Accesing Databases with JPA

Introduction to Service Design and Engineering 2013/2014.

Lab session #8

University of Trento

Cristhian Parra: Lab Lectures for IntroSDE-2013/2014
Some slides from LabSDE-2012/2013

Outline

- JPA Overview
- JPA Example
- CRUD Restful with JPA support

Pre-Requisites (1):

- Download and Install SQLite
- Download the SQLite JDBC driver
- Download and Install H2 (all platforms version, the jar inside bin is the jdbc driver)
- JAXB and Jersey libraries from previous lab sessions
- Download SQLite Studio (we will use it to create/explore an sqlite DB)

Pre-Requisites (2):

- JPA Support:
  - Eclipse: make you sure to have the **Dali Java Persistence Tools**
  - Without Eclipse: Download EclipseLink jars -> eclipselink.jar, javax.persistence_*.jar

JPA Overview (1)

- JPA stands for **Java Persistence API**
- JPA is a **Java specification** for **ORM** (another is JDO).
  - ORM stands for **Object-relational mapping:** the process of mapping objects to relational tables (and vice versa).
  - ORM allow developers to **work directly with objects** rather then with SQL statements.
  - The JPA implementation is typically called **persistence provider**
  - Some implementations are: Hibernate, EclipseLink and Apache OpenJPA

JPA Overview (2)

- Mapping between Java objects and database tables is defined via **persistence metadata**.
- JPA providers use persistence metadata to perform the correct database operations.
- Persistence metadata is tipically specified via **annotations in the Java class**.
  - Alternatively the metadata can be defined via XML files or a combination of both. A XML configuration overwrites the annotations (see example)

Tutorial JPA: Simple project (1)

- Create a new JPA project in Eclipse
  - If you do not use eclipse, all you need is EclipseLink libraries you downloaded before to be in the classpath of your project
  - In your source folder (e.g., src) create a folder named "**META-INF**"
  - Add a file named "**persistence.xml**" in this folder.
- Add the JDBC SQLite Driver to the classpath of the project
- Open the **persistence.xml**. You should see the following, with the persistence-unit name equal to the name of your project (if not, add it)

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence data-brackets-id='11' version="2.1" xmlns="http://xmlns.jcp.org/xml/ns/persistence"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/persistence
                        http://xmlns.jcp.org/xml/ns/persistence/persistence_2_1.xsd">
    <persistence-unit data-brackets-id='12' name="introsde-jpa">
    </persistence-unit>
</persistence>
```

Tutorial JPA: Simple project (2)

- "**persistence.xml**" defines the information needt for connecting to the database including:
  - *JDBC driver:* a sofware component enbaling Java applications to interact with specific database engines (e.g., mysql, oracle, H2, sqlite)
  - *Database url:* the url to the database (e.g., "*jdbc:mysql:localhost:3306/databasename*)
  - *User and password:* to connect to the database
- Add the following to your persistence.xml, as part of the persistence-unit and replace PATH_TO_THE_SQLITE_FILE for the path to the "*lifecoach.sqlite*" file you will find in this session resources folder

```
<properties data-brackets-id='13'>
    <property data-brackets-id='14' name="javax.persistence.jdbc.driver" value="org.sqlite.JDBC" />
    <property data-brackets-id='15' name="javax.persistence.jdbc.url"
        value="jdbc:sqlite:PATH_TO_THE_SQLITE_FILE" />
</properties>
```

Tutorial JPA: Adding Models (1)

- Open the **lifecoach.sqlite** in Sqlite Studio and explore the schema.
- After you know very well what tables are in the database and how they relate to each other, the next step is to create java classes that will represent this data model. **This is what we call the MODEL**
- Create a package named **model** and add the first model for the table **Person**

```
import java.io.Serializable;
import javax.persistence.*;
@Entity  // indicates that this class is an entity to persist in DB
@Table(name="Person") // to whate table must be persisted
public class Person implements Serializable {
    private static final long serialVersionUID = 1L;
    @Id // defines this attributed as the one that identifies the entity
    @GeneratedValue(strategy=GenerationType.AUTO)
    @Column(name="idPerson") // maps the following attribute to a column
    private Long idPerson;
    @Column(name="lastname")
    private String lastname;
    @Column(name="name")
    private String name;
    @Column(name="username")
    private String username;
    @Temporal(TemporalType.DATE) // defines the precision of the date attribute
    @Column(name="birthdate")
    private Date birthdate;
    @Column(name="email")
    private String email;
    // getters and setters of all the private attributes
```

Tutorial JPA: Adding Models (2)

- Classes that will be persisted in a database must be annotated with **@Entity** annotation.
- By default, the table name corresponds to the class name. You can change this with the addition to the annotation **@Table(name="NEWTABLENAME")**.
- An entity represents a *table* in the database. Instances of the class will represents its *rows*.
- Fields of the Entity will be saved in the database and JPA can use either instance variables (fields) or the corresponding getters and setters to access the fields. **It is not allowed to mix both methods**.
- By default each field is mapped to a column with the name of the field. You can change the default name via **@Column (name="newColumnName").**

Tutorial JPA: Adding Models (3)

- The following annotations can be used in fields:
  - **@Id:** Identifies the unique ID of the database entry
  - **@GeneratedValue:** Together with ID defines that this value is generated automatically.
  - **@Transient** Field will not be saved in database

Exercise 1: Creating JPA Models

- Create the model for the *LifeStatus* table in Lifecoach DB

Tutorial JPA: Accessing the Database (1)

- Create a package named **dao (data access objects)** and the following class to it.

```
package dao;
import java.util.List;
import javax.persistence.*;
import model.Person;
public enum LifeCoachDao {
  instance;
  private EntityManagerFactory emf;
  private LifeCoachDao() {
      if (emf!=null) {
          emf.close();
      }
      emf = Persistence.createEntityManagerFactory("introsde-jpa");
  }
  public EntityManager createEntityManager() {
      return emf.createEntityManager();
  }
  public void closeConnections(EntityManager em) {
      em.close();
  }
  public EntityTransaction getTransaction(EntityManager em) {
      return em.getTransaction();
  }
  public EntityManagerFactory getEntityManagerFactory() {
      return emf;
  }
}
```

Tutorial JPA: Operations in the Database (1)

- The **LifeCoachDao** is a singleton java instance that contains an **EntityManagerFactory**, which is configured by the persistence unit "**introsde-jpa**"
- This class will be used to create and **Entity Manager** whenever we need to execute an operation in the Database.
- The entity manager provides the operations from and to the database,
  - e.g. find objects, persists them, remove objects from the database, etc.
- In JavaEE applications, the entity manager is automatically inserted in the web application. Outside JavaEE you need to manage the entity manager yourself.

Tutorial JPA: Operations in the Database (2)

- To save objects in the database, the Entity Manager provides the method **save()**
- To synchronize objects again with the database a Entity Manager provides the **merge()** method.
- If the Entity Manager is closed (via **close()**) then the managed entities are in a detached state.

Tutorial JPA: Operations in the Database (3)

- Add the following methods to the Person model

```
public static Person getPersonById(Long personId) {
    EntityManager em = LifeCoachDao.instance.createEntityManager();
    Person p = em.find(Person.class, personId);
    LifeCoachDao.instance.closeConnections(em);
    return p;
}
public static List<Person data-brackets-id='16'> getAll() {
    EntityManager em = LifeCoachDao.instance.createEntityManager();
    List<Person data-brackets-id='17'> list = em.createNamedQuery("Person.findAll", Person.clas
    LifeCoachDao.instance.closeConnections(em);
    return list;
}
public static Person savePerson(Person p) {
    EntityManager em = LifeCoachDao.instance.createEntityManager();
    EntityTransaction tx = em.getTransaction();
    tx.begin();
    em.persist(p);
    tx.commit();
    LifeCoachDao.instance.closeConnections(em);
    return p;
}
...
```

Tutorial JPA: Operations in the Database (4)

```
...
public static Person updatePerson(Person p) {
    EntityManager em = LifeCoachDao.instance.createEntityManager();
    EntityTransaction tx = em.getTransaction();
    tx.begin();
    p=em.merge(p);
    tx.commit();
    LifeCoachDao.instance.closeConnections(em);
    return p;
}
public static void removePerson(Person p) {
    EntityManager em = LifeCoachDao.instance.createEntityManager();
    EntityTransaction tx = em.getTransaction();
    tx.begin();
    p=em.merge(p);
    em.remove(p);
    tx.commit();
    LifeCoachDao.instance.closeConnections(em);
}
```

Tutorial JPA: Testing the connection (1)

- Add JUnit support to your project
  - In Eclipse: *Project -> Build Path -> Configure Build Path -> Add Library -> JUnit*
  - Without IDE: download the jars from the JUnit Website
- Create a package named **test**

Tutorial JPA: Testing the connection (2)

- Create the following junit test class in the test package and run it as a JUnit class

```
package test;
import static org.junit.Assert.*;
import model.Person;
import org.junit.Test;
import dao.LifeCoachDao;
import java.util.List;
public class JPAStarterTest {
  @Test
  public void readPerson() {
      Person p = Person.getPersonById(new Long(1));
      assertEquals("Table has correct name", "Chuck", p.getName());
      List<Person data-brackets-id='18'> list = Person.getAll();
      assertEquals("Table has one entity", 1, list.size());
  }
  ...
```

Tutorial JPA: Testing the connection (3)

```
...
@Test
public void addPersonWithDao() {
    // Arrange
    Person p = new Person();
    p.setName("Pinco");
    p.setLastname("Pallino");
    // Act
    Person.savePerson(p);
    // Assert
    assertNotNull("Id should not be null", p.getIdPerson());
    List<Person data-brackets-id='19'> list = Person.getAll();
    assertEquals("Table has two entities", 2, list.size());
    assertEquals("Table has correct name", "Pinco", list.get(1).getName());
    Person created = list.get(1);
    Person.removePerson(created);
    list = Person.getAll();
    assertEquals("Table has two entities", 1, list.size());
    assertEquals("Table has correct name", "Chuck", list.get(0).getName());
}
```

Tutorial JPA: SQLite sequences sidenote

- SQLITE implements auto increment ids through named sequences that are stored in a special table named "sqlite_sequence"
- For this reason, you need to use the following @GeneratedValue annotation

```
@GeneratedValue(generator="sqlite_person")
@TableGenerator(name="sqlite_person", table="sqlite_sequence",
    pkColumnName="name", valueColumnName="seq",
    pkColumnValue="Person")
@Column(name="idPerson")
```

Exercise 2:

- Add CRUD operations to models using transactions and persist, merge and remove operations from the entity manager

Tutorial JPA: Generating Entities (1)

- This part of the tutorial is **for eclipse only**
  - If not using eclipse, jump to the part of mapping relationships with JPA
- Open data source view and add a new conection (Right Click on *Database connections -> New..*)
- **Add the correct SQLite Driver**
  - Click on the add button right of the "Drivers" select
  - Select "SQLite JDBC Driver"
  - If eclipse is unable to locat the jar of the driver, go to the tab *JAR List*, remove the current jar and add the SQLite JDBC driver you have downloaded
- **Configure the connection**
  - Database = lifecoach
  - Database Location = PATH_TO_YOUR_LOCAL_LIFECOACH_SQLITE_FILE
- Right click on your project and select **JPA tools --> Generate Entities from Tables**

Tutorial JPA: Generating Entities (2)

- Select the connection, get the list of tables and select the tables that you want to use to generate entities

Tutorial JPA: Generating Entities (3)

- The second step for generating entities is to specify **Table Associations** between entities to map relationship between tables. Add a relationship between *LifeStatus* and *MeasureDefinition*

Tutorial JPA: Generating Entities (4)

- Specify the columns that define the association

Tutorial JPA: Generating Entities (5)

- Specify the cardinality of the association

Tutorial JPA: Generating Entities (6)

- Finally, you can choose whether which entities in the association should have a property referencing the other entity (it is not required that both have a property referencing the other entity)

Tutorial JPA: Generating Entities (7)

- The next step to select some defaults properties for the mapping, like the default key generator. Since we are using SQLite and there is a special generator for this, let's choose "None"

Tutorial JPA: Generating Entities (8)

- The last step is to define how each property will be mapped (name of the attributes int the model classes, type of each attribute, etc.). Make sure all the primary keys have the mapping kind to "id".

Tutorial JPA: Relationships (1)

- You should have now all the entity classes generated in the model package.
  - If you didn't use the generation wizard, copy the model classes from the introsde-jp this lab code.
- Open the LifeStatus entity. You should have the following attribute:

```
@ManyToOne
@JoinColumn(name = "idMeasureDef", referencedColumnName = "idMeasureDef", insertable = true, up
private MeasureDefinition measureDefinition;
```

Tutorial JPA: Adding Relationships (2)

- JPA allows to define relationships between classes, e.g. it can be defined that a class is part of another class (containment).
- Classes can have one to one, one to many, many to one, and many to many relationships with other classes.
- A relationship can be bidirectional or unidirectional, e.g. in a bidirectional relationship both classes store a reference to each other while in an unidirectional case only one class has a reference to the other class.
- Within a bidirectional relationship you need to specify the owning side of this relationship in the other class with the attribute "mappedBy", e.g. **@ManyToMany(mappedBy="attributeOfTheOwningClass".**
- JPA Relationship annotations:

```
@OneToOne
@OneToMany
@ManyToOne
@ManyToMany
```

- asdf

``java // mappedBy must be the name of the attribute mapping this // relationship in the referenced entity @OneToMany(mappedBy="person",cascade=CascadeType.ALL,fetch=FetchType.EAGER) private List lifeStatus; ... ```

Tutorial JPA: Testing Relationships

Tutorial Jersey + JPA: Adding Resources (1)

- Add Jersey-Bundle libraries to the build path

Example 2

Tutorial JPA: Database from model (2)

- Open data source view in eclipse and add a new conection
-

- Select H2 (or Oracle if H2 is not available) name the connection lifecoach (H2) add driver

- Add H2 jar

- name = H2 Driver
- Database name = lifecoach
- Class driver = org.h2.Driver
- URL = jdbc:h2:~/lifecoach

convert to JPA project use created connection

or, without eclipse wizard add persistence xml to META-INF folder in src

Other Resources

- JUnit Tutorial
- JPA tutorial from where we took some of the explanations
- Checkout also mashape.com and signup with your GitHub account (we will try to use an API from there in the future sessions)