz napomenu da je neophodno uneti naziv paketa kojem će klasa da pripada (slika 7). Posao se završava klikom na dugme **Finish**.



Slika 3.1.7 Informacije o klasi entiteta koja se kreira



Slika 3.1.8 Klasa entiteta je kreirana

PRIMER 3 - KREIRANA JPA KLASA - DODATNA RAZMATRANJA

Automastki generisan kod JPA klase zahteva dodatnu analizu.

Na prethodnoj slici je moguće primetiti sa se u folderu **Source Packages** projekta pojavila Java klasa pod nazivom **Customer.java**. Datoteka odgovara kreiranom JPA entitetu iz tabele baze podataka **JPAPrimer** i njen kod je priložen sledećim listingom:

```
package com.metropolitan.JPACRUDDemo;

import java.io.Serializable;
import javax.persistence.Basic;
import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.Id;
import javax.persistence.NamedQueries;
import javax.persistence.NamedQuery;
import javax.persistence.Table;
import javax.validation.constraints.NotNull;
import javax.validation.constraints.Size;
import javax.xml.bind.annotation.XmlRootElement;

/**
 *
 * @author Vladimir Milicevic
 */
@Entity
@Table(name = "customer")
@XmlRootElement
@NamedQueries({
    @NamedQuery(name = "Customer.findAll", query = "SELECT c FROM Customer c")
    , @NamedQuery(name = "Customer.findById", query = "SELECT c FROM Customer c WHERE c.id = :id")
    , @NamedQuery(name = "Customer.findByFirstname", query = "SELECT c FROM Customer c WHERE c.firstname = :firstname")
    , @NamedQuery(name = "Customer.findByLastname", query = "SELECT c FROM Customer c WHERE c.lastname = :lastname")})
```

```
public class Customer implements Serializable {

    private static final long serialVersionUID = 1L;
    @Id
    @Basic(optional = false)
    @NotNull
    @Column(name = "ID")
    private Long id;
    @Size(max = 255)
    @Column(name = "FIRSTNAME")
    private String firstname;
    @Size(max = 255)
    @Column(name = "LASTNAME")
    private String lastname;

    public Customer() {
    }

    public Customer(Long id) {
        this.id = id;
    }

    public Long getId() {
        return id;
    }

    public void setId(Long id) {
        this.id = id;
    }

    public String getFirstname() {
        return firstname;
    }

    public void setFirstname(String firstname) {
        this.firstname = firstname;
    }

    public String getLastname() {
        return lastname;
    }

    public void setLastname(String lastname) {
        this.lastname = lastname;
    }

    @Override
    public int hashCode() {
        int hash = 0;
        hash += (id != null ? id.hashCode() : 0);
        return hash;
    }
}
```

[illegible]

Za početak je moguće primetiti da je klasa obeležena anotacijom **@Entity** što nedvosmisleno ukazuje da se radi o JPA entitetu. Dalje, anotacijom **@Table *istaknuta*** je tabela sa kojom je JPA entitet povezan.

Takođe, moguće je primetiti da je jedno polje automatski obeleženo anotacijom **@Id**. Ovo polje je odgovaralo primarnom ključu tabele iz koje je generisan JPA entitet. U ovom slučaju se ne primenjuje strategija generisanja primarnog ključa i anotaciju **@GeneratedValue** je neophodno manuelno implementirati. Anotacijom **@Basic** su polja označena kao neopcionalna.

Anotacijama **@Column** su polja JPA entiteta povezana sa odgovarajućim kolonama tabele baze podataka.

O upitima, koji su obeleženi anotacijom, **@NamedQueries** biće više diskusije u izlaganju koje sledi.

Ova lekcija sadrži video materijal. Ukoliko želite da pogledate ovaj video morate da otvorite LAMS lekciju.

▼ 3.1 Zadaci za samostalni rad

ZADACI ZA SAMOSTALNI RAD STUDENTA

Cilj je provežbavanje naučenog

Zadatak 3:

Nad postojećom šemom baze podataka iz Zadatka 1, kreirati JPA automatski. Uporediti JPA entitete. Koji način je po Vašem mišljenju adekvatniji.

▼ Poglavlje 4

Obeleženi upiti i JPQL

PRIMER 4 - PRIMENA OBELEŽENIH UPITA

Anotacijom `@NamedQueries` obeležena je lista upita koje izvršava kao obeleženi upit.

Za početak neophodno je, za potrebe izlaganja, izolovati deo koda iz prethodno kreirane [JPA](#) klase. Kod je priložen sledećim listingom:

```
@NamedQueries({  
  
    @NamedQuery(name = "Customer.findAll", query = "SELECT c FROM Customer c")  
  
    , @NamedQuery(name = "Customer.findById", query = "SELECT c FROM Customer c  
WHERE c.id = :id")  
  
    , @NamedQuery(name = "Customer.findByFirstname", query = "SELECT c FROM  
Customer c WHERE c.firstname = :firstname")  
  
    , @NamedQuery(name = "Customer.findByLastname", query = "SELECT c FROM Customer  
c WHERE c.lastname = :lastname")  
})
```

Akcentat je na primeni anotacije `@NamedQueries` koja sadrži **value** atribut. Vrednost ovog atributa predstavlja niz `@NamedQuery` anotacija koje prihvataju dva argumenta **name** i **query**. Atributom **name** određen je logički naziv koji se pridružuje anotaciji (po konvenciji naziv JPA klase se koristi kao deo naziva upita), dok je atributom **query** priložen [JPQL](#) upit kojeg će izvršiti obeleženi upit.

Anotacija `@NamedQueries` će biti generisana samo u slučaju kada se klikne na CheckBox pod nazivom Generate Named Query Annotations for Persistent Fields u čarobnjaku New Entity Classes from Database.

JPQL je JPA specifični jezik upita čija je sintaksa veoma slična SQL jeziku. Čarobnjak New Entity Classes from Database generiše JPQL upit za svako polje u entitetu. Kada se upit izvrši biće vraćena lista svih instanci entiteta koje odgovaraju zadatom kriterijumu upita.

Sledećim kodom je priložena jedna DAO klasa:

```
import java.util.List;  
import javax.persistence.EntityManager;
```

```
import javax.persistence.Query;
public class CustomerDAO {
    public List findCustomerByLastName(String someLastName)
    {
        //deo koda je izostavljen zbog jasnoće i preglednosti

        Query query =
            em.createNamedQuery("Customer.findByLastName");
        query.setParameter("lastName", someLastName);
        List resultList = query.getResultList();
        return resultList;
    }
}
```

Moguće je primetiti da DAO objekat sadrži metodu koja će vratiti listu **Customer** entiteta čije prezime odgovara vrednosti parametra **lastName**. Da bi navedeno bilo realizovano, neophodno je obezbediti instancu klase **javax.persistence.Query**. Ovo je obavljeno angažovanjem metode **createNamedQuery()** **EntityManager** objekta i prosleđivanjem naziva upita.

Kada su određeni svi parametri upita, pozivom metode **getResultList()**, **Query** objekta, vraćaju se svi rezultati koji odgovaraju kriterijumu upita.

▼ 4.1 Zadaci za samostalni rad

ZADACI ZA SAMOSTALNI RAD STUDENTA

Cilj je provera naučenog

Zadatak 4:

Ispraviti kod Zadataka 3 tako što za svaki obleženi upit treba kreirati upit JPQL-om koristeći value parametre po uzoru na primer 4.

▼ Poglavlje 5

Validacija zrna

PRIMER 5 - ANOTACIJE ZA PROVERU PODATAKA

Validacija zrna je uvedena sa Java EE 6.

Validacija zrna ([bean validation](#)) je uvedena sa Java EE 6 kao deo specifikacije Java [Specification Request \(JSR 303\)](#). Specifikacija provere zrna uključuje skup anotacija kojima je moguće vršiti proveru podataka. Koristeći čarobnjak za generisanje JPA klasa, razvojno okruženje NetBeans je u potpunosti iskoristilo prednosti koje validacije zrna nosi, dodavanjem navedenih anotacija odgovarajućim poljima koja odgovaraju kolonama tabele baze podataka iz koje je generisana JPA klasa.

U narodnom izlaganju biće prezentovane neke anotacije za proveru podataka koje su integrisane u [Customer](#) entitet. Polje [customerId](#) je obeleženo anotacijom [@NotNull](#) koja ne dozvoljava polju da prihvati [null](#) vrednost.

Nekoliko polja entiteta [Customer](#) je obeleženo anotacijom [@Size](#) koja ukazuje na maksimalan broj karaktera kojih osobina zrna može da prihvati. Ova vrednost je dobijena iz tabele kojom je entitet generisan.

Biće analizirano još nekoliko validacionih anotacija. Na primer, anotacija [@Pattern](#) obezbeđuje da se obeleženo polje podudara sa datim regularnim iskazom.

Sledećim listingom je izolovan kod iz JPA klase entiteta sa linijama koda koje sadrže anotacije za proveru vrednosti polja.

```
@Id
@Basic(optional = false)
@NotNull
@Column(name = "CUSTOMER_ID")
private Integer customerId;
@Size(max = 20)
@Column(name = "FIRST_NAME")
private String firstName;
@Size(max = 20)
@Column(name = "MIDDLE_NAME")
private String middleName;
@Size(max = 20)
@Column(name = "LAST_NAME")
private String lastName;
//
@Pattern(regexp="[a-z0-9!#$%&~*+/?^_`{|}~-]+(?:\\.
```

```
[a-zA-Z0-9!#$%&~*'*/=?^_`{|}~-]+)@(?:[a-zA-Z0-9](?:[a-zA-Z0-9-]*[a-zA-Z0-9])?\.\.)+[a-zA-Z0-9](?:[a-zA-Z0-9-]*[a-zA-Z0-9])?", message="Invalid email")//
if the field contains email address consider using this annotation to
enforce field validation
@Size(max = 30)
@Column(name = "EMAIL")
private String email;
```

▼ 5.1 Zadaci za samostalni rad

ZADACI ZA SAMOSTALNI RAD STUDENTA

Cilj je primena stečenog znanja

Zadatak 5:

Ispraviti kod Zadatka 4 da sadrži sve neophodne validacije.

▼ Poglavlje 6

Relacije entiteta

PRIMER 6 - ANOTACIJE RELACIJA - @ONETOMANY I @MANYTOONE ANOTACIJE

Za definisanje relacija između JPA entiteta moguće je primeniti nekoliko anotacija.

Za definisanje relacija između JPA entiteta moguće je primeniti nekoliko anotacija. Za potrebe analize i izlaganja, ponovo će u centru pažnje biti JPA entitet **Customer** kojem je demonstrirani čarobnjak za kreiranje JPA stranica mogao da dodeli izvesne anotacije koje ukazuju na kardinalnost relacija CUSTOMER tabele iz baze podataka.

Prva anotacija koja će biti obrađena je @OneToMany. Svako polje obeleženo anotacijom @OneToMany pripada tipu podataka java.util.Collection. Na jednoj strani relacije je entitet Customer (potrošač) koji može da ima više porudžbina, adresa, telefonskih bojeva i slično. U ovom scenariju rada sa kolekcijama, čarobnjak koristi generike za specificiranje tipova objekata koje je moguće dodati u kolekciju. Objekti ovih kolekcija su JPA entiteti povezani sa odgovarajućim tabelama iz aktuelne šeme baze podataka. Anotacija **@OneToMany** poseduje atribut **mappedBy**. Vrednost ovog atributa mora da odgovara nazivu polja sa druge strane relacije.

Drugu stranu relacije može da čini entitet **Address**. Ovaj entitet će svojim listingom da demonstrira mogućnost posedovanja više adresa za jednog potrošača. Entitet će biti prikazan odgovarajućim listingom u kojem se ističe još jedna anotacija **@JoinColumn**. Atribut **name** ove anotacije ukazuje na kolonu u bazi podataka sa kojom je entitet mapiran za definisanje ograničenja stranog ključa između tabela **ADDRESS** i **CUSTOMER** (konkretno, CUSTOMER_ID kolona za ADDRESS tabelu). Sledi listing JPA klase **Address.java** koji sadrži opisane linije koda:

```
package com.metropolitan.jpa;
import java.io.Serializable;
import javax.persistence.Basic;
import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.Id;
import javax.persistence.JoinColumn;
import javax.persistence.ManyToOne;
import javax.persistence.NamedQueries;
import javax.persistence.NamedQuery;
import javax.persistence.Table;
import javax.validation.constraints.NotNull;
import javax.validation.constraints.Size;
@Entity
```

```
@Table(name = "ADDRESS")
@NamedQueries({
    @NamedQuery(name = "Address.findAll",
        query = "SELECT a FROM Address a"),
    @NamedQuery(name = "Address.findById",
        query = "SELECT a FROM Address a WHERE a.addressId =
            :addressId"),
    @NamedQuery(name = "Address.findByAddrLine1",
        query = "SELECT a FROM Address a WHERE a.addrLine1 =
            :addrLine1"),
    @NamedQuery(name = "Address.findByAddrLine2",
        query = "SELECT a FROM Address a WHERE a.addrLine2 =
            :addrLine2"),
    @NamedQuery(name = "Address.findByCity",
        query = "SELECT a FROM Address a WHERE a.city = :city"),
    @NamedQuery(name = "Address.findByZip",
        query = "SELECT a FROM Address a WHERE a.zip = :zip"))})
public class Address implements Serializable {
    private static final long serialVersionUID = 1L;
    @Id
    @Basic(optional = false)
    @NotNull
    @Column(name = "ADDRESS_ID")
    private Integer addressId;
    @Size(max = 100)
    @Column(name = "ADDR_LINE_1")
    private String addrLine1;
    @Size(max = 100)
    @Column(name = "ADDR_LINE_2")
    private String addrLine2;
    @Size(max = 100)
    @Column(name = "CITY")
    private String city;
    @Size(max = 5)
    @Column(name = "ZIP")
    private String zip;
    @JoinColumn(name = "ADDRESS_TYPE_ID",
        referencedColumnName = "ADDRESS_TYPE_ID")
    @ManyToOne
    private AddressType addressType;
    @JoinColumn(name = "CUSTOMER_ID",
        referencedColumnName = "CUSTOMER_ID")
    @ManyToOne
    private Customer customer;
    @JoinColumn(name = "US_STATE_ID",
        referencedColumnName = "US_STATE_ID")
    @ManyToOne
    private UsState usState;
    //metode i konstruktori zbog jasnoće i preglednosti su izostavljeni
}
```

PRIMER 6 - ANOTACIJE RELACIJA - MANY-TO-MANY RELACIJA

Pored podrške za obrađene relacije, JPA ima podršku i za many-to-many i one-to-one relacije.

Kao nastavak na izlaganje o anotacijama `@OneToMany` i `@ManyToOne` naslanja se diskusija na temu anotacije `@ManyToMany`. Na primer, baza podataka, koja je predmet analize, može da sadrži tabele `CUSTOMER_ORDER` i `ITEM` u kojima se čuvaju podaci o porudžbinama potrošača i stavkama porudžbine, respektivno. Jedna porudžbina može da sadrži više proizvoda i jedan proizvod može da bude sadržan u više porudžbina.

Sada se prilaže kodovi datoteka `Item.java` i `CustomerOrder.java` koje odgovaraju JPA entitetu koji je deo many-to-many relacije.

```
package commetropolitan.jpa;
import java.io.Serializable;
import java.util.Collection;
import javax.persistence.Basic;
import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.Id;
import javax.persistence.ManyToMany;
import javax.persistence.NamedQueries;
import javax.persistence.NamedQuery;
import javax.persistence.Table;
import javax.validation.constraints.NotNull;
import javax.validation.constraints.Size;
@Entity
@Table(name = "ITEM")
@NamedQueries({
    @NamedQuery(name = "Item.findAll", query = "SELECT i FROM Item i"),
    @NamedQuery(name = "Item.findById", query = "SELECT i FROM Item i WHERE i.itemId = :itemId"),
    @NamedQuery(name = "Item.findByIdNumber", query = "SELECT i FROM Item i WHERE i.itemNumber = :itemNumber"),
    @NamedQuery(name = "Item.findByIdShortDesc", query = "SELECT i FROM Item i WHERE i.itemShortDesc = :itemShortDesc"),
    @NamedQuery(name = "Item.findByIdLongDesc", query = "SELECT i FROM Item i WHERE i.itemLongDesc = :itemLongDesc")})
public class Item implements Serializable {
    private static final long serialVersionUID = 1L;
    @Id
    @Basic(optional = false)
    @NotNull
    @Column(name = "ITEM_ID")
    private Integer itemId;
    @Size(max = 10)
    @Column(name = "ITEM_NUMBER")
    private String itemNumber;
```

```
@Size(max = 100)
@Column(name = "ITEM_SHORT_DESC")
private String itemShortDesc;
@Size(max = 500)
@Column(name = "ITEM_LONG_DESC")
private String itemLongDesc;
@ManyToMany(mappedBy = "itemCollection")
private Collection<CustomerOrder> customerOrderCollection;
//metode i konstruktori izostavljeni zbog preglednosti.
}
```

```
package com.metropolitan.jpa;
import java.io.Serializable;
import java.util.Collection;
import javax.persistence.Basic;
import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.Id;
import javax.persistence.JoinColumn;
import javax.persistence.JoinTable;
import javax.persistence.ManyToMany;
import javax.persistence.ManyToOne;
import javax.persistence.NamedQueries;
import javax.persistence.NamedQuery;
import javax.persistence.Table;
import javax.validation.constraints.NotNull;
import javax.validation.constraints.Size;
@Entity
@Table(name = "CUSTOMER_ORDER")
@NamedQueries({
    @NamedQuery(name = "CustomerOrder.findAll",
        query = "SELECT c FROM CustomerOrder c"),
    @NamedQuery(name = "CustomerOrder.findByIdByCustomerOrderId",
        query = "SELECT c FROM CustomerOrder c WHERE c.customerOrderId = :customerOrderId"),
    @NamedQuery(name = "CustomerOrder.findByIdByOrderNumber",
        query = "SELECT c FROM CustomerOrder c WHERE c.orderNumber = :orderNumber"),
    @NamedQuery(name = "CustomerOrder.findByIdByOrderDescription",
        query = "SELECT c FROM CustomerOrder c WHERE c.orderDescription = :orderDescription")})
public class CustomerOrder implements Serializable {
    private static final long serialVersionUID = 1L;
    @Id
    @Basic(optional = false)
    @NotNull
    @Column(name = "CUSTOMER_ORDER_ID")
    private Integer customerOrderId;
    @Size(max = 10)
    @Column(name = "ORDER_NUMBER")
    private String orderNumber;
    @Size(max = 200)
```



```
@Column(name = "ORDER_DESCRIPTION")
private String orderDescription;
@JoinTable(name = "ORDER_ITEM", joinColumns = {
@JoinColumn(name = "CUSTOMER_ORDER_ID",
referencedColumnName = "CUSTOMER_ORDER_ID")),
inverseJoinColumns = {
@JoinColumn(name = "ITEM_ID",
referencedColumnName = "ITEM_ID"))
@ManyToMany
private Collection<Item> itemCollection;
@JoinColumn(name = "CUSTOMER_ID", referencedColumnName =
"CUSTOMER_ID")
@ManyToOne
private Customer customer;
//konstruktori i metode su izostavljeni iz razloga preglednosti
}
```

PRIMER 6 - ANOTACIJE RELACIJA - ONE-TO-ONE RELACIJA

Neophodno je obraditi još jednu anotaciju.

Na kraju moguće je kreirati one-to-one tip relacije između dva JPA entiteta. Strana koja poseduje relaciju mora da poseduje polje JPA entiteta sa druge strane relacije i ova polja moraju da budu obeležena anotacijama **@OneToOne** i **@JoinColumn**.

U nedostatku adekvatne ilustracije, u tekućem primeru, za relaciju tipa **@OneToOne** biće kreirane dve nove datoteke kojima se demonstrira primena ove anotacije.

Na primer, osoba oblači košulju. Svaka osoba poseduje jedno dugme na pupku, jedno dugme na pupku može biti na jednoj osobi. To je prikazano sledećim datotekama.

```
@Entity
public class Person implements Serializable {
@JoinColumn(name="BELLY_BUTTON_ID")
@OneToOne
private BellyButton bellyButton;
public BellyButton getBellyButton(){
return bellyButton;
}
public void setBellyButton(BellyButton bellyButton){
this.bellyButton = bellyButton;
}
}
```

```
@Entity
@Table(name="BELLY_BUTTON")
public class BellyButton implements Serializable(
{
@OneToOne(mappedBy="bellyButton")
```

```
private Person person;  
public Person getPerson(){  
    return person;  
}  
public void setPerson(Person person){  
    this.person=person;  
}  
}
```

▼ 6.1 Zadaci za samostalni rad

ZADACI ZA SAMOSTALNI RAD STUDENTA

Cilj je provežbavanje stečenog znanja

Zadatak 6:

Ispraviti kod Zadatka 5 i dodati sve neophodne relacije entiteta.

▼ Poglavlje 7

Kreiranje JSF aplikacija iz JPA entiteta

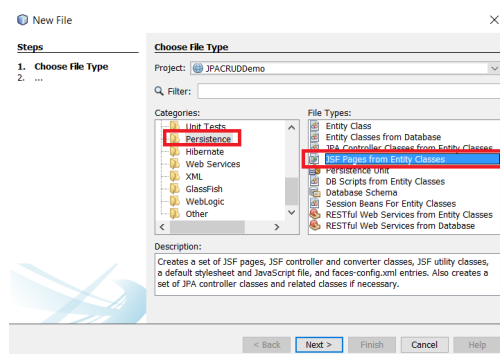
KREIRANJE JSF DATOTEKA

Kreiranjem JSF datoteka kompletira se veb aplikacija.

Budući da su u prethodnom radu kreirani JPA entiteti, koji predstavljaju objektno - orijentisanu implementaciju tabela baze podataka, a zatim su kreirani i odgovarajuće DAO datoteke za upravljanje podacima u okviru aplikacije, neophodno je razviti mehanizme pomoću kojih će korisnim moći da interaguje sa aplikacijom.

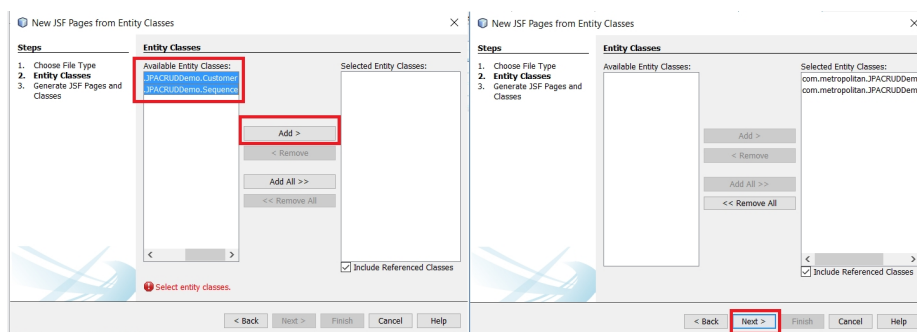
U aktuelnom primeru, primenom razvojnog alata NetBeans IDE, biće pokazano kako je moguće automatski kreirati JSF stranice i na taj način učiniti veb aplikaciju koja se razvija kompletno funkcionalnom.

U razvojnom okruženju, za aktuelni projekat, bira se opcija **File**, a zatim i **New File**. Otvara se prozor **New File** u kojem se bira kategorija datoteke (u ovom slučaju **JavaServer Faces Persistence**) i tip datoteke (u ovom slučaju **JSF Pages from Entity Classes**) kao na sledećoj slici.

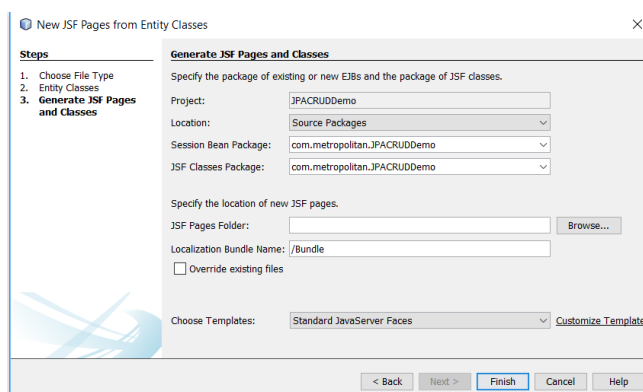


Slika 7.1.1 Kreiranje novih JSF Pages from Entity Classes datoteka

Klikom na dugme **Next** otvara se prozor za izbor klasa JPA entiteta. Klase se markiraju i klikom na dugme **Add** dodaju za dalji proces kreiranja JSF stranica (Slika2). Klikom da **Next** odlazi se u prozor gde se završava ovaj proces kreiranja JSF stranica (Slika3).



Slika 7.1.2 Izbor JPA entiteta za generisanje JSF datoteka



Slika 7.1.3 Kreiranje JSF stranica - završni korak

JSF DATOTEKE - POČETNA STRANICA

Prolaskom kroz prikazani čarobnjak kreirane su JSF stranice.

Prolaskom kroz prikazani čarobnjak kreirane su JSF stranice za sve klase entiteta koje su bile izabrane. Kreirana je i **index.xhtml** stranica od koje počinje navigacija kroz kreiranu veb aplikaciju. Listing stranice **index.xhtml** je priložen:

```
<?xml version='1.0' encoding='UTF-8' ?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:h="http://xmlns.jcp.org/jsf/html">
  <h:head>
    <title>Facelet Title</title>
    <h:outputStylesheet name="css/jsfcrud.css"/>
  </h:head>
  <h:body>
    Hello from Facelets

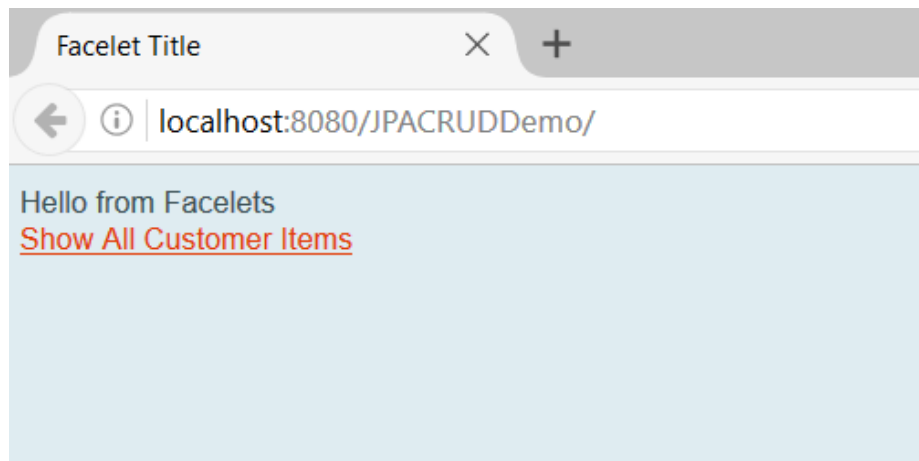
    <h:link outcome="/customer/List" value="Show All Customer Items"/>
  </h:body>
```

```
</html>
```

Na početnoj stranici definisan je link kojim se vrši navigacija ka stranici `List.xhtml`, koja je generisana u folderu `customer` i koja je nastala iz JPA klase `Customer`. Još neke datoteke su na isti način našle svoje mesto u ovom folderu, a to su: `Create.xhtml`, `Edit.xhtml` i `View.xhtml`.

Stranicom `List.xhtml` se prikazuje lista dostupnih korisnika iz baze podataka; `Create.xhtml` kreira novog korisnika; `Edit.xhtml` dozvoljava da se podaci o korisniku ažuriraju i `View.xhtml` prikazuje podatke o korisniku.

Zbog jednostavnosti, napravljena je prosta veb aplikacija čije JSF stranice se odnose na JPA klasu `Customer.java`. Pokretanjem aplikacije, učitava se stranica `index.xhtml` na način prikazan sledećom slikom:

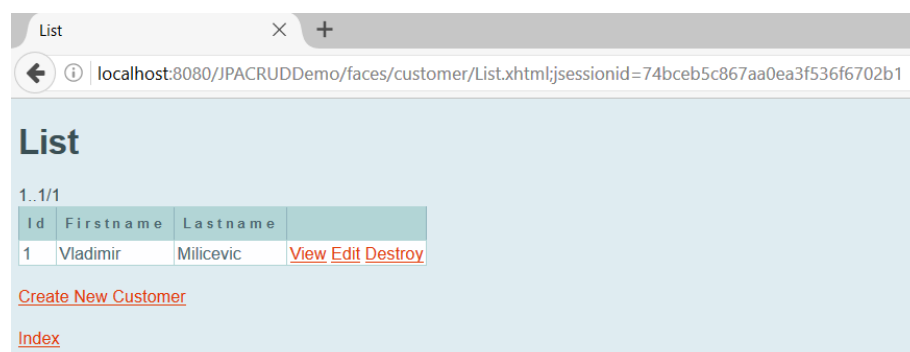


Slika 7.1.4 Automatski generisan JSF stranica `index.xhtml`

OSTALE JSF DATOTEKE - LIST.XHTML

Demonstracija ostalih automatski generisanih JSF datoteka iz JPA klasa entiteta.

Klikom na link `Show All Customer Items`, na početnoj stranici, vrši se navigacija do sledeće automatski generisane stranice `List.xhtml`. Ovom stranicom se prikazuje lista svih korisnika koji su dostupni u bazi podataka (sledeća slika).



Slika 7.1.5 Lista korisnika iz baze podataka

Kada se pogleda priložena stranica primećuju se linkovi ka ostalim stranicama, pored prikazane liste korisnika. Moguće je kreirati novog korisnika i dodati ga u bazu podataka (stranica [Create.xhtml](#)) klikom na link [Create New Customer](#). Moguće je, klikom na link [View](#), otići na stranicu sa istim nazivom i pogledati podatke za konkretnog korisnika. Takođe, klikom na link [Edit](#), omogućen je odlazak na stranicu za ažuriranje korisničkih podataka.

Na kraju, moguće je brisanje korisnika ([Destroy](#)) i povratak na početnu stranicu ([Index](#)).

Sledećim listingom priložen je JSF kod stranice [List.xhtml](#).

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/
TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:ui="http://xmlns.jcp.org/jsf/facelets"
      xmlns:h="http://xmlns.jcp.org/jsf/html"
      xmlns:f="http://xmlns.jcp.org/jsf/core"
      xmlns:p="http://primefaces.org/ui">

  <ui:composition template="/template.xhtml">

    <ui:define name="title">
      <h:outputText value="#{bundle.ListCustomerTitle}"></h:outputText>
    </ui:define>

    <ui:define name="body">
      <h:form id="CustomerListForm">
        <p:panel header="#{bundle.ListCustomerTitle}">
          <p:dataTable id="datalist" value="#{customerController.items}"
            var="item"
            selectionMode="single"
            selection="#{customerController.selected}"
            paginator="true"
            rowKey="#{item.id}"
            rows="10"
            rowsPerPageTemplate="10,20,30,40,50"
            >

            <p:ajax event="rowSelect" update="createButton viewButton
editButton deleteButton"/>
            <p:ajax event="rowUnselect" update="createButton viewButton
editButton deleteButton"/>

            <p:column>
              <f:facet name="header">
                <h:outputText
value="#{bundle.ListCustomerTitle_firstName}">
              </f:facet>
              <h:outputText value="#{item.firstName}">
            </p:column>
```

```

        <p:column>
            <f:facet name="header">
                <h:outputText
value="#{bundle.ListCustomerTitle_lastName}"/>
            </f:facet>
            <h:outputText value="#{item.lastName}"/>
        </p:column>
        <p:column>
            <f:facet name="header">
                <h:outputText
value="#{bundle.ListCustomerTitle_id}"/>
            </f:facet>
            <h:outputText value="#{item.id}"/>
        </p:column>
        <f:facet name="footer">
            <p:commandButton id="createButton"
icon="ui-icon-plus" value="#{bundle.Create}"
actionListener="#{customerController.prepareCreate}" update=":CustomerCreateForm"
oncomplete="PF('CustomerCreateDialog').show()"/>
            <p:commandButton id="viewButton"
icon="ui-icon-search" value="#{bundle.View}" update=":CustomerViewForm"
oncomplete="PF('CustomerViewDialog').show()" disabled="#{empty
customerController.selected}"/>
            <p:commandButton id="editButton"
icon="ui-icon-pencil" value="#{bundle.Edit}" update=":CustomerEditForm"
oncomplete="PF('CustomerEditDialog').show()" disabled="#{empty
customerController.selected}"/>
            <p:commandButton id="deleteButton"
icon="ui-icon-trash" value="#{bundle.Delete}"
actionListener="#{customerController.destroy}" update=":growl,datalist"
disabled="#{empty customerController.selected}"/>
        </f:facet>
    </p:dataTable>
</p:panel>
</h:form>

    <ui:include src="Create.xhtml"/>
    <ui:include src="Edit.xhtml"/>
    <ui:include src="View.xhtml"/>
</ui:define>
</ui:composition>

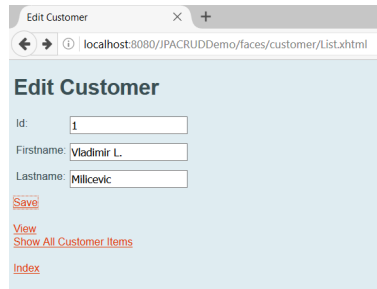
</html>

```

OSTALE JSF DATOTEKE - EDIT.XHTML

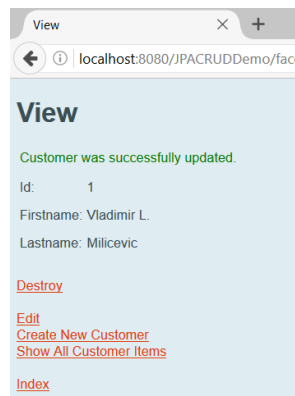
Ovom stranicom je omogućeno ažuriranje podataka, iz baze podataka, o korisniku.

U prethodnom izlaganju je opisan način kako se stiže do stranice za ažuriranje korisničkih podataka koja je prikazana sledećom slikom.



Slika 7.1.6 Ažuriranje korisničkih podataka

Kada se izvesne informacije promene, klikom na link **Save** one će biti zapamćene (sledeća slika)



Slika 7.1.7 Informacija o ažuriranju podataka.

Kao što je moguće videti i ova stranica dozvoljava navigaciju ka početnoj stranici, ali i stranicama za pregled korisnikovih podataka, kao i liste svih korisnika. Sledi listing stranice **Edit.xhtml**.

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/
TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:ui="http://xmlns.jcp.org/jsf/facelets"
      xmlns:h="http://xmlns.jcp.org/jsf/html"
      xmlns:f="http://xmlns.jcp.org/jsf/core">

  <ui:composition template="/template.xhtml">
    <ui:define name="title">
      <h:outputText value="#{bundle.EditCustomerTitle}"></h:outputText>
    </ui:define>
    <ui:define name="body">
      <h:panelGroup id="messagePanel" layout="block">
        <h:messages errorStyle="color: red" infoStyle="color: green"
          layout="table"/>
      </h:panelGroup>
      <h:form>
```



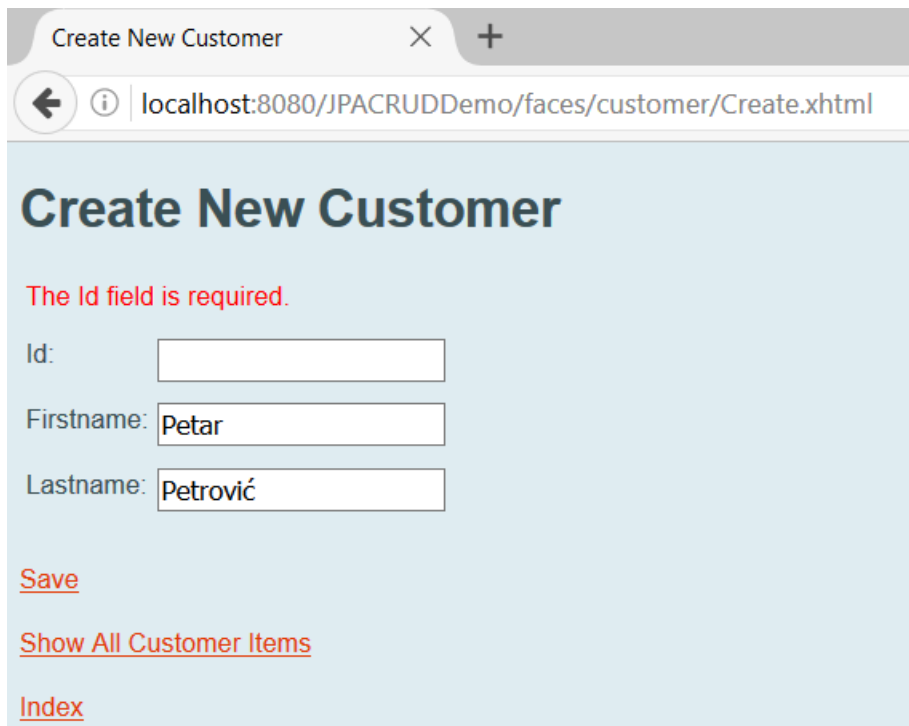
```
<h:panelGrid columns="2">
    <h:outputLabel value="#{bundle.EditCustomerLabel_id}" for="id"
/>
    <h:inputText id="id" value="#{customerController.selected.id}"
title="#{bundle.EditCustomerTitle_id}" required="true"
requiredMessage="#{bundle.EditCustomerRequiredMessage_id}"/>
    <h:outputLabel value="#{bundle.EditCustomerLabel_firstname}"
for="firstname" />
    <h:inputText id="firstname"
value="#{customerController.selected.firstname}"
title="#{bundle.EditCustomerTitle_firstname}" />
    <h:outputLabel value="#{bundle.EditCustomerLabel_lastname}"
for="lastname" />
    <h:inputText id="lastname"
value="#{customerController.selected.lastname}"
title="#{bundle.EditCustomerTitle_lastname}" />
</h:panelGrid>
<h:commandLink action="#{customerController.update}"
value="#{bundle.EditCustomerSaveLink}"/>
<br/>
<br/>
<h:link outcome="View" value="#{bundle.EditCustomerViewLink}"/>
<br/>
<h:commandLink action="#{customerController.prepareList}"
value="#{bundle.EditCustomerShowAllLink}" immediate="true"/>
<br/>
<br/>
<h:link outcome="/index" value="#{bundle.EditCustomerIndexLink}" />
</h:form>
</ui:define>
</ui:composition>

</html>
```

OSTALE JSF DATOTEKE - CREATE.XHTML

Automatski je generisana i JSF stranica za kreiranje i unos novih korisnika u bazu podataka.

Automatski je generisana i JSF stranica za kreiranje i unos novih korisnika u bazu podataka, koja je dobila naziv **Create.xhtml** i prikazana je sledećom slikom.



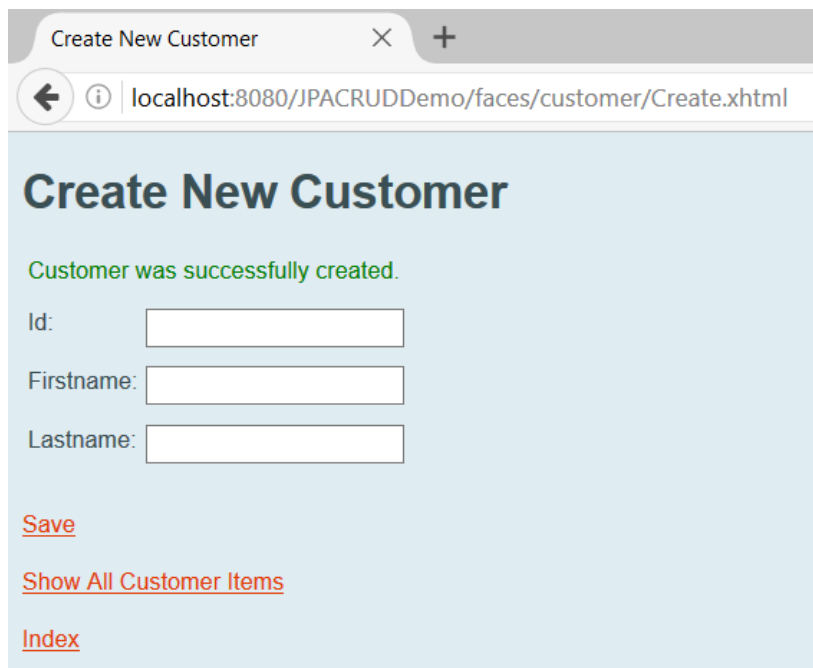
The screenshot shows a web browser window with the title 'Create New Customer' and the URL 'localhost:8080/JPACRUDDemo/faces/customer/Create.xhtml'. The page has a light blue background and a dark blue header. The main content area contains the title 'Create New Customer' in large, bold, dark blue font. Below the title, a red error message reads 'The Id field is required.' There are three input fields: 'Id:' (empty), 'Firstname:' (containing 'Petar'), and 'Lastname:' (containing 'Petrović'). Below the input fields, there are three links: 'Save' (underlined, red), 'Show All Customer Items' (underlined, red), and 'Index' (underlined, red).

Slika 7.1.8 Kreiranje novog korisnika

Na slici je moguće primetiti da je sva polja, koja su obavezna, neophodno popuniti inače se javlja poruka sa greškom.

Takođe, stranica obuhvata navigaciju ka početnoj stranici i ka stranici koja prikazuje listu svih postojećih korisnika iz baze podataka. Link **Save** realizuje snimanje unetih podataka o novom korisniku.

Forma sa Slike 8, korektno je popunjena, izvršeno je snimanje korisnika i dobijena je informacija da je korisnik uspešno sačuvan u bazi podataka.



The screenshot shows the same web browser window as before, but the error message has been replaced by a green success message: 'Customer was successfully created.' The input fields are now empty. The 'Save' link is still underlined and red, and the 'Show All Customer Items' and 'Index' links are also underlined and red.

Slika 7.1.9 Dodati je nov korisnik

KREIRANJE NOVOG KORISNIKA - DEMO

Ponovno učitavanje stranice List.xhtml za prikazivanje ažurirane liste korisnika.

Prethodno prikazana stranica **Create.xhtml** realizovana je kodom koji je priložen sledećim listingom.

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/
TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:ui="http://xmlns.jcp.org/jsf/facelets"
      xmlns:h="http://xmlns.jcp.org/jsf/html"
      xmlns:f="http://xmlns.jcp.org/jsf/core"
      xmlns:p="http://primefaces.org/ui">

  <ui:composition template="/template.xhtml">

    <ui:define name="title">
      <h:outputText value="#{bundle.ListCustomerTitle}"></h:outputText>
    </ui:define>

    <ui:define name="body">
      <h:form id="CustomerListForm">
        <p:panel header="#{bundle.ListCustomerTitle}">
          <p:dataTable id="datalist" value="#{customerController.items}"
            var="item"
              selectionMode="single"
            selection="#{customerController.selected}"
              paginator="true"
              rowKey="#{item.id}"
              rows="10"
              rowsPerPageTemplate="10,20,30,40,50"
            >

            <p:ajax event="rowSelect" update="createButton viewButton
            editButton deleteButton"/>
            <p:ajax event="rowUnselect" update="createButton viewButton
            editButton deleteButton"/>

            <p:column>
              <f:facet name="header">
                <h:outputText
            value="#{bundle.ListCustomerTitle_firstName}" />
              </f:facet>
              <h:outputText value="#{item.firstName}" />
            </p:column>
```

```

        <p:column>
            <f:facet name="header">
                <h:outputText
value="#{bundle.ListCustomerTitle_lastName}"/>
            </f:facet>
            <h:outputText value="#{item.lastName}"/>
        </p:column>
        <p:column>
            <f:facet name="header">
                <h:outputText
value="#{bundle.ListCustomerTitle_id}"/>
            </f:facet>
            <h:outputText value="#{item.id}"/>
        </p:column>
        <f:facet name="footer">
            <p:commandButton id="createButton"
icon="ui-icon-plus" value="#{bundle.Create}"
actionListener="#{customerController.prepareCreate}" update=":CustomerCreateForm"
oncomplete="PF('CustomerCreateDialog').show()"/>
            <p:commandButton id="viewButton"
icon="ui-icon-search" value="#{bundle.View}" update=":CustomerViewForm"
oncomplete="PF('CustomerViewDialog').show()" disabled="#{empty
customerController.selected}"/>
            <p:commandButton id="editButton"
icon="ui-icon-pencil" value="#{bundle.Edit}" update=":CustomerEditForm"
oncomplete="PF('CustomerEditDialog').show()" disabled="#{empty
customerController.selected}"/>
            <p:commandButton id="deleteButton"
icon="ui-icon-trash" value="#{bundle.Delete}"
actionListener="#{customerController.destroy}" update=":growl,datalist"
disabled="#{empty customerController.selected}"/>
        </f:facet>
    </p:dataTable>
</p:panel>
</h:form>

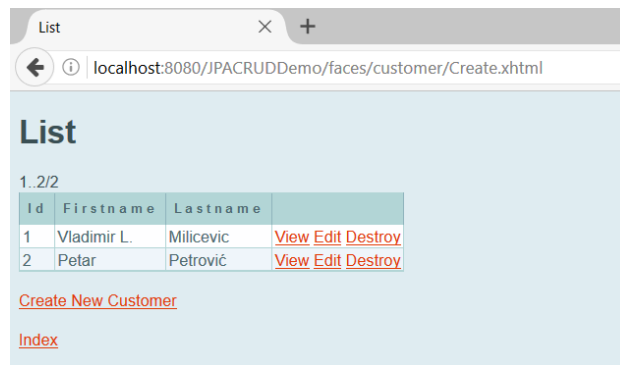
    <ui:include src="Create.xhtml"/>
    <ui:include src="Edit.xhtml"/>
    <ui:include src="View.xhtml"/>
</ui:define>
</ui:composition>

</html>

```

Kao što je moguće primetiti odavde je moguće ponovna navigacija ka stranici koja prikazuje listu korisnika iz baze podataka. Klikom na link **Show All Customer Items** još jednom će se proveriti funkcionalnost aplikacije i zaokružiti izlaganje o bazama podataka u JPA veb aplikacijama.

Ažurirana lista korisnika prikazana je sledećom slikom.



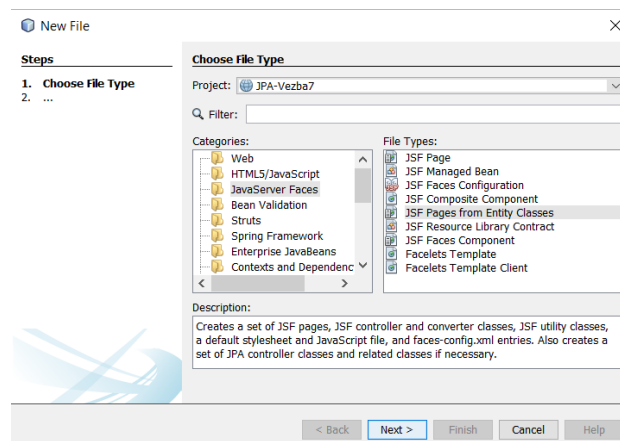
Ova lekcija sadrži video materijal. Ukoliko želite da pogledate ovaj video morate da otvorite LAMS lekciju.

7.1 Pokazni primeri

PRIMER 7 - KREIRANJE JSF STRANICA IZ JPA ENTITETA

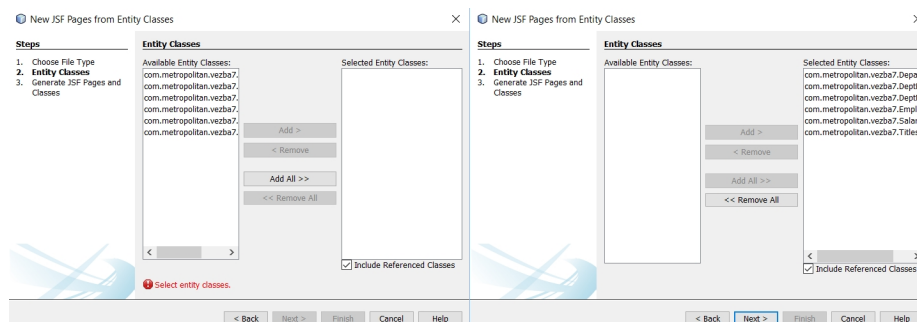
Veb aplikacija se zaokružuje generisanjem JSF stranica.

U prozoru New File bira se kategorija datoteke (u ovom slučaju JavaServer Faces ili Persistence) i tip datoteke (u ovom slučaju JSF Pages from Entity Classes) kao na sledećoj slici.



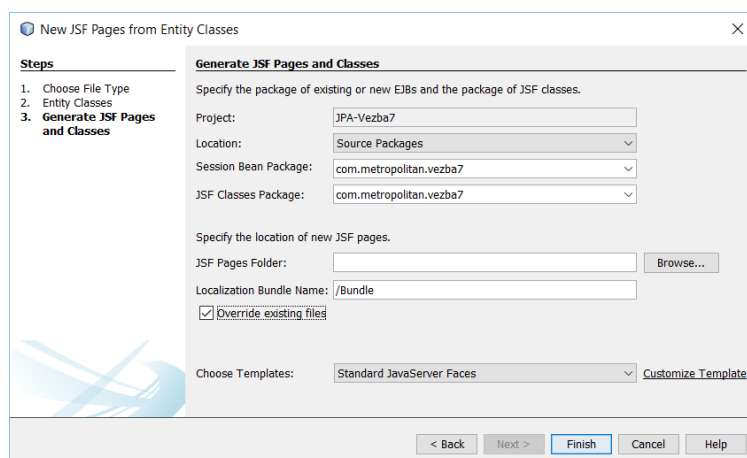
Slika 7.2.1 Izbor kreiranja JSF stranica iz JPA entiteta

Dalje, ponove se srećemo za prozorom za izbor JPA klasa entiteta, ali ovaj put za generisanje JSF stranica.



Slika 7.2.2 Izbor klasa JPA entiteta

Konačno biraju se paketi za zrna sesije i JSF klase. Aplikacija je generisana.



Slika 7.2.3 Kraj zadatka

▼ 7.2 Zadaci za samostalni rad

ZADACI ZA SAMOSTALNI RAD STUDENTA

Pokušajte sami

Zadatak 7 :

Nad JPA entitetima Zadatka 6 kreirati JSF stranice. Pokrenuti aplikaciju.

Zadatak 8:

Pokušajte da sredite GUI kreiranih JSF stranica iz prethodnog primera 7.

Zadatak 9:

Pokušajte da iskoristite neke od obrađenih biblioteka komponenata.

Zadatak 10:

Uposlite aplikaciju i proverite korektnost urađenih zadataka.

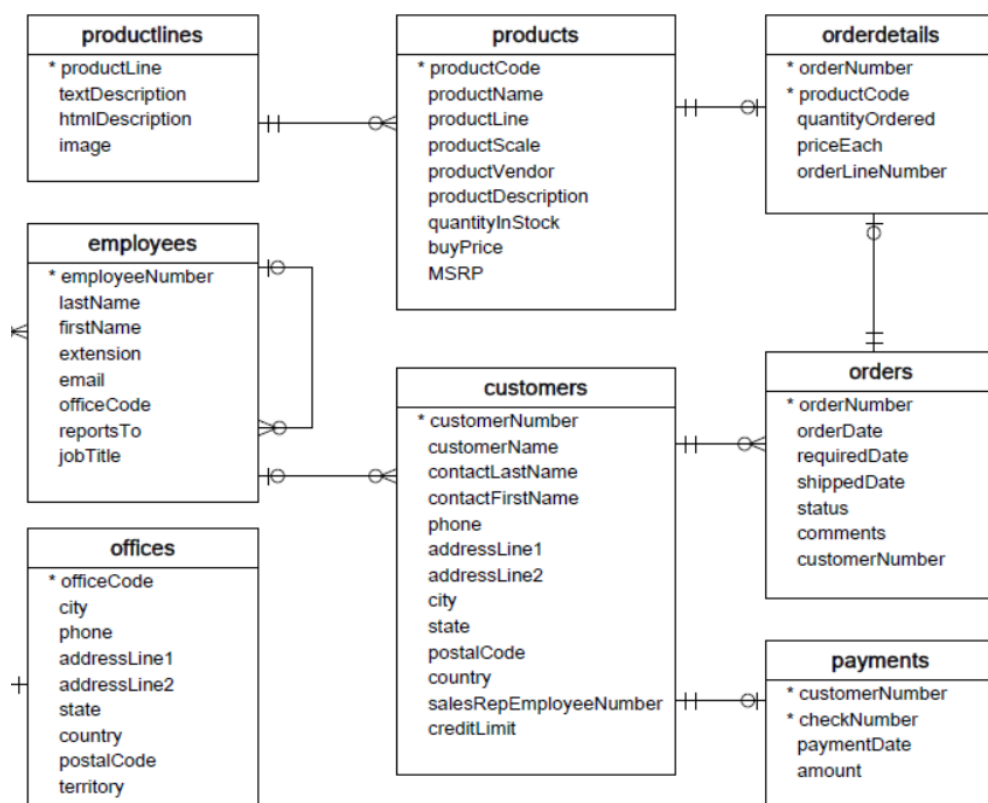
▼ Poglavlje 8

Domaći zadatak

DOMAĆI ZADACI

Za ove zadatke se ne daje rešenje i očekuje se da svaki student pokuša samostalno rešavanje istih

Po uzoru na primere 1-7 kreirajte veb aplikaciju nad bazom podataka koja je data šemom sa sledeće slike.



Slika 8.1 Šema baze podataka

▼ Poglavlje 9

Zaključak

ZAKLJUČAK

Lekcija će se bavila radom sa bazama podataka primenom Java Persistence API (JPA).

U prethodnom izlaganju lekcija je istakla da **Java Persistence API (JPA)** predstavlja API za objektno - relaciono mapiranje (**object - relational mapping** - ORM). Takođe, posebno je u lekciji naglašeno da ORM alati pomažu prilikom automatizovanja mapiranja Java objekata u tabele relacionih baza podataka. Prve verzije J2EE koristile su entitetska zrna kao standardni ORM pristup. Entitetska zrna (entity beans) su uvek pokušavala da drže sinhronizovanim podatke koji se čuvaju u memoriji sa podacima baza podataka. Iako je ovo odlična ideja, u praksi je pokazala ozbiljne nedostatke koji su se reflektovali kroz loše performanse aplikacija.

Lekcija ističe da je JPA integrisala ideje nekoliko ORM alata i postavila ih kao standard. Upravo tim alatima se ova lekcija detaljno bavila.

Lekcija je stavila fokus na sledeće teme:

- Kreiranje JPA entiteta;
- Interakcija sa JPA entitetima primenom klase EntityManager;
- Generisanje JPA entiteta iz postojećih šema baza podataka;
- Korišćene JPA upita i Java Persistence Query Language (JPQL);
- Kreiranje JSF aplikacije sa JPA entitetima.

Savladavanjem ove lekcije studenti razumeju ORM pristup i vladaju primenom ORM alata u radu sa bazama podataka složenih Java EE veb aplikacija..

LITERATURA

Za pripremu lekcije korišćena je najnovija literatura.

1. Eric Jendrock, Ricardo Cervera-Navarro, Ian Evans, Kim Haase, William Markito. 2014. Java Platform, Enterprise Edition The Java EE Tutorial, Release 7, ORACLE
2. David R. Heffelfinger. 2015. Java EE7 Developomnet With NetBeans 8, PACK Publishing
3. Yakov Fain. 2015. Java 8 programiranje, Kombib (Wiley)
4. Josh JUneau. 2015. Java EE7 Recipes, Apress
5. <https://www.tutorialspoint.com/jsf/>
6. <https://www.tutorialspoint.com/jpa/>
7. <http://www.mysqltutorial.org/mysql-sample-database.aspx>