**JPA - JAVA PERSISTENCE API / Hibernate**

טכנולוגית JPA היא הדרך של java להתחבר ל data base בצורה נוחה.

JPA היא טכנולוגיה שעובדת תחת ה JAVA EE , כלומר ב APPLICATION SERVER אך ניתן להשתמש בה גם באפליקציה רגילה. Persistence – היכולת לשמור נתונים לאורך זמן.

אנו רגילים לצורה המקובלת והשימושית של שמירת נתונים בצורה טבלאית. השימוש הזה הוא לא לגמרי טבעי, כיון שאנו כמפתחות שפה בשיטת ה OOP רגילות לעבוד עם אובייקטים, קלאסים וכו' ואילו שפת הDB מדברת בצורה שונה: טבלאות, שורות ועמודות, מפתחות ראשיים ומשניים וכו'. - סט מושגים שונה לחלוטין בין שתי השפות. היה טבעי הרבה יותר שיהיה אפשר לשמור "אובייקט"

JPA נותן את האפשרות לעבוד עם DB בלי להתעסק עם ה DB עצמו. –אלא רק עם אובייקטים. כיצד זה נעשה ?

באופן כללי JPA ממפה את הקלאסים באופן שכל קלאס ממופה לטבלה, וכל שדה (או מאפיין) ממופה לעמודה. כמובן שהמיפוי לא תמיד פשוט כ"כ וזה נראה בהמשך.

הקלאסים נכתבים כרגיל, רק חובה להקפיד על 2 דברים:

* חייב להיות constructor ללא ארגומנטים.
* אין אפשרות שהקלאס או אחת המתודות יהיו final.

דוגמת קוד קצרה בשימוש בטכנולוגיה זו:

EntityManagerFactory emf= Persistence.createEntityManagerFactory("lessonJPA1PU");

EntityManager em=emf.createEntityManager();

Customer c1=new Customer("dina", 123);

Customer c2= new Customer("rina", 456);

em.getTransaction().begin();

em.persist(c1);

em.persist(c2);

em.getTransaction().commit();

Vector lst= (Vector) (em.createQuery("select c.name from Customer c where c.ID=123").getResultList());

System.out.print(lst);

em.close();

את המיפוי ניתן לעשות או ע"י קובץ של XML , או ע"י annotations. בד"כ הדרך היותר נוחה היא בדרך השניה – annotation שזוהי דרך חדשה ומקובלת כיום מאוד.

ה annotion החשוב והראשי שיש להשתמש הוא Entity@. אותו יש לרשום מעל לשם הקלאס. Annotion זה אומר שקלאס זה יעבור מיפוי לטבלה. שם הטבלה ברירת מחדל יהיה שם הקלאס. וכל מאפיין יתמפה לעמודה בטבלה, ושוב כברירת מחדל שם העמודה יהיה כשם השדה בקלאס. כדי לשנות שם לטבלה או לעמודה יש להשתמש בannotions Table@ / Column @

כל קלאס שהוא entity, חובה לציין בו גם מיהו השדה שיהווה מפתח ראשי. ע"י הannotion ID@

את המיפוי ניתן לעשות או על השדות שבקלאס, או על מאפיינים (מעל פונקציית ה get ).

שדה או מאפיין שלא רוצים שיתמפה לטבלה רושמים מעליו את הannotion Transient @

Annotions נוספים:

**Basic@** - מתאים לרשום מעל שדות שהם מטיפוס בסיסי (וגם מחרוזות). ניתן לקבוע בו 2 דברים: fetch: eager / lazy

Optional: true / false - האם השדה יכול להיות null.

**Temporal@** מתאים לשדות מטיפוס date או calendar (וגם long שיכול לשמש לתאריך) איזה סוג של תאריך ישמר ב DB

.**Enumerated**@ - האם טיפוסים שהם enum יישמרו כמחרוזת או כמספר.

.**GeneratedValue@** - קובע ש ID יווצר ע"י ה DB. ישנם מס' פרמטרים להגדיר כיצד ייווצר המפתח הראשי. מהיכן לקחת את הID . תלוי בסוג של הDB .

EMBEDED

אם יש בקלאס משתנה שהוא מטיפוס מחלקה. בטבלה רושמים הכל בטבלה אחת.

ולכן המיפוי לא תמיד כ"כ פשוט. כדי ש 2 העולמות יעבדו בצורה נכונה. פותרים את הבעיה ע"י embbed. ניתן לומר שברמת המיפוי לשטח את האובייקט. עי הannotion Embeddeble@ ו Embeded@ מסביר לJPA למפות את השדות שנמצאות באובייקט. אם י ש מס' שדות בעלות אותו טיפוס מחלקתי, יש לשלוט על שמות הטורים כי אין אפשרות כמובן ליצור אותו טור פעמיים באותו שם. וזה נעשה ע"י ה annotion AttributeOverrides@.

דוגמא:

מחלקת point:

@Embeddable

public class Point implements Serializable{

private int x;

private int y;

public Point() {

}

public Point(int x, int y) {

this.x = x;

this.y = y; }

public int getX() {

return x;}

public int getY() {

return y; }

public void setX(int x) {

this.x = x; }

public void setY(int y) {

this.y = y; }

@Override

public String toString() {

return "x: "+ x +"y: "+y; }

מחלקת Person:

public class Person implements Serializable {

@Id

@GeneratedValue(strategy = GenerationType.AUTO)

private Long id;

@Embedded

@AttributeOverrides(

{@AttributeOverride(name="x", column = @Column (name="x1") ),

@AttributeOverride(name="y", column = @Column (name="y1") ) })

private Point pnt1;

@Embedded

@AttributeOverrides(

{@AttributeOverride(name="x", column = @Column (name="x2") ),

@AttributeOverride(name="y", column = @Column (name="y2") )

} )

private Point pnt2;

public Person() {

}

**Composite keys**

עד עכשיו היה לנו מפתח שהוא טור אחד, אם יש מפתח ראשי שמורכב מכמה שדות ה JPA מחייב אותנו לעשות class שמכיל את כל השדות הנצרכות. קלאס זה חייב לענות על מספר דרישות. ישנם שני דרכים לעשות זאת:

**דרך א': @EmbeddedId**

דוגמא:

המחלקה שמשמשת עבור שדות הPK:

@Embeddable

public class MyPK1 implements Serializable{

private int id1;

private int id2;

public MyPK1() {

}

@Override

public boolean equals(Object obj) {

return super.equals(obj);

}

@Override

public int hashCode() {

return super.hashCode();

}

מחלקה שמשתמשת בו:

@Entity

public class UseCompositeEmbedded implements Serializable {

@EmbeddedId

private MyPK1 myPK;

**דרך ב': @IdClass**

דוגמא:

המחלקה שמשמשת עבור שדות הPK:

public class MyPK1 implements Serializable{

private int id1;

private int id2;

public MyPK1() {

}

@Override

public boolean equals(Object obj) {

return super.equals(obj);

}

@Override

public int hashCode() {

return super.hashCode();

}

המחלקה שמשתמשת בו:

הערה: חייבת להיות התאמה מלאה בין מחלקת הPK למחלקה שמשתמשת בו.

@Entity

@IdClass(MyPK1.class)

public class UseCompositeIdClass implements Serializable {

@Id private int id1;

@Id private int id2;

**Inheritance**

אחד השינויים המרכזיים בין עולם הoop לעולם ה db הוא היכולת לבצע ירושה.

JPA נותן את התמיכה המלאה, כיצד להתמודד עם ההבדל הזה ע"י צורות שונות של מיפויים.

יכול להיות מס' מצבים:

1. Entity שהוא אבסטרקטי – מתמפה לטבלה רגיל כמו כל entity.
2. אנטיטי יורש מקלאס אחר שהוא לא אנטיטי. אם ב"אבא" אין שדות שאמורות להשמר בטבלה, אין בעיה. אך אם מעונינים שהשדות ב"אבא" כן יישמרו יש להגדיר באבא (שהוא לא entity) את האנוטישטן MappedSuperClass @
3. קלאס שהוא לא entity שיורש מקלאס שהוא כן entity, רק השדות של הsuper class יתמפו לטבלה.
4. אנטיטי שיורש מאנטיטי- 2 ישויות שכל אחת ולידית בפני עצמה.

JPA נותן 3 אסטרטגיות שונות:

* **A table per hierarchy-** לקחת את כל הארירכיה לטבלה אחת. בטבלה זו יהיו כל השדות. בחלק מהרשומות חלק מהשדות יהיו רקים. בטבלה זו יווצר עוד שדה שנקרא type שכתוב בו לאיזה סוג שייכת הרשומה. היתרון של שיטה זו, שהכל מרוכז בטבלה אחת ולכן השליפה תהיה פשוטה ביותר. החיסרון הוא חוסר נרמול של הנתונים. (א"א להגדיר notNull על שדות מסוימים.
* **A table per class**- לכל קלאס יהיה טבלה משלו המכיל את כל השדות הקשורות לקלאס זה בלבד. החיסרון: כל שליפה מסובכת הרבה יותר, נוצר union כדי לחפש את המידע מכל הטבלאות. היתרון: הנרמול יהיה מקסימלי.
* **A table per concrete class**- נוצרות טבלאות מתאימות לכל קלאס, כל טבלה מכילה את כל הנתונים הנצרכים עבור הקלאס כולל השדות של ה superClass.

כדי לקבוע באיזו אסטרטגיה ייוצרו הטבלאות יש להשתמש באנוטיישן מתאים מעל שורת הגדרת הקלאס, לדוגמא:

@Inheritance(strategy= InheritanceType.TABLE\_PER\_CLASS)

**relationShips**

יחסים בין קלאסים שונים, מחלקה מסוימת מחזיקה reference למחלקה אחרת.

הקשר בין המחלקות נמדד ב2 פרמטרים:

1. כיוון הקשר :

* Unidirectional – רק צד אחד מכיר את הצד השני. (אנו נתמקד רק בכוון זה.)
* BIdirectional – 2 המחלקות מכירות אחת את השניה.

1. סוג הקשר:

ישנם 4 סוגים של יחסים:

* One to one
* Many to one
* One to many
* Many to many

פרוט:

* **One to one**

כל רשומה במחלקה מקושרת עם רשומה אחת ממחלקה אחרת:

דוגמא: לכל בנ"א יש כתובת אחת

@Entity

public class Person implements Serializable {

private static final long serialVersionUID = 1L;

@Id

@GeneratedValue(strategy = GenerationType.AUTO)

private Long id;

private String name;

@OneToOne

private Address addrs;

מחלקת Address כתובה כמחלקה רגילה ללא תוספות.

בDB ייוצר FK בטבלת person ע"מ לקשרו לaddress.

אם רוצים לבצע פעולות זהות על 2 אובייקטים שאחד מהם מכיל קשר לאובייקט השני, ניתן לעשות זאת ע"י המאפיין cascade באנוט' שמגדיר את סוג הקשר בין 2 האובייקטים, שמשמעותו לחלחל את הפעולה פנימה גם לאובייקט המקושר.

לדוגמא: במחלקת Person ניתן לרשום:

@OneToOne(cascade={CascadeType.PERSIST})

private Address addrs;

במחלקה בה יוצרים את האובייקטים ומכניסים אותם לdb ניתן לרשום:

Person p1=new Person("moshe", new Address(456));

….

em.persist(p1);

* **Many to one**

דוגמא: מורה ותלמידים. לכל מורה ישנם הרבה תלמידים. לכל תלמיד יש מורה אחד.

@Entity

public class Person implements Serializable {

@Id

@GeneratedValue(strategy = GenerationType.AUTO)

private Long id;

@ManyToOne

private Teacher teac;

בטבלה שתיווצר עבור אובייקט זה יתווסף שדה נוסף עבור FK לטבלת Teacher.

דוגמא להכנסת רשומות לטבלה:

Teacher t=new Teacher("baharir");

Person pup1=new Person( t, "pup1");

Person pup2=new Person(t, "pup2");

Person pup3=new Person( t, "pup3");

Person pup4=new Person( t, "pup4");

… … .

em.persist(t);

em.persist(pup1);

em.persist(pup2);

em.persist(pup3);

em.persist(pup4);

* **One to many**

דוגמא: מחלקת הזמנות שמכילה מערך של מוצרים. הזמנה אחת – הרבה מוצרים.

במצב זה קיימת בעיה היכן ליצור את הFK כיון שא"א ליצור אותו בטבלת הone , מצד שני ליצור אותו בצד הmany יגרום לכך שבעת שינוי במחלקת הone נצרך לבצע שינוי במחלקת הmany למרות שבמחלקה זו לא נעשה דבר. לכן החליטו לבצע את סוג הקשר ע"י טבלת join וכך לא נאלץ לשנות את טבלת הmany.

דוגמא:

במחלקת Order נגדיר מערך של הזמנות ונציין את סוג הקשר:

@OneToMany

private ArrayList<Product> myProds;

* **many to many**

דוגמא: תלמידים וקורסים. כל תלמיד יכול להופיע במס' קורסים. וכל קורס יכול להכיל הרבה תלמידים.

מסמנים את הקשר עם האנוט' : @ManyToMany

תווצר טבלת join ע"מ לקשר בין הרשומות.

**ENTITY MENEGER**

entityManager- הרכיב שמנהל את ה s entity . באמצעותו ניתן לעשות את ה CRUD OPERATIONS (create, read, update, delete)

בקובץ XML כתוב לאיזה DB להתחבר ואיך לנהל אותו.

כל אובייקט שהוא Entity יכול להיות מנוהל ע"י הem באחד מ 4 מצבים.

**1.New** – נוצר instance חדש אך לא חיברתי אותו עדיין לDB . כלומר, אין רשומה מקבילה לו בDB

כדי להכניסו ל DB יש להשתמש במתודה persist()(מקבילה לinsert). זה יגרום לפקודת sql לDB ואז הוא עובר למצב הבא:

**Managed .2**- הentity manager מכיר את ה entity ואז כל שינוי שיעשה על האוביקט יבוא לידי ביטוי בdb (כמו שימוש בפונקציה setName() ישפיע על הטבלה).

כדי לדעת האם הentity במצב nemage יש את המתודה contains()

כאשר הem נסגר האובייקט הופך להיות detached (ה em נסגר כאשר נסגרת הטרנזקציה)

**3. detached**- לאובייקט יש רשומה ב db אך אין קשר לdb – הוא לא מנוהל ע"י ה .em

אם רוצים להחזיר שוב את הentity להיות מנוהל ע"י הem כי נעשו שינויים ורוצים לעדכן את ה DB משתמשים עם המתודה merge()שמייצרת פקודת update.

**4. remove –** שמוש במתודה remove של ה Em , תגרום למחיקת הרשומה של האובייקט מהטבלה.

**QUERIES - - שאילתות**

ביצוע שאילתא פשוטה באמעות מפתח ראשי:

ישנה מתודה שנקראת find() שמבצעת select רק לפי מפתח ראשי מקבלת את הקלאס ואת המפתח הראשי ומחזירה את האובייקט במצב של managed.

ביצוע שאילתות מורכבות – שפת **JPQL**.

כאמור, JPA נולד למטרה בו נוכל לנהל נושאים הקשורים לDB דרך שפת java. לשם כך נוצרה שפה המקבילה לשפת sql שנקראת jpql , בשפה זו אנו מבצעים שאילתות מקבילות לשאילתות של שפת sql , אך במקום להשתמש בשם טבלה ושם טור בטבלה וכו' משתמשים במושגים של שפת תכנות: שם של קלאס, שם של שדה בקלאס וכו'.

לדוגמא:

* Select p from Person p
* Select d from Computer c INNER join c.disks d
* Select d from Computer c, IN(c.disks) d
* Select c from Computer c where c.type LIKE 'Pentium%'

דוגמאות נוספות:

The EntityManager.createQuery and EntityManager.createNamedQuery methods are

used to query the datastore by using Java Persistence query language queries.

The createQuery method is used to create **dynamic queries**, which are queries defined

directly within an application's business logic:

public List findWithName(String name) {

return em.createQuery(

"SELECT c FROM Customer c WHERE c.name LIKE :custName")

.setParameter("custName", name)

.setMaxResults(10)

.getResultList();

}

The createNamedQuery method is used to create **static queries**, or queries that are

defined in metadata by using the javax.persistence.NamedQuery annotation. The

name element of @NamedQuery specifies the name of the query that will be used with the

createNamedQuery method. The query element of @NamedQuery is the query:

@NamedQuery(

name="findAllCustomersWithName",

query="SELECT c FROM Customer c WHERE c.name LIKE :custName"

)

Here's an example of createNamedQuery, which uses the @NamedQuery:

...

customers = em.createNamedQuery("findAllCustomersWithName")

.setParameter("custName", "Smith")

.getResultList();

public List findWithName(String name) {

return em.createQuery(

"SELECT c FROM Customer c WHERE c.name LIKE :custName")

.setParameter("custName", name)

.getResultList();

}

public List findWithName(String name) {

return em.createQuery(

"SELECT c FROM Customer c WHERE c.name LIKE ?1")

.setParameter(1, name)

.getResultList();

}

קימיים סוגים רבים ומגוונים של שאילתות (יותר מאשר sql) רשימה מלאה ניתן למצוא בטוטוריאל של java ee (פרק של persistence **39.4 Example Queries**)