HIBERNATE & JPA

↓

Java persistence API

1. Define entities.
2. Relationship blw entities.

JPAL: We write queries using entities.

Spring Boot:

1. Enable building production ready applications quickly
2. Spring Boot is neither an application server nor a web server.
3. It provides great integration with embedded servers like Tom Cat Jetty or undertow

Features:

Quick Started projects with auto-configuration.

@SpringBootApplication indicates that spring context file

↳ It enables autoconfiguration.

↳ It enables Component scan.

SpringApplication.run method is used to run Spring

run method returns ApplicationContext.

```
ApplicationContext applicationContext = SpringApplication.
                                        (  .class).
for(String name : applicationContext.getBeanDefinitionNames
{
    Sop(name)
}
```

logging.level.org.springframework=Debug

Spring:-

Responsible to solve Dependency Injection.

Loosly coupling using autowired

Spring MVC:-

Concepts like.

Dispatcher servlet.

Model and view and

View resolver

JDBC to JPA:-

Hibernate:

Entity is the specialization of Bean.

```
public class CourseRepository {
@Autowired EntityManager em;
public Course findById() {
em.find(Course.class, id);
```

```
class ... implements CLR {
@Auto
{
Logger logger = LoggerFactory.getLogger
Logger.(this.getClass());
}
```

application.properties:

spring.h2.console.enabled=true.

↳ turn on statistics on.

spring.jpa.properties.hibernate.generate-statistics=true

logging.level.org.hibernate.stat=debug

#To show all queries

spring.jpa.show-sql=true ↳ how many queries are executed.

spring.jpa.properties.hibernate.format-sql=true.

logging.level.org.hibernate.type=~~debug~~ trace

```
Class CourseRepository {

@Autowired
EntityManager em;

Public void deleteById(long id) {
Course course = findById(id);
em.remove(course);

Public Course save(Course course)
{
  if (course.getId() == null) {
    em.persist(course);
      └save.
  }
  else {
    em.merge(course);
      └update.
  }
    return course;
}

Public void test with EntityManager()
{
```

Unit test for save method:-

```
@Test
@DirtiesContext
public void save() {
Course course = repository.findById(10001L);
assertEquals("JPA Course", course.getName());
//update details
course.setName("JPA in steps - updated");
repo.save(course);
Course course1 = repo.findById(10001L);
assertEquals("JPA in 50 steps - updated", course1.getName());
}
```

```
public void playWithEntityManager() {
    em.persist(new
Course course = new Course("Web services in 100 steps");
em.persist(course);
course.setName("Web services in 100 steps - updated");
```

↳ this will saved in database due to the @Transactional annotation. If anyth

```
Public void playWith() {
    Course course1 = new Course("Web Services in 100 steps");
    em.persist(course1);
    em.flush();

    course1.setName("Web Services in 100 steps - updated");
    em.flush();
    em.clear();
    Course course2 = new Course("Angular JS in 100 Steps");
    em.persist(course2);
    em.flush();
    em.detach(course2);
    course2.setName("Angular JS in 100 steps - updated");
    em.flush();
  }
}
```

em.flush(); → updates or not saved.

em.clear(); clears out the tracked by entity manager.

em.flush(); ↓ not updated after detach

after detach method is called

no other will be updated

em.flush(); → It is used changes up to that point can sent to data

em.refresh(course1);

↳ data will be refreshed

↳ change will be lost

↳ previous data will be present.

Entity manager is a interface.

↳ PersistentContext

JPQL

Java persistence Query language:-

In JPQL query from entities
↳ classes

Select c FROM Course c

)Test

ublic void jpql_typed() {

ypedQuery<Course> query =

em.createQuery("select c FROM
Course c", Course.class);

st<Course> resultList = query.getResultList();

ogger.info("select c from Course c -> {}", resultList);

UnitTest

```
@Test
Public void find()
{
 List rs=em.CreateQuery
("select c FROM Course
c").getResultList();
 Course.class;
loggel.info("select c
From Course c ->{}",
 resultList);
```

```
@Test
Public void jpql_where() {
TypedQuery<Course> query = em.createQuery("select c
    Course c where name like '%100 Steps'", Course.cla
List<Course> resultList = query.getResultList();
log.info("select c From Course c where name like
    '%100 Steps' -> {}", resultList);
```

JPA and Hibernate annotations:

@Table(name = "CourseDetails")
↳ to define name of table

@Column(name = "full name", nullable = false)
↳ means name can not be null
↳ to define column name

name of column in table is full name

```
Public class Course {
@UpdateTimestamp
private LocalDateTime lastUpdatedDate;
@CreationTimestamp
private LocalDateTime CreateDate;
```

```
@NamedQuery(name = "query-get-all-courses", query = "Select
class Course {                                  c From Course c");

}

@Test
public void query() {

Query query = em.createNamedQuery("query-get-all-courses");

List resultList = query.getResultList();

logger.info("select c From Course c -> {}", resultList);

@Test
public void jpql_typed() {

TypedQuery<Course> query =

    em.createNamedQuery("query-get-all-courses", Course.cla

List<Course> resultList = query.getResultList();

logger.info("Select c From Course c -> {}", resultList);

}
```

@NamedQuery is not reusable

We have to use

```
@NamedQueries({@NamedQuery(name="query-ge
```

query = "select c From Course c where name like '%...

Native Queries:-

Creat new NativeQueryTest;

↓

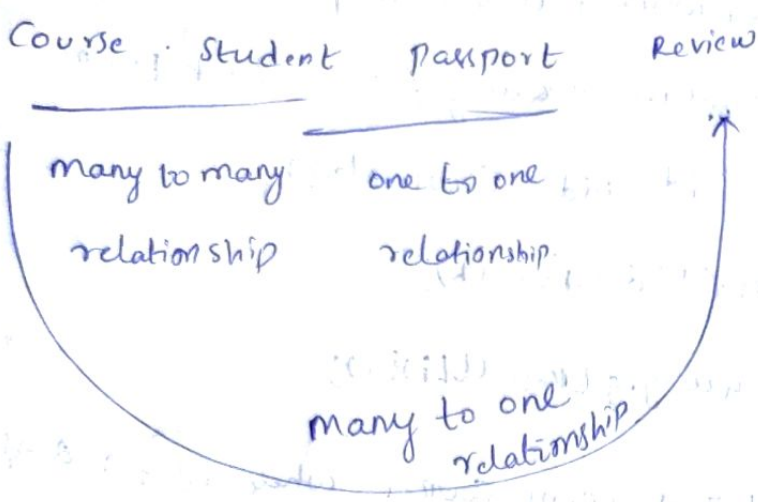
Sending direct sql out to JPA.

```
@RunWith(SpringRunner.class)
Public class NativeQueriesTest {

@Autowired

EntityManager em;

@Test

Public void native_queries_basic() {

Query query = em.createNativeQuery("select * From Cour

List resultList = query.getResultList(); Course.cla

logger.info("Select c From Course c -> {}", result

@Test

Public void native_queries_with_Parameter() {

Query query = em.createNativeQuery("SELECT * FROM

COURSE where id = ?", Course.class);

query.setParameter(1, 10001L);

List resultList = query.getResultList();
```

```
@Test
Public void native_queries_with_named_parameter() {
Query query = em.createNativeQuery ("select * FROM
    COURSE where id = :id", Course.class);
query.setParameter("id", 10001L);
List resultList = query.getResultList();
logger.info("SELECT * FROM COURSE where id = :id -> {}",
    resultList);
```

```
@Test
@Transactional
Public void native_queries_to_update() {
Query query = em.createNativeQuery("update COURSE set
    last_updated_date = sysdate()", Course.class);
List resultList = query.executeUpdate();
Int no.of Rows.
logger.info("no.of ROWS updated -> {}", no.of Rows");
```

Entities and Relationships:-



Eagerfetch:-

fetching student details, passport details also fetched is called eager fetching

```
class Student{
@OneToone(fetch = FetchType.LAZY)
private Passport passport;
}
```

Persistence Context contains entities and also stored in it

We can interact with persistence Context using Entity manager

persistence Context is created at the start of transaction, all the changes tracked by persistence

Persistence ... access to database.

Persistence Context Stores entities as we manage any changes.

One to one mapping:-

@OneToOne(fetch = FetchType.LAZY, mappedBy = "passport");

↓

to make student owning side of relation

Passport is non owning side of relationship

why do we need @Transaction?

A: whenever we make changes on database then we need

@Transactional.

when we are ~~insert or~~ updak or delete.

Do read only methods need a transaction?

```
List<Comment> someReadonlyMethod(){

User user = em.find(User.class, 1L);

List<Comment> comments = user.getComments();

return comments;
```

In entity manager there is default Transaction.

Without transaction there is no connection to database.

Why do we use @DirtiesContext.

---

to reset the database.

Many to one

```
Public class Course {

@OneToMany
private List<Review> reviews = new ArrayList<>();

Public void addReview(Review review) {
    this.reviews.add(review);
}
Public void removeReview(Review review) {
    this.reviews.remove(review);
```

done by us.

Not to manipulate

Not used by others

```
Public class Course {

@ManyToOne
Course course;
```

```
Public void addReviewsForCourse() {                    CourseRepositoryTest

    Course course = findById(10003L);

    logger.info("course.getReviews() -> {}", course.getReviews());

    Review review1 = new Review("5", "Great");

    Review review2 = new Review("5", "Halsoff");

    // setting the relationship
    course.addReview(review1);   review1.setCourse(course);

    course.addReview(review2);

    review2.setCourse(course);

    em.persist(review1);
    em.persist(review2);          // adding to database.
```

Public class Student {

@manyToMany we can create JoinTable like COURSE_STUDENT

@ManyToMany

@JoinTable(name="STUDENT_COURSE", joinColumn=@JoinColumn(name="STUDENT_ID")) | creating a join with table

// JoinColumn - STUDENT_ID

// inverseJoinColumn - COURSE_ID

JoinTable(name="STUDENT_COURSE",
    joinColumns=@JoinColumn(name="STUDENT_ID"),
    inverseJoinColumns=@JoinColumn(name="COURSE_

```
Public void insertStudentAndCourse() {
Student student = new Student("Jack");
Course course = new Course("microservices in 100 steps");
em.persist(student);
em.persist(course);
Student.addCourse(course);
Course.addStudent(student);
em.persist(student);
}
```

Map inheritance relations with table

Single table option.

Both PartTime and FullTime employee stored in single table.

```
@DiscriminatorColumn(name="EmployeeType")
@Inheritance(strategy=InheritanceType.SINGLE_TABLE)
Public class abstract Employe {
}
```

```
@Inheritance(strategy=InheritanceType.TABLE_PER_CLASS);
```

↳ Individual tables are created.

TABLE-PER-CLASS:

Each concrete class has it own table.

~~Column~~ Common columns will be repeated.

Retrieval is done using UNION

```
@Inheritance(strategy = InheritanceType.JOINED)
Public class Employee{

}
```

↓

Separate tables will be created for parent class and sub classes

To get details of subclass Join will performed to get details,

@mappedsuperclass

@Entity ↳ Create table for subclass,

↳ there will be no relation ship

```
Pubclass Employee(){

}
```

JOINS:

JOIN -> select c, s from Course c JOIN c.students s

LEFTJOIN -> select c, s from course c LEFT JOIN c.students

CROSSJOIN -> Selet c, s from Course c, student s

Criteria Queries:

Public class CriteriaQuery Test {

@Test

Public void jpql_basic(){

CriteriaBuilder cb = em.getCriteria Builder();

cb.~~createQuery~~(

CriteriaQuery<Course> cq = cb.create Query(Course.class);

~~Cq.from~~

Root<Course> courseRoot = Cq.from(Course.class)

TypedQuery<Course> query = em.createQuery(Cq.select(courseRoot));

1. Use criteria builder to create a query returning the expected result object

2. Define roots for table which are involved in the query

3. Define predicates using Criteria Builder

4. Add predicates to the Criteria query

5. Build the Typed

```
List<Course> resultList = query.getResultList();
logger.info("Typed Query -> {}", resultList);
```

```
public void all_courses_having() {

CriteriaBuilder cb = em.getCriteriaBuilder();                    1.

CriteriaQuery<Course> cq = cb.createQuery(Course.class);         2.

Root<Course> courseRoot = cq.from(Course.class);                 3.

= cb.like(courseRoot.get("name"), "%100 steps");  // Define the Predicate
predicate like.                                                  4.

                cq.where(like);                                  5.

TypedQuery<Course> query = em.createQuery(cq.select(courseRoot));

List<Course> resultList = query.getResultList();

logger.info("TypedQuery -> {}", resultList);
```

```
public void all_courses_without_students() {

1,2,3 steps are common.

Predicate ~~list~~. studentsIsEmpty = cb.isEmpty(courseRoot.get
                                        ("students"));

cq.where(studentsIsEmpty);
```

Criteria query using Join:

→ Criteria Join.

Public void join() {

1, 2, 3 steps are common.

Co~~urseRoot.join("students")~~

Join<object, object> join = CourseRoot.join("students");

4, 5, 6 are common.

@Test

public void leftJoin()

{

Join <object, object> join = CourseRoot.join("students",

JoinType.LEFT);

}

Transaction management:

When making changes for data.

Transaction Management – Acid properties:

A – Atomicity

C – consistency

I – Isolation

D – Durability

Atomic: Either the transaction should be completely successful or all the changes done by transaction should be reverted back

Consistency: Leaving the system in consistent state whether the transaction is successful or fails.

Isolation: Multiple levels of isolation are different. If one transaction is updating and other transaction is observing the updation

ability: once the transaction is completed, even whe

1. Dirty Read
2. Non Repeatable Read
3. Phanthom Read

1. Dirty Read :- Another Transaction reading unpublished value before a given transaction is committed.

Transaction 1 & 2 running parallely

Dirty Read	Description	A's account	B's
Transact 1 - Step 1	Deduct A's account by 50$		
Transaction 2 - Step 2	Deduct A's account by 100$		
Transaction 1 - Step 2	Deposit 50$ in to B's Account		
Transaction 2 - Step 2	Deposit 100$ in to B's Account		

Non Repeatable Read.

When reading same thing we get two different Description. Value. is called non repeatable read.

Transaction - 1 - Step 1 : Select * from person where id = 10

Transaction - 2 - Step 1 : Update person set age = 30 where id

Transaction - 1 - Step 2 : Select * from person where id = 10

Phanthom Read :- At different time we get different no. of rows Description in same transaction

Transaction - 1 Step 1 : Select * from person where age

Transaction - 2 Step 1 : Insert into values (13, 'Ravi')

4 Isolation levels:

	Dirty Read	Non Repeatable Read	Phantom Read
Read Uncommitted	possible	possible	possible
Read Committed	Solved	possible	possible
Repeatable Read	solved	solved	possible
Serializable	solved	solved	solved

Now will be locked

Other transaction not able to update

After completing the first transaction then only another transaction is possible

until transaction 1 will complete the only other transaction will take action

Locking Isolation levels:

Using Spring Transaction,

We can give isolation levels

There are two Transactional annotations

@Transactional for javax and spring.

Talking with database 1

database 2

// mg an interface.

JPA able to manage changes only in one database

Even we are making multiple updates

If update 2 fails data change by update 1

will get roll back

@JPA @Transaction is sufficient for single database

If we want to manage transaction across multiple databases use Spring @Transactional.

@Test

@Transactional(isolation = REPEATABLE_READ)

spring data JPA: → JPA specific implementation of Spring JDA

Aims to provide simple abstraction.

```
@Test
public void findById_CourseIsPresent()
{
    Optional<Course> course = repo.findById(1000L);
    assertTrue(course.isPresent());
}

public void findById_CourseNotPresent(){
    Optional<Course> courseOptional = repo.findById(2000L);
    assertFalse(courseOptional.isPresent());
}

@Test
public void testing() {
    Course course = new Course("microservices in 100 steps");
    repo.save(course);
course.setName("microservices in 100 steps - updated");
```

```
logger.info("courses -> {}", repo.findAll());
logger.info("Count -> {}", repo.count());
```

Sorting using Spring Data JPA repositories

```
@Test
Public void sort() {
Sort sort = new Sort(Sort.Direction.DESC, "name");
logger.info("sorted courses {}", repo.findAll(sort));
```

@Test — ⟵ divide them in to pages.

```
Public void Pagination() {
PageRequest pageRequest = ~~new PageRequest(0,~~
                        = PageRequest.of(0,3);
~~logger.info("First -> {}", repo.findAll(~~
Page<Course> firstPage = repo.findAll(pageRequest);
logger.info("First Page -> {}", firstPage.getContent());
Pageable secondPage = firstPage.nextPageable();
Page<Course> secondPage = repo.findAll(secondPageable);
logger.info("second Page -> {}", Pageable);
```

Custom queries using Spring Data JPA

```
Public interface CoursesSpringRepository extends JPARepository<Course,
                                                    Long>
{
    List<Course> findByName(String name);
    List<Course> countByName(String name);
    List<Course> findByNameANDId(String name, Long id);
    List<Course> findByNameorderByIdDesc(String name);
@Test
Public void test()
{
    logger.info("{}", repo.findByName("WebServices"))
}
```

JPQL Query

```
@Query("select c FROM Course c where name like '%10
                                              steps'
List<Course> CourseWith10StepsInName();

@Query(value="select * from Course c where name
    '%100 Step'", nativeQuery=true)

List<Course> courseWith100StepsInNameUsingNative
```

Spring Data REST :-

@JsonIgnore

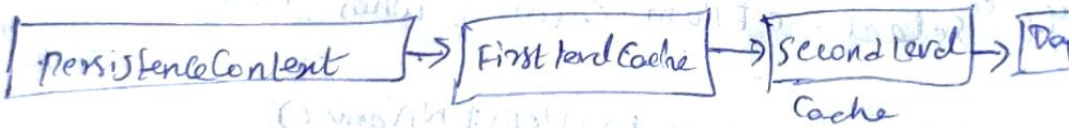
for multi repeating

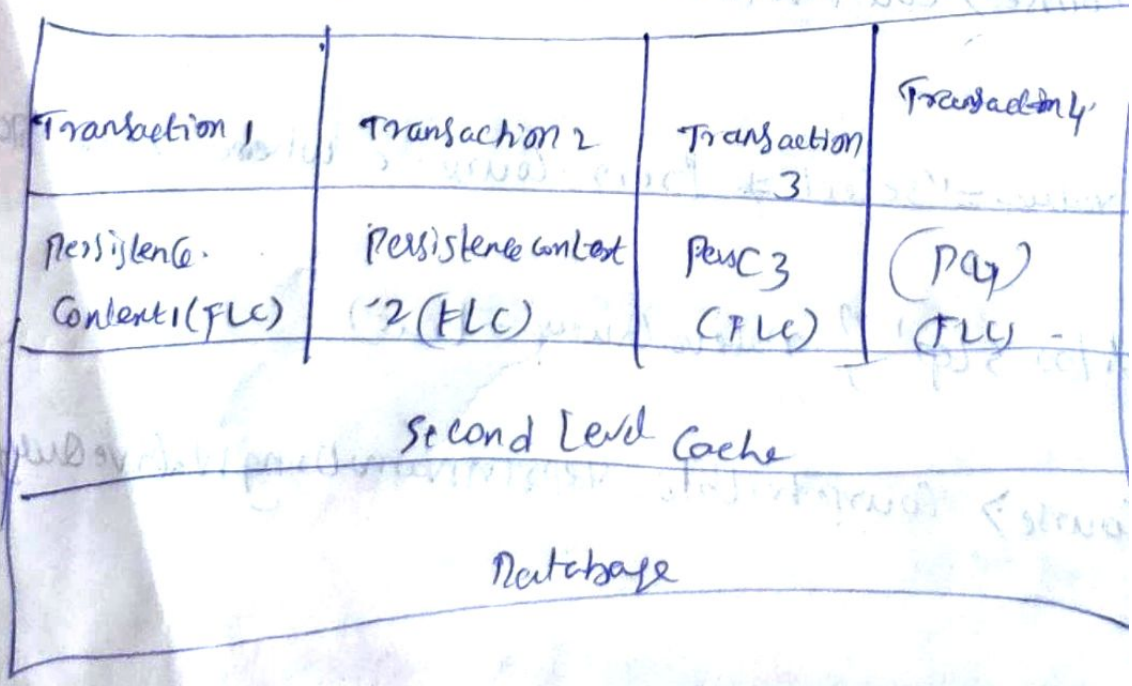
Caching with hibernate.

Two levels of Caching:

First level Cache

Second level Cache.





First level Cache: when we want retrive data with same course again and again

First time it will go to the database and put it in Persistence context.

↳ (FLC) → For single transaction

Second level Cache: Across multiple application

multiple users using applications.

List of users in country is same.

Stores common information for all the users of application

Data common across multiple users

First level cache: First level cache is within the Boundary of single transaction

@Test → In CourseRepositoryTest.

@Transactional

Public void findById-firstLevelCacheDemo(){

Course course = repo.findById(10001L);

logger.info("First Course Retrieved {}", course);

Course course1 = repo.findById(10001L);

logger.info("First Course Retrieved again {}", course1)

assertEquals("JPA in 50 Steps", course.getName());

To enable second level cache - ehcahe

1. enable. Second. level. cache

2. Specify the caching framework - Ehcahe.

3. only cache what I tell to Cache.

4. what data to cache.

application. properties

Spring. ja. properties. hibernate. cache. use-second_level_cache
= true

Spring. jpa. properties. hibernate. cache. region. factory_class

= hibernate. cache. ehcache. EhCacheRegionFactory.

Spring. jpa. properties. javax. persistence. sharedCache. mode =

ENABLE - SELECTIVE

logging. level. net. sf. ehcache = debug

---

t deletes! . None by adding a column to database
to track it is delete or not.

bernate specific annotation

@NDelete (sql = "update course set is_deleted=true
where id = 9")

0 -> false

1 -> true

@preRemove → method will be called whenever a row of particular method will be deleted

```
Private void preRemove()
{   logger.info("setting isDeleted to true");
    this.isDeleted = true;
}
```

JPA life cycle methods

postPersist

postRemove

```
Public class student
{
Private Addres address
}
```

@Embeddable → embeded to other entity.

class

```
Public Address {
    private String line1;
    private String line2;
    private String city;
```

→ To store data of other objects in to current entity.

```
Public enum ReviewRating {

ZERO, ONE, TWO, THREE, FOUR, FIVE

  @Entity
  Public class Review {

  @Enumerated(EnumType.STRING)

Private ReviewRating rating;

}
```

JPA is transaction frontend application.

When do you use JPA?

1. SQL Database
2. Static Domain model
3. Mostly Crud.
4. mostly simple Queries/mapping

Performance Tunings:

Measure the performance and after tune it.

ZERO PERformance Tuning without measuring

- Enable and monitor Stats in atleast one environment.

Performance Tuning - Indexes:

1. Add the right indexes on database
2. Execution plan.

Use Appropriate Caching:

1. First level caching.
2. Second level caching → different transaction on same server or same instance of application share common data
3. Distributed Cache
4. Be careful about the size of first level Cache.

Eager vs Lazy fetch:

1. Use Lazy fetching mostly
2. remember all mapping
   (@ manyTOone and
   @ oneTOone - all EAGER
   by default

Useful for caching across all multiple ~~caches~~ instances

Ex: HazelCache

Avoid N+1

Entity Graphs & Named Entity Graphs & Dynamic

```
public class PerformanceTuningTest {
    @Autowired
    EntityManager em;

    @Test
    @Transactional
    public void creatingNplusoneproblem() {
        List<Course> courses = em.createNamedQuery("query_get
            _courses", Course.class).getResultList();

        for(Course course : courses)
        {
            logger.info("course -> {} . Students -> {}", course,
                course.getStudents());
        }
```

To fix N+1 problem make it EAGER Fetch

```
@Test
@Transactional
public void solvingNplusoneproblem_EntityGraph() {
    EntityGraph<Course> entityGraph = em.createEntityGraph
        (Course.class);
    Subgraph<Object> subGraph = entityGraph.addSubgraph
```

```
List<Course> courses = em.createNamedQuery("query_get_all
        _Courses", Courses.class);
    .setHint("javax.persistence.loadgraph", entityGraph).
        getResultList();

for(Course course : courses) {
    logger.info("Course -> {} Students -> {}", course,
                                course.getStudent());
```

```
@Test
@Transactional
Public Void joinFetch()
{
List<Course> courses = em.CreatNamed
    @Query("query_get_all_Courses_join
        feth", Course.class).getRes
                    ultList();
```

```
query = "Select c FRO
Course c JOIN FETCH
    c.students s"
```

```
for(Course course : courses)
{
    logger.info("Course -> {} students -> {}", Course
                    Course.getStudents());
}
```