**Hibernate**

Hibernate – ը ինքնուրույն գրադարան է և կախված չէ Spring Framework – ից:

Կառուցվածքը կարելի է բաժանել հետևյալ մակարդակների՝

* Java app
* Hibernate framework layer (core objects)
* Backhand layer – այն api – ն որը աշխատում է hibernate – ի տակից (JDBC, JNDI, JTA)
* Database layer

Քանի որ java կլասսները և աղյուսակները DB – ում իրար շատ նման են ստեղծվել է ORM (Object-relational Mapping) գաղափարը: Hibernate – ը իրենից ներկայացնում է ORM – ի ռեալիզացիա որը իր վրա է վերցնում աղյուսակների ավտոմատ convert – ը java օբյեկտների և հակառակը, բայց Hibernate – ը ընդամենը կարգավորում է իսկ տակից միևնույն է աշխատում է JDBC API – ն:

Hibernate օգտագործելիս կարիք չկա գրելու sql կոդ: Hibernate- ը ունի գործողութոյւններին համանուն մեթոդներ օր՝ save որոնք տակից կատարում են sql query – ի գործողությունները:

Աղյուսակների միջև կապերը java կլասսի մեջ կունենան հետևյալ տեսքը՝ Parent\_Table – ի տողը նկարագրող կլասսը իր մեջ կպահի child – երի տողերի լիստ, իսկ child – ի տողը իր մեջ parent – ի տողի տիպի օբյեկտ: Այս դեպքում նշված դաշտերը get անելիս Hibernate – ը տակից ավտոմատ կաշխատեցնի join query – ն և կվերադարձնի իրար հետ կապված տվյալները:

Hibernate – ի առացքային օբյեկտներն են՝

SessionFactory, Session, Transaction, ConnectionProvider, TransactionFactory

Hibernat – ում օգտագործվող օբեյկտներից են՝

Session session = sessionFactory.getCurrentSession();

Օբյեկտը ունի save, update, get և այլն մեթոդներ որոնց միջոցով ապահովում է անհրաժեշ աշխատանքային ինտերֆեյսը app – ի և DB – ի միջև: Հանդիսանում է JDBC – ի wrapper և իր մեջ պահում է նաև քեշ: Ունի նաև մեթոդներ տրանզակցիաների համար:

Մինչև այդ մեթոդների կանչը պետք է session օբյեկտի վրա կանչել beginTransaction() – ը, իսկ ավարտելուց հետո՝ getTransaction().commit(); Տրանզակցիաները ապահովում են միասնական աշխատանքը: Transaction օբյեկտը ունի համապատասխան մեթոդներ տրանզակցիաների կառավարման համար:

Commit և rollback – ը աշխատում են նույն սկզբունքով, ինչ որ JDBC - ում

Տրանզակցիաները կարող են նաև արգելափակել DB – ի հասանելիությունը, օր՝ քանի դեռ insert է արվում select մի արա, կամ քանի դեռ update է արվում ուրիշ update թույլ մի տուր և այլն:

Արգելափակման պրոցեսը ունի տարբեր մակարդակներ (Transaction Isolation Level) և շատ նման է thread – ների դեպքում synchronize – ի ֆունկցիոնալությանը:

Hibernate – ով աշխատելու համար պետք է hibernate core գրադարանը:

Spring – ից անկախ աշխատելու դեպքում կարիք կա տալ Hibernate – ի config – ներ resources – ում hibernate.properties – ի մեջ: Ֆայլի մեջ նշվում է`

hibernate.driver.class=com.microsoft.sqlserver.jdbc.SQLServerDriver  
hibernate.connection.url=jdbc:sqlserver://localhost:1433;database=TestHibernate  
hibernate.connection.username=  
hibernate.connection.password=  
  
hibernate.dialect=org.hibernate.dialect.SQLServerDialect  
hibernate.show\_sql=true  
hibernate.current\_session\_context\_class=thread

Դրանից հետո պետք է ստեղծել աղյուսակին համապատասխան java կլասս և դրա վրա դնել **@Entity** անոտացիան, որը ցույց է տալիս որ կլասսը կապված է DB – ի հետ: Կարելի է ավելացնել նաև @Table(name=tablename), բայց սրա կարիքը չկա եթե կլասի և աղյուսակի անունները համընկնում են:

Աղյուսակները նկարագրող Entity կլասսներին համապատասխան Hibernate – ը կարող է ավտոմատ ստեղծել աղյուսակներ (դրա համար պետք է միացված լինի hbm2ddl.auto config - ը) որոնց միջև կարելի է սահմանել հիերարխիա որը կարող է լինել հետևյալ տեսակների՝ (կոնֆիգուրացիան անոտացիաներից բացի կարող է լինել նաև xml – ի միջոցով)

* Table Per Hierarchy (TPH) – այս տեսակի իմաստը նրանում է որ բոլոր աղյուսակների համար ընդհանուր դաշտերը դուրս են բերվում մեկ parent կլասս և մնացած entity կլասսները ժառանգվում ես դրանից: Բոլոր subclass – ները պահվում եմ նույն աղյուսակումորն ունի հատուկ սյուն որում նշվում է subclass – ի տիպը: Այս սւպերկլասսի համար կիրառվում են @Entity, @Inheritance(strategy = InheritanceType.SINGLE\_TABLE), @DiscriminatorColumn(name = "animal\_type") – տիպերը նշելու համար սյուն, իսկ subclass – ի վրա նշվում է նաև @DiscriminatorValue(value = "cat") տիպի անունը աղյուսակի մեջ (այս անոտացիան կարող էլինել նաև սուպերկլասսի վրա):
* Table Per Concrete (TPC) – այս տարբերակի դեպքում ստեղծվում են առանձին աղյուսակներ միայն subclass – ների համար որոնք պարունակում են և ժառանգված և ունիկալ դաշտերը: superclass – ի վրա նշվում է միայն @Inheritance(strategy = InheritanceType. TABLE\_PER\_CLASS)
* Table Per Subclass (TPS) – այս դեպքում superclass – ի աղյուսակում կպահվեն միայն ընդհանուր դաշտերը իսկ subclass – ների համար ստեղծված կլասսներում կպահվեն բոլոր դաշտերը ներառյալ ընդհանուրները:@Inheritance(strategy=InheritanceType.TABLE\_PER\_CLASS)

@Entity – ով կլասսները պարտադիր պետք է ունենան դատարկ կոնստրուկտոր և id դաշտ որը պետք է նշվ @Id անոտացիայով:

Դաշտերը պետք է անոտավորվեն @Column(name = columnName)

Hibernate – ի Configuration տիպի օբյեկտում նշվում է model class – ը, որով Hibernate – ը հասկանում է որ DB- ում կա Person աղյուսակ որի հետ պետք է աշխատել: Այս ընթացքում ավտոմատ միանում է նաև hibernate.properties - ը

Configuration configuration = new Configuration().addAnnotatedClass(Person.class);

Configuration – ից կարելի է ստանալ SessionFactory

SessionFactory sessionFactory = configuration.buildSessionFactory();

Session session = sessionFactory.getCurrentSession();

session.get(Class clazz, int id);

session.save(Object o);

Որպեսզի կլասսի մեջ id – ի արժեքը ստացվի DB – ի գեներացրաց արժեքին համապատասխան պետք է @GeneratedValue(strategy = GenerationType.IDENTITY) անոտացիան, սրանով Hibernate – ին ասվում է որ պետք չէ ձեռք տալ այս դաշտին դրա արժեքը գեներացվում է DB – ում: save – ից անմիջապես հետո կարելի է ստանալ դրա id քանի որ Hibernate – ը ավտոմատ այն կավելացնի օբյեկտի մեջ save – ի հետո:

GenerationType.SEQUENCE -կարող է օգտագործվել երբ DB – ն չունի id գեներացնելու ֆունկցիա և id – ն պետք է գեներացվի java – ում:

Update – ի համար պետք է get – ով ստանալ տողը java – ում դրա օբյեկտի վրա կատարել անհրաժեշտ փոփոխությունները set մեթոդների միջոցով և վերջ տրանզակցիայի commit – ից հետո փոփոխությունները կանցնեն DB:

Delete – ի համար նորից get – ի միջոցով ստանալ օբյեկտը և փոխանցել session.remove(Object o) մեթոդին:

**HQL**

Hibernate – ում query գրելու համար օգտագործվում է HQL սինտաքսը:

Query գրելու կարիք կարող է առաջանալ սպեցիֆիկ գործողություններ կատարելու համար, այն դեպքորում երբ անհրաժեշտ գործողության համարժեք մեթոդը Hibernate – ը չունի, օր՝ ստանալ տվյալները որևէ պայմանով (age > 20), կամ ջնջել տողերը որոնցում անունը սկսվում է որևէ տառով, update անել բոլորին և այլն:

SQL – աշխատում է աղյուսակների հետ DB – ում, իսկ HQL ոչինչ չգիտի աղյուսակների մասին և աշխատում է java entity – ների հետ: Մնացած հարցերում SQL – ը և HQL – ը շատ նման են: Օր՝

SQL – SELECT \* FROM Person WHERE name = ‘Test’

HQL - FROM Person WHERE name = ‘Test’

HQL – ը ստեղծվել է որպեսզի աղյուսակի մակարդակ իջնելու կարիք չլինի և աշխատենք java entity – ների հետ:

HQL- ը տակից կոնվերտացվում է SQL – ի

Hibernate- ում կարելի է HQL – ի փոխարեն օգտագործել նաև սովորական SQL

Գրելու համար` session.createQuery(String query).getResultList()

Update – ի համար ՝ ` session.createQuery(String updateQuery).executeQuery()

Օր՝

currentSession.createQuery("UPDATE Person SET name = 'Test' WHERE age > 30").executeUpdate();

List<Person> people = currentSession.createQuery("FROM Person WHERE name LIKE 'J%'").getResultList();

currentSession.createQuery("DELETE FROM Person WHERE name LIKE 'J%'").executeUpdate();

createQuery() մեթոդը վերադարձնում է Query տիպի օբյեկտ որը ունի մեթոդներ՝

**public List list()** վերադարձնում է query – ի արդյունքը List – ով

**public Query setFirstResult(int rowno)** նշվում է այն տողը որից հետո պետք է ստանալ արդյունքները

**public Query setMaxResult(int rowno)** նշում է քանակ թե քանի տող պետք է ստանալ աղյուսակից

**public Query setParameter(int position, Object value)** query – ի պարամետրերը set անելու համար

**public Query setParameter(String name, Object value)** query – ի պարամետրերը set անելու համար

այսինքն Query վերադարձնող մեթոդներըի միջոցով կարելի է config անել query – ն և դրա վրա կանչել list() մեթոդը վերջնական արդյունքը ստանալու համար:

**Hibernate entity – ն ունի կյանքի հետևյալ ցիկլերը՝**

* **Transient –** երբ ստեղծվում է entity – ի օբյեկտը և դրա համար դեռևս չի կանչվել save մեթոդը այն գտնվում է այս state – ում, այսինքն այն դեռևս կապված չէ Hibernate – ի հետ և դրա կողմից track չի լինում, չի գտնվում է այսպես կոչված Persistance context – ում: Այս օբյեկտի վրա setter – ների կանչը ոչ մի մերպ դեռևս չի ազդում DB – ի վրա: Այսինքն transient օբյեկտը կարելի է ընկալել որպես սովորական java օբյեկտ:
* **Persistent (Managed) –** երբentity օբյեկտի համար կանչվում է save մեթոդը և այն տարնզակցիայի commit() մեթոդով Hibernate – ի միջոցով save է լինում DB - ում օբյեկտը անցնում է այս state – ին: Օբյեկտը այս վիճակին է անցնում նաև երբ այն get մեթոդով ստանում ենք DB – ից: Օբյեկտը track է լինում Hibernate – ի կողմից և գտնվում է persistence context – ում: Այս օբյեկտների վրա կանչված setter մեթոդները գեներացնում են query և ազդում են DB- ի վրա: persistence context – ում օբյեկտը գտնվում է այնքան ժամանակ քանի դեռ աշխատում է Hibernate session – ն:
* **Detached** – բնութագրվում է նրանով որ օբյեկտը դուրս է գալիս persistence contex – ից և նորից դառնում է սովորական java օբյեկտ: Նման է Transient state – ին: Այս state – ին անցնելու համար` session.detach(entityObject): ԵՎ հակառակը կարելի է օբյեկտը վերադարձնել persistence context session.merge(entityObject) մեթոդով:
* **Removed –** այսstate – ին օբյեկտը անցնում է երբ այն commit() – ից հետո ջնջվում է սեսսիայից՝ session.remove(entityObject)

**Cascading – ը Hibernate – ում**

Կարող ենք կարգավորել այնպես որ օր՝ parent օբյեկտ ստեղծելուց և այն նոր child – օբյեկտի կոնստրուկտորի մեջ նշելոց հետո բավարար լինի save անել միայն parent – ը իսկ դրա հետ կապված child – ը արդեն save լինի cascading – ի միջոցով այլ ոչ թե առանձի դրա համար save մեթոդով: Կասկադավորումը սովորաբար կատարվում է parent – ում: @OneToMany անոտացիան ունի cascade պարրամետր որտեղ նշվում են կասկադավորման ենթակա գործողությունները:

Օր՝ CascadeType.Persist – նշանակում է save – ի ժամանակ

@OneToMany(mappedBy = ‘owner’, cascade = CascadeType.Persist)

Տարբերությունը այն է որ cascade ատրիբուտը օգտագործելիս session.save(Object savingObject) – ի փոխարեն պետք է լինի session.persist(Object savingObject): Մեթոդները իրար շատ նման են, բայց ունեն հետևյալ տարբերությունները՝

|  |  |
| --- | --- |
| svae | persist |
| Պատկանում է միայն Hibernate գադարանին և չկա այլ JPA գրադարաններում | Ստանում ենք JPA գրադարանից |
| Վերադարձնում է save եղած entity – ի primary key - ը | Void է և ոչինչ չի վերադարձնում |
| Քանի որ վերադարձնում է արժեք ապա Երաշխավորված է որ primary key – ը արդեն նշանակված կլինի միանգամից մեթոդի կանչից հետո | Նման բան չի երաշխավորում, այսինքն արժեքը ինչ որ պահի կնշանակվի բայց ոչ անմիջապես կանչից հետո |

save – ը օգտագործելու դեպքում cascade ատրիբուտի փոխարեն պետք է ավելացվի @Cascade(CascadeType.Save\_Update), արգումենտը ընդունում է vararg և մնացած տիպերը կարելի է նշել ստորակետով: Այս դեպքում persist() – չի կասկադավորվի իսկ save – ը կկասկադավորվի:

CascadeType.REFRESH – ը օգտագործվում է քեշը թարմացնելու համար

**Join - ը Hibernate – ում:**

Join – երի համար կան հատուկ անոտացիաներ որոնք դրվում են դաշտերի վրա՝

**One To Many**

@OneToMany(mappedBy = '') – իրենից ներկայացնում է child – երի List parent աղյուսակում, mappedBy – ում նշվում է childTable – ում parentTable – ի օբյեկտի անունը:

@ManyToOne, @JoinColumn(name = '', referencedColumnName = '') – իրենից parent տիպի օբյեկտ child աղյուսակում, name – ում նշվում է childTable – ում foreign key – ի սյան անունը, referencedColumnName – ում նշվում է parentTable – ում id – ի սյան անունը: @JoinColumn – ն ով կլասսը կոչվում է նաև owning side այսինքն այն կողմը որ իր մեջ ունի foreign key:

Նշված անոտացիաներից հետո եթե կանչենք համապատասխան դաշտերի getter – ները hibernate – ը տակի կկատարի անհրաժեշտ join – ը և կվերադարձնի List: Նշված տարբերակը աշխատում է միայն տրանզակցիայի շրջանակում:

Hibernate – ը ունի քեշավորման մեխանիզմ և get – ից հետո հաջորդ get – ի ժամանակ արժեքը վերադրձվում է քեշից: Այսինքն օրինակ childTable – որևէ նոր օբյեկտ ավելացնելուց հետո պետք է դրա foreign key արժեքով տողը get անել parent աղյուսակից և դրա child – երի list – ի մեջ ավելացնել նոր ստեղծված օբյեկտը որը ավելացվել է childTable – ում:

Քեշի թարմացման համար կարելի է օգտագործել session.refresh() մեթոդը:

Child – ից parent – ի հետ կապված տողը կամ բոլոր տողերը ջնջելու համար բավական է ստանալ parent – ի մեջ սահմանված child - երի List – ը և դրանում ջնջել:

Hibernate – ում DB query – ները գեներացվում են միայն entity – ի getter մեթոդները օգտագործելիս, մնացած դեպքերում օր՝ childList.clear() query չի գեներանում:

**One To One**

Այս դեպքում օգտագործվում է @OneToOne անոտացիան և parent և child կլասսներում, իսկ own side կլասսում (այն կլասսում որի աղյուսակը պարունակում է foreign key) field – ի վրա նաև @JoinColumn – անոտացիան: ոչ own side կլասսում անոտացիան @OneToOne(mappedBy = ''): Այս դեպքում 2 աղյուսակներն էլ հավասարազոր են և OneToMany -> ManyToOne հաելային անոտացիաների կարիքը չկա:

@JoinColumn(name = '', referencedColmnName = '') – name – ը դա child – ում foreign key – ի սյունն է իսկ referencedColumnname – ը parent – ում id – ի սյունը:

Այս դեպքում կարող է parent – ի օբյեկտի id – ն միաժամանակ լինել նաև child – ի id, այսինքն java կլասսում @Id դաշտի տակ կունենանք parent տիպի ոբյեկտ: Նման դեպքերում երբ կլասսում @Id – ի տակ ոչ թե թվային արժեք է այլ օբյեկտ պետք է կլասսը իմպլեմենտացնի Serializable ինտերֆեյսը: Այս տարբերակի թերությունը այն է որ child – ի id – ն եթե թիվ չէ ապա այն հնարավոր չէ ստանալ Hibernate – ի միջոցով, այդ պատժառով ցանկալի է որ այնուամենայնիվ child – ում id – ն և foreign key – ը լինեն իրարից տարբեր, իսկ foreign key – ի վրա DB - ում դրված լինի unique սահմանափակումը:

**Many To Many**

Այս կապի համար օգտագործվում են @ManyToMany և @JoinTable անոտացիաները: Այս դեպքում 2 աղյուսակներն էլ նկարագրող կլասսները ունեն միմյանց List – երի դաշերը որոնք նշվում են @ManyToMany անոտացիայով, իսկ @JoinTable – ը դրանցից մեկի վրա (կարևոր չէ թե որ մեկի) նկարագրում է աղյուսակները միավորող join աղյուսակը: many to many կապի կոնտեքստում չկան parent և child աղյուսակներ, չկա owning side քանի որ աղյուսակները իրենց մեջ չեն պարունակում foreign key:

@JoinTable(name = ‘Actor Movie’,

joinColumns = @JoinColumn(name = 'actor\_id'),

inverseJoinColumns = @JoinColumn(name = movie\_id'))

name – join աղյուսակի անունը

joinColumns –նշվում է այն կլասսի id սյան անունը որի մեջ որ օգտագործվել է անոտացիան

inverseJoinColumns – նշվում է մյուս աղյուսակի id սյունի անունը

**Transaction management**

Տրանզակցիան կապված է սեսսիայի հետ և ստեղծվում է Transaction sesion.beginTransaction() մեթոդով: Transaction ինտերֆեյսը ունի հետևյալ մեթոդները՝

void begin() – տրանզակցիա բացելու համար

void commit() – ավարտում է տրանզակցիան

void rollback() – տրանզակցիան հետ է տալիս

boolean isAlive() – տրանզակցիան ակտիվ է թե ոչ

boolean wasCommited() – ստուգում է հաջող ավարտը

boolean wasRolledBack() – ստուգում է հաջող հետ տալը

**Caching**

Քեշավորումը բարձրացնում է application – ում ռեսուրսների օպտիմալ օգտագործումը: Կա քեշավորման 2 մակարդակ 1-ին և 2-րդ:

Առաջին մակարդակի քեշը պարունակվում է session օբյեկտում, այն միացած է default: Այս մակարդակի քեշը հասանելի չէ ամբողջ application – ի համար: Application – ը կարող է օգտագործել բազմաթիվ session օբյեկտներ:

Երկրորդ մակարդակի քեշը պարունակվում է SessionFactory օբյեկտում: Այս քեշում պահվող data – ն հասանելի է ամբողջ application – ի համար, բայց այն պետք է միացնել:

**Lazy loading vs Eager Loading**

Ենթադրենք կա OneToMany կապ և parent – ը իր մեջ պարունակում է մեծ քանակությամբ child: Քանի որ ամեն անգամ parent օբյեկտը ստանալիս միշտ չէ որ պետք են նաև դրա հետ կապված օբյեկտները ապա ռեսուրսների օպտիմալ ծախսման նպատակով ստեղծվել է lazy loading գաղափարը:

Eadge loading – ի ժամանակ կապված օբյեկտները load են լինում անմիջապես parent – ը get անելիս` @OneTyMany(mappedBy = '', fetch = FetchType.Eager), այս դեպքում քանի որ get – ի ժամանակ join – ի միջոցով արդեն load են եղել բոլոր կապված օբյեկտները դրանք getter – ի միջոցով հասանելի կլինեն նույնիսկ տրանզակցիայի փակվելուց հետո այսինքն երբ օբյեկտը կանցնի detouch state - ի

Lazy loading – ի ժամանակ կապված օբյեկտները load են լինում միայն parent – օբյեկտի վրա getter մեթոդը կանչելուց հետո`

@OneTyMany(mappedBy = '', fetch = FetchType.Lazy), այս դեպքում տրանզակցիայի փակվելուց հետո getter – ի միջոցով այլևս չենք կարողանա ստանալ կապված օբյեկտները, քանի որ ի տարբերություն eager – ի getter – ը գեներացնում է առանձին query:

Default աշխատում են հետևյալ type – երը՝

@OneToMany - lazy

@ManyToOne - eager

@ManyToMany - lazy՝ 2 կողմից

@OneToOne - eager՝ 2 կողմից

Այսինքն fetch type -երը default կարգավորված են այնպես որ աշխատանքը լինի արդյունավետ:

Կա նաև հետևյալ նրբությունը, եթե getter մեթոդի մեջ արժեքը վերադարձնելուց բացի ուրիշ բան տեղի չի ունենում և getter - ի կանչը որևէ փոփոխականի չի վերագրվում կամ օրինալ printLn չի արվում ապա կոմպիլյատորը ուղակի անտեսում է getter մեթոդի կանչը: Այս դեպքում եթե parent – ի getter – ով <<օդի մեջ>> փորձենք get անել child – երը ապա նույն սկզբունքով java կոմպիլյատորը ignore կանի геттер – ի կանչը: Դրա համար child - երը առանց որևէ փոփոխականի վերագրման ուղակի load անելու համար օգտագործվում է Hibernate.initializer(parent.getChildes()) մեթոդը:

Lazy loading – ի ժամանակ տրանզակցիայի փակվելուց հետո child-երին ստանալու միակ տարբերակը դա Hibernate session – ը նորից բացնելն է: Դրա համար session օբյեկտին ուղակի sessionFactory.getCurrnetSession() – ի միջոցով վերագրվում է նոր սեսսիա և դրանից հետո newSession.beginTransaction(), որից հետո նախորդ արդեն փակվոծ սեսսիային կապված օբյեկտը պետք է կապել նոր բացված սեսսիայի հետ newSession.merg(object) մեթոդով:

**Ինտեգրացիա Spring – ի հետ**

Այստեղ էլ պետք են Hibernate core, spring orm dependency – ները:

Պետք է նշել որ բարդ app – ներում միաժամանակ կարող են օգտագործվել և Hibernate և JdbcTemplate` համապատասխանաբար պարզ և ավելի բարդ query – ների համար:

Configuration կլասսում ավելացվում է @EnableTransactionManagment անոտացիան, իսկ DataSource – ի bean – ը ստեղծելիս նշվում են hibernate.properties ֆայլի համապատասխան փոփոխականները: Hibernate – ի հետ աշխատելու համապր պետք են 1 Properties օբյեկտ որը bean չէ և 2 bean – ները՝

Properties – նշվում են dialect – ի մասի config - ները

@Bean LocalSessionFactoryBean – sessionFactory ստեղծելու համար, որը նաև սքան է անում համապատասխան pacag – ները @Entity – ները գտնելու համար

@Bean PlatformTransactionManagment – թե ինչպես է պետք աշխատել տրանզակցիաների հետ:

Hibernate – ով աշխատելու համար DAO կլասսում ուղակի inject է արվում sessionFactory – ն դրանից հետո մեթոդում ստանում ենք session - ը իսկ տրանզակցիան որևէ մեթոդի մեջ բացելու համար այդ մեթոդի վրա օգտագործվում է @Transactional անոտացիան, ընդ որում եթե մեթոդը միայն select է անում և չի փոփոխում տվյալները դրա արգումնտում ընդունված է նշել

@Transactional(readOnly = true)

**Spring Data JPA (Java Persistance Api)**

JPA – ն աբստրակցիայի ավելի բարձր մակարդակ ունեցող տեխնոլոգիա է DB – ների հետ աշխատելու համար: JPA- ն տակից օգտագործում է Hibernate:

JPA – ում բոլոր CRUD և ոչ միայն մեթոդները գերենացվում են ավտոմատ ինտերֆեյսների և անոտացիաների միջոցով:

JPA -օգտագործելիս DAO – ին փոխարինում է @Repository գաղափարը`

* Երկուսն էլ տվյալների հետ աշխատանքի համար են
* Repository – ն ավելի բարձր մակարդակի է, աշխատանքը միայն entity – ների հետ է առանց sql կոդի, իսկ DAO – ն ավելի ցածր մակարդակի է ավելի մոտ slq կոդին
* Բարդ app – ներում սովորաբար օգտագործվում են և repository – ն և DAO – ն: Repository – ն CRUD գործողությունների իսկ DAO – ն ավելի բարդ slq – ների համար:

Controller – ի և repository – ի արանքում նախատսվում է սերվիսի շերտ՝ @Service անոտացիայով:

Պետք է ավելացնել spring data jpa dependency – ն:

Config ֆայլում LocalSessionFactory – ն փոխարինվում է EntityManagerFactory – ով իսկ hibernateTransactionManager – ը jpaTransactionManager - ով: Տարբերությունը այն է որ սեսսիան և sessionFactory – ն առկա են hibernate – սպեցիֆիկացիայում, իսկ JPA սպեցիֆիկացիայում չկան նման հասկացություններ, դրանց փոխարեն կան entityManager և enityManagerFactory:

Repostiory ստեղծելու համար պետք է ստեղծել ինտերֆեյս @Repository անոտացիայով և այն ժառանգել JpaRepository<T, V> – ից: T – այն օբյեկտն է որի հետ պետք է աշխատի repository – ն իսկ V- ն primary key – ի տիպը՝ Integer:

D8anic hetw petq 1 stex]el se8vis \e8ty @Service անոտացիայով և քանի որ տրանզակցիոն գործողությունները պետք է տեղի ունենան այս մակարդակում ապակ կլասսը նշվում է նաև @Transactional անոտացիայով: