

IŠ PRAEITOS PASKAITOS - LOMBOK

- @Data anotacija leidžia nerašyti get/set metodų
- Norint, kad neraudonuotų sakiniai IDE, reikia
 - atsisiųsti [lombok.jar failą](#)
 - uždaryti Eclipse ir paleisti

```
$ java -jar lombok.jar
```

- nurodyti, kur yra eclipse, ir instaliuoti
 - pvz. ~/eclipse/jee-2018-09/eclipse/
 - paleisti Eclipse ir atnaujinti projektus refresh/clean
- Idea turėtų padėti [Lombok plugin'as](#)





AKADEMIJA.IT

INFOBALT IR TECH CITY

ORM. JPA. SPRING DATA/DB PRIJUNGIMAS PRIE SPRING BOOT

Andrius Stašauskas

andrius@stasauskas.lt

<http://stasauskas.lt/itpro2018/>

TURINYS

- Technologijos
 - nuo JDBC iki Hibernate
- ORM
- JPA
- EntityManager
 - Transakcijos
- Servisų kūrimas
- Spring Data
 - DAO
- JPQL užklausų kalba



TECHNOLOGIJOS. NUO JDBC IKI HIBERNATE



AKADEMIJA.IT

INFOBALT IR TECH CITY

JDBC 1.X 1997 JAVA 1

- Java Database Connectivity

```
CREATE TABLE 'employee' (
  'CURR_ID' int(10) unsigned NOT NULL AUTO_INCREMENT,
  'NAME' varchar(40) NOT NULL,
  'AGE' int(10) unsigned NOT NULL,
  PRIMARY KEY ('CURR_ID')
) ENGINE=InnoDB AUTO_INCREMENT=2 DEFAULT CHARSET=utf8;

import java.sql.*;

public class EmployeeDAO implements EmployeeDAOI {
    public void save(int empno, String name, double sal, int dept) throws Exception {
        //Here we want to insert the given employee details to the DB
        //To do this we need to execute the following sql statement to DB
        String sql = "insert into emp values('"+empno+"','"+name+"','"+sal+"','"+dept+"')";
        Connection con = null;
        try {
            Class.forName("oracle.jdbc.driver.OracleDriver");
            con = DriverManager.getConnection("jdbc:oracle:thin@localhost:1521:orcl","system","manager");
            Statement st = con.createStatement();
            st.executeUpdate(sql);
        } catch (Exception e) {
            e.printStackTrace();
        } finally {
            try {
                if (con != null)
                    con.close();
            } catch (Exception e) {}
        }
    }

    public boolean updateSal(int empno, double newSal) throws Exception {
        String sql = "update emp set sal="+newSal+" where empno="+empno;
        Connection con = null;
        boolean flag = false;
        try {
            Class.forName("oracle.jdbc.driver.OracleDriver");
            con = DriverManager.getConnection("jdbc:oracle:thin@localhost:1521:orcl","system","manager");
            Statement st = con.createStatement();
            if (st.executeUpdate(sql) > 0) {
                flag = true;
            }
        } catch (Exception e) {
            e.printStackTrace();
        } finally {
            try {
                if (con != null)
                    con.close();
            } catch (Exception e) {}
        }
        return flag;
    }

    public String getEmp(int empno) throws Exception {
        String sql = "select * from emp where empno="+empno;
        StringBuilder sb = new StringBuilder();
        Connection con = null;
        try {
            Class.forName("oracle.jdbc.driver.OracleDriver");
            con = DriverManager.getConnection("jdbc:oracle:thin@localhost:1521:orcl","system","manager");
            Statement st = con.createStatement();
            ResultSet rs = st.executeQuery(sql);
            if (rs.next()) {
                sb.append(rs.getInt(1));
                sb.append(" ");
                sb.append(rs.getString(2));
                sb.append(" ");
                sb.append(rs.getDouble(3));
                sb.append(" ");
                sb.append(rs.getInt(4));
            }
        } catch (Exception e) {
            e.printStackTrace();
        } finally {
            try {
                if (con != null)
                    con.close();
            } catch (Exception e) {}
        }
        return sb.toString();
    }

    public boolean delete(int empno) throws Exception {
        String sql = "delete from emp where empno="+empno;
        Connection con = null;
        boolean flag = false;
        try {
            Class.forName("oracle.jdbc.driver.OracleDriver");
            con = DriverManager.getConnection("jdbc:oracle:thin@localhost:1521:orcl","system","manager");
            Statement st = con.createStatement();
            if (st.executeUpdate(sql) > 0) {
                flag = true;
            }
        } catch (Exception e) {
            e.printStackTrace();
        } finally {
            try {
                if (con != null)
                    con.close();
            } catch (Exception e) {}
        }
        return flag;
    }
}
```



JDBC 2.X/DATASOURCE/DRIVERMANAGER 1998 JAVA 2

- DB drivers, parameters validation

```
CREATE TABLE `employee` (
  `CUST_ID` int(10) unsigned NOT NULL AUTO_INCREMENT,
  `NAME` varchar(100) NOT NULL,
  `AGE` int(10) unsigned NOT NULL,
  PRIMARY KEY (`CUST_ID`)
) ENGINE=InnoDB AUTO_INCREMENT=2 DEFAULT CHARSET=utf8;

package com.mkyong.customer.dao.impl;

import java.sql.Connection;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Statement;
import com.mkyong.customer.dao.CustomerDAO;
import com.mkyong.customer.model.Customer;

public class JdbcCustomerDAO implements CustomerDAO
{
    private DataSource dataSource;

    public void setDataSource(DataSource dataSource) {
        this.dataSource = dataSource;
    }

    public void insert(Customer customer){
        String sql = "INSERT INTO CUSTOMER " +
            "(CUST_ID, NAME, AGE) VALUES (?, ?, ?)";
        Connection conn = null;

        try {
            conn = dataSource.getConnection();
            PreparedStatement ps = conn.prepareStatement(sql);
            ps.setInt(1, customer.getCustId());
            ps.setString(2, customer.getName());
            ps.setInt(3, customer.getAge());
            ps.executeUpdate();
            ps.close();

        } catch (SQLException e) {
            throw new RuntimeException(e);
        } finally {
            if (conn != null) {
                try {
                    conn.close();
                } catch (SQLException e) {}
            }
        }
    }

    public Customer findByIdCustomerId(int custId){
        String sql = "SELECT * FROM CUSTOMER WHERE CUST_ID = ?";
        Connection conn = null;

        try {
            conn = dataSource.getConnection();
            PreparedStatement ps = conn.prepareStatement(sql);
            ps.setInt(1, custId);
            Customer customer = null;
            ResultSet rs = ps.executeQuery();
            if (rs.next()) {
                customer = new Customer(
                    rs.getInt("CUST_ID"),
                    rs.getString("NAME"),
                    rs.getInt("AGE")
                );
            }
            rs.close();
            ps.close();
            return customer;
        } catch (SQLException e) {
            throw new RuntimeException(e);
        } finally {
            if (conn != null) {
                try {
                    conn.close();
                } catch (SQLException e) {}
            }
        }
    }
}
```



HIBERNATE 2001

- Cross-DB SQL clauses/types, persistence of POJOs

JAVA PASAULIS PATOBULĖJO

- EJB 2 -> EJB 3 entities
 - JPA 2006
- Spring 2001
- Hibernate įgyvendino JPA specifikaciją



HIBERNATE/SPRING 2005/2011

- daugiau konfigūracijos, bet daugiau ir automatizacijos, o ir mažiau kodo
- kelių CRUD metodų+paieškos pavyzdys (CRD be update):

```
@Repository
public class DBUserDAO implements UserDao {
    @PersistenceContext private EntityManager entityManager;
    public List<User> getUsers() {
        return entityManager.createQuery("SELECT u from User u", User.class).getResultList();
    }
    public void createUser(User user) { entityManager.persist(user); }
    public void deleteUser(String username) {
        User user = entityManager
            .createQuery("SELECT u from User u where username = :un", User.class)
            .setParameter("un", username).getSingleResult();
        if (entityManager.contains(user)) { entityManager.remove(user); }
        else { entityManager.remove(entityManager.merge(user)); }
    }
}
```



HIBERNATE/SPRING DATA 1.0-1.09 2011/2016

[illegible]

2017-2018. HIBERNATE. SPRING BOOT. SPRING DATA
1.11+,2.0+



AKADEMIJA.IT

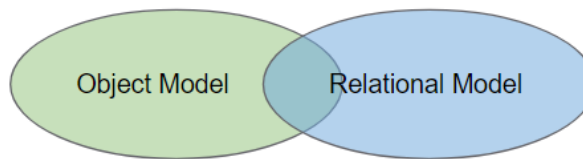
INFOBALT IR TECH CITY

ORM



OBJEKTINIO-RELIACINIO MODELIO SUSIEJIMAS (ORM)

- Dauguma programų sąveikauja su RDBVS
- Dauguma DBVS gali dirbti tik su primityviais tipais – tekstiniais laukais, datomis, skaičiais.
- RDBVS neoperuoja objektais ar sudėtiniais tipais.
- Programos autorius sprendžia kaip lentelėje esančius duomenis perkelti į objektų grafą.
- Technika, padedanti sujungti šiuos modelius (objektinį ir reliacinį) vadinama object-relation mapping (ORM).



OBJEKTINIO-RELIACINIO MODELIO SUSIEJIMAS (ORM)

- Impedance mismatch: iššūkio esmė susieti modelius, nors tam tikros koncepcijos egzistuojančios viename modelyje neturi atitikmens kitame modelyje. Pvz. Asmuo <-> Darbovietė susiejimas.
 - Polimorfizmas
 - Identitetas (equals(), ==, = (SQL))
 - Ryšiai (Java nuorodos ir išoriniai raktai).
 - Enkapsuliacija
 - Navigavimas ir skaitymas (nuorodos vs SQL rezultatų sąrašai)



OBJEKTINIO-RELIACINIO MODELIO SUSIEJIMAS (ORM)

- Duomenų saugojimo būdai RDBVS
 - Išreikštinai „gauti“ reikšmes iš objektų ir išreikštinai išsaugoti naudoti SQL INSERT sakinius.
 - Tam naudojama žemesnio lygio specifikacija JDBC.
 - Naudoti aukštesnio abstrakcijos lygio susiejimą siejant Klases iš objektinio programavimo su Lentelėmis iš RDBVS. Toks susiejimas ir vadinamas ObjectRelational Mapping (ORM).
 - Tam naudojama aukštesnio lygio specifikacija JPA.



JDBC

- JDBC atsirado pats pirmas, kaip būdas saugoti duomenis DB.
- JDBC pateikia standartizuotą abstrakciją, kaip bendrauti su įvairių gamintojų duomenų bazių sąsajomis.
 - JDBC kodas yra portabilus, **tačiau SQL kalba – nėra portabili**
 - Realybėje praktiškai nepavyks parašyti SQL kodo, kuris nepakeistas veiktų ant bet kurių dviejų populiariųjų DB platformų



JPA



AKADEMIJA.IT

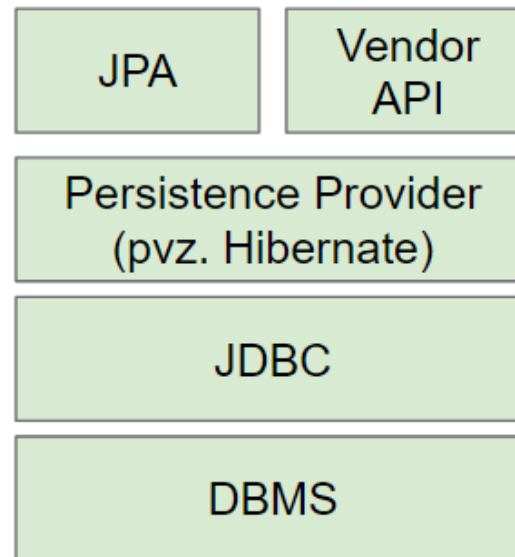
INFOBALT IR TECH CITY

JDBC VS JPA

- JPA (Java Persistence Application programming interface arba API) suteikia priemones paprastų Java objektų saugojimui duomenų bazių valdymo sistemose.
- JPA gali būti naudojamas Java SE ir EE aplinkose
- JPA gali automatiškai susieti Java objektus ir duomenų bazių lentelių (ir kitų elementų) įrašus (eilutes).
- JPA yra aukštesnio lygio abstrakcija, nei JDBC.
- Java kodas gali būti* izoliuotas nuo DBVS specifikos.
- Naujausia JPA standarto versija - 2.2 (hibernate - 2.1)



JPA ARCHITEKTŪRA



- Persistence provider rūpinasi ORM
- JDBC suteikia metodus užklausoms ir duomenų atnaujinimui duomenų bazėje

JPA SAVYBĖS

- Pakeičiama realizacija, galima naudoti skirtingų gamintojų produktus.
 - JPA – tai specifikacija. JPA realizacija – konkretus produktas, atitinkantis specifikaciją
 - JPA 2.1 Hibernate EclipseLink DataNucleus(2.2)
- Dinaminės, Type-safe užklausos.
- JPQL – gimininga SQL kalbai užklausų kalba, leidžianti daug paprasčiau operuoti objektais ir jų atributais, o ne duomenų bazės lentelėmis ir stulpeliais.
 - `SELECT c FROM Category c WHERE c.items is NOT EMPTY`



JPA SAVYBĖS

- Gali sugeneruoti DB schemų kūrimo skriptus, juos įvykdyti. Gali perkurti visą DB.
- POJO (Plain Old Java Object) Saugojimas:

```
@Entity
public class Customer implements Serializable {
    @Id protected Long id;
    protected String name;
    public Customer() {}
    public Long getId() {return id;}
    public String getName() {return name;}
    public void setName(String name) {this.name = name;}
}
```



ESYBĖS (ENTITIES)

- Reliacinėje paradigmoje esybė (Entity) dažniausiai yra atpažystamas ir identifikuojamas realaus pasaulio objekto atitikmuo, turintis atributus bei sąryšius su kitomis esybėmis. Pvz. Darbdavys, darbuotojas, darbovietė, prekė...
- Objektinio-reliacinio susiejimo pagrindas:
 - Entity klasės atitinka lenteles
 - Entity objektai atitinka įrašus lentelėse
 - Susiejimo detalės kontroliuojamos Java anotacijomis



ENTITY PAVYZDYS

```
@Entity
@Table(name = "customer")
public class Customer {
    @Id
    public int id;
    ...
    public String name;
    @Column(name="CREDIT")
    public int c_rating;
    @LOB public Image photo;
    ...
}
```



REIKALAVIMAI ENTITY KLASĖMS

- Entity klasės privalo atitikti reikalavimus:
 - Entity klasė privalo turėti bent vieną public arba protected konstruktorių be parametų
 - Entity klasė negali būti pažymėta final
 - Entity savybės ar metodai negali būti pažymėti final
 - Klasės kintamieji privalo būti private/protected/package-private pasiekiamumo ir kreipimasis į juos galimas tik per klasės metodus



REIKALAVIMAI ENTITY KLASĖMS

- Entity klasių laukai turi būti šių tipų:
 - Primityvūs
 - `java.lang.String`
 - Serializuojami tipai
 - Wrappers of Java primitive types
 - `java.math.BigInteger`, `java.math.BigDecimal`
 - `java.util.Date`, `java.util.Calendar`, `java.sql.Date`, `java.sql.Time`, `java.sql.Timestamp`
 - User-defined serializable types
 - `byte[]`, `Byte[]`, `char[]`, `Character[]`
 - Enumeruojami tipai
 - Kiti Entities tipų laukai arba jų kolekcijos
 - `@Embeddable` pažymėtos klasės



REIKALAVIMAI ENTITY KLASĖMS

- Laukai, pažymėti `javax.persistence.Transient` arba Java raktažodžiu `transient` NEbus saugomi į db.
- Entity klasės turi laikytis JavaBeans pavadinimų suteikimo taisyklių (`setProperty`, `getProperty`, `isProperty`).
- Naudojamos kolekcijos turi būti vienos iš:
 - `java.util.Collection`
 - `java.util.Set`
 - `java.util.List`
 - `java.util.Map`



TABLE

- Lentelei duomenų bazėje galime suteikti kitą pavadinimą

```
@Entity
@Table(name = "klientas")
public class Customer {
```

- Visur java ir JPA naudosime Customer, tačiau duomenų bazėje lentelė bus saugoma kaip klientas



COLUMN

- Lentelės stulpeliai žymimi @Column anotacija

```
@Column // nebūtina anotacija  
public int c_rating;  
@Column(name="DB_column", nullable=false, length=5)  
private String someProperty;
```

- Kaip ir Table atveju, pavadinimą tikroje duomenų bazėje galime pakeisti
- someProperty duomenų bazėje saugomas stulpelyje DB_column, turi ribojimą NOT NULL ir yra VARCHAR(5) tipo.



COLUMN

- Lentelės stulpeliai žymimi @Column/@Temporal anotacija

```
@Column(precision=4, scale=1)
private Double someNumber;
@Temporal(TemporalType.DATE)
private Date someDate;
@Temporal(TemporalType.TIMESTAMP)
private Date someOtherDate;
```

- DECIMAL(4, 1), max skaičius: 999.9
- someDate saugomas DATE tipo stulpelyje
- someOtherDate saugomas TIMESTAMP tipo stulpelyje



ENUMERATED

- Lentelės stulpeliai žymimi @Enumerated anotacija

```
public enum SalaryRate { JUNIOR, SENIOR, MANAGER, EXECUTIVE }  
..  
@Enumerated(String)  
private SalaryRate salaryRate;
```

- Nėra tipo atitikmens duomenų bazėse, todėl anotacija leidžia parinkti duomenų bazės atitikmenis
 - raidinę (string)
 - enumeratoriaus eilės numerio (ordinal)



TRANSIENT

- Lentelėje nesaugomi duomenys žymimi @Transient anotacija

```
@Transient  
private int myCounter;
```

- myCounter nebus saugomas į DB ir nebus valdomas



ENTITYMANAGER



ENTITYMANAGER

- Entitymanager klasė yra centrinis entity klasių valdymo taškas.
- Persistence context – registras kur saugoma entity objektų būsenos.
- Jis valdo susiejimą tarp iš anksto žinomų entity klasių ir ORM duomenų šaltinio.
 - Suteikia API užklausoms, objektų paieškai, sinchronizacijai, duomenų saugojimui DB.



ENTITYMANAGER

- `find(Class, Object)` - surasti objektą pagal pirminį raktą
- `persist(Object)` - išsaugoti ir pradėti valdyti objektą
- `merge(Object)` - išsaugoti objekto pakeitimus
- `refresh(Object)` - gauti objekto pasikeitimus iš DB
- `remove(Object)` - pašalinti objektą
- `createQuery(String)` - sukurti JPQL užklausą
- Visos operacijos gali būti taikomos ir pilnam objektų grafui



EM.PERSIST

```
Customer customer = new Customer(1, "John Bow");  
em.persist(customer);
```

- EntityManager'io klasės persist() metodas užregistruoja entity į persistence context.
- Nuo šio momento entity yra valdomas ir EntityManager užtikrina, kad šio objekto duomenys būtų sinchronizuoti su duomenų baze



EM.FIND

```
Customer customer = em.find(Customer.class, id);
```

- EntityManager find() metodas leidžia pagal entity klasę ir pirminio rakto identifikatorių surasti entity duomenų bazėje.
- Jeigu operacija sėkmingai pasibaigia, grąžintas Customer objektas taps valdomas/prižiūrimas.
 - Tačiau, jei objektas nebus surastas - find() grąžins null



EM.REMOVE

```
Customer customer = em.remove(customer);
```

- Entitymanager remove() metodas pašalina duomenis susijusius su šiuo objektu iš duomenų bazės.
- remove() metodas pašalins objektą iš persistence context. T.y. jo būseną nebebus sekama



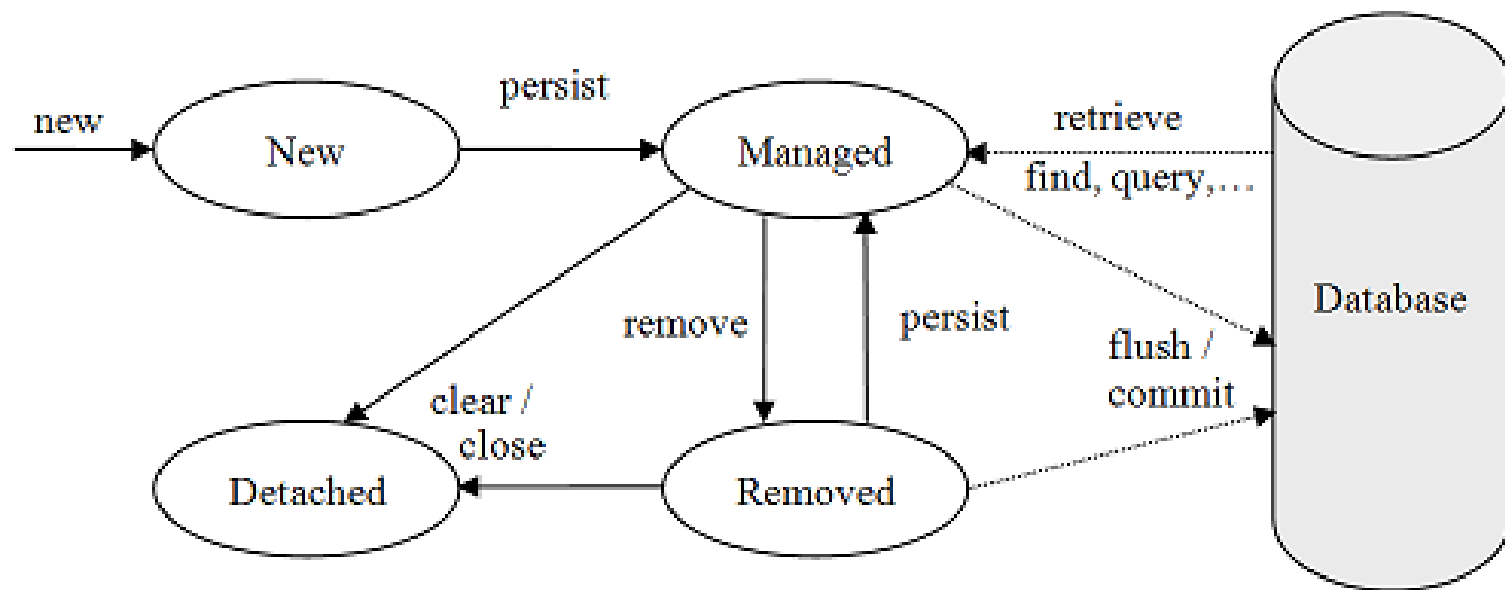
EM.CREATEQUERY

```
Query q = entityManager.createQuery("select g from Good g");  
List<Good> goods = q.getResultList();
```

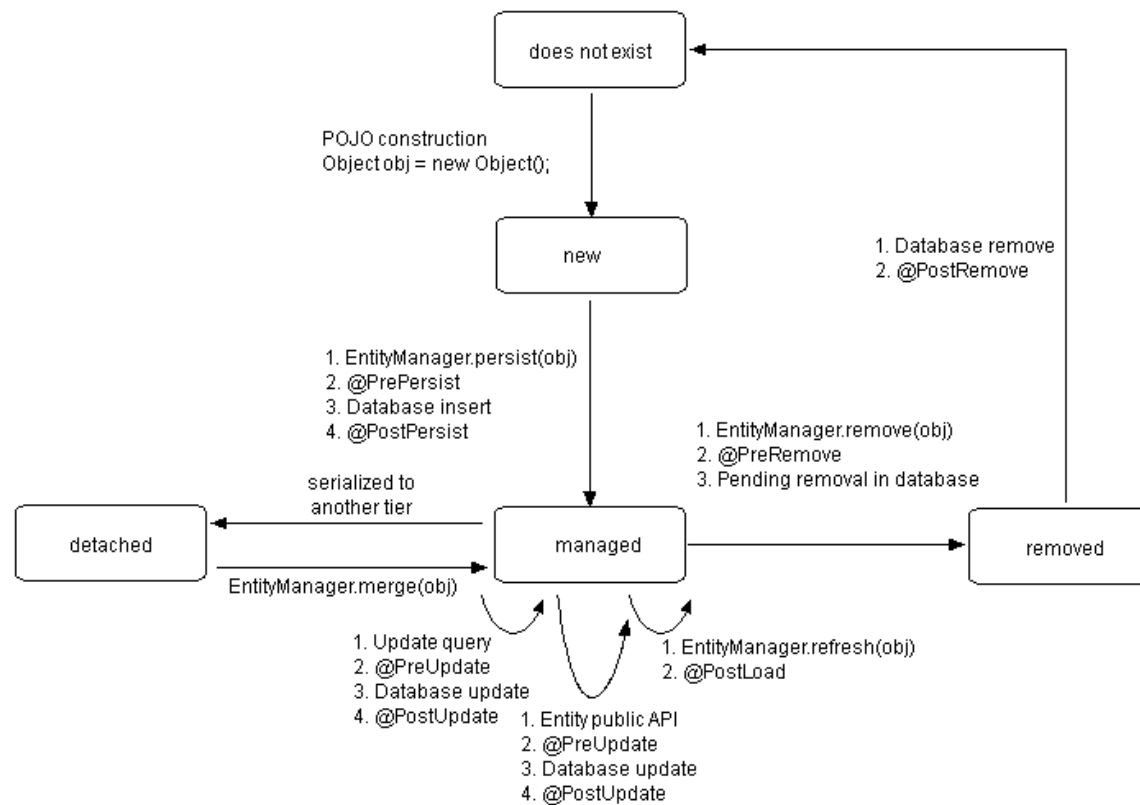
- createQuery() konvertuos JPQL užklausą į SQL užklausą ir ją įvykdys
- SQL užklauso rezultata konvertuos į objektus



ENTITY LIFECYCLE



ENTITY LIFECYCLE



ENTITY IDENTIFIKATORIAUS SUTEIKIMAS (GENERAVIMAS)

- Identifikatorius Entity objektams gali būti sugeneruojamas JPA realizacijos (persitence provider) automatiškai, pagal nurodytą tipą.
- Tam naudojama @GeneratedValue anotacija
- Programuotojas gali pasirinkti vieną iš kelių strategijų:
 - AUTO - provider chooses for us
 - TABLE - naudoja generator table
 - SEQUENCE - DB feature
 - IDENTITY - DB feature



ENTITY IDENTIFIKATORIAUS SUTEIKIMAS (GENERAVIMAS)

```
@Entity
public class User {
    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private Long id;

    @Column
    private String name;

    // geteriai ir seteriai
}
```

- Auto naudoti reikėtų tik kuriant, realiems produktams - nurodyti konkretų tipą



TRANSAKCIJOS (OPERACIJOS)



TRANSAKCIJOS (OPERACIJOS)

- Metodai, kurių veiksmas vykdomi transakcijose pažymimi `@Transactional`
- Transakcija yra būtina visiems metodams dirbantiems su duomenų baze
- `@Transactional` pradės transakciją tik jei komponentas sukurtas per Spring ir kviečiamas iš kitos klasės



TRANSAKCIJŲ VALDYMAS

- JTA ir Resource Local tipai:
 - JTA - transakcijos valdomos per JTA API. Transakcijas valdo išorinis komponentas. Palaikoma Java EE aplinkoje. Transakcijos gali būti pradėtos ir baigtos išoriniuose (mūsų programuojamiems) komponentams.
 - Resource Local – transakcijos valdomos EntityManager pagalba. Palaikoma tiek Java EE, tiek Java SE aplinkose.



TRANSAKCIJŲ VALDYMAS

```
@Transactional
public void updateGood(Integer goodId) {
    Good good = entityManager.find(Good.class, goodId);
    good.setName("Test");
}
```

- Pakeitimai bus išsaugoti automatiškai pasibaigus transakcijai - nereikės kviesti em.persist()



TRANSAKCIJŲ VALDYMAS

```
@Transactional
public void updateGood(Good good) {
    good.setName("Test");
}
```

- Pakeitimas automatiškai į duomenų bazę išsaugotas nebus, nes good greičiausiai buvo užkrautas kitoje transakcijoje ir todėl nepriklauso esamai sesijai
 - čia gautas Good yra detached



TRANSAKCIJŲ VALDYMAS

```
@Transactional
public void updateGood(Good good) {
    good = entityManager.merge(good);
    good.setName("Test");
}
```

- Pakeitimai bus išsaugoti, nes po merge(...) operacijos grąžintas good jau yra šios sesijos dalis



TRANSAKCIJŲ VALDYMAS

```
@Transactional
public void updateGood(Good good) {
    good.setName("Test");
    Good savedGood = goodRepository.save(good);
}
```

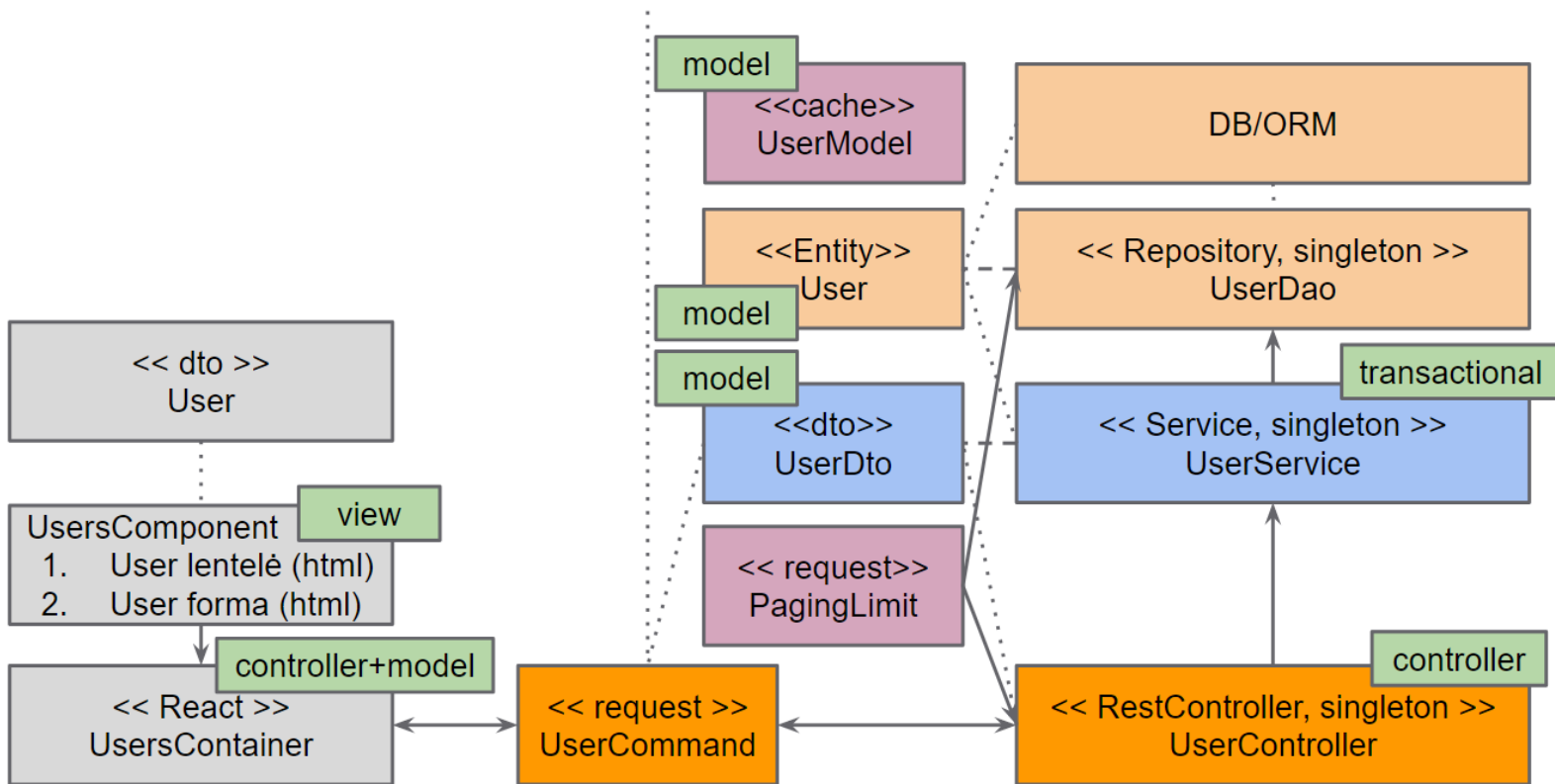
- naudojant Spring Data save() metodą, merge būtų iškvieistas automatiškai
- čia good yra detached, o savedGood - jau attached (valdomas)



SERVISŲ KŪRIMAS



MVC - TRANSACTIONAL



SERVISŲ SLUOKSNIS

- SoC - separation of concerns, dizaino principas
 - TCP turi 7 sluoksnius..
 - .. arba HTML/CSS/JS, etc.
- Pagal REST standartą entity objektai neturi būti perduoti į per Rest servisą atgal klientui
 - klientas dirba su Rest resursais. Resursas != Esysbė
 - Esysbės ID, vidiniai/transient parametrai ne resurso dalis. Pvz. jei turime 2 servusus ir jie dirba su skirtinga DB, kiekvienoje DB jų ID gali būti vienodas



SERVISŲ SLUOKSNIS

- serviso užduotis - atlikti biznio transakciją
 - pridėti User, priskirti user'ui rolę
- jei ji nepavyko, tai užduotis: atlikti rollback
- RestController gali kviesti keletą verslo transakcijų
- RestController užduotis - priimti užklausas ir pateikti atsakymus
- Repository/DAO užduotis - atlikti atomines operacijas su esybėmis



SERVISŲ SLUOKSNIS

- norint atlikti skirtingas validacijas
 - užklauso duomenims
 - servisui reikalingiems duomenims
 - DB reikalingiems duomenims
- pvz. gal mes priimam ir 254 simbolių ilgio vardą, bet servisas dirba su simbolių eilute, kurią turi užkoduoti, tai jam 200 gali būti maksimumas
 - o DB pvz. jei prijungiam H2 - leidžiam 254, o jei prijungiam seną Mainframe vietoj DB - tai tik 50



SERVISŲ SLUOKSNIS

- darbui su DB reikalinga transakcija, bet RestController atitikmuo nebūtinai kviečiamas iš Spring. Kitaip sukonfigūravus aplikaciją, tai gali būti tiesiog klasė, todėl joje transakcijos su @Transactional neveiks, nes jos veikia tik kai yra kviečiamos iš Spring
- beje, ne visada naudojama ir Jackson biblioteka JSON kodavimui ar atkodavimui
 - pvz JsonIgnore su kitomis bibliotekomis neveiks, tai kitur ID nebus galima tiesiog ignoruoti



UŽDUOTIS #1 - DAO SU EM

- pirmiausia reikia sukonfigūruoti Spring Boot aplikaciją
- konfigūracijos tikslas: turėti H2 duomenų bazę, veikiančią su Hibernate, kurį naudoja Spring Data, kad viskas veiktų Spring Boot aplikacijoje, o duomenų bazę mums sukurtų automatiškai mūsų nurodytoje vietoje ir išjungiant aplikaciją duomenys išliktų
- į `pom.xml` prisidėti priklausomybę

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



UŽDUOTIS #1 - DAO SU EM

- į `application.properties` failą įsidėti DB nustatymus:

```
###
#   Database Settings
###
spring.datasource.url=jdbc:h2:file://tmp/test2235.db;AUTO_SERVER=TRUE;DB_CLOSE_DELAY=-1;DB_CLOSE_ON_EXIT=FALSE
# ;AUTO_SERVER=TRUE;DB_CLOSE_DELAY=-1;DB_CLOSE_ON_EXIT=FALSE
spring.datasource.platform=h2
spring.datasource.username = sa
spring.datasource.password =
spring.datasource.driverClassName = org.h2.Driver
spring.jpa.database-platform=org.hibernate.dialect.H2Dialect
```

- DB bus sukurta `/tmp/test2235.db`, todėl atnaujinant DB struktūrą reiks ištrinti šiuos failus



UŽDUOTIS #1 - DAO SU EM

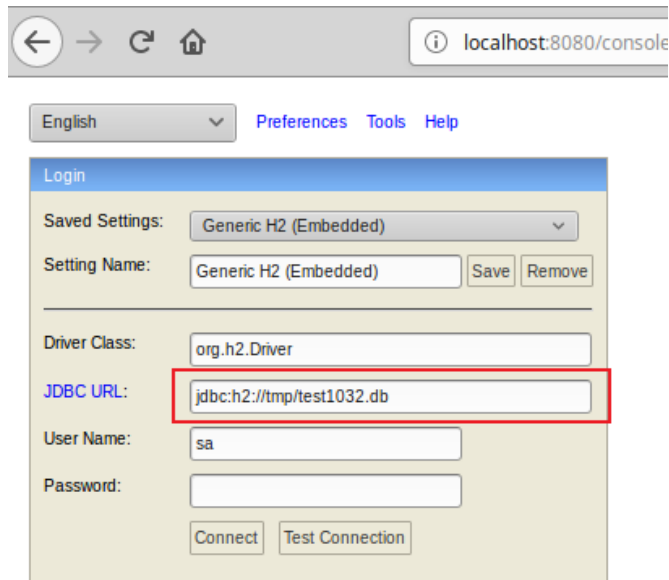
- į `application.properties` failą įsidėti H2 DB nustatymus:

```
###  
#   H2 Settings  
###  
spring.h2.console.enabled=true  
spring.h2.console.path=/console  
spring.h2.console.settings.trace=false  
spring.h2.console.settings.web-allow-others=false
```

- prie konsolės prieiti galima <http://localhost:8081/console>



UŽDUOTIS #1 - DAO SU EM



- naudodamiesi H2 konsole, nepamirškite įsivesti savo DB kelią, kurį nurodėte `application.properties` faile

```
spring.datasource.url=jdbc:h2:file://tmp/test2235.db;AUTO_<..>
```



UŽDUOTIS #1 - DAO SU EM

- į `application.properties` failą įsidėti Hibernate nustatymus:

```
###
#   Hibernate Settings
###
spring.jpa.hibernate.ddl-auto = update
spring.jpa.properties.hibernate.show_sql=true
spring.jpa.properties.hibernate.use_sql_comments=false
spring.jpa.properties.hibernate.format_sql=true
spring.jpa.properties.hibernate.hibernate.dialect=org.hibernate.dialect.H2Dialect
spring.jpa.properties.hibernate.hbm2ddl.auto=update
spring.jpa.properties.hibernate.temp.use_jdbc_metadata_defaults=true
spring.jpa.properties.javax.persistence.validation.mode=auto
```



UŽDUOTIS #1 - DAO SU EM

- User klasę pažymėti @Entity
 - atsargiai - mes naudojame paprastą JPA, todėl anotacijos taip pat iš `javax.persistence.*`, pvz.:

```
import javax.persistence.Entity;
```

- User klasės atributus pažymėti @Column
- taip pat sukurti ir auto generated id
- pastaba: klasė negali būti final



UŽDUOTIS #1 - DAO SU EM

- implementuoti DAO su EM - galime pasinaudoti prieš tai rodytu kodu:

```
@Repository
public class DBUserDAO implements UserDao {
    @PersistenceContext private EntityManager entityManager;
    public List<User> getUsers() {
        return entityManager.createQuery("SELECT u from User u", User.class).getResultList();
    }
    public void createUser(User user) { entityManager.persist(user); }
    public void deleteUser(String username) {
        User user = entityManager
            .createQuery("SELECT u from User u where username = :un", User.class)
            .setParameter("un", username).getSingleResult();
        if (entityManager.contains(user)) { entityManager.remove(user); }
        else { entityManager.remove(entityManager.merge(user)); }
    }
}
```



UŽDUOTIS #1 - DAO SU EM

- sukurti UserService
- iš UserController kviesti tik UserService metodus
 - userDao iš UserController pašalinti
- iš UserService kviesti UserDao

```
@Service
public class UserService {
    @Autowired @Qualifier("repoUserDao")
    private UserDao userDao;

    @Transactional(readOnly = true)
    public List<User> getUsers() {
        return userDao.getUsers();
    }

    @Transactional
    public void createUser(User user) {
        userDao.createUser(user);
    }
    ...
}
```



UŽDUOTIS #2 - REQUEST SCOPE

- susikurti request scope bean PagingData

```
@Component
@Scope(value = "request", proxyMode = ScopedProxyMode.TARGET_CLASS)
public class PagingData {
    private int limit;
    public PagingData() {
        this.limit = 5; // <numatytasis filtras>
    }
    // getters and setters
}
```

- UserController įsidėti kaip Autowired, o getUsers nustatyti pvz setLimit(10)
- UserDao įsidėti kaip Autowired ir pries getResultList pakviesti .setMaxResults() su getLimit()



SPRING DATA. DAO



SPRING DATA

- o kas jeigu nenorime rūpintis paprastomis užklausomis, juk jos visada vienodos
 - mes nenorime išradinėti dviračio
- Spring Data leidžia
 - atsisakyti paprastų užklausų
 - lengviau rašyti vidutinio sudėtingumo užklausas
 - visiškai nekurti repository klasės



DAO SU SPRING DATA

- CRUD įgyvendinantis DAO su Spring Data atrodo taip:

```
public interface UserRepository extends JpaRepository<User, Long> { }
```

- nereikia implementacijos!!
- jei kitoje klasėje pasiimsime su Autowired, galėsime kviesti įvairius metodus, pvz.:
 - `userRepository.save(user);` Create/Update
 - `userRepository.findAll();` Read
 - `userRepository.delete(User);` Delete
 - ir kt.
- interfeisas: `JpaRepository<Entity, ID>`



DAO SU SPRING DATA

- `JpaRepository` yra `CrudRepository` + `PagingAndSortingRepository`
 - `Paging/Sorting`: galima naudoti `Pageable` ir `Sort` parametrus
 - `Jpa`: persistence contexto detalesnis valdymas, įrašų įterpimas/trynimas/ieškojimas batch'u (po daug vienu metu)
 - pvz. susirasti įrašus pagal jų ID viena užklausa:

```
List<User> findAllById(Iterable<ID> ids)
```



DAO SU SPRING DATA

- Tampa ypatingai svarbus servisų sluoksnis
 - įsitikinkite, kad kviečiate Dao ar Repository tik iš servisų sluoksnio, ne iš rest controller'io
 - kitu atveju tiesiog galite nesuprasti, kodėl vienas ar kitas dalykas neveikia
 - serviso sluoksnio metodai turi būti anotuoti @Transactional
 - tuomet teisingai veikia DB transakcijų automatizavimas



DAO SU SPRING DATA

- papildome metodu, kuris ieško User pagal tikslų email:

```
public interface UserRepository extends JpaRepository<User, Long> {  
    User findByEmail(String email);  
}
```



DAO SU SPRING DATA

- papildome metodu, kuris ieško visų User, kurių email dalis yra partOfEmail:

```
public interface UserRepository extends JpaRepository<User, Long> {  
    List<User> findUsersByEmailContaining(String partOfEmail);  
}
```



DAO SU SPRING DATA

- papildome metodu, kuris trina visus User pagal username:

```
public interface UserRepository extends JpaRepository<User, Long> {  
    void deleteByUsername(String username);  
}
```



SPRING DATA TAISYKLĖS

- galima susikurti bet kokius metodus, sujungiant operacijos pavadinimą (pvz `delete`), tuomet tai, ką norime rasti, pvz `User`, žodį `by` bei reikalingus laukus
- galimos operacijos:
 - `find`, `read`, `query`, `count`, `get`, `delete`, `exists`
- neįrašius ieškomojo, bus pasirinkta iš interfeiso
- galima patikslinti, ko tiksliau ieškome:
 - `findFirst..`, `findTop..`, `findDistinct..`, `findUser..`, `findFirstUser..`, `findDistinctUser..`



SPRING DATA TAISYKLĖS

- laukus galima jungti su And ir Or, pvz.:
 - `findUsersByUsernameAndEmail`,
`findByUsernameOrEmail`
- po laukų galime nurodyti jų naudojimo taisykles, pvz. `Containing`, `IgnoreCase`
- taip pat galime nurodyti rūšiavimą `OrderByEmailAsc` ar `OrderByUsernameDesc`
- daugiau informacijos ir visi raktiniai žodžiai, kuriuos galime naudoti:
 - [Spring Data Query Creation](#)



SPRING DATA TAISYKLĖS

- keletas pavyzdžių, ką galime parašyti ir kaip komponuoti:

```
List<Person> findByEmailAddressAndLastname(EmailAddress emailAddress, String lastname);  
// Enables the distinct flag for the query  
List<Person> findDistinctPeopleByLastnameOrFirstname(String lastname, String firstname)  
List<Person> findPeopleDistinctByLastnameOrFirstname(String lastname, String firstname)  
// Enabling ignoring case for an individual property  
List<Person> findByLastnameIgnoreCase(String lastname);  
// Enabling ignoring case for all suitable properties  
List<Person> findByLastnameAndFirstnameAllIgnoreCase(String lastname, String firstname)  
// Enabling static ORDER BY for a query  
List<Person> findByLastnameOrderByFirstnameAsc(String lastname);  
List<Person> findByLastnameOrderByFirstnameDesc(String lastname);
```



SPRING DATA TAISYKLĖS

- norėdami naudoti savo SQL, naudojame `@Query`:

```
@Query("select u from User u where u.emailAddress = ?1")  
User findByEmailAddress(String emailAddress);
```

- norėdami modifikuoti duomenis, naudojame `@Modifying`:

```
@Modifying  
@Query("update User u set u.firstname = ?1 where u.lastname = ?2")  
int setFixedFirstnameFor(String firstname, String lastname);
```

- arba kuriame savo DAO ir jame naudojame EntityManager bei rašome savo Java kodą bei JPQL



PAVYZDYS EXISTSBY

- Spring Data `existsBy` galima apsirašyti taip:

```
boolean existsByNickname(String nickname);
```

- Bet galima panaudoti ir `Query` anotaciją:

```
@Query(value =  
    "SELECT CASE WHEN count(u)> 0 THEN true ELSE false END"  
    +" FROM User u WHERE nickname = ?1")  
boolean existsByNickname(String nickname);
```

- arba `findBy` metoda:

```
User findByNickName(String nickname);  
default boolean existsByNickname(String nickname) {  
    return findByNickName(nickname) != null;  
}
```



UŽDUOTIS #3 - SPRING DATA

- sukurkite Rest servisą ieškoti naudotojams pagal vardą ir pavardę
- paiešką suprogramuokite sukurdami Spring Data interfeiso metodą



JPQL UŽKLAUSŲ KALBA



JPQL

- The Java Persistence Query language (JPQL) tai nuo platformos nepriklausoma objektiškai orientuota užklausų kalba. Ji yra dalis JPA specifikacijos.
- Pagrindinis skirtumas nuo SQL yra:
 - JPQL operuoja esybėmis ir objektais (grafais)
 - SQL operuoja lentelėmis, stulpeliais ir įrašais
- JPQL palaiko trijų tipų sakinius: SELECT, UPDATE, DELETE



JPQL

- JPQL panašus į SQL, bet tai ne tas pats SQL:

```
SELECT c FROM Customer c
```

- FROM sąlyga JPQL kalboje nurodo entity, kuris bus naudojamas užklausoje.
- Pavyzdyje Customer yra entity klasė, c – objekto identifikatorius.
 - Identifikatorius gali būti naudojamas toliau sąlygoje nurodant sąryšius ar savybes (where sąlygoje)



JPQL

- Išraiškose galima naudoti taško sintaksę prieiti prie objekto laukų (path expressions):

```
SELECT c FROM Customer c where c.address IS NOT EMPTY
```

- Path expressions gali būti naudojama:
 - WHERE sąlygose
 - ORDER BY sąlygose
- JPQL galima naudoti JOIN'us, tačiau užtenka susieti su objektu, nereikia nurodyti pirminio/išorinio rakto.

```
SELECT r.loginName FROM Customer c JOIN c.credentials r  
WHERE c.address.street = 'Fast'  
SELECT r.loginName FROM Customer c, Credentials r  
WHERE c=r.customer AND c.address.street = 'Fast'
```



QUERY API

- JPA technologija leidžia TRIMIS skirtingais būdais pasiekti duomenis:
 - `EntityManager.find()`
 - JPQL užklausomis
 - SQL užklausomis
- JPQL ir SQL užklausos vykdomos JPA Query API priemonėmis
- Naudojant hibernate, JPQL išplečia HQL



QUERY API

- Query API palaiko dviejų rūšių užklausas. Abiem atvejais naudojamas Entity manager: Named Query ir Query
- Vardinės užklausos (Named queries) rašomos, kai užklausos kriterijai žinomi iš anksto
- Dinaminės užklausos konstruojamos ir vykdomos dinamiškai
- Užklausas, kurios naudojamos daugiau nei vienoje vietoje, geriau realizuoti kaip vardines.
- Jos pasižymi geresne greitimeika, nes yra suformuojamos vieną kartą ir vėliau pernaudojamos.
- Vardinės užklausos aprašomos entity klasėje naudojant anotaciją `@NameQuery` ir vėliau naudojamos siejant Spring Data metodus arba `em.createNamedQuery("uzklausosPavadinimas")`



QUERY API

- vardinė užklausa @Entity esybėje:

```
@Entity
@NamedQuery(name = "User.findByEmailAddress",
    query = "select u from User u where u.emailAddress = ?1")
public class User { }
```

- ir Spring Data repository ją panaudos aprašius taip:

```
public interface UserRepository extends JpaRepository<User, Long> {
    User findByEmailAddress(String emailAddress);
}
```



QUERY API

- Dinaminės užklausos kuriamos vykdymo metu ir yra naudojamos, kai pati užklausa priklauso nuo situacijos (pvz. kiek duomenų įvesta į paieškos laukelį)

```
Query query = em.createQuery("SELECT c FROM Customer c");
```

- Query klasės objektai atitinka vieną užklausą ir pateikia keletą metodų nurodyti užklausos parametrus
- Parametrus galima nurodyti dviem būdais:
 - Pagal parametro vardą
 - Pagal eilės numerį
- Spring Data šios užklausos rašomos naudojant @Query



QUERY API

- parametru pvz.

```
Query query=em.createQuery("SELECT i FROM Item i WHERE i.name = ?1");
query.setParameter(1, "Car");
List<Item> items = query.getResultList();

Query query = em.createQuery("SELECT i FROM Item i WHERE i.name = :itemName");
query.setParameter("itemName", "Car");
List<Item> items = query.getResultList();

@Query("select u from User u where u.firstname like %?1")
List<User> findByFirstnameEndsWith(String firstname);

@Query("select u from User u where u.firstname like %:name")
List<User> findByNameEndsWith(@Param("name") String name);
```



UŽDUOTIS #4 - JPQL

- User sukurkite age lauką
- sukurkite Rest servisą seniausiam naudotojui gauti
 - nepamirškite ir UserService
- parašykite dinaminę SQL užklausą, kuri išrenka seniausią naudotoją
- užklausą realizuokite repozitorijos interfeise pasinaudodami Spring Data @Query
- sukurkite default User findOldestUser() interfeiso metodą
 - implementacijoje tiesiog pakvieskite Spring Data metodą paieškai



UŽDUOTIS #5 - PARDUOTUVĖS DAO

- sukurkite DAO tvarkytis su produktais
- įsivaizduokime, jog norėsime ir krepšelį saugoti duomenų bazėje
 - sukurkite kitą DAO bei krepšelio tarpinį servisą, skirtą dirbti su krepšeliu
- panaudokite abu repository DAO Rest servisuose
 - kol kas krepšelyje saugomi produktai bus atskiri krepšelio produktai, turintys tikrųjų produktų ID
 - .. vėliau išmoksime susieti kelis @Entity tarpusavyje



KITOJE PASKAITOJE

JPA priklausomybės. Live coding. Užduotys

