B.Sc. In Software Development. Year 4.
Semester I. Enterprise Development.
JPQL Overview

LIMERICK INSTITUTE OF TECHNOLOGY
SCHOOL OF SCIENCE, ENGINEERING & I.T.
Department of Information Technology

Retrieving Data

- Retrieving data using JPA is significantly easier that retrieving data with JDBC.
- Unlike JDBC, JPA automatically converts between objects and SQL.
- As a result you don’t have to write your own code to do this.
- JPA automatically performs any joins necessary to satisfy the relationships between entities.

How to get an Entity Manager Factory

- When you are using a full Java EE server like Glassfish or JBoss the server provides built-in entity managers for you.

- When you're not using these full servers, JPA provides a class named EntityManagerFactory that you can use to get entity managers.

- Entity manager factories are thread safe.
  - Entity managers are not – because of that, you need to request a new entity manager for each method where you need to access to one.
  - The easiest way to do this is to code a class like the one on the following slide.

How to get an Entity Manager Factory

```
12  public class DBUtil {
13
    private static final EntityManagerFactory emf =
15              Persistence.createEntityManagerFactory("books_PU");
16
17      public static EntityManagerFactory getEmf() {
18          return emf;
19      }
20  }//end DBUtil
```

- It makes it easier to get an entity manager factory whenever you need one.

Note

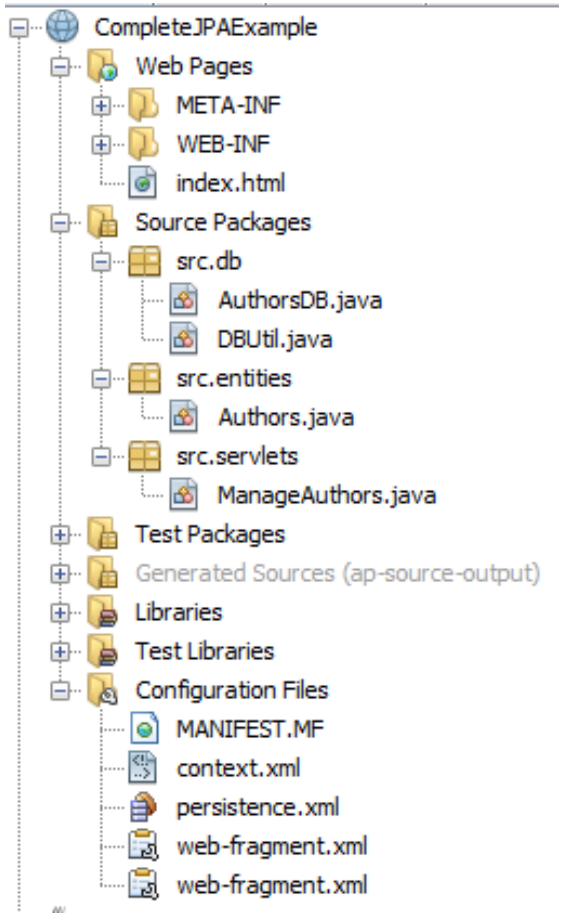
- These examples assume that a persistence unit and an entity class has also been generated.

- The state of the authors table in the books DB at the beginning of these examples is as follows:

AuthorID	FirstName	LastName	Image	YearBorn
1	Harvey	Deitel	1.jpg	1946
2	Paul	Deitel	2.jpg	1968
3	Tem	Nieto	3.jpg	1969

Note

- The anatomy of the Netbeans project that these examples are based on:



How to get an Entity by PK

- Code in the AuthorsDB class:

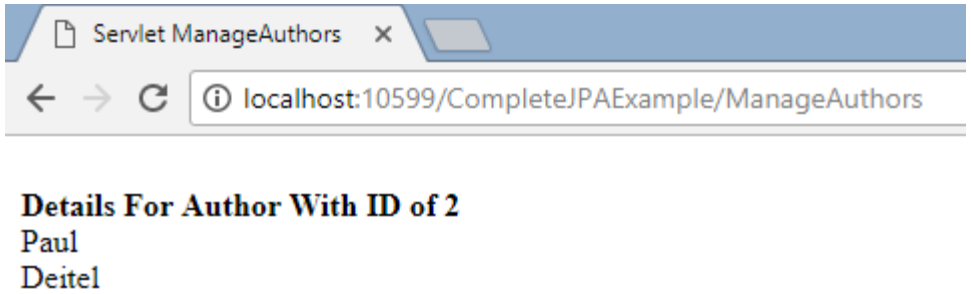
```
153     public static Authors getAuthorByID(Integer id) {
154
155         EntityManager em = DBUtil.getEmf().createEntityManager();
156
157         try {
158
159             Authors a = em.find(Authors.class, id);
160             return a;
161         }//end try
162         finally {
163             em.close();
164         }//end finally
165
166     }//end getAuthorByID()
```

How to get an Entity by PK

- Code in the Servlet:

```
out.println("<br><b>Details For Author With ID of 2</b><br>");
Authors a = AuthorsDB.getAuthorByID(new Integer(2));

out.println(a.getFirstName());
out.println("<br>");
out.println(a.getLastName());
```

- Output:



How to get all Entities

- Code in the AuthorsDB class:

```
128     public static List<Authors> getAllAuthors() {
129         EntityManager em = DBUtil.getEmf().createEntityManager();
130
131         String q = "SELECT a from Authors a";
132
133         TypedQuery<Authors> tq = em.createQuery(q, Authors.class);
134
135         List<Authors> list;
136
137         try {
138             list = tq.getResultList();
139
140             if (list == null || list.isEmpty())
141                 list = null;
142
143         }
144         finally {
145             em.close();
146         }
147
148
149         return list;
150
151     }//end getAllAuthors
```

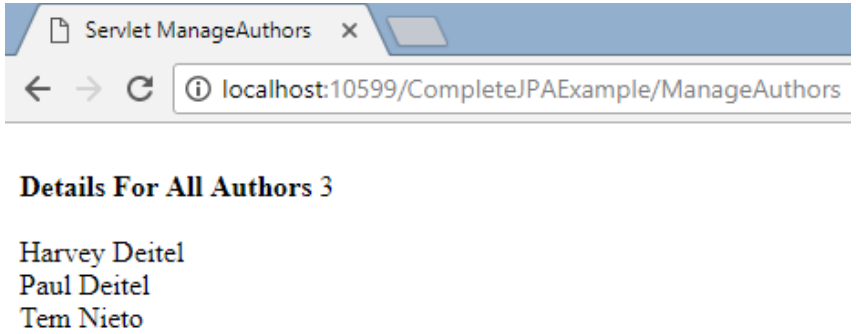
JPQL

How to get all Entities

- Code in the Servlet:

```
List<Authors> allAuthorsList = AuthorsDB.getAllAuthors();

out.println("<br><b>Details For All Authors</b> " + allAuthorsList.size() + "<br>");

for (Authors anAuthor: allAuthorsList) {

    out.println("<br> " + anAuthor.getFirstName() + " " + anAuthor.getLastName() );
}
```

- Output:



How to get all Entities based on a criteria

- Code in the AuthorsDB class:

```
171     public static List<Authors> getAuthorsBornBefore(Integer year) {
172
173         EntityManager em = DBUtil.getEmf().createEntityManager();
174
175          String q = "SELECT a from Authors a where a.yearBorn < " + year;
176
177          TypedQuery<Authors> tq = em.createQuery(q, Authors.class);
178
179          List<Authors> list;
180
181          try {
182              list = tq.getResultList();
183
184              if (list == null || list.isEmpty())
185                  list = null;
186
187          }
188          finally {
189              em.close();
190          }
191
192          return list;
193
194     }//end getAuthorsBornBefore()
```

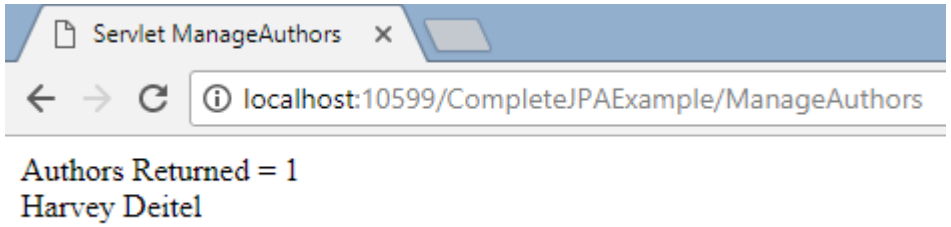
JPQL

How to get all Entities based on a criteria

- Code in the Servlet:

```
List<Authors> authorsList = AuthorsDB.getAuthorsBornBefore(1950);
out.println("Authors Returned =  " + authorsList.size() + "<br>");

for (Authors author : authorsList) {
    out.println(author.getFirstName() + " " + author.getLastName() + "<br>");
}
```

- Output:



How to get all Entities based on a criteria

- You often need to use JPQL to retrieve entities based on a column other than the primary key.
- JPQL looks like SQL, its an OO query language defined as part of the JPA specification.
- The method shown in the code in the previous example fetches all authors that were born prior to 1950.
- Here the variable named *a* refers to the *Authors* object, not the *Authors* table in the DB.
- Similarly, *a.yearBorn* refers to the *yearBorn* field of the *Authors* object.
  - This is known as a path expression.

How to get all Entities based on a criteria

- JPA doesn't specify whether a query should return a null value or an empty list if there are no results.
- Because of that, some implementations of JPA return null and others return an empty list.
- To ensure your code works consistently, you can use an *if* statement to check for both conditions.
  - When you do that you can avoid a NullPointerException by checking for the null value before you check for the empty list.
  - Then, if either condition is true you can return a null value.

Parameterised queries

```
public static List<Authors> getAuthorsBornBefore(Integer year) {

   EntityManager em = DBUtil.getEmf().createEntityManager();

    String q = "SELECT a FROM Authors a WHERE a.yearBorn < :year";


    TypedQuery<Authors> tq = em.createQuery(q, Authors.class);
    tq.setParameter("year", year);

    List<Authors> list;

    try {
        list = tq.getResultList();

        if (list == null || list.isEmpty())
            list = null;

    }
    finally {
        em.close();
    }

    return list;

}//end getAuthorsBornBefore()
```

Getting Single Results

- Code in the AuthorsDB class:

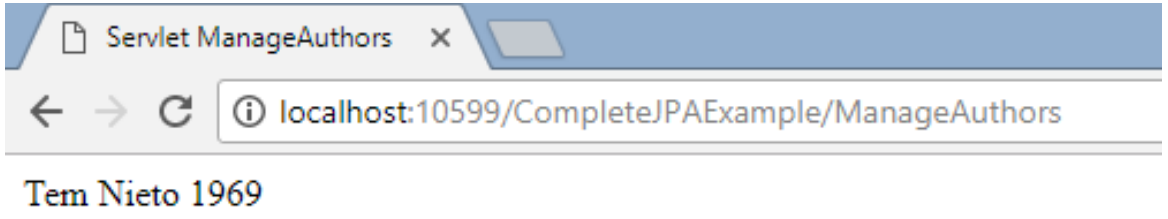
```
public static Authors getAuthorByLastName(String name) {
    EntityManager em = DBUtil.getEmf().createEntityManager();

    String q = "SELECT a FROM Authors a WHERE a.lastName = :name";

    TypedQuery<Authors> tq = em.createQuery(q, Authors.class);

    tq.setParameter("name", name);

    Authors a;
    try {
        a = tq.getSingleResult();

    } finally {
        em.close();
    }

    return a;
}//end getAuthorByLastName()
```

Getting Single Results

- Code in the Servlet:

```
Authors a = AuthorsDB.getAuthorByLastName("Nieto");
out.println(a.getFirstName() + " " + a.getLastName() + " " + a.getYearBorn() );
```

- Output:



Getting Single Results

- The code is similar to that for retrieving multiple entities, however to get a single entity you call the getSingleResult method from the query instead of getResultList.
- Unlike getResultList, the getSingleResult method can throw two exceptions.
- NoResultException (query returns no results) and NonUniqueResultException (the query returns more than one result).

Calling Named Queries

- Any entity class generated by Netbeans will contain a number of Names Queries.

```
@Entity
@Table(name = "authors")
@XmlRootElement
@NamedQueries({
    @NamedQuery(name = "Authors.findAll", query = "SELECT a FROM Authors a")
    , @NamedQuery(name = "Authors.findByLastName", query = "SELECT a FROM Authors a WHERE a.lastName = :name")
    , @NamedQuery(name = "Authors.deleteByName", query = "DELETE FROM Authors a WHERE a.lastName = :name")
    , @NamedQuery(name = "Authors.findByAuthorID", query = "SELECT a FROM Authors a WHERE a.authorID = :authorID")
    , @NamedQuery(name = "Authors.findByImage", query = "SELECT a FROM Authors a WHERE a.image = :image")
    , @NamedQuery(name = "Authors.findByYearBorn", query = "SELECT a FROM Authors a WHERE a.yearBorn = :yearBorn")})
public class Authors implements Serializable {
```

Getting Single Results

- Code in the AuthorsDB class:

```
public static Authors getAuthorByLastName(String name) {
    EntityManager em = DBUtil.getEmf().createEntityManager();

    TypedQuery<Authors> tq = em.createNamedQuery("Authors.findByLastName", Authors.class);

    tq.setParameter("name", name);

    Authors a;
    try {
        a = tq.getSingleResult();

    } finally {
        em.close();
    }

    return a;
}//end getAuthorByLastName()
```

Inserting a Record

- Modifying entities (updating, deleting or inserting) using JPA is far easier than JDBC because JPA understand the relationships between your entities.

- When doing this you will need to use a transaction to commit all operations to the database.
  - If any of the operations fail, you can use the transaction to roll back any changes.
  - This ensures data integrity.

Inserting a Record

- Code in the AuthorsDB class:

```
public static void insertAuthor(Authors a ) {

    EntityManager em = DBUtil.getEmf().createEntityManager();

    EntityTransaction trans = em.getTransaction();

    try {
        trans.begin();
        em.persist(a);
        trans.commit();
    }
    catch(Exception ex) {
        System.out.println(ex);
    }
    finally {
        em.close();
    }

}//end insertAuthor()
```

Inserting a Record

- Code in the Servlet:

```
Authors insertAuthor = new Authors();
insertAuthor.setFirstName("Alison");
insertAuthor.setLastName("Ryan");
insertAuthor.setYearBorn(2016);
insertAuthor.setImage("123.jpg");
AuthorsDB.insertAuthor(insertAuthor);
```

Updating a Record

- Code in the AuthorsDB class:

```
public static void updateAuthor(Authors a) {

    EntityManager em = DBUtil.getEmf().createEntityManager();

    EntityTransaction trans = em.getTransaction();

    try {
        trans.begin();
        em.merge(a);
        trans.commit();
    } catch (Exception ex) {
        System.out.println(ex);
    } finally {
        em.close();
    }

}//end updateAuthor
```

Updating a Record

- Code in the Servlet:

```
Authors updateAuthor = AuthorsDB.getAuthorByID(3);
updateAuthor.setFirstName("Francis");
updateAuthor.setLastName("Ryan");
updateAuthor.setYearBorn(2016);
updateAuthor.setImage("123.jpg");
AuthorsDB.updateAuthor(updateAuthor);
```

Updating a Record

- Code in the AuthorsDB class:

```
public static Authors getAuthorByID(Integer id) {

    EntityManager em = DBUtil.getEmf().createEntityManager();

    try {

        Authors a = em.find(Authors.class, id);
        return a;
    }//end try
    finally {
        em.close();
    }//end finally

}//end getAuthorByID()
```

Deleting a Record

- Code in the AuthorsDB class:

```
public static void deleteAuthor(Authors a) {

    EntityManager em = DBUtil.getEmf().createEntityManager();

    EntityTransaction trans = em.getTransaction();

    try {
        trans.begin();
        em.remove(em.merge(a));
        trans.commit();
    }
    catch(Exception ex) {
        System.out.println(ex);
    }
    finally {
        em.close();
    }

}//end deleteAuthor()
```

Deleting a Record

- Code in the Servlet:

```
Authors deleteAuthor = AuthorsDB.getAuthorByID(3);
AuthorsDB.deleteAuthor(deleteAuthor);
```

Deleting Multiple Records

- Named Query:

```
@Entity
@Table(name = "authors")
@XmlRootElement
@NamedQueries({
    @NamedQuery(name = "Authors.findAll", query = "SELECT a FROM Authors a")
    , @NamedQuery(name = "Authors.findByLastName", query = "SELECT a FROM Authors a WHERE a.lastName = :name")
    , @NamedQuery(name = "Authors.deleteByName", query = "DELETE FROM Authors a WHERE a.lastName = :name")
    , @NamedQuery(name = "Authors.findByAuthorID", query = "SELECT a FROM Authors a WHERE a.authorID = :authorID")
    , @NamedQuery(name = "Authors.findByImage", query = "SELECT a FROM Authors a WHERE a.image = :image")
    , @NamedQuery(name = "Authors.findByYearBorn", query = "SELECT a FROM Authors a WHERE a.yearBorn = :yearBorn")})
public class Authors implements Serializable {
```

Deleting Multiple Records

- Code in the AuthorsDB class:

```
public static int deleteAllAuthorByLastName(String name) {

    int count = 0;

    EntityManager em = DBUtil.getEmf().createEntityManager();
    EntityTransaction trans = em.getTransaction();

    TypedQuery<Authors> tq = em.createNamedQuery("Authors.deleteByName", Authors.class);

    tq.setParameter("name", name);

    try {
        trans.begin();
        count = tq.executeUpdate();
        trans.commit();
     } catch (Exception ex) {
        System.out.println(ex);
        trans.rollback();
    } finally {
        em.close();
    }

    return count;
}//end getAuthorByLastName()
```

Deleting Multiple Records

- Code in the Servlet

```
int recordsDeleted = AuthorsDB.deleteAllAuthorByLastName("Deitel");
out.println("Records Deleted " + recordsDeleted);
```

Using Aggregate Functions (max)

- Code in the AuthorsDB class

```
public static int getMax() {
    EntityManager em = DBUtil.getEmf().createEntityManager();

    //query SHOULD BE implemented as a NamedQuery in Entity class
    String q = "SELECT MAX(a.yearBorn) from Authors a";

    TypedQuery tq = em.createQuery(q, Integer.class);

    return (Integer) tq.getSingleResult();

}//end getMax()
```

- Code in the Servlet

```
out.println("Max value for yearBorn is " + AuthorsDB.getMax());
```

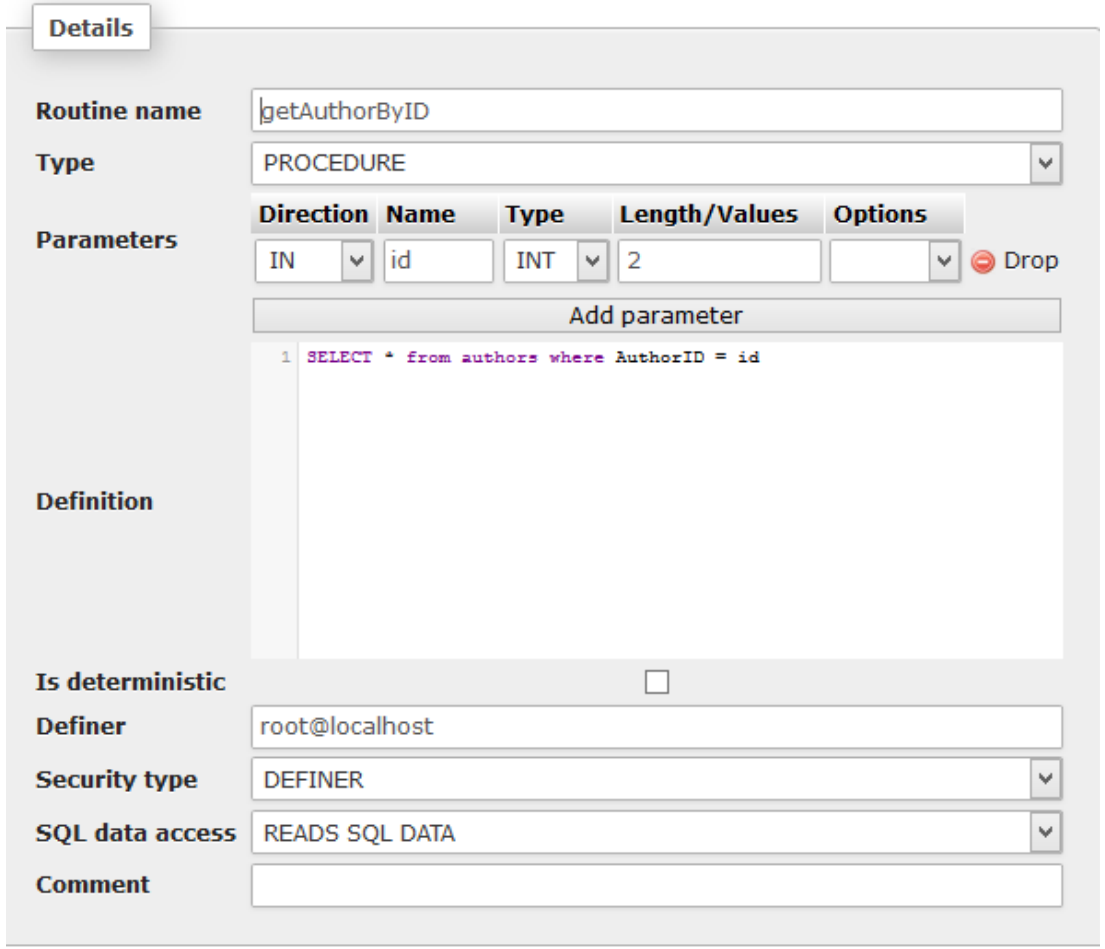
Other Aggregate Functions (avg, min, sum, count)

- Sample JPQL queries.

```
String avgQ   = "SELECT AVG(a.yearBorn) from Authors a"; //returns a Double
String minQ   = "SELECT MIN(a.yearBorn) from Authors a"; //returns an Integer
String sumQ   = "SELECT SUM(a.yearBorn) from Authors a";   //returns a Long
String countQ = "SELECT COUNT(a) from Authors a WHERE a.yearBorn < 2000"; //returns a Long
```

Using Stored Procedures

- Stored Procedure in the Books DB



Using Stored Procedures

- Two ways to call it:

```
public static Authors getAuthorByIDStoredProc(Integer id) {
    EntityManager em = DBUtil.getEmf().createEntityManager();

    StoredProcedureQuery query =
            em.createStoredProcedureQuery("getAuthorByID", Authors.class);

    query.registerStoredProcedureParameter(1, Integer.class, ParameterMode.IN);
    query.setParameter(1, id);

    Authors a;

    try {
        a = (Authors) query.getSingleResult();
    }
    finally {
        em.close();
    }
    return a;
}//end getAuthorByIDStoredProc
```

Using Stored Procedures

- Since JPA 2.1
- In the Authors (entity class)

```
@Entity
@Table(name = "authors")
@XmlRootElement
@NamedStoredProcedureQuery(
        name = "Authors.getAuthorByID",
        resultClasses = Authors.class,
        procedureName = "getAuthorByID",
        parameters = {
            @StoredProcedureParameter(mode=IN, name="id", type=Integer.class)
        }
)
@NamedQueries({
    @NamedQuery(name = "Authors.findAll", query = "SELECT a FROM Authors a")
    , @NamedQuery(name = "Authors.findByLastName", query = "SELECT a FROM Aut
    , @NamedQuery(name = "Authors.deleteByName", query = "DELETE FROM Authors
    , @NamedQuery(name = "Authors.findByAuthorID", query = "SELECT a FROM Aut
    , @NamedQuery(name = "Authors.findByImage", query = "SELECT a FROM Autho
    , @NamedQuery(name = "Authors.findByYearBorn", query = "SELECT a FROM Aut
public class Authors implements Serializable {

    private static final long serialVersionUID = 1L;
```

Using Stored Procedures

- In the AuthorsDB class

```
public static Authors getAuthorByIDStoredProc(Integer id) {

    EntityManager em = DBUtil.getEmf().createEntityManager();

    Authors a;

    try {
        StoredProcedureQuery query =
                em.createNamedStoredProcedureQuery("Authors.getAuthorByID");
        query.setParameter("id", id);
        a = (Authors) query.getSingleResult();
    } finally {
        em.close();
    }
    return a;
}//end getAuthorByIDStoredProc
```

Wrap Up

- We have only scratched the surface here of what can be achieved with JPA.
- The skills you will have gathered however, would be sufficient for most small to medium sized web applications.
- For larger sites you may want to take advantage of additional features offered by a full Java EE server such as automatic transaction commits and rollbacks and the automatic management of entity managers.
  - Many of these features are outlined here.
- JPA introduced another way to perform queries called the Criteria API.
- This API (not discussed here) is designed to make dynamic queries easier and safer.

References

Murach, J., (2014) *Murachs JavaServlets JSP*, 3rd edn. Mike Murach and Associates, Inc.

Keith, M., Schincariol, M. (2013) *Pro JPA 2: Second Edition (Expert's Voice in Java)*, 2nd edn. Apress.

http://www.objectdb.com/api/java/jpa

https://www.tutorialspoint.com/jpa/

https://docs.oracle.com/javaee/6/tutorial/doc/bnbpz.html