# Spring Boot

* @GeneratedValue annotation lets the framework to generate value and not by manually.
* Java class should have a public or protected non argument constructor.
* Auto-Configuration is the heart of spring boot. Let’s see tomcat is running without any configuration, that is done by auto-configuration.

## @EnableAutoConfiguration

* Allows for configuration classes to be scanner dynamically
* Often based on JARs loaded onto class path
* **@AutoConfgureBefore** and **@AutoConfigureAfter**
* **@ConditionalOnClass**
* **@ConditionalOnBean**
* **@ConditionalOnProperty**
* **@ConditionalOnMissingBean**

## @SpringBootApplication

* It states that it is context file
* Enables auto-configurations
* Enables component scan – scan all the classes in class path to find beans

## Spring Boot Actuator

* It does monitoring of Spring Boot Application
* You can see a lot of meta data of the application, you want to see how many beans has configured, or how many times this specific service has been called
* To enable Actuator, add its dependency in pom.xml and hal browser dependency
* Go to properties file and add following property, however it will impact performance

Management.endpoints.web.exposure.include=\*

## Spring Boot Developer Tools

* When the application is up and running and you change the code, you do not need to stop and start again the application to pick up the change, just add the dependency of spring boot devtools, and then when you save the change it automatically picks up the change in minimal time.
* To fill up database automatically with data file, create a file with extension .sql and save under resources folder where application.properties is saved

## World before JPA and Hibernate

## Entity Manager

* **Em.persist( obj )** is used to save or insert a record.
* **Em.merge( obj )** is used to update a record.
* When with entity manager you save a record in database with method **persisit()** and later on you changes any property of that object, that is also changed in database.

Course course = **new** Course(**"My Java Course"**);**entityManager**.persist(course);  
course.setName(**"My java course updated"**);

* **EntityManager.flush()** will send the data to database immediately and does not wait for transaction to commit.
* **EnitityManager.detach ( obj ):** By calling this method on an object, entity manager will stop tracking that object, whatever changes occurs to that object will not be persisted in database.
* **EntityManager.clear():** instead of calling detach method on each object, you call call this **clear()**  method and entity manager will stop tracking all changes occurring to all objects.
* **EntityManager.**refresh(): again start tracking

## Entity

* Entity is a pojo class , it represents a row in a table which we want to persist.

## Persistence Context

**Persistence Context** is an **environment** or **cache** where entity instances(which are capable of holding data and thereby having the ability to be persisted in a database) are managed by **Entity Manager**.It syncs the entity with database.All objects having **@Entity** annotation are capable of being persisted. **@Entity** is nothing but a class which helps us create objects in order to communicate with the database.And the way the objects communicate is using methods.And who supplies those methods is the Entity Manager.

## CommandLineRunner vs ApplicationRunner

## @Transactional

This annotation is applied on repository class to take care of transactions.

When we manipulate data, perform any crud operation, that should be done via Transaction. A transaction takes care of **ACID** property of Database. We do not need transaction for find methods.

Transaction take care that if any error happens in performing crud operation, everything will be rolled back.

## Annotations

# Udemy Course: JPA and Hibernate with Spring Boot

# JPA: Java Persistence API

JPA says that you don’t need to worry about writing queries. Just tell me about your objects and how they map to your tables in database. Tell me about the relationships between tables and I’ll write queries on behalf of you.

## JDBC or Spring JDBC

Approaches present there before JPA. In which we were writing a lot of queries manually. Developer was responsible of writing right queries and giving right values in those queries.

## Entity

A uniquely identifiable object that we want to persist.

@Id, @GeneratedValue

* For @Id, we only generate getter method, because hibernate sets the value automatically and we don’t want others to set the value.
* Hibernate requires a no argument constructor.

## Repository

* In Repository, we add EntityManager

## JPQL :

* In SQL we query from tables while in JPQL we query from Entities.
* The queries written in JPQL are converted in SQL queries by JPA implementation that is hibernate.

## Relationship

**ManyToMany** : for ManytoMany a bridge table is formed.

* @OneToOne is always retrieved **Eagerly.**
* JoinTable comes in result of ManyToMany relationship.
* Customizing the Join Table

## Inheritance Relationship

## Transaction Management

* Without Transaction management, if any of the statement fail, system would be left in an inconsistent state.
* Transaction ensure ACID property of database.
* **Atomicity:** Either all of none of the transaction should be successful.
* **Consistency:** After all transaction are committed, whether all the transactions are succeeded or failed, system should in consistent state. A **Consistent state** is that
* **Isolation:** How queries in the same transaction are isolated from each other.
* **Durability:** Once a transaction is successful, even the system crashes or something wrong happen, the change should be persisted even there are other system failures.

**When one transaction is writing a data and another transaction is reading the exact data, in this condition, there are three scenarios**

**Dirty Read:**

Dirty read is 2nd transaction is reading the value modified by 1st transaction when the 1st transaction is not completed yet. In Case the 1st transaction fails and reverts the value to original, 2nd transaction is still holding the modified value, which is wrong.

**Non-Repeatable Read:**

* A non-repeatable read is the one in which data is read twice inside the same transaction cannot be guaranteed to have the same value. Depending on the isolation level, another transaction could have nipped in and updated the value between these two reads.

**Scenario:**

Statement A reads the age value of a person is 50;

Statement B update the age value of the same person to 60;

Statement C reads the values and now it is 60.

And its all happening in the same transaction.

**Phantom Read:**

* A phantom read occurs when, in the course of a transaction, two identical queries are executed, and the collection of rows returned by the second query is different from the first.

**Scenario:**

First query to get all persons with age between 20 and 50; it returns 3 records.

Second query insert a person with age 30;

3rd query to get all person with age between 20 and 50, now it returns 4 records.

Two identical queries return different number of rows returned within the same transaction; this is phantom read.

## Spring Boot Features

## Auto Configuration

# @SpringBootApplication

* It indicates that it is a Spring Context file
* It performs auto configuration
* It performs component scan

## @ComponentScan

* Performs scan for packages and sub package to search for Beans

## SpringApplication.run()

* It returns application context

## Spring vs spring boot vs spring mvc

These resolve different problems and they are not competing each other for the same thing

<https://www.springboottutorial.com/spring-boot-vs-spring-mvc-vs-spring>

**What is the Core Problem the Spring Resolves?**

Most important feature of Spring Framework is Dependency Injection. At the core of al Spring modules is Dependency Injection or IOC Inversion of Control.

**What is the Core Problem the Spring Boot Resolves?**

Auto configuration.

**What is the Core Problem the Spring MVC Resolves?**

Spring MVC Framework provides decoupled way of developing web applications. With simple concepts like dispatcher servlet, ModelAndView and View Resolved, it makes it easy to develop web applications.

* The spring framework resolve the problem of dependency injection or inversion of control, this the at the core of spring framework

## Caching

* Cache is needed to boost performance
* In hibernate there is two levels of cache
* First level cache is maintained in persistence context
* First level cache is for the single transaction, if we don’t add @Transactional annotation, both find queries will hit the database and not the cache.
* Second level cache is for multiple transaction
* Distributed cache works among the applications

**Steps for enabling second level cache**

* + Add dependency of eh cache
  + Add enable property for second level cache
  + Tell which cache framework to use, usually it is eh cache
  + Which cache mode to use, all, none or selective
  + Tell which entities to cache using @cachable on the entity

Persistence Context 🡪 first level cache 🡪 second level cache 🡪 database

## Soft Deletes

* Soft deletes means instead of removing record from database, you set a key/column to indicate that whether this is record is active or not.
* On Entity level/ class name, add following annotations, this means on delete statement, this sql query should be executed
* @SQLDelete(sql = “update entity\_name set is\_deleted = true where id = ?”), but this will not work with native queries, its your responsibility to set is\_deleted false in query.
* The problem with this approach is that, it still return this record in find query, and we need to filter it manually, to resolve this problem you set an annotation @Where on entity level and give a clause to it like following and it will not return the data where is\_deleted = true

@Where(clause = “is\_deleted= false”) // return rows where is\_deleted is false, but native queries don’t use @where annotation, so write this condition in native query.

* @SQLDelete only update column in database, it does not update attribute value of the entity, keep this in mind, Solution is that, there is an annotation called @PreRemove which is set on a method, this method is called before deleting any row of the entity, change the values of attribute in this method

@PreRemove

Private void preRemove(){

This.isDeleted = true;

}

## Entity Life cycle methods

**PostLoad:**

The method annotated with this will be called after entity is retrieved and loaded in memory, in result of a select query.

**PostPersist:**

The method annotated with this will be called right after entity is persisted in database , it is called of em.persist() method.

**PostRemove**

**PreLoad**

**PrePersist**

**PreRemove**

## Embedded and Embeddable

* This embedded field will be directly present in table, no separate table will be created
* Purpose is code cleanup and maintainability

## Performance Tuning

* Enable statistics on debug level to measure performance

**Indexing**

* Add right indexes on the databases, create index on most searched by attribute in table

**Caching**

* Use appropriate caching,
* Be aware of the size of first level cache, regularly clear using entity manager

**Using lazy or Eager fetch**

* Use lazy fetching mostly
* Remember that mapping \*toOne (ManyToOne, OneToOne) are Eager by default
* Any mapping of \*toMany is by default Lazy fetch

## N+1 Problem and solution

**Problem:**

When there is \*toMany mapping, number of queries are fire to get record of dependent data.

**Solution:**

* Its first and simple solution is using Eager fetch instead of Lazy fetch. But its drawback is that you will be getting data that you don’t require at that time, and it will decrease performance, so it is not the best solution.
* Using Entity graph and sub graph, the name of main entity will be the entity graph, and name of subsequent entity, will be the sub graph and we will set hint the graph
* When we use graph, it will use joins to get record, instead of using multiple select queries
* Third approach is using JoinFetch

Select c from Course Join Fetch c.students s

## Criteria Queries

* Instead of writing JPQL queries, define some criteria using Criteria APIs and that will do the work.

# Spring Security

1. Custom class that extends WebSecurityConfigureAdapter
2. Create authenticate API endpoint
   * Accept user name and password
   * Return jwt token
3. Intercepts all incoming request

* Extract JWT token from all incoming request
* Validate and set in execution context

## Using @Autowired with constructor

## @PostConstruct

## @PreDestroye

# Aspect Oriented Programming