**Properties**

spring.jpa.hibernate.ddl-auto=

Aceasta proprietate permite lui Hibernate sa lucreze cu schema bazei de date

* **update** – compara schema bazei de date cu cea definita de noi si i da update la cea din baza de date, adica vede ce diferente sunt si o modifica, dar niciodata nu va sterge ceva din baza de date(coloana, tabel, constraint etc.), chiar daca e diferenta
* **create** – drop tabelele din baza de date si creaza altele pe baza la cele pe care noi le avem cu @Entity
* **validate** – verifica daca schemele coincid. Daca nu, arunca exceptie
* **create-drop** – face ceea ce face create si la final de program, sterge baza

**@Transient**

* Chiar daca nu punem anotatia @Column la un field, Hibernate oricum va crede ca acel field e o coloana din tabel. Daca nu vrem ca un field sa fie considerat ca colana, dar ca simplu field, folosim @Transient

**@Basic**

* Este exact ca @Column, dar permite sa decidem daca un field sa fie lazy sau eager initialized.

Basic(fetch=FetchType.LAZY)

Daca nu e specificat fetch, default e eager, si in asa caz nu e diferenta intre @Column si @Basic

**@Temporal**

* @Temporal ne permite sa lucram cu data, timp, adica cu DATE, TIME, TIMESTAMP din baza de date
* Lucrul cu datele si orele din Java in SQL e cam problematic, dar anotatia data ne ajuta

@Temporal(TemporalType.TIMESTAMP) – echivalent pentru TIMESTAMP din SQL

@Temporal(TemporalType.TIMES) – echivalent pentru TIME din SQL

@Temporal(TemporalType.DATE) – echivalent pentru DATE din SQL

* Aceasta anotatie e destinata mai mult fieldurilor ce sunt @Column

ManyToMany multiple fields

**Insertable and Updatable**

* La @Column sau @OneToOne etc. putem pune parametrii insertable=true/falsei si updatable=true/false
* **insertable=false** – inseamna ca Hibernate nu va include acest atribut cand entitatea va fi inserata in baza de date. Adica, pur si simplu nu va popula coloana si o va lasa cu null sau pe mana bazei de date, daca e id. In plus true e default
* **updatable=false** – Hibernate nu va da update la coloana data. Va da update la toate celelalte, dar asta va ramane asa cum a fost initial, indiferent ca noi o setam cu altceva prin setter.
* Sunt folosite mai mult pentru primaary key sau fk

**Caching**

* JpaRepository are 2 metode de save:

- save(Onject)

saveAndFlush(object)

Diferenta e ca cu save obiectul se va salva cand se va da commit la tranzactie, dar cu saveAndFlush, el se va salva deodata cand se executa metoda

* Acum, hibernate mereu pastreaza in cache rezultatele ce le ia din tabel, ca sa nu faca foarte multe interogari in baza de date. Totusi, daca de ex avem in BD 50 de clienti si hibernate i va stoca in cache, se poata intampla ca noi sa stergem din BD un client manual, accesand DBMS, insa Hibernate va pastra in cache inca 50 de clienti, desi noi deja am sters unul

**States oj Object**

* **Transient** – este statusul unui obiect creat ce nu este legat nici de cum cu baza de date. Hibernate nu are nimic cu cel. De ex, cand cream un obiectu cu new, si nu facem nimic cu el in Hibernate. El e doar in codul nostru si nu are nimic cu Hibernate sau baza de date



* **persistent** – odata ce un transient object e pus in baza de date, el devin persistent. In general, un obiect pus sau scos din BD, e persistent. Hibernate colecteaza date despre el si are grija ca ele sa ajunga in baza de date fara a mai folosi metode. Un persistent object este



* **Detached** – un persistent object este detached cand el nu mai este asociat cu nicio Session. Asta se poate intampla cand sesiunea e inchisa sau cand obiectul este detached explicit. Modificarile facute asupra unui obiect detached nu sunt puse in baza de date



Deoarece am folosit detach(), faptul ca noi am modificat Surname in “Montblanc” nu influenta nicidecum baza de date. Odata ce am folosit detach(), Hibernate nu mai urmareste modificarile facute asupra la obiectul scos din baza de date. El nu atinge obiectul din baza de date, dar pur si simplu putem modificat cum vrem obiectul student, si nimic din modificarile facute nu se vor duce in baza de date

* **Removed** – un persistent object devine removed cand folosim metoda remove(). El nu mai este asociat bazei de date.

**Merge vs Persist**

* persist() salveaza un obiect in baza de date, unul nou, dar merge() da update la el daca obiectul exista in baza de date, sau daca nu exista, il adauga.



Am putea si nu folosi merge(), asa cum oricum obiectul va fi modificat in baza de date.

* merge() mai este folosit si pentru a intoarce un obiect din starea detached in persistent. Daca am dat detach la un obiect ce a fost scos din baza de date, si vrem ca el sa fie inapoi controlat de Hibernate, vom folosi merge(), se va crea un nou obiect ce va fi in persistent stage, si va fi pus in baza de date ca obiect nou sau va da update daca deja exista.
* persist() nu returneaza nimic, deci el chiar iar obiectul nostru si il pune in persistent state, dar merge(), chiar daca salveaza un obiect inextistent, el oricum va crea o copie persistent la obiectul nostru.

**De ce merge() returneaza o copie**

* merge() returneaza un nou obiect.
* Daca noi avem un obiect care este detached sau transient, apelarea metodei merge nu va face ca acest obiect propriu zis sa fie in persistent stage. merge va crea un nou obiect ce va fi deja persistent, si il va returna, dar primul obiect folosit tot e detached sau transient

**merge() vs update()**

* Si merge si update sun folosite pentru a readuce un obiect din detached sau transient state in persistent
* update() nu poate da update la un obiect daca un alt obiect identic deja exista in persistent state. De ex:

Student student1 = session.getById(10,Student.class);

Student student2 = new Student(10,”A”,”B”);

session.update(student2)

Vom primi exceptie, deoarece deja exista student1 care pastreaza acest tuple din baza de date, si nu poate face ca si student2 sa fie in persistent, si deci nu poate da update la tuple in BD. **Hibernate merge pe principiul ca fiecare tuple din baza de date trebuie sa aiba un singut object in persistent state!**

* update() doar da update la un obiect ce trebuie neaparat sa existe in baza de date. Daca asa obiect nu exista, avem eroare. El foloseste UPDATE tabel SET ....
* merge() oricum creaza o copie la obiect, ca sa fie in persistent state,si chiar daca deja avem un obiect ce corespunde acelui tuple, nu e nicio problema.

**Person** person = **new** **Person**(); person.setName("John"); session.save(person); session.evict(person); person.setName("Mary"); **Person** mergedPerson = (Person) session.merge(person);

person si mergedPerson sunt 2 obiecte diferite!

Teacher teacher1 = new Teacher("Dumitru","Valuta",10);  
Teacher teacher2 = session.merge(teacher1);  
System.*out*.println(teacher1==teacher2);

false

* merge() da update la un obiect existent deja, si save la unul neexisent, update() doar modifica un obiect ce la sigur exista
* merge() creaza un nou obiect peristent si il returneaza, update nu, caci update deodata il ia pe cel oferit si il face persistent
* merge accepta si obiecte ce nu sunt in persistence state. El le adauga in baza de date,daca nu exista, sau le da update daca exista. Update tot accepta obiecte ce sunt in transient state, si le accepta pe cele in detached si le face persistent inapoi
* Teacher teacher1 = new Teacher("Dumitru","Valuta",10);  
  Teacher teacher2 = session.merge(teacher1);

Teacher teacher = new Teacher("Mititiuc","Eduard",10);  
teacher.setId(21);  
  
session.update(teacher);

Teacher teacher = new Teacher("Mititiuc","Eduard",10);  
teacher.setId(21);  
session.update(teacher);  
session.evict(teacher);  
teacher.setFirstName("Azaza");  
  
session.update(teacher);

* Cand vine vorba de a actualiza datele, cand facem merge(), EntityManager va verifica daca id oferit de noi exista in baza de date. Evident, daca id e 0 sau NULL, e clar ca va salva direct itemul in BD. Daca nu e 0 sau NULL, va verifica intai daca asa entitate exista in baza de date. Daca da, va face update, daca nu, va da save.

@GetMapping("/")  
@Transactional  
public Teacher teacher(){  
 Teacher teacher = new Teacher();  
 teacher.setId(135);  
 teacher.setFirstName("Alex");  
 teacher.setLastName("Gon");  
 teacher.setExpirience(5);  
  
 return entityManager.merge(teacher);  
}

1, Intai entityManager a verificat daca exista un Teacher cu id 135

2. Daca da, da update, daca nu, da save

La fel e si cu update()

**Save vs persist**

* Cea mai mare diferenta e ca save() functioneaza si in afara unei tranzactii definite de noi!!Ea o creaza pe a ei. Asta e rau si de aceea persist e preferata, caci persist merge doar in interiorul unei tranzactii definite de noi:

Student student = new Student("Mititiuc","Eduard");  
session.save(student);

Fara a deschide o tranzactie, tot va merge, caci ea automat va crea una.

* persist() returneaza void, si lucreaza direct pe obiectul trimis si va modifica anume id in el. La fel face si save, doar ca el returneaza id la obiectul salvat, desi tot il va salva si in el
* In cazul la persist(), daca dam persist() la un obiect ce deja e in persistent state, nu se va intampla nimic.

Teacher teacher = new Teacher("Dumitru","Valuta",25);  
session.persist(teacher);  
session.persist(teacher);

In baza de date se va salva un singur obiect, nu 2 caci acesta deja este in persistent state si e in cache

* Cu persist(), daca avem un obiect in detached state, si incercam sa il salvam, primi eroare. Daca e in trasient, putem liber sa il salvam, asa si trebuie. In cache, obiectele detached sunt pastrate, si Hibernate va sti ca un obiect a fost detached:

Teacher teacher = new Teacher("Dumitru","Valuta",25);  
session.persist(teacher);  
  
session.detach(teacher);  
  
session.persist(teacher);



* Daca dam save() de 2 ori la acelasi obiect, nu se va intampla nimic, se va salva oricum doar un obiect, caci acesta deja este in persistent state si e in cache

Teacher teacher = new Teacher("Dumitru","Valuta",25);  
  
session.save(teacher);  
session.save(teacher);

* evict() = detach()

Daca dam save la un object apoi detach si iar save, nu va aparea exceptie, ci se vor salva 2 obiecte cu id diferit in BD:

[Hibernate: save,persist, update, merge | Baeldung](https://www.baeldung.com/hibernate-save-persist-update-merge-saveorupdate)

**More Method of Session**

* refresh() - repopuleaza o entitate cu cele mai recente date dintr-o baza de date
* lock(obj,type) - tranzactia curenta da lock la obiect ca nicio alta tranzactie sa nu il poata modifica cat ea o face.

type:

LockModeType.PESSIMISTIC\_WRITE - tranzactia curenta poate citi,modifica,sterge datele, dar alta nu va putea pana cea curenta nu elibereaza lockul

LockModeType.PESSIMISTIC\_READ - permite datelor doar sa fie citite, nu si modificate in vreun fel

* load() - creaza un proxy object. Metoda va prelua obiectul din baza de date in mod lazy, adica va accesa propriu zis datele lui BD doar cand obiectul va fi folosit undeva
* byId(Clasa.class).load(id) - returneaza un obiect din BD, dar modificarile lui nu vor mai fi puse in BD. Daca modificam ceva in el, aceste modificari nu sunt puse in BD.
* flush() - forteaza ca obiectul sa fie salvat in baza de date fara a mai ajunge la commit()

**Caching**

* First Cache e mereu activat in mode default in Hibernate



Aici propriu zis s-a executat un singur query de select, nu 2! Hibernate mereu pastreaza in cache entitatile extrase, si modificarile asupra lor deci tot se salveaza in cache.

**MapStruct**

* Transferul de date dintre entitati si DTO poate fi adesea anevoios, caci necesita mult cod
* MapStruct ne ajuta sa evitam scrierea la mult cod manual
* Dependenta:

<dependency>

<groupId>org.mapstruct</groupId>

<artifactId>mapstruct</artifactId>

<version>1.4.2.Final</version>

</dependency>

<build>

<plugins>

<!-- ... -->

<plugin>

<groupId>org.apache.maven.plugins</groupId>

<artifactId>maven-compiler-plugin</artifactId>

<version>3.5.1</version>

<configuration>

<source>11</source> <!-- depending on your project. In this example, Java 11 is used -->

<target>11</target> <!-- depending on your project. In this example, Java 11 is used -->

<annotationProcessorPaths>

<path>

<groupId>org.mapstruct</groupId>

<artifactId>mapstruct-processor</artifactId>

<version>1.3.1.Final</version>

</path>

</annotationProcessorPaths>

</configuration>

</plugin>

</plugins>

</build>

**Caching**

* **First Level**

**-** este activat in mod default

- nu poate fi dezactivat

- este asociat cu Session object. Un Session memoreaza in cache toate query si obiectele returnate, si daca de ex facem un query intr-o Session, se va efectua un query in baza de date, dar query si object returnat vor fi cached si daca mai face inca odata acelasi query in aceeasi Session, nu se va mai face un query in baza de date, ci primim acelasi object deja cached

- Odata ce Session se termina, se sterge si cache asociat ei

- evict() sau detahc() fac si ca obiectul sa fie scos din cache

- clear() - face sa se curete tot cache la session

- fiecare Session are query al ei.

**Exemplu**:

@GetMapping  
@Transactional  
public void test() throws InterruptedException {  
 System.*out*.println(entityManager.find(Student.class,1));  
 Thread.*sleep*(10000);  
 System.*out*.println(entityManager.find(Student.class,1));  
}

Daca in aceste 10 secunde cat threadul e activ modificam valoarea in tabel la user cu id1, peste 10 secunde oricum vom primi obiectul vechi din cache

* **Second level cache**

**-** este disabled by default

- este un cache nu doar pentru un Session, ci pentru toate Sessions

-