Spring JDBC and JPA (Hibernate)

Step 01 - Setting up a project with JDBC, JPA, H2 and Web Dependencies

Step 02 - Launching up H2 Console

SpringBoot will automatically create a connection to H2 database with the help of the below dependency.

```
          <dependency>
               <groupId>com.h2database</groupId>
               <artifactId>h2</artifactId>
               <scope>runtime</scope>
          </dependency>
```

To enable H2 console, (in application.properties):

```
spring.h2.console.enabled=true
```

Restart the application and check the console for below log;

```
 Servlet webServlet mapped to [/h2-console/*]
```

H2 console is accessible at the below URL:
http://localhost:8080/h2-console

Use defaults entries in the H2 console for now. In-memory database is only live as long as the application is running.

Step 03 - Creating a Database Table in H2
create a data.sql file in source/main/resources folder. Write queries in data.sql for table creation.

Step 04 - Populate data into Person Table
If you insert some data into H2 table by instantiating the entity objects, after application restart all the data will be lost. Just to counter this, enter the insert statement in data.sql so that every time the application is restarted, insert query runs automatically.

Step 05 - Implement findAll persons Spring JDBC Query Method

PersonJbdcDao.java

PersonJbdcDao.java will talk to the database and get the values. Annotate this class with @Repository as it talks to the database.

```
@Repository
public class PersonJbdcDao {

     // JdbcTemplate is spring's way to provide database connection
     @Autowired
     JdbcTemplate jdbcTemplate;
     public List<Person> findAll() {
          // This will get the resultset and map individual rows to the
Person class
          return jdbcTemplate.query("select * from person",
                    new BeanPropertyRowMapper(Person.class));
     }
}
```

Step 06 - Execute the findAll method using CommandLineRunner

To fire the finalAll() method at the start of the application.When we implement CommandLineRunner interface, the code inside run() method will launch as soon as the ApplicationContext is ready.

```
@SpringBootApplication
public class DatabaseDemoApplication implements CommandLineRunner {

     private Logger logger = LoggerFactory.getLogger(this.getClass());

     @Autowired
     PersonJbdcDao dao;

     public static void main(String[] args) {
          SpringApplication.run(DatabaseDemoApplication.class, args);
     }

     @Override
     public void run(String... args) throws Exception {
          logger.info("All users -> {}", dao.findAll());
     }
}
```

O/P:

```
All users ->
[com.personal.kunj.database.databasedemo.entity.Person@51fc862e,
com.personal.kunj.database.databasedemo.entity.Person@a7cf42f,
com.personal.kunj.database.databasedemo.entity.Person@fe09383]
```

How to avoid the above Hashcode output? → Override toString() in Person.java (Entity class)

O/P:

```
All users -> [Person [id=10001, name=Ranga, location=Hyderabad,
birthDate=2018-09-05 10:52:53.274], Person [id=10002, name=James,
location=New York, birthDate=2018-09-05 10:52:53.29], Person [id=10003,
name=Pieter, location=Amsterdam, birthDate=2018-09-05 10:52:53.29]]
```

Note: The bean on which BeanPropertyRowMapper is defined should have a default constructor. So Person class should have a default constructor. Otherwise we will get below exception:

```
Caused by: java.lang.NoSuchMethodException:
com.personal.kunj.database.databasedemo.entity.Person.<init>()
```

Step 07 - A Quick Review - JDBC vs Spring JDBC

JDBC	Spring JDBC
More lines of code	Less lines of code
ResultSet to object mapping difficult	Easy (using RowMapper classes)
	No need to handle connection, statement etc, JdbcTemplate takes care of these.
	If an Exception occurs then JdbcTemplate closes connections automatically.

Step 08 – What is in the background? Understanding Spring Boot Autoconfiguration

How is JdbcTemplate getting autowired in?

How is JdbcTemplate knowing connection to the in-memory database?

Ans: SpringBoot auto configuration

Set root logging level to debug. logging.level.root=debug.

To see how spring boot auto configuration works, go to eclipse console log and

Look for ============================

CONDITIONS EVALUATION REPORT

============================ in the console log and see the positive matches (auto configuration). Spring Boot looks for the available classes on the classpath and it would automatically configure things based on that. If it sees an in-memoty database on the classpath it automatically creates a connection to it. If web app on the classpath, it automatically configures a DispatcherServlet. If JPA on the classpath, it configures EntityManagerFactory and TransactionManager.

Step 09 - Implementing findById Spring JDBC Query Method

PersonJbdcDao.java

```
public Person findById(int id) {
           return jdbcTemplate.queryForObject
                      ("select * from person where id=?", new Object[]
{ id },
                      new BeanPropertyRowMapper<Person>(Person.class));
      }
```

DatabaseDemoApplication.java

```
@Override
      public void run(String... args) throws Exception {
           logger.info("All users -> {}", dao.findAll());
           logger.info("User id 10001 -> {}", dao.findById(10001));

      }
```

Step 10 - Implementing deleteById Spring JDBC Update Method

In case of update and delete operations, we need to use jdbcTemplate.update().

PersonJbdcDao.java

```
public int deleteById(int id) {
// It returns no of rows rows affected by the query
           return jdbcTemplate.update
                      ("delete from person where id=?", new Object[]
{ id });
      }
```

DatabaseDemoApplication.java

```
      @Override
      public void run(String... args) throws Exception {
           logger.info("All users -> {}", dao.findAll());
           logger.info("User id 10001 -> {}", dao.findById(10001));
           logger.info("Deleting 10002 -> No of Rows Deleted - {}",
dao.deleteById(10002));
      }
```

Console o/P: Deleting 10002 -> No of Rows Deleted – 1

Step 11 - Implementing insert and update Spring JDBC Update Methods

Console O/P:

```
All users -> [Person [id=10001, name=Ranga, location=Hyderabad,
birthDate=2018-09-05 11:53:25.747], Person [id=10002, name=James,
location=New York, birthDate=2018-09-05 11:53:25.762], Person [id=10003,
name=Pieter, location=Amsterdam, birthDate=2018-09-05 11:53:25.762]]
```

```
User id 10001 -> Person [id=10001, name=Ranga, location=Hyderabad,
birthDate=2018-09-05 11:53:25.747]
Deleting 10002 -> No of Rows Deleted - 1
Inserting 10004 -> 1
Update 10003 -> 1
```

Step 12 - Creating a custom Spring JDBC RowMapper

PersonJbdcDao.java

```
package com.personal.kunj.database.databasedemo.jdbc;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Timestamp;
import java.util.List;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.jdbc.core.BeanPropertyRowMapper;
import org.springframework.jdbc.core.JdbcTemplate;
import org.springframework.jdbc.core.RowMapper;
import org.springframework.stereotype.Repository;

import com.personal.kunj.database.databasedemo.entity.Person;

@Repository
public class PersonJbdcDao {
      // JdbcTemplate is spring's way to provide database connection
      @Autowired
      JdbcTemplate jdbcTemplate;


//Creating inner class bcz this PersonRowMapper will only be used inside
PersonJbdcDao class.
      class PersonRowMapper implements RowMapper<Person> {
            @Override
            public Person mapRow(ResultSet rs, int rowNum) throws
SQLException {
                  Person person = new Person();
                  person.setId(rs.getInt("id"));
                  person.setName(rs.getString("name"));
                  person.setLocation(rs.getString("location"));
                  person.setBirthDate(rs.getTimestamp("birth_date"));
                  return person;
            }
      }
            public List<Person> findAll() {
            return jdbcTemplate.query("select * from person", new
PersonRowMapper());
```

```
	}
		public Person findById(int id) {
		return jdbcTemplate.queryForObject
				("select * from person where id=?", new Object[]
{ id },
				new BeanPropertyRowMapper<Person>(Person.class));
	}
		public int deleteById(int id) {
		// It returns how many rows are affected by the query
		return jdbcTemplate.update
				("delete from person where id=?", new Object[]
{ id });
	}
	public int insert(Person person) {
		return jdbcTemplate.update("insert into person (id, name,
location, birth_date) " + "values(?,  ?, ?, ?)", new Object[]
{ person.getId(), person.getName(), person.getLocation(), new
Timestamp(person.getBirthDate().getTime()) });
	}

	public int update(Person person) {
		return jdbcTemplate.update("update person " + " set name = ?,
location = ?, birth_date = ? " + " where id = ?", new Object[]
{ person.getName(), person.getLocation(), new
Timestamp(person.getBirthDate().getTime()), person.getId() });
	}

}
```

O/P:

```
All users -> [Person [id=10001, name=Ranga, location=Hyderabad,
birthDate=2018-09-05 12:36:01.952], Person [id=10002, name=James,
location=New York, birthDate=2018-09-05 12:36:01.967], Person [id=10003,
name=Pieter, location=Amsterdam, birthDate=2018-09-05 12:36:01.967]]
```

Step 13 - Quick introduction to JPA

Why do you map a query and try and map values and get the data back? Why don't you map the entity? Why don't you map an object to a row in the table?

In JPA we define entity and relationship between entities.

We will map a field in the object to a column in the database.

If a Person can have multiple addresses, you can define a relationship between Person and address as well.

The JPA will take care of identifying the entities and creating the right queries for you based on the operations you would want to perform.

The job of writing the query shifts from the developer to JPA implementation framework.

Step 14 - Defining Person Entity

```
package com.personal.kunj.database.databasedemo.entity;
import java.util.Date;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.Id;

@Entity
public class Person {
      @Id
      @GeneratedValue
      private int id;
      private String name;
      private String location;
      private Date birthDate;
      // The bean on which BeanPropertyRowMapper is defined should have a
default constructor.
            public Person() {
                  }

      public Person(int id, String name, String location, Date birthDate)
{
            super();
            this.id = id;
            this.name = name;
            this.location = location;
            this.birthDate = birthDate;
      }
      public Person(String name, String location, Date birthDate) {
            super();
            this.name = name;
            this.location = location;
            this.birthDate = birthDate;
      }

      public int getId() {
            return id;
      }
```

```
	public void setId(int id) {
		this.id = id;
	}

	public String getName() {
		return name;
	}

	public void setName(String name) {
		this.name = name;
	}

	public String getLocation() {
		return location;
	}

	public void setLocation(String location) {
		this.location = location;
	}

	public Date getBirthDate() {
		return birthDate;
	}

	public void setBirthDate(Date birthDate) {
		this.birthDate = birthDate;
	}

	@Override
	public String toString() {
		return "Person [id=" + id + ", name=" + name + ", location=" +
location + ", birthDate=" + birthDate + "]";
	}

}
```

Step 15 - Implementing findById JPA Repository Method

Person.java

```
@Entity
public class Person {
…….•.…….•
}
```

PersonJpaRepository.java

```
// This is a repository. We will do all person related operations here.
@Repository

// we will do transaction management in here
// whenever we do insert, delete, update rows (or do database
transaction). If you do 3-4 transaction at a time, you would want all of
them to be successful or all of them to fail together.
// For now we will implement transaction at the level of repository.
Ideally transaction have to be implemented around your business services.
@Transactional
public class PersonJpaRepository {

//connect to the database
// Entity Manager manages the entities. All the operations that you are
performing inside a session are stored in PersistenceContext.
EntityManager is the interface to the PersistenceContext. All operations
have to be going through EntityManager.
      @PersistenceContext
      EntityManager entityManager;

      public Person findById(int id) {
            return entityManager.find(Person.class, id); // JPA
      }
}
```

Note: SpringBoot auto configuration knows that we are using in-memory database, it knows that JPA is in the classpath. It knows that I am defining entities as well. What it does is that it triggers a schema update (one of the hibernate features). It creates schema for us. So from now on we do not need to create table bcz the table would be created by schema update.

How to see automatically generated query?

```
spring.jpa.show-sql=true
```

Console O/P:

```
Hibernate: select person0_.id as id1_0_0_, person0_.birth_date as
birth_da2_0_0_, person0_.location as location3_0_0_, person0_.name as
name4_0_0_ from person person0_ where person0_.id=?
2018-09-06 07:53:13.547  INFO 2992 --- [main]
ication$$EnhancerBySpringCGLIB$$c54dca3e : User id 10001 -> Person
[id=10001, name=Ranga, location=Hyderabad, birthDate=2018-09-06
07:53:12.766]
```

Step 16 - Implementing insert and update JPA Repository Methods

entityManager.merge(Entity entity) → In update/insert operations. Merge() method knows whether the id is set inside person or not. If id is already set it will update that person, if not, it will insert it in.

Step 17 - Implementing deleteById JPA Repository Method

```
public void deleteById(int id) {s
           Person person = findById(id);
           entityManager.remove(person);
      }
```

Step 18 - Implementing findAll using JPQL Named Query

Use JPQL (Java Persistence Query Language)

```
@Entity
// NamedQuery is defined on the entity for which it is used
// JPQL does not use database table to fetch data. It uses entities.
Query=”JPQL query”
@NamedQuery(name="find_all_persons", query="select p from Person p")
public class Person {
….
}
```

PersonJpaRepository.java

```
public List<Person> findAll() {
           // createNamedQuery will take; 1. Name of the Query 2. What
kind of entity it would return
           TypedQuery<Person> namedQuery =
entityManager.createNamedQuery("find_all_persons", Person.class);
           return namedQuery.getResultList();
      }
```

Step 01 - Create a JPA Project with H2 and Spring Boot

dependency added: H2, Web, JPA, DevTools

Step 02 - Create JPA Entity Course

```java
@Entity
public class Course {

    // To define primary key
    @Id
    @GeneratedValue // want JPA to generate id for us
    private Long id;

    private String name;
// Default constructor will be used by JPA to create bean
    protected Course() {
    }

    // We want others only to provide the name, not id
    public Course(String name) {
        this.name = name;
    }
//Getters and setters
}
```

If you run application at this point, ‘course’ table will be created in H2 database.

Step 03 - Create findById using JPA Entity Manager

Create CourseRepository.java. In any repository class we will try to talk to the EntityManager.

Step 04 - Configuring application.properties to enable H2 console and additional logging

application.properties

```properties
# Enabling H2 Console
spring.h2.console.enabled=true

#Turn Statistics on
# "generate_statistics" will tell no of queries fired
spring.jpa.properties.hibernate.generate_statistics=true
# keep logging level for "org.hibernate.stat" as trace
logging.level.org.hibernate.stat=trace

# Show all queries (automatically generated by hibernate)
spring.jpa.show-sql=true
# Format the queries
spring.jpa.properties.hibernate.format_sql=true
# What parameters are being set to the queries
logging.level.org.hibernate.type=trace
```

Step 05 - Writing Unit Test for findById method

CourseRepositoryTest.java

```
package com.personal.Kunj.jpa.advancedjpa.repository;

import static org.junit.Assert.assertEquals;
import org.junit.Test;
import org.junit.runner.RunWith;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.context.SpringBootTest;
import org.springframework.test.context.junit4.SpringRunner;
import com.personal.Kunj.jpa.advancedjpa.AdvancedJpaApplication;
import com.personal.Kunj.jpa.advancedjpa.entity.Course;

// Used to launch SpringContext in unit test
@RunWith(SpringRunner.class)
/*
 * The SpringContext that we would want to launch is a 'SpringBootTest'.
We
 * would want to launch entire SpringBootContext which is present
 * in'AdvancedJpaApplication'. (classes=AdvancedJpaApplication.class)
will
 * launch the entire context
 */
@SpringBootTest(classes = AdvancedJpaApplication.class)
public class CourseRepositoryTest {

      private Logger logger = LoggerFactory.getLogger(this.getClass());

      @Autowired
      CourseRepository repository;

      /*
       * If we do right click on contextLoads(), it will launch the
entire context, ie
      * it will do the same thing as what is done at the application run.
       */

      // After the above it will run the code present in @Test method
      // Then context is killed.
      // Unit test is run between the Context Launch and Destroy.
      @Test
      public void contextLoads() {
            logger.info("Test is running");
      }
      // These JUnit tests will run as part of your build

      @Test
	public void findById_basic() {
		Course course = repository.findById(10001L);
		assertEquals("JPA in 50 Steps", course.getName());

	}

}
```

Step 06 - Writing a deleteByID method to delete an Entity

Exception:

Caused by: org.springframework.dao.InvalidDataAccessApiUsageException: No EntityManager with actual transaction available for current thread - cannot reliably process 'remove' call; nested exception is javax.persistence.TransactionRequiredException: No EntityManager with actual transaction available for current thread - cannot reliably process 'remove' call

Fix: Annotate CourseRepository.java with @Transactional

Step 07 - Writing Unit Test for deleteById method

```
@Test

// This unit test is modifying the database. By using @DirtiesContext, spring will automatically reset the data after the test is run. Data will remain consistent even after this unit test execution.

@DirtiesContext

	public void deleteById_basic() {
		repository.deleteById(10002L);
		assertNull(repository.findById(10002L));
	}
```

Step 08 - Writing a save method to update and insert an Entity

Step 09 - Writing Unit Test for save method

```
	@Test

	@DirtiesContext

	public void save_basic() {

		// get a course
		Course course = repository.findById(10001L);
		assertEquals("JPA in 50 Steps", course.getName());

		// update details
		course.setName("JPA in 50 Steps - Updated");

		repository.save(course);

		// check the value
		Course course1 = repository.findById(10001L);
		assertEquals("JPA in 50 Steps - Updated", course1.getName());
```

```
      }
```

Step 10 - Quick Review and Debugging Tips

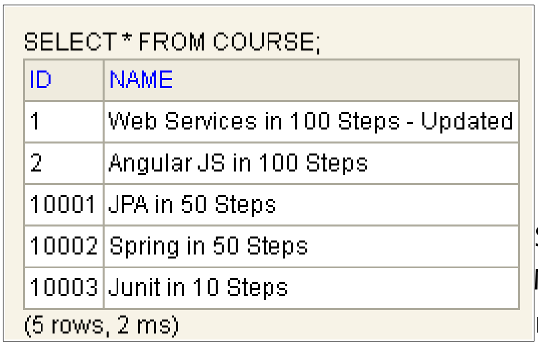
Step 11 - Playing with Entity Manager

```
@Repository
@Transactional
public class CourseRepository {

      @Autowired
      EntityManager em;
public void playWithEntityManager() {
           Course course = new Course("Web Services in 100 Steps");
           // persist() is used to create a new entity
           em.persist(course);
           /*
            * An update query is fired bcz of the below statement without
even asking for a
            * save. HOW?
            *
            * BCZ OF @Transactional annotation, this entire method is in
a
            * single transaction. And while we are within the scope of a
transaction,
            * EntityManager keeps track of all the things that were
updated/modified
            * through it. In this example Course is updated/inserted
through the
            * EntityManager. So changes made to the course are tracked by
the
            * EntityManager.
            */
           course.setName("Web Services in 100 Steps - Updated");
      }
}
```

Step 12 - Entity Manager Methods - clear and detach

```
// both the courses course1 and course2 will be updated.

public void playWithEntityManager() {
           Course course1 = new Course("Web Services in 100 Steps");
           em.persist(course1);

           course1.setName("Web Services in 100 Steps - Updated");

           Course course2 = new Course("Angular JS in 100 Steps");
           em.persist(course2);

           course2.setName("Angular Js in 100 Steps - Updated");

      }
public void playWithEntityManager() {
```

```
          Course course1 = new Course("Web Services in 100 Steps");
          em.persist(course1);
          // The changes done until that point is sent out to the
database
          em.flush();

          course1.setName("Web Services in 100 Steps - Updated");
          em.flush();

          Course course2 = new Course("Angular JS in 100 Steps");
          em.persist(course2);
          em.flush();

          // Let's say I do not want course2 changes to be going to the
database after this step.
          // The changes to course2 will no longer be tracked after this
stage
          //em.detach(course2);

          // The other way of detaching the entity is by clearing
everything up. This will clear everything that is there in the
EntityManager.
          em.clear();

          course2.setName("Angular Js in 100 Steps - Updated");
          em.flush();
     }
```



SELECT * FROM COURSE;

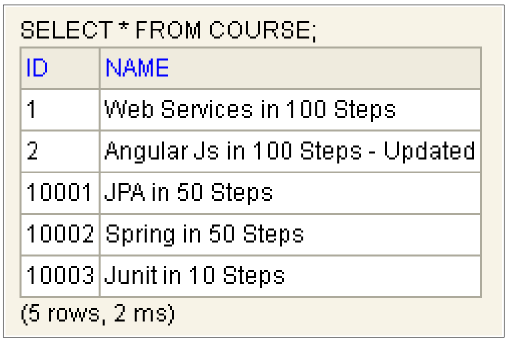
ID	NAME
1	Web Services in 100 Steps - Updated
2	Angular JS in 100 Steps
10001	JPA in 50 Steps
10002	Spring in 50 Steps
10003	Junit in 10 Steps

(5 rows, 2 ms)

Step 13 - Entity Manager Methods – refresh

```
public void playWithEntityManager() {

          Course course1 = new Course("Web Services in 100 Steps");
          em.persist(course1);
          Course course2 = new Course("Angular JS in 100 Steps");
          em.persist(course2);

          em.flush();

          course1.setName("Web Services in 100 Steps - Updated");
          course2.setName("Angular Js in 100 Steps - Updated");

          /*
           * Refresh course1 with the content that is there in the
database. Take
```

```
             * course1 details as it is in the database. I do not want the updated data of
             * course1 to go through.
             */
            em.refresh(course1);;

            // Only course2 changes will go the database
            em.flush();

      }
```



SELECT * FROM COURSE;

ID	NAME
1	Web Services in 100 Steps
2	Angular Js in 100 Steps - Updated
10001	JPA in 50 Steps
10002	Spring in 50 Steps
10003	Junit in 10 Steps

(5 rows, 2 ms)

Step 14 - A Quick Review of Entity Manager

EntityManager is an interface to PersistenceContext.

All the entities that are saved through EntityManager are saved through PersistenceContext.

PersistenceContext keeps track of all the entities which are changes during a specific transaction and also keeps track of the changes that neds to be stored back to the database.

Step 15 - JPQL – Basics

JPQL → Java Persistence Query language

In SQL we query from the database table, whereas in JPQL we query from entities. Whatever JPQL query we write are converted into SQL query by JPA implementation.

SQL → Select * from course

JPQL → select c from course c

```
package com.personal.Kunj.jpa.advancedjpa.repository;
import java.util.List;

import javax.persistence.EntityManager;
import javax.persistence.Query;
import javax.persistence.TypedQuery;

import org.junit.Test;
import org.junit.runner.RunWith;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.context.SpringBootTest;
import org.springframework.test.context.junit4.SpringRunner;

import com.personal.Kunj.jpa.advancedjpa.AdvancedJpaApplication;
import com.personal.Kunj.jpa.advancedjpa.entity.Course;

@RunWith(SpringRunner.class)
@SpringBootTest(classes = AdvancedJpaApplication.class)
public class JPQLTest {

     private Logger logger = LoggerFactory.getLogger(this.getClass());

     @Autowired
     EntityManager em;

     @Test
     public void jpql_basic() {
           Query query = em.createQuery("Select  c  From Course c");
           List resultList = query.getResultList();
           logger.info("Select  c  From Course c -> {}",resultList);
     }

     @Test
     public void jpql_typed() {

           // Returning a Course back. typed queries are always better as
they make your program clear
           TypedQuery<Course> query =
                           em.createQuery("Select  c  From Course c",
Course.class);

           List<Course> resultList = query.getResultList();
           logger.info("Select  c  From Course c -> {}",resultList);
     }

     @Test
     public void jpql_where() {
           TypedQuery<Course> query =
                           em.createQuery("Select  c  From Course c
where name like '%100 Steps'", Course.class);

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

           logger.info("Select  c  From Course c where name like '%100
Steps'-> {}",resultList);
           //[Course[Web Services in 100 Steps], Course[Spring Boot in
100 Steps]]
     }

}
```

Step 16 - JPA and Hibernate Annotations - @Table

```
@Entity
// To define the name of the table
@Table(name="CourseDetails")
public class Course {
..........…........….…
}
```

Step 17 - JPA and Hibernate Annotations - @Column

```
// Set all the constraints here that you have in your data in dB to
prevent bad data from entering into DB
	@Column(name="fullname", nullable=false)
	private String name;
```

Step 18 - JPA and Hibernate Annotations - @UpdateTimestamp and @CreationTimestamp

Sometime some of the applications have the requirement that I want to store when this specific row is updated/inserted last time.

I would like to store created time and updated time of a particular row.

This is not a solution provided by JPA. It is provided by hibernate.

Course.java

```
@Entity
// To define the name of the table
@Table(name="CourseDetails")
public class Course {

	// To define primary key
	@Id
	@GeneratedValue // want JPA to generate for us
	private Long id;

	// Set all the constraints here that you have in your data in dB to
prevent bad data from entering into DB
	@Column(name="fullname", nullable=false)
	private String name;

	@UpdateTimestamp
	private LocalDateTime lastUpdatedDate;

	@CreationTimestamp
	private LocalDateTime createdDate;
// Getters and setters, contructors
}
```

Step 19 - JPA and Hibernate Annotations - @NamedQuery and @NamedQueries

JPQLTest.java

```
package com.personal.Kunj.jpa.advancedjpa.repository;
import java.util.List;

import javax.persistence.EntityManager;
import javax.persistence.Query;
import javax.persistence.TypedQuery;

import org.junit.Test;
```

```java
import org.junit.runner.RunWith;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.context.SpringBootTest;
import org.springframework.test.context.junit4.SpringRunner;

import com.personal.Kunj.jpa.advancedjpa.AdvancedJpaApplication;
import com.personal.Kunj.jpa.advancedjpa.entity.Course;

@RunWith(SpringRunner.class)
@SpringBootTest(classes = AdvancedJpaApplication.class)
public class JPQLTest {

    private Logger logger = LoggerFactory.getLogger(this.getClass());

    @Autowired
    EntityManager em;

    @Test
    public void jpql_basic() {
        Query query = em.createQuery("Select  c  From Course c");
        List resultList = query.getResultList();
        logger.info("Select  c  From Course c -> {}",resultList);
    }

    @Test
    public void jpql_typed() {

        /*
         * Here we are hard coding the query. If we need to use the
same query again,
         * we need to write the query again. @NamedQuery can help us
here where we can give
         * name to a query.
         *
         * @NamedQuery is always defined on the entity class to which
it is directed.
         */
        //TypedQuery<Course> query = em.createQuery("Select  c  From
Course c", Course.class);
        TypedQuery<Course> query =
em.createNamedQuery("query_get_all_courses", Course.class);

        List<Course> resultList = query.getResultList();
        logger.info("Select  c  From Course c -> {}",resultList);
    }

    @Test
    public void jpql_where() {
        TypedQuery<Course> query =
      em.createNamedQuery("query_get_100_step_courses", Course.class);

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

           logger.info("Select  c  From Course c where name like '%100
Steps'-> {}",resultList);
           //[Course[Web Services in 100 Steps], Course[Spring Boot in
100 Steps]]
      }

}
```

Course.java

```
@Entity
// To define the name of the table
@Table(name="CourseDetails")
// we can either use multiple @NamedQuery or @NamedQueries

@NamedQueries(value = { @NamedQuery(name = "query_get_all_courses", query
= "Select  c  From Course c"),
           @NamedQuery(name = "query_get_100_step_courses", query =
"Select  c  From Course c where name like '%100 Steps'") })

/*@NamedQuery(name="query_get_all_courses", query="Select  c  From Course
c")
@NamedQuery(name="query_get_100_step_courses", query="Select  c  From
Course c where name like '%100 Steps'")*/
public class Course {
….
}
```

Step 20 - Native Queries - Basics

Native Queries is sending native sql directly from JPA.

Situations where we have to go NativeQuery.

1) Setting tuning parameters

2) Using some DB specific features that are not supported by JPA

3) While doing a mass update. Ex; suppose we want to update all the wows of a table in a specific query. In this situation if you use JPA, you have to et the row and update and repeat. You cannot do a mass update using JPA.

Note: Whenever you are making use of Native Query, you are not making use of PersistenceContext. So, if you have all the entities directly present in your PersistenceContext, then you will have to make sure that you will refresh them so that you get the latest data from the database.

NativeQueriesTest.java

```
package com.personal.Kunj.jpa.advancedjpa.repository;
import java.util.List;
import javax.persistence.EntityManager;
import javax.persistence.Query;

import org.junit.Test;
import org.junit.runner.RunWith;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.context.SpringBootTest;
import org.springframework.test.context.junit4.SpringRunner;
import org.springframework.transaction.annotation.Transactional;

import com.personal.Kunj.jpa.advancedjpa.AdvancedJpaApplication;
import com.personal.Kunj.jpa.advancedjpa.entity.Course;

@RunWith(SpringRunner.class)
@SpringBootTest(classes = AdvancedJpaApplication.class)
public class NativeQueriesTest {

     private Logger logger = LoggerFactory.getLogger(this.getClass());

     @Autowired
     EntityManager em;

     @Test
     public void native_queries_basic() {
           Query query = em.createNativeQuery("SELECT * FROM COURSE",
Course.class);
           List resultList = query.getResultList();
           logger.info("SELECT * FROM COURSE  -> {}", resultList);
           //SELECT * FROM COURSE  -> [Course[Web Services in 100 Steps],
Course[JPA in 50 Steps - Updated], Course[Spring in 50 Steps],
Course[Spring Boot in 100 Steps]]
     }

     @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();
           logger.info("SELECT * FROM COURSE  where id = ? -> {}",
resultList);
           //[Course[JPA in 50 Steps - Updated]]
     }

     @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);
           //[Course[JPA in 50 Steps - Updated]]
      }

      // Mass update where Native SQL queries are handy
      @Test
      @Transactional // Bcz we are trying to change the data
      public void native_queries_to_update() {
           Query query = em.createNativeQuery("Update COURSE set 
last_updated_date=sysdate()");
           int noOfRowsUpdated = query.executeUpdate();
           logger.info("noOfRowsUpdated  -> {}", noOfRowsUpdated);
           //SELECT * FROM COURSE  -> [Course[Web Services in 100 Steps], 
Course[JPA in 50 Steps - Updated], Course[Spring in 50 Steps], 
Course[Spring Boot in 100 Steps]]
      }


}
```

Establishing Relationships with JPA and Hibernate – OneToOne

Step 21 - Entities and Relationships - An overview

Tables envolved;

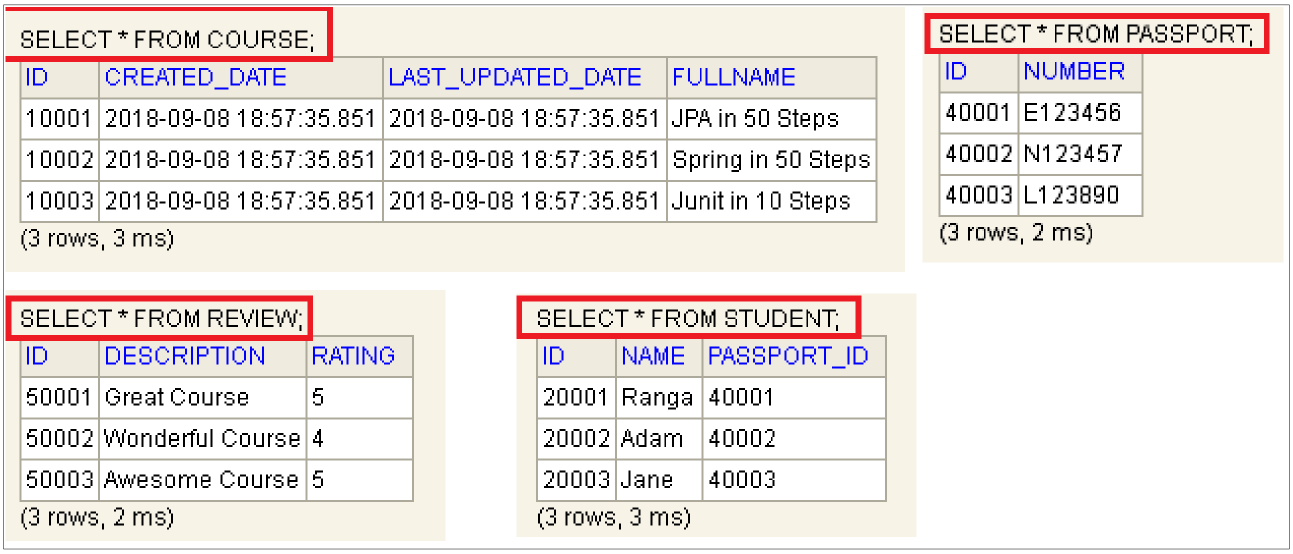
course, student, passport, review

A course can have multiple students enrolling for it. A student can enroll in multiple courses at a time. Course and student have many to many relationship.

A course can have multiple reviews from different students. On student giving a 5 star review, other giving a 4 star review. A review is always associated with a single course. The relationship between course and review is many to one.

A student can have only one passport and passport is associated with only one student. The relation between student and passport is one to one.

Step 22 - Defining Entities - Student, Passport and Review



SELECT * FROM COURSE;

ID	CREATED_DATE	LAST_UPDATED_DATE	FULLNAME
10001	2018-09-08 18:57:35.851	2018-09-08 18:57:35.851	JPA in 50 Steps
10002	2018-09-08 18:57:35.851	2018-09-08 18:57:35.851	Spring in 50 Steps
10003	2018-09-08 18:57:35.851	2018-09-08 18:57:35.851	Junit in 10 Steps

(3 rows, 3 ms)

SELECT * FROM PASSPORT;

ID	NUMBER
40001	E123456
40002	N123457
40003	L123890

(3 rows, 2 ms)

SELECT * FROM REVIEW;

ID	DESCRIPTION	RATING
50001	Great Course	5
50002	Wonderful Course	4
50003	Awesome Course	5

(3 rows, 2 ms)

SELECT * FROM STUDENT;

ID	NAME	PASSPORT_ID
20001	Ranga	40001
20002	Adam	40002
20003	Jane	40003

(3 rows, 3 ms)

Step 23 - Introduction to One to One Relationship

student – passport → one to one

We can either create a student_id column in passport table or create a passport_id column in student table. **Here student table is** owning the passport_id relationship. Similarly passport table is owning the student_Id relationship.

Now, define the above relationship in the code.

```
@Entity
public class Student {

	@Id
	@GeneratedValue
	private Long id;

	@Column(nullable = false)
	private String name;

	@OneToOne // one to one relationship from student to passport

	private Passport passport;
// …....…........….......…..
}
```

Step 24 - OneToOne Mapping - Insert Student with Passport

StudentRepository.java

```
package com.personal.Kunj.jpa.advancedjpa.repository;

import javax.persistence.EntityManager;

import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Repository;
import org.springframework.transaction.annotation.Transactional;

import com.personal.Kunj.jpa.advancedjpa.entity.Passport;
import com.personal.Kunj.jpa.advancedjpa.entity.Student;

@Repository
@Transactional
public class StudentRepository {

	private Logger logger = LoggerFactory.getLogger(this.getClass());

	@Autowired
	EntityManager em;

	public void saveStudentWithPassport() {
		Passport passport = new Passport("Z123456");
		/*
		 * passport has to be there in the database before you want to
create a
		 * relationship between student and passport.
		 */

		/*
		 * Caused by:
org.springframework.dao.InvalidDataAccessApiUsageException:
		 * org.hibernate.TransientPropertyValueException: object
references an unsaved
		 * transient instance - save the transient instance before
flushing :
		* com.personal.Kunj.jpa.advancedjpa.entity.Student.passport ->
```

```
            * com.personal.Kunj.jpa.advancedjpa.entity.Passport; nested
exception is
            * java.lang.IllegalStateException:
            * org.hibernate.TransientPropertyValueException: object
references an unsaved
            * transient instance - save the transient instance before
flushing :
           * com.personal.Kunj.jpa.advancedjpa.entity.Student.passport ->
            * com.personal.Kunj.jpa.advancedjpa.entity.Passport
            */

           /*
            * Before we are creating a student we have to create a
passport (create an id
            * for it) as student is at the owning side of the
relationship
            */

           /*
            * Note: Hibernate is Lazy! it will wait as long as it can
before inserting the
            * passport in.
            */

           // Here hibernate will just generate the next sequence. Query
will not go the DB
           em.persist(passport);

           Student student = new Student("Mike");

           student.setPassport(passport);
           em.persist(student);
           // At the end of the transaction, hibernate will send the
changes down to the database
      }

}
```

Step 25 - OneToOne Mapping - Retrieving Student with Passport and Eager Fetch

Any one to one relation is always eager fetch ie, if you have one to one relationship , the student details as well as passport details are retrieved.

```
package com.personal.Kunj.jpa.advancedjpa.repository;
import javax.persistence.EntityManager;

import org.junit.Test;
import org.junit.runner.RunWith;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.context.SpringBootTest;
import org.springframework.test.context.junit4.SpringRunner;
import org.springframework.transaction.annotation.Transactional;

import com.personal.Kunj.jpa.advancedjpa.AdvancedJpaApplication;
import com.personal.Kunj.jpa.advancedjpa.entity.Student;


@RunWith(SpringRunner.class)
@SpringBootTest(classes = AdvancedJpaApplication.class)
public class StudentRepositoryTest {

      private Logger logger = LoggerFactory.getLogger(this.getClass());

      @Autowired
      StudentRepository repository;

      @Autowired
      EntityManager em;

      @Test
      public void retrieveStudentAndPassportDetails() {
            // em.find() will fetch the student as well as passport (eager
fetch)
            Student student = em.find(Student.class, 20001L); // (1)
            logger.info("student -> {}", student);
            logger.info("passport -> {}",student.getPassport());
      }
}
```

Query fired in case of eager fetch **(1)**:

```
Hibernate:
    select
        student0_.id as id1_3_0_,
        student0_.name as name2_3_0_,
        student0_.passport_id as passport3_3_0_,
        passport1_.id as id1_1_1_,
        passport1_.number as number2_1_1_
    from
        student student0_
    left outer join
        passport passport1_
            on student0_.passport_id=passport1_.id
    where
        student0_.id=?
```

Step 26 - OneToOne Mapping - Lazy Fetch

In some cases, eager fetch can give you performance issues bcz I might want to only retrieve student, not passport. How to avoid eager fetch?

```
@Entity
public class Student {

      @Id
      @GeneratedValue
      private Long id;

      @Column(nullable = false)
      private String name;

      @OneToOne(fetch=FetchType.LAZY) // one to one relationship from
student to passport
      private Passport passport;

// ….…

}
```

StudentRepositoryTest.java

```
      @Test
      public void retrieveStudentAndPassportDetails() {

      Student student = em.find(Student.class, 20001L); // (2)
      logger.info("student -> {}", student);
      logger.info("passport -> {}",student.getPassport()); //(3)
      }
```

Query fired in case of lazy fetch **(2)**:

```
Hibernate:
    select
        student0_.id as id1_3_0_,
        student0_.name as name2_3_0_,
        student0_.passport_id as passport3_3_0_
    from
        student student0_
    where
        student0_.id=?
```

Statement (3) throws below exceptioin. Why?

```
org.hibernate.LazyInitializationException: could not initialize proxy
[com.personal.Kunj.jpa.advancedjpa.entity.Passport#40001] - no Session
```

Reason: By the time em.find() is run, the transaction ends right then and there. Transaction is needed for us to get the details of passport (student.getPassport()). We need to have a session. Bcz we are not having any transaction here, the session gets ended as soon as em.find() gets executed.

Solution: Put @Transactional annotation on retrieveStudentAndPassportDetails() method. Now the Hibernate session is only killed at the end of the test ie, at the end of the method.

StudentRepositoryTest.java

```
@Test
@Transactional
public void retrieveStudentAndPassportDetails() {

Student student = em.find(Student.class, 20001L); // (2)
logger.info("student -> {}", student);
logger.info("passport -> {}",student.getPassport()); //(3)
}
```

Query fired in case of lazy fetch **(statement (3))**:

```
 Hibernate:

    select
        passport0_.id as id1_1_0_,
        passport0_.number as number2_1_0_
    from
        passport passport0_
    where
        passport0_.id=?
```

Lazy Fetch → You get the details only when they are needed.

Step 27 - Transaction, Entity Manager and Persistence Context

StudentRepositoryTest.java

```
@Test
          @Transactional
          public void someTest() {
               //Database Operation 1 - Retrieve student
               Student student = em.find(Student.class, 20001L);


               //Database Operation 2 - Retrieve passport
               // As we already have student object, we are not using
EntityManager to het data.

               Passport passport = student.getPassport();

               //Database Operation 3 - update passport
               passport.setNumber("E123457");

               //Database Operation 4 - update student
               student.setName("Ranga - updated");
          }
```

Let us assume that student.setName("Ranga - updated") failed.Should the while thing succeed or fail? What should be the state of the database after the above transaction? Ideally when we are talking about a transaction, everything should succeed or nothing should succeed. So if student.setName("Ranga - updated") fails, all the changes done to the database before this should be rolled back. This is the reason we use the concept called @Transactional.

In JPA, whenever you define a @Transactional, you also create PersistenceContext. PersistenceContext is a place where all the entities that you are operating upon are being stored.

```
public void someTest() {
                //Database Operation 1 - Retrieve student
                Student student = em.find(Student.class, 20001L);
 // PersisitenceContext will store student after the above statement execution
                //Persistence Context (student)

                //Database Operation 2 - Retrieve passport
                Passport passport = student.getPassport();
// PersistenceContext will store student and passport after the above statement
execution
                //Persistence Context (student, passport)

                //Database Operation 3 - update passport
                passport.setNumber("E123457");
// Status of PersistenceContext
                //Persistence Context (student, passport++)

                //Database Operation 4 - update student
                student.setName("Ranga - updated");
// Status of PersistenceContext
                //Persistence Context (student++ , passport++)
// Only after the entire transaction is completed, the database changes are sent
out to the database
        }
```

The way we interact with PersistenceContext is by using EntityManager. Whenever we call a method on EntityManager, we are actually playing with PersistenceContext. PersistenceContext is created at the start of the transaction and killed as soon as the transaction is ended.

If there is no @Transactional at the method, each call will act as its own transaction. For ex, at the start of em.find(Student.class, 20001L) a transaction will be opened and at the end of the execution of the em.find() the transaction will be closed. So in this case student.getPassport() will generate an exception as there is no transaction/persistencecontext opened as this is not called by EntityManager.

In Hibernate terminology Session= Persistence Context.

So far we have @Transactional annotation on all the repository methods. What if we remove it from StudentrepositoryTest.java method.

StudentRepositoryTest.java

```
package com.personal.Kunj.jpa.advancedjpa.repository;
import javax.persistence.EntityManager;

import org.junit.Test;
import org.junit.runner.RunWith;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.context.SpringBootTest;
import org.springframework.test.context.junit4.SpringRunner;
import org.springframework.transaction.annotation.Transactional;

import com.personal.Kunj.jpa.advancedjpa.AdvancedJpaApplication;
import com.personal.Kunj.jpa.advancedjpa.entity.Passport;
import com.personal.Kunj.jpa.advancedjpa.entity.Student;


@RunWith(SpringRunner.class)
@SpringBootTest(classes = AdvancedJpaApplication.class)
public class StudentRepositoryTest {

     private Logger logger = LoggerFactory.getLogger(this.getClass());

     @Autowired
     StudentRepository repository;

     @Autowired
     EntityManager em;

           //Session & Session Factory
           //EntityManager & Persistence Context
           //Transaction

           @Test
           // Let us remove @Transactional and move the method body to a
new method in StudentRepository.java
     /*
      * After removing @Transactional, the transaction support will be
provided by
      * StudentRepository.java and all the queries in
      * someOperationToUnderstandPersistenceContext() will run fine.
      */
           //@Transactional
           public void someTest() {
                repository.someOperationToUnderstandPersistenceContext();
           }
```

```
    @Test
    /*
     * If @Transactional is not here, student.getPassport() will throw
below
     * exception in case of lazy fetch:
     * org.hibernate.LazyInitializationException: could not initialize
proxy
     * [com.personal.Kunj.jpa.advancedjpa.entity.Passport#40001] - no
Session
     */
    @Transactional
    public void retrieveStudentAndPassportDetails() {

        Student student = em.find(Student.class, 20001L);
        logger.info("student -> {}", student);
        logger.info("passport -> {}",student.getPassport());
    }
}
```

StudentRepository.java

```
package com.personal.Kunj.jpa.advancedjpa.repository;

import javax.persistence.EntityManager;

import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Repository;
import org.springframework.transaction.annotation.Transactional;

import com.personal.Kunj.jpa.advancedjpa.entity.Passport;
import com.personal.Kunj.jpa.advancedjpa.entity.Student;

@Repository
@Transactional
public class StudentRepository {

    private Logger logger = LoggerFactory.getLogger(this.getClass());

    @Autowired
    EntityManager em;

    public Student findById(Long id) {
        return em.find(Student.class, id);
    }

    public Student save(Student student) {

        if (student.getId() == null) {
            em.persist(student);
```

```
          } else {
               em.merge(student);
          }

          return student;
     }

     public void deleteById(Long id) {
          Student student = findById(id);
          em.remove(student);
     }

     public void saveStudentWithPassport() {
          Passport passport = new Passport("Z123456");
          /*
           * passport has to be there in the database before you want to
create a
           * relationship between student and passport.
           */

          /*
           * Caused by:
org.springframework.dao.InvalidDataAccessApiUsageException:
           * org.hibernate.TransientPropertyValueException: object
references an unsaved
           * transient instance - save the transient instance before
flushing :
          * com.personal.Kunj.jpa.advancedjpa.entity.Student.passport ->
           * com.personal.Kunj.jpa.advancedjpa.entity.Passport; nested
exception is
           * java.lang.IllegalStateException:
           * org.hibernate.TransientPropertyValueException: object
references an unsaved
           * transient instance - save the transient instance before
flushing :
          * com.personal.Kunj.jpa.advancedjpa.entity.Student.passport ->
           * com.personal.Kunj.jpa.advancedjpa.entity.Passport
           */

          /*
           * Before we are creating a student we have to create a
passport (create an id
           * for it) as student is at the owning side of the
relationship
           */

          /*
           * Note: Hibernate is Lazy! it will wait as long as it can
before inserting the
           * passport in.
           */
```

```
        // Here hibernate will just generate the next sequence. Query
will not go the DB
        em.persist(passport);

        Student student = new Student("Mike");

        student.setPassport(passport);
        em.persist(student);
        // At the end of the transaction, hibernate will send the
changes down to the database
    }

    public void someOperationToUnderstandPersistenceContext() {
        // Database Operation 1 - Retrieve student
        Student student = em.find(Student.class, 20001L);
        // Persistence Context (student)

        // Database Operation 2 - Retrieve passport
        Passport passport = student.getPassport();
        // Persistence Context (student, passport)

        // Database Operation 3 - update passport
        passport.setNumber("E123457");
        // Persistence Context (student, passport++)

        // Database Operation 4 - update student
        student.setName("Ranga - updated");
        // Persistence Context (student++ , passport++)
    }

}
```

PersistenceContext acts as 2 things:

1) It acts as store for different entities that are being managed. All the changes done to entities are tracked by PersistenceContext.

2) PersistenceContext also gives you access to the database. If you do student.getPassport(), PersistenceContext ensures that the queries are fires to the DB.

Hibernare uses Session and SessionFactory. If you want to use JPA you do not need to worry about Session and SessionFactory.

Step 28 - OneToOne Mapping - Bidirectional Relationship - Part 1

Get the passport using EntityManager and get the details of the student the passport is associated with. Previously we went to passport from student, now we wan to achieve the opposite.

```
@Entity
```

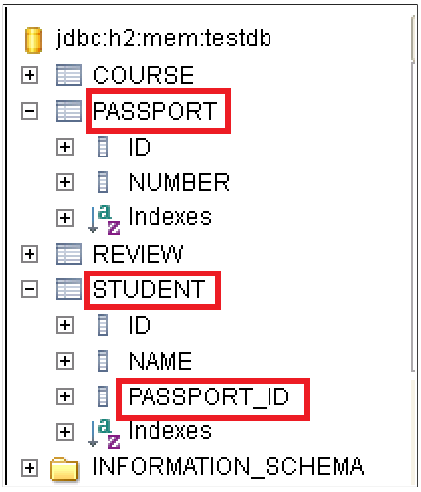
```
public class Passport {

        @Id
        @GeneratedValue
        private Long id;

        @Column(nullable = false)
        private String number;

        @OneToOne(fetch=FetchType.LAZY)
        private Student student;
// ..
}

@Entity
public class Student {

        @Id
        @GeneratedValue
        private Long id;

        @Column(nullable = false)
        private String name;

        @OneToOne(fetch=FetchType.LAZY)
        private Passport passport;
// …
}
```

Note: The two @OneToOne mapping will create data duplication in the DB.

Solution: Make one of these (Student or Passport) above 2 entities the owning side of the relationship, ie, either student should have the passport_id or passport should have the student_id. If student has the passport_id, then student is the owning side of the relationship.
For now make Student owning side of the relationship.The way we can do this by adding a mappedBY=”What is the name of the variable” to the non-owning side of the relationship.

```
@Entity
public class Passport {

        @Id
        @GeneratedValue
        private Long id;

        @Column(nullable = false)
        private String number;

        @OneToOne(fetch=FetchType.LAZY, mappedBy="passport")
        private Student student;
// ..
}
```

You will see that the Student table has a passport_id column but Passport table does not have a student_id column.



jdbc:h2:mem:testdb
- COURSE
- PASSPORT
  - ID
  - NUMBER
  - Indexes
- REVIEW
- STUDENT
  - ID
  - NAME
  - PASSPORT_ID
  - Indexes
- INFORMATION_SCHEMA

Step 29 - OneToOne Mapping - Bidirectional Relationship - Part 2

To get the details of a student associated with psspport.

Passport.java

```
@Entity
public class Passport {

	@Id
	@GeneratedValue
	private Long id;

	@Column(nullable = false)
	private String number;

	/*
	 * Even though there is no studnt_id column associated with passport, this gives
	 * us a way to navigate from passport to student. This is Bidirectional
	 * relationship
	 */
	@OneToOne(fetch=FetchType.LAZY, mappedBy="passport")
	private Student student;
//..….
}
```

StudentRepositoryTest.java

```
@Test
	@Transactional
	public void retrievePassportAndAssociatedStudent() {
		Passport passport = em.find(Passport.class, 40001L);
		logger.info("passport -> {}", passport);
		logger.info("student -> {}", passport.getStudent());
	}
```

7. FAQ’s about Hibernate and JPA

FAQ 1) When does Hibernate send updates to the database?

```
@Transactional
void someMethodWithChange() {
// The transaction starts here
// create objects

// Only queries for sequence generation (ids) get fired here. Only  id’s are assigned to the users
em.persist(user1);
em.persist(user2);
// em.flush(); → All the above transaction gets sent to the DB

// Persistence context recognizes the changes made to users. It does not even fire update queries here
// change user1 → If this fails, even though we have done em.flush(), Hibernate will rollback entire changes done before this point.
// change user2

// Persistence context keeps track of all the above changes
// Hibernate waits until the last possible moment before it would start persisting the changes. (Bcz if something fails, since all the changes are part of one transaction, it can be rolled back)
} // The transaction ends here. All changes are saved to DB.
```

Note: If you want to send some changes to DB in the meanwhile, you can use em.flush().

FAQ 2) When do we need @Transactional in an unit test

Whenever we are making a change to DB, we need **@Transactional.** Whenever we put **@Transactional on a method.** The entire logic in that method and the method it calls are within the boundaries of transaction.

If from a unit test we are calling a repository and the repository makes use of EntityMAnager, then we do not have to annotate the method in unit test with @Transactional. If we are directly making changes from unit test (directly using EntityManager to insert/update row), we need to add @Transactional on the unit test method.

If you want to do any change to the DB, you need a transaction.

FAQ 3) Do read only method need a transaction

Let us take 2 entities: User, Comment. User is posting lot of comments. A single comment is always associated with one user.

```
// Assume there is no @Transactional here
List<Comment> someReadOnlyMethod() {
```

```
// This line executes fine. In EntityManager there is a default
transaction that you will make use of.
User user = em.find(User.class, 1); // This transaction will end as soon
as this method is executed.
// This line will throw exception.As we are not making use of
EntityManager here, there is nothing to provide a transaction here.
Suppose comments are Lazily loaded. So we need to fire a query to DB to
get comments. If we need to fire a query to DB, we need a connection to
DB, we need a transaction.
List<Comment> comments  = user.getComments();
return comments;
}
```

FAQ 4) Why do we use @DirtiesContext in an unit test
To roll back the changes made to the DB by unit tests.

8. Establishing relationships with JPA and hibernate – OneToMAny and ManyToMany

Step 30 - ManyToOne Mapping - Designing the database
Course, Review
Each course can have multiple or no review present.
A review can only be associated with one course.
The best possible design for ManyToOne relationship is to go to the one side (Review : one review is associated with one course). We will have a course_id column in the review table to store the relationship between course and review.
Step 30 - Part 2 - ManyToOne Mapping - Implementing the Mapping
Review will have the course_id. Review is the owning side of the relationship. MappedBy will be on the non-owning side (Course) of the relationship.
Change data.sql to accommodate the new column in review table.
Note: There is no change in the course table but review table has an additional column ‘ course_id’. course_id is a foreign key on table course.
Course.java

```
package com.personal.Kunj.jpa.advancedjpa.entity;
import java.time.LocalDateTime;
import java.util.List;

import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.Id;
import javax.persistence.OneToMany;

import org.hibernate.annotations.CreationTimestamp;
import org.hibernate.annotations.UpdateTimestamp;

@Entity
public class Course {

      @Id
      @GeneratedValue
      private Long id;

      @Column(name="fullname", nullable=false)
      private String name;

      @OneToMany(mappedBy="course")
      private List<Review> reviews;

      public List<Review> getReviews() {
            return reviews;
      }

      /*
       * I do not want others to set reviews. I want others to add one
review at a
       * time.I do not want somebody to take the list of the revies and
manipulate it
       * and then give me the complete list.
       */

      public void addReview(Review review) {
            this.reviews.add(review);
      }

      public void removeReview(Review review) {
            this.reviews.remove(review);
      }

      @UpdateTimestamp
      private LocalDateTime lastUpdatedDate;

      @CreationTimestamp
      private LocalDateTime createdDate;


      // Default constructor will be used by JPA to create bean
      protected Course() {
      }


      // WE want others only to provide the name , not id
```

```java
    public Course(String name) {
        this.name = name;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public Long getId() {
        return id;
    }
    // Override toString() to get rid of hashcode being printed
    @Override
    public String toString() {
        return "Course [id=" + id + ", name=" + name + "]";
    }
}
package com.personal.Kunj.jpa.advancedjpa.entity;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.Id;
import javax.persistence.ManyToOne;
```

Review.java

```java
@Entity
public class Review {

    @Id
    @GeneratedValue
    private Long id;

    private String rating;

    private String description;

    @ManyToOne
    private Course course;

    public Course getCourse() {
        return course;
    }

    public void setCourse(Course course) {
        this.course = course;
    }

    protected Review() {
      }

      public Review(String rating, String description) {
           this.rating = rating;
           this.description = description;
      }

      public String getDescription() {
           return description;
      }

      public void setDescription(String description) {
           this.description = description;
      }

      public String getRating() {
           return rating;
      }

      public void setRating(String rating) {
           this.rating = rating;
      }

      public Long getId() {
           return id;
      }

      @Override
      public String toString() {
           return String.format("Review[%s %s]", rating, description);
      }

}
```

```
Hibernate:

    create table course (
       id bigint not null,
        created_date timestamp,
        last_updated_date timestamp,
        fullname varchar(255) not null,
        primary key (id)
    )
Hibernate: s
    create table review (
       id bigint not null,
        description varchar(255),
        rating varchar(255),
        course_id bigint,
        primary key (id)
    )
```

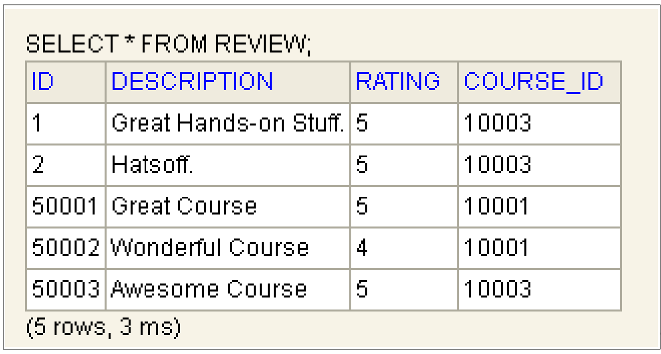
```
Hibernate: 
    alter table review 
       add constraint FKprox8elgnr8u5wrq1983degk 
       foreign key (course_id) 
       references course
```

Step 31 - ManyToOne Mapping - Retrieving and inserting Reviews for Course

We are getting a couple of reviews for the course ‘[redacted]’ and we would like to insert them in.

CourseRepository.java

```
public void addReviewsForCourse() {
		//get the course 10003
		Course course = findById(10003L);
		logger.info("course.getReviews() -> {}", course.getReviews());
		
		//add 2 reviews to it
		Review review1 = new Review("5", "Great Hands-on Stuff.");
		Review review2 = new Review("5", "Hatsoff.");
		
		//setting the relationship
		course.addReview(review1);
		review1.setCourse(course);
		
		course.addReview(review2);
		review2.setCourse(course);
		
		//save it to the database. Mind here that we are only 
persisting Review, not course
		em.persist(review1);
		em.persist(review2);
	}
```



SELECT * FROM REVIEW;

ID	DESCRIPTION	RATING	COURSE_ID
1	Great Hands-on Stuff.	5	10003
2	Hatsoff.	5	10003
50001	Great Course	5	10001
50002	Wonderful Course	4	10001
50003	Awesome Course	5	10003

(5 rows, 3 ms)

Step 32 - ManyToOneMapping - Generalizing

Insert Reviews

CourseRepository.java

```
public void addReviewsForCourse(Long courseId, List<Review> reviews) {
		
		Course course = findById(courseId);
		logger.info("course.getReviews() -> {}", course.getReviews());
		for(Review review:reviews)
		{			
			//setting the relationship
```

```
            course.addReview(review);
            review.setCourse(course);
            // Only persisting the review, not the course
            em.persist(review);
        }
    }
```

Step 33 - ManyToOne Mapping - Wrapping up

NOTE: ***ToOne → Default is eager fetching
***ToMany → Default is lazy fetching

CourseRepositoryTest.java

```
// Test for @OneToMany side of the relationship
    @Test
    @Transactional // In the below method by default Lazy fetch will take place
    // by default on @OneToMany side of the relationship, fetch strategy is Lazy
    // In every @OneToMany side of the relationship, you have to decide which type of fetching you want to go for

    public void retrieveReviewsForCourse() {
        Course course = repository.findById(10001L);
        // The below statement will throw "exception" if we do not have @Transactional
        logger.info("{}",course.getReviews());
    }
    // Test for @ManyToOne side of the relationship
    @Test
    @Transactional

    public void retrieveCourseForReview() {
        // On the @ManyToOne side of the relationship The fetching is always EAGER
        Review review = em.find(Review.class, 50001L);
        logger.info("{}",review.getCourse());
    }
```

Step 34 - ManyToMany Mapping - Table Design

Entities: Course, Student

A course can have multiple student. A student can enroll into multiple courses.

Creating student_id in course and course_id in student will have multiple entries which is not a good design. This is where concept of JoinTable comes into picture. We will create a join table (course_student or student_course) and have columns student_id and course_id stored.

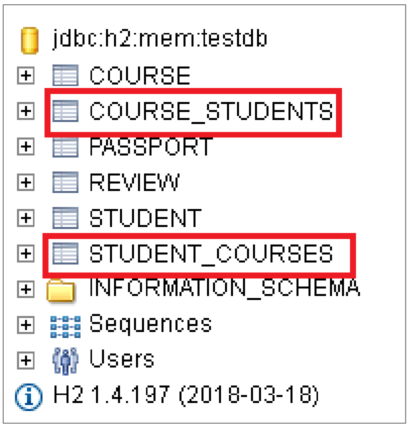
ManyToMany should always be established using a join table.

Step 35 - ManyToMany Mapping - Adding Annotations on Entities

```
package com.personal.Kunj.jpa.advancedjpa.entity;
import java.time.LocalDateTime;
import java.util.ArrayList;
import java.util.List;

import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.Id;
import javax.persistence.ManyToMany;
import javax.persistence.OneToMany;

import org.hibernate.annotations.CreationTimestamp;
import org.hibernate.annotations.UpdateTimestamp;

@Entity
public class Course {

    @Id
    @GeneratedValue
    private Long id;

    @Column(name="fullname", nullable=false)
    private String name;

    @UpdateTimestamp
    private LocalDateTime lastUpdatedDate;

    @CreationTimestamp
    private LocalDateTime createdDate;

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

    @ManyToMany
    private List<Student> students = new ArrayList<>();

    // Default constructor will be used by JPA to create bean
        protected Course() {
        }

        // We want others only to provide the name , not id
        public Course(String name) {
            this.name = name;
        }

    public List<Review> getReviews() {
        return reviews;
    }

    /*
     * I do not want others to set reviews. I want others to add one
review at a
     * time.I do not want somebody to take the list of the revies and
manipulate it
     * and then give me the complete list.
     */
    public void addReview(Review review) {
        this.reviews.add(review);
    }

    public void removeReview(Review review) {
        this.reviews.remove(review);
    }
    public List<Student> getStudents() {
        return students;
    }

    public void addStudent(Student student) {
        this.students.add(student);
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public Long getId() {
        return id;
    }
    // Override toString() to get rid of hashcode being printed
    @Override
    public String toString() {
        return "Course [id=" + id + ", name=" + name + "]";
    }
}

package com.personal.Kunj.jpa.advancedjpa.entity;
import java.util.ArrayList;
import java.util.List;

import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.FetchType;
import javax.persistence.GeneratedValue;
import javax.persistence.Id;
import javax.persistence.ManyToMany;
import javax.persistence.OneToOne;

@Entity
public class Student {

    @Id
    @GeneratedValue
```

```
	private Long id;
	@Column(nullable = false)
	private String name;
	@OneToOne(fetch=FetchType.LAZY)
	private Passport passport;
	@ManyToMany
	private List<Course> courses = new ArrayList<>();
	protected Student() {
	}
	public List<Course> getCourses() {
		return courses;
	}
	public void addCourse(Course course) {
		this.courses.add(course);
	}
	public Passport getPassport() {
		return passport;
	}
	public void setPassport(Passport passport) {
		this.passport = passport;
	}
	public Student(String name) {
		this.name = name;
	}
	public String getName() {
		return name;
	}
	public void setName(String name) {
		this.name = name;
	}
	public Long getId() {
		return id;
	}
	@Override
	public String toString() {
		return String.format("Student[%s]", name);
	}
}
```

DB:



Note: In the above screenshot, a separate relationship table is created for both the tables. 2 tables with the same design are created to establish a single relationship.

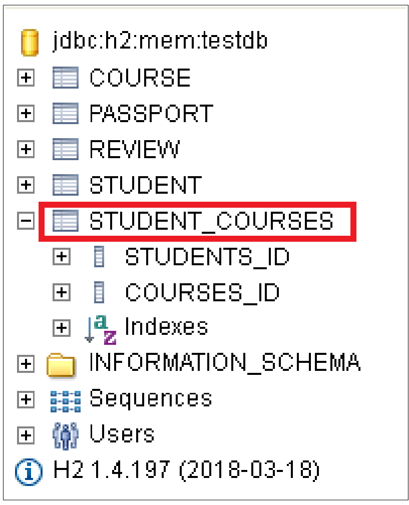
Step 36 - ManyToMany Mapping - Fixing two join tables problem

Solution of the step 35 : Make one of the entities the owning side of the relationship. In ManyToMany it does not matter which entity is a t the owning side of the relationship unlike @ManyToOne and @OneToMany.

Let us make student owning side of the relationship.

```
@Entity
public class Course {

      @ManyToMany(mappedBy="courses")
      private List<Student> students = new ArrayList<>();
// ….
}
@Entity
public class Student {

      @ManyToMany
      private List<Course> courses = new ArrayList<>();
//.....…......…
}
```

DB:



jdbc:h2:mem:testdb
⊞ COURSE
⊞ PASSPORT
⊞ REVIEW
⊞ STUDENT
⊟ STUDENT_COURSES
  ⊞ STUDENTS_ID
  ⊞ COURSES_ID
  ⊞ Indexes
⊞ INFORMATION_SCHEMA
⊞ Sequences
⊞ Users
ⓘ H2 1.4.197 (2018-03-18)

How to change the join table and it’s column’s name?

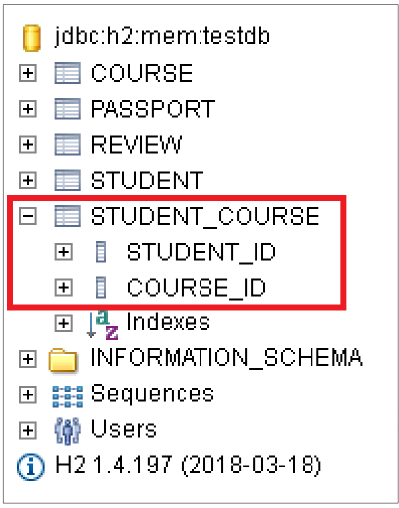
Step 37 - ManyToMany Mapping - Customizing the Join Table

At the owning side of the relationship (Student) we can add an annotation @JoinTable.

Student.java

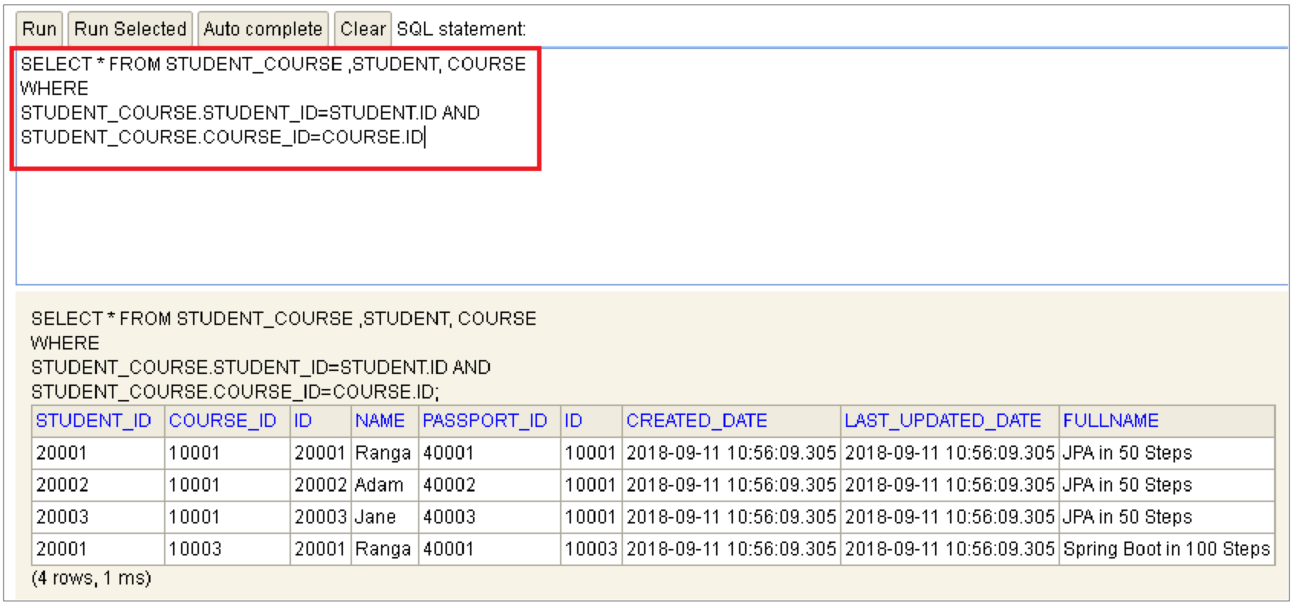
```
@ManyToMany
      // JoinColumn for this entity is Student_id.
      // InverseJoinColumn is course_id
      @JoinTable(name = "STUDENT_COURSE",
      joinColumns = @JoinColumn(name = "STUDENT_ID"),
      inverseJoinColumns = @JoinColumn(name = "COURSE_ID")
      )
```

DB:



jdbc:h2:mem:testdb
COURSE
PASSPORT
REVIEW
STUDENT
STUDENT_COURSE
STUDENT_ID
COURSE_ID
Indexes
INFORMATION_SCHEMA
Sequences
Users
H2 1.4.197 (2018-03-18)

Step 38 - ManyToMany Mapping - Insert Data and Write Join Query
Populate Student_course table.



```
SELECT * FROM STUDENT_COURSE ,STUDENT, COURSE
WHERE
STUDENT_COURSE.STUDENT_ID=STUDENT.ID AND
STUDENT_COURSE.COURSE_ID=COURSE.ID
```

```
SELECT * FROM STUDENT_COURSE ,STUDENT, COURSE
WHERE
STUDENT_COURSE.STUDENT_ID=STUDENT.ID AND
STUDENT_COURSE.COURSE_ID=COURSE.ID;
```

STUDENT_ID	COURSE_ID	ID	NAME	PASSPORT_ID	ID	CREATED_DATE	LAST_UPDATED_DATE	FULLNAME
20001	10001	20001	Ranga	40001	10001	2018-09-11 10:56:09.305	2018-09-11 10:56:09.305	JPA in 50 Steps
20002	10001	20002	Adam	40002	10001	2018-09-11 10:56:09.305	2018-09-11 10:56:09.305	JPA in 50 Steps
20003	10001	20003	Jane	40003	10001	2018-09-11 10:56:09.305	2018-09-11 10:56:09.305	JPA in 50 Steps
20001	10003	20001	Ranga	40001	10003	2018-09-11 10:56:09.305	2018-09-11 10:56:09.305	Spring Boot in 100 Steps

(4 rows, 1 ms)

How to get the above data without using query?
Step 39 - ManyToMany Mapping - Retrieve Data using JPA Relationships
StudentRepositoryTest.java

```
@Test
	@Transactional
	public void retrieveStudentAndCourses() {
		// Executes only the student query (Lazy fetch)
		// ManyToMany by default uses lazy fetch
		Student student = em.find(Student.class, 20001L);

		logger.info("student -> {}", student);
		// It will run join query on student_course and course table
		logger.info("courses -> {}", student.getCourses());
	}
```

Step 40 - ManyToMany Mapping - Insert Student and Course
Inserting Student, course and their relationship
StudentRepository.java

```
public void insertHardcodedStudentAndCourse(){
		Student student = new Student("Jack");
		Course course = new Course("Microservices in 100 Steps");
```

```
        em.persist(student);
        em.persist(course);

        // Persisting the relationship between student and course
        student.addCourse(course);
        course.addStudent(student);
        em.persist(student); // Persisting the owning side
    }

    public void insertStudentAndCourse(Student student, Course course){

        student.addCourse(course);
        course.addStudent(student);

        em.persist(student);
        em.persist(course);
    }
```

Step 41 - Relationships between JPA Entities - A summary

Use fetch type judiciously.

Always think of the fetch type whrn you are thinking about the relationships.

Designing table in case of various (OneToOne, ManyToOne etc) relations

9. Inheritance Hierarchies with JPA and Hibernate

Step 42 - Introduction to Inheritance Hierarchies and Mappings

In OOPS inheritance is an imp concept.

PartTimeEmployee → inheriting from → FullTimeEmployee → inheriting from → Employee

The Employee class has all the common attributes. Attribute which are specific to FullTimeEmployee is salary and for PartTimeEmployee hourlyWage. How do we map this relation to DB table?

Step 43 - JPA Inheritance Hierarchies and Mappings - Setting up entities

```
@Entity
public abstract class Employee {
     @Id
     @GeneratedValue
     private Long id;

     @Column(nullable = false)
     private String name;
// ….
}
@Entity
public class FullTimeEmployee extends Employee {

     private BigDecimal salary;
//..…….......…..…..
}
@Entity
public class PartTimeEmployee extends Employee {

     private BigDecimal hourlyWage;
// ........…
}
```

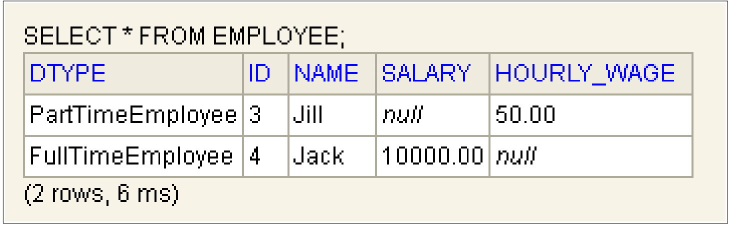
Step 44 - JPA Inheritance Hierarchies and Mappings - Setting up a Repository

```
@Repository
@Transactional
public class EmployeeRepository {

     private Logger logger = LoggerFactory.getLogger(this.getClass());

     @Autowired
     EntityManager em;

     public void insert(Employee employee) {
          em.persist(employee);
     }

     public List<Employee> retrieveAllEmployees() {
          return em.createQuery("select e from Employee e",
Employee.class).getResultList();
     }
}
```

Step 45 - JPA Inheritance Hierarchies and Mappings - Single Table

We would like to store FullTimeEmployee and PartTimeEmployee as different tables.

Single table strategy is the default strategy.



```
SELECT * FROM EMPLOYEE;
```

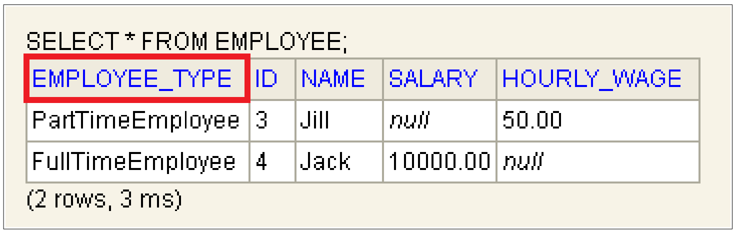
DTYPE	ID	NAME	SALARY	HOURLY_WAGE
PartTimeEmployee	3	Jill	*null*	50.00
FullTimeEmployee	4	Jack	10000.00	*null*

(2 rows, 6 ms)

Both the

FullTimeEmployee and PartTimeEmployee are stored in single table.
From the performance perspective this is good bcz we are quering from a single table. But from data integrity perspective, we have to define both the columns (salary, hourly_wage) as nullable. If there is a defect in code, there is a chance that an invalid data can come in. Somebody enters null in hourly_wage for PartTimeEmployee, that is a bad data. My database is allowing bad data.
The problem with single table hierarchy is that you will have lot of nullable columns.
In the Employee table there is an additional coulumn DTYPE that will say what kind of employee is being stored.
You can even give a name to DTYPE column by @DiscriminatorColumn.

```
@Entity
@Inheritance(strategy=InheritanceType.SINGLE_TABLE)
@DiscriminatorColumn(name="EmployeeType")
public abstract class Employee {

// ……..…..……..……..……..
}
```

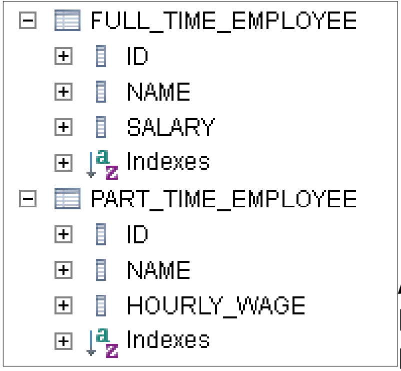


SELECT * FROM EMPLOYEE;

EMPLOYEE_TYPE	ID	NAME	SALARY	HOURLY_WAGE
PartTimeEmployee	3	Jill	null	50.00
FullTimeEmployee	4	Jack	10000.00	null

(2 rows, 3 ms)

Step 46 - JPA Inheritance Hierarchies and Mappings - Table Per Class
TABLE_PER_CLASS → A table per concrete entity class
Employee is an abstract class. FullTimeEmployee and PartTimeEmployee are concrete classes, so individual tables will be created for them as soon as we enter data in to table.
The good thing about JPA is that insertion and retrieval style is irrespective of what inheritance type we are using.

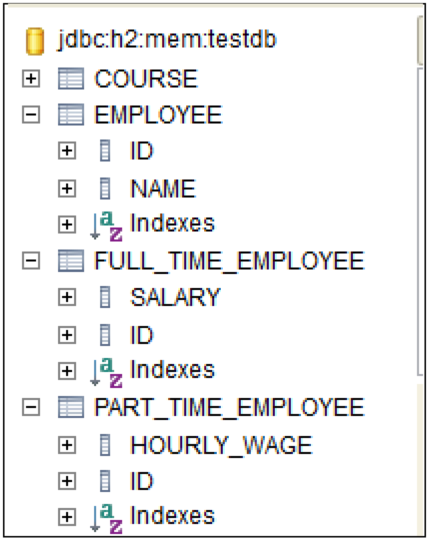


```
⊟ FULL_TIME_EMPLOYEE
    ⊞ ID
    ⊞ NAME
    ⊞ SALARY
    ⊞ Indexes
⊟ PART_TIME_EMPLOYEE
    ⊞ ID
    ⊞ NAME
    ⊞ HOURLY_WAGE
    ⊞ Indexes
```

A union will be done on both FullTimeEmployee and PartTimeEmployee to get the details of Employee.
The problems:
1) with this strategy is that the common columns (name) are repeated.
2) If we have 10 different classes, we will have as many no of tables. This is not good.
3) Retrieval is done using union. This is OK
Step 47 - JPA Inheritance Hierarchies and Mappings – Joined
JOINED: /**
     * A strategy in which fields that are specific to a

```
     * subclass are mapped to a separate table than the fields
     * that are common to the parent class, and a join is
     * performed to instantiate the subclass.
     */
```

Separate table created for super as well as sub class. Whenever we are trying to get the details of sub class, there will be a join performed.



```
jdbc:h2:mem:testdb
⊞ COURSE
⊟ EMPLOYEE
    ⊞ ID
    ⊞ NAME
    ⊞ Indexes
⊟ FULL_TIME_EMPLOYEE
    ⊞ SALARY
    ⊞ ID
    ⊞ Indexes
⊟ PART_TIME_EMPLOYEE
    ⊞ HOURLY_WAGE
    ⊞ ID
    ⊞ Indexes
```

Joined strategy is good in terms of the database design. However from performance perspective it not the best as it has to join columns to fetch the data.

```
Hibernate:
    select
        employee0_.id as id1_1_,
        employee0_.name as name2_1_,
        employee0_1_.salary as salary1_2_,
        employee0_2_.hourly_wage as hourly_w1_3_,
        case
            when employee0_1_.id is not null then 1
            when employee0_2_.id is not null then 2
            when employee0_.id is not null then 0
        end as clazz_
    from
        employee employee0_
    left outer join
        full_time_employee employee0_1_
            on employee0_.id=employee0_1_.id
    left outer join
        part_time_employee employee0_2_
            on employee0_.id=employee0_2_.id
```

Step 48 - JPA Inheritance Hierarchies and Mappings - Mapped Super Class

This strategy is about not to use inheritance at all, rather use @MappedSuperClass.

```
// Employee is a mapped super class which is present just for the sake of mapping.
@MappedSuperclass
/*
```

```
 * When a class is a mapped super class, it cannot be an entity. There
will not
 * be any table for this class.
 */
@Entity
// @Inheritance(strategy = InheritanceType.JOINED)
public abstract class Employee {
//.
}
```

If you have a class annotated with both @MappedSuperclass and @Entity, you will get below exception.

```
Caused by: org.hibernate.AnnotationException: An entity cannot be
annotated with both @Entity and @MappedSuperclass:
com.personal.Kunj.jpa.advancedjpa.entity.Employee
```

@MappedSuperclass :

```
* <p> A class designated with the <code>MappedSuperclass</code>
 * annotation can be mapped in the same way as an entity except that the
 * mappings will apply only to its subclasses since no table
 * exists for the mapped superclass itself. When applied to the
 * subclasses the inherited mappings will apply in the context
 * of the subclass tables. Mapping information may be overridden
 * in such subclasses by using the <code>AttributeOverride</code> and
 * <code>AssociationOverride</code> annotations or corresponding XML
elements.
 *
```

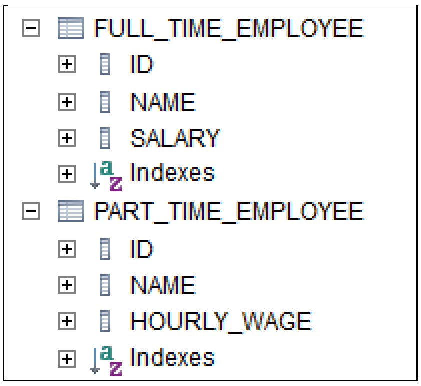
As there is no Employee class, application will throw below error bcz we are trying to use an instance of Employee class in the code:

```
java.lang.Error: Unresolved compilation problem:
	The method retrieveAllEmployees() is undefined for the type
EmployeeRepository
```

FIXED CODE:

```
package com.personal.Kunj.jpa.advancedjpa.repository;

import java.util.List;

import javax.persistence.EntityManager;

import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Repository;
import org.springframework.transaction.annotation.Transactional;

import com.personal.Kunj.jpa.advancedjpa.entity.Employee;
import com.personal.Kunj.jpa.advancedjpa.entity.FullTimeEmployee;
import com.personal.Kunj.jpa.advancedjpa.entity.PartTimeEmployee;

@Repository
@Transactional
```

```
public class EmployeeRepository {

      private Logger logger = LoggerFactory.getLogger(this.getClass());

      @Autowired
      EntityManager em;

      public void insert(Employee employee) {
            em.persist(employee);
      }
// Will work no more as There is no Employee entity anymore
      /*
       * public List<Employee> retrieveAllEmployees() { return
       * em.createQuery("select e from Employee e",
Employee.class).getResultList(); }
       */
      public List<PartTimeEmployee> retrieveAllPartTimeEmployees() {
            return em.createQuery("select e from PartTimeEmployee e",
PartTimeEmployee.class).getResultList();
      }

      public List<FullTimeEmployee> retrieveAllFullTimeEmployees() {
            return em.createQuery("select e from FullTimeEmployee e",
FullTimeEmployee.class).getResultList();
      }
}
```

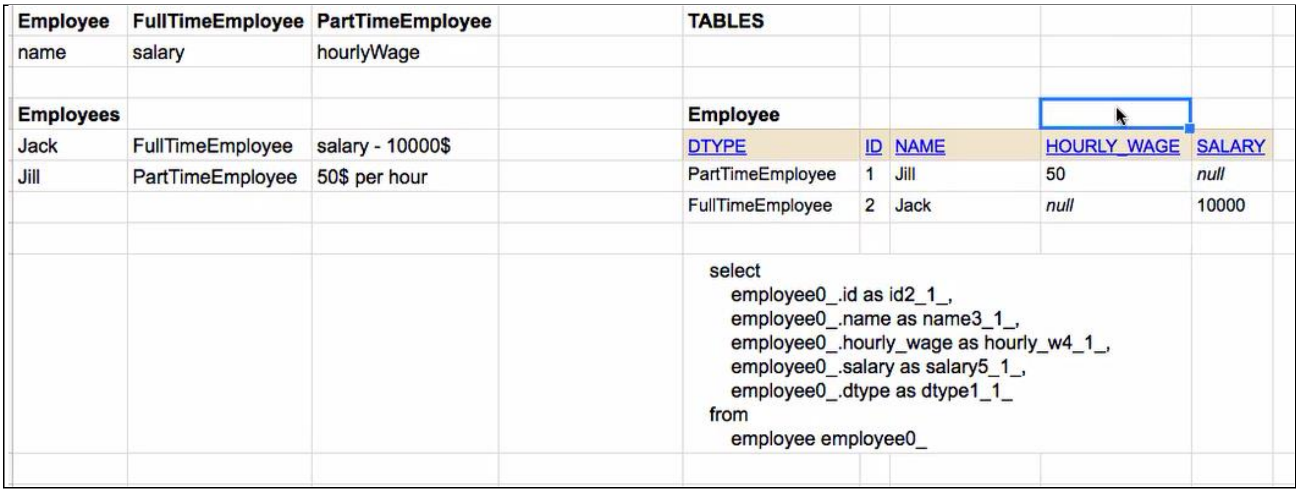


```
⊟ FULL_TIME_EMPLOYEE
    ⊞ ID
    ⊞ NAME
    ⊞ SALARY
    ⊞ Indexes
⊟ PART_TIME_EMPLOYEE
    ⊞ ID
    ⊞ NAME
    ⊞ HOURLY_WAGE
    ⊞ Indexes
```

```
Hibernate:
    select
        fulltimeem0_.id as id1_1_,
        fulltimeem0_.name as name2_1_,
        fulltimeem0_.salary as salary3_1_
    from
        full_time_employee fulltimeem0_
Hibernate:
    select
        parttimeem0_.id as id1_2_,
        parttimeem0_.name as name2_2_,
        parttimeem0_.hourly_wage as hourly_w3_2_
    from
        part_time_employee parttimeem0_
```

Step 49 - JPA Inheritance Hierarchies and Mappings - How to Choose

Single Table



Employee	FullTimeEmployee	PartTimeEmployee
name	salary	hourlyWage
Employees		
Jack	FullTimeEmployee	salary - 10000$
Jill	PartTimeEmployee	50$ per hour

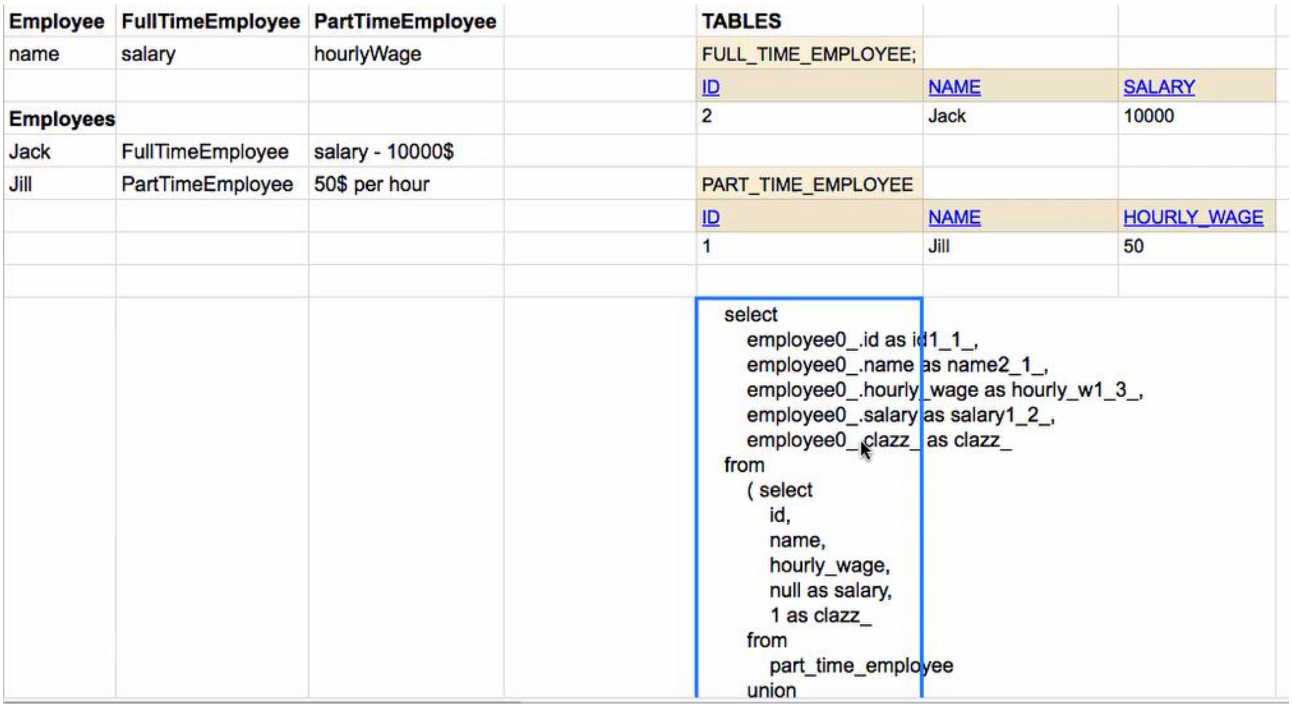
TABLES

Employee

DTYPE	ID	NAME	HOURLY_WAGE	SALARY
PartTimeEmployee	1	Jill	50	*null*
FullTimeEmployee	2	Jack	*null*	10000

```
select
   employee0_.id as id2_1_,
   employee0_.name as name3_1_,
   employee0_.hourly_wage as hourly_w4_1_,
   employee0_.salary as salary5_1_,
   employee0_.dtype as dtype1_1_
from
   employee employee0_
```

Table Per Class



Employee	FullTimeEmployee	PartTimeEmployee
name	salary	hourlyWage
Employees		
Jack	FullTimeEmployee	salary - 10000$
Jill	PartTimeEmployee	50$ per hour

TABLES

FULL_TIME_EMPLOYEE;

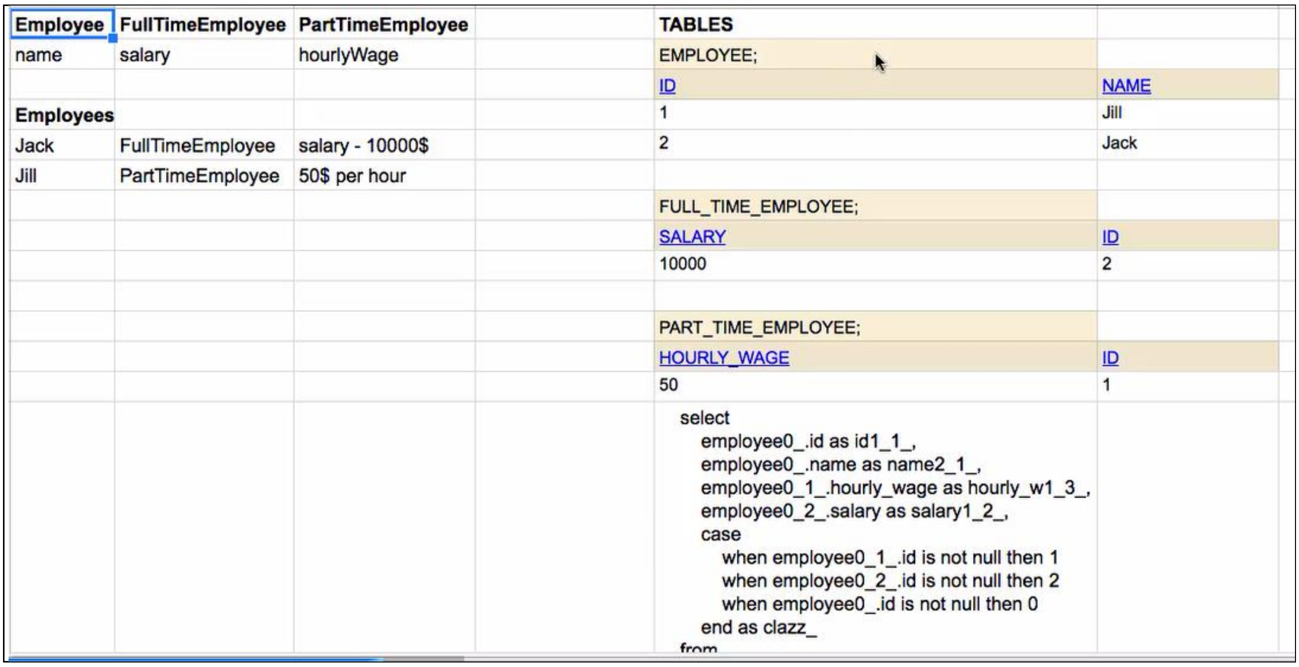
ID	NAME	SALARY
2	Jack	10000

PART_TIME_EMPLOYEE

ID	NAME	HOURLY_WAGE
1	Jill	50

```
select
   employee0_.id as id1_1_,
   employee0_.name as name2_1_,
   employee0_.hourly_wage as hourly_w1_3_,
   employee0_.salary as salary1_2_,
   employee0_.clazz_ as clazz_
from
   ( select
      id,
      name,
      hourly_wage,
      null as salary,
      1 as clazz_
   from
      part_time_employee
   union
```

Joined



Employee	FullTimeEmployee	PartTimeEmployee
name	salary	hourlyWage
Employees		
Jack	FullTimeEmployee	salary - 10000$
Jill	PartTimeEmployee	50$ per hour

TABLES

EMPLOYEE;

ID	NAME
1	Jill
2	Jack

FULL_TIME_EMPLOYEE;

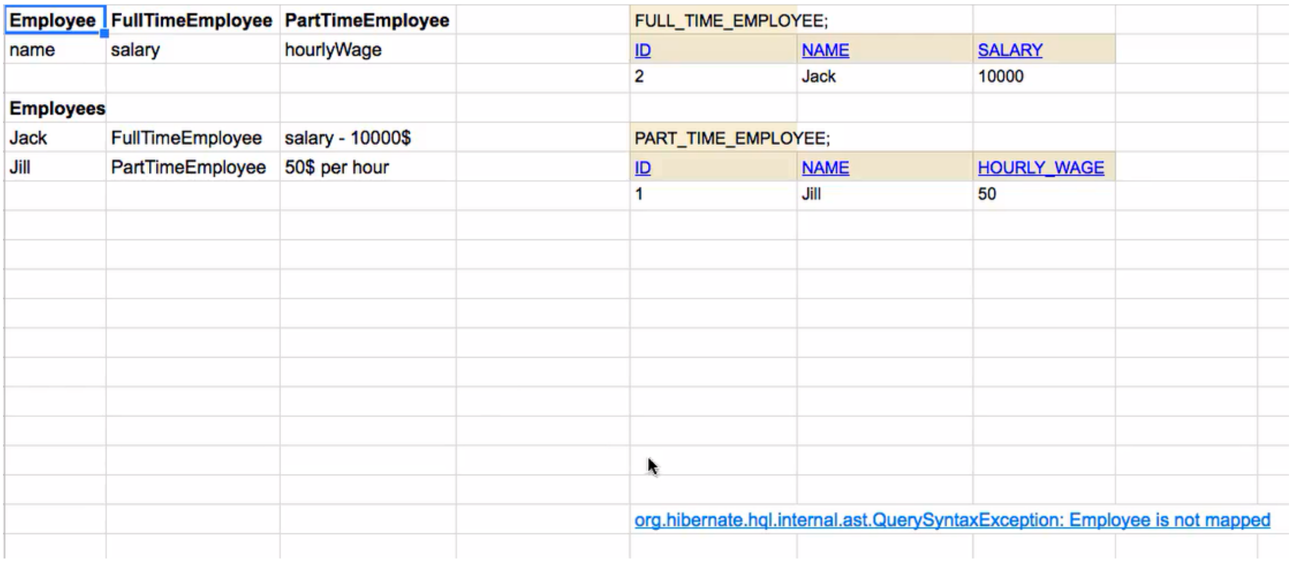
SALARY	ID
10000	2

PART_TIME_EMPLOYEE;

HOURLY_WAGE	ID
50	1

```
select
   employee0_.id as id1_1_,
   employee0_.name as name2_1_,
   employee0_1_.hourly_wage as hourly_w1_3_,
   employee0_2_.salary as salary1_2_,
   case
      when employee0_1_.id is not null then 1
      when employee0_2_.id is not null then 2
      when employee0_.id is not null then 0
   end as clazz_
from
```

MappedSuperClass



Employee	FullTimeEmployee	PartTimeEmployee
name	salary	hourlyWage
Employees		
Jack	FullTimeEmployee	salary - 10000$
Jill	PartTimeEmployee	50$ per hour

FULL_TIME_EMPLOYEE;

ID	NAME	SALARY
2	Jack	10000

PART_TIME_EMPLOYEE;

ID	NAME	HOURLY_WAGE
1	Jill	50

org.hibernate.hql.internal.ast.QuerySyntaxException: Employee is not mapped

How to choose?

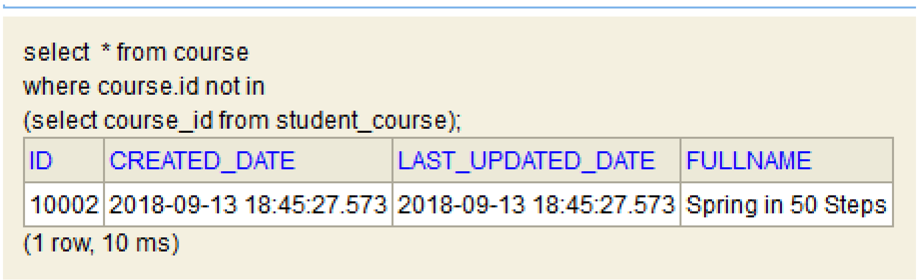
If you are concerned about data integrity.
Choose JOINED.

If you really worried about performance.
Single Table

10. Queries with Entities using JPQL

Step 50 - JPQL - Courses without Students

Find out the courses which do not have the students.



```
select * from course
where course.id not in
(select course_id from student_course);
```

ID	CREATED_DATE	LAST_UPDATED_DATE	FULLNAME
10002	2018-09-13 18:45:27.573	2018-09-13 18:45:27.573	Spring in 50 Steps

(1 row, 10 ms)

To do the above kind of stuff we used to like query in jdbc/ spring jdbc like above. With JPA we are not needed to write such a complex query.

We have already mapped course and student as ManyToMany. We will use this relationship to write query.

JPQLTest.java

```
@Test
	public void jpql_courses_without_students() {
		// Here we are refering to the entities and relation behind
them.
		// We are not really worried about the tables.
		// In c.students below, students is the variable defined in
the course class
		TypedQuery<Course> query = em.createQuery("Select c from
Course c where c.students is empty", Course.class);
```

```
          List<Course> resultList = query.getResultList();
          logger.info("Results -> {}", resultList);
          // [Course[Spring in 50 Steps]]
     }
```

Step 51 - JPQL - Courses with atleast 2 Students and order by

JPQLTest.java

```
@Test
     public void jpql_courses_with_atleast_2_students() {
          TypedQuery<Course> query = em.createQuery("Select c from
Course c where size(c.students) >= 2", Course.class);
          List<Course> resultList = query.getResultList();
          logger.info("Results -> {}", resultList);
          // [Course[JPA in 50 Steps]]
     }

     @Test
     public void jpql_courses_ordered_by_students() {
          TypedQuery<Course> query = em.createQuery("Select c from
Course c order by size(c.students) desc",
                    Course.class);
          List<Course> resultList = query.getResultList();
          logger.info("Results -> {}", resultList);
     }
```

Step 52 - JPQL - Courses like 100 Steps

Student has passport. We want to find those students whose passport numbers are in a certain pattern (containing 1234).

JPQLTest.java

```
@Test
     public void jpql_students_with_passports_in_a_certain_pattern() {
          TypedQuery<Student> query = em.createQuery("Select s from
Student s where s.passport.number like '%1234%'",
                    Student.class);
          List<Student> resultList = query.getResultList();
          logger.info("Results -> {}", resultList);
     }
```

You can also use JPQL for the below operations:

```
     //like
     //BETWEEN 100 and 1000
     //IS NULL
     //upper, lower, trim, length
```

Step 53 - JPQL - Using Joins

```
/*
```

```
	 * We are not worried about the tables. We look for entities. In
c.students,
	 * students is the field decalred in the course class
	 */
//JOIN => Select c, s from Course c JOIN c.students s
		//LEFT JOIN => Select c, s from Course c LEFT JOIN c.students
s
		//CROSS JOIN => Select c, s from Course c, Student s
		//3 and 4 =>3 * 4 = 12 Rows
```

JPQLTest.java

```
@Test
	public void join() {
// Cannot use a typed query as the result does not only contain student
		Query query = em.createQuery("Select c, s from Course c JOIN
c.students s");
		List<Object[]> resultList = query.getResultList();
		logger.info("Results Size -> {}", resultList.size());
		for (Object[] result : resultList) {
			logger.info("Course{} Student{}", result[0], result[1]);
		}
	}

	@Test
	public void left_join() {
		Query query = em.createQuery("Select c, s from Course c LEFT
JOIN c.students s");
		List<Object[]> resultList = query.getResultList();
		logger.info("Results Size -> {}", resultList.size());
		for (Object[] result : resultList) {
			logger.info("Course{} Student{}", result[0], result[1]);
		}
	}

	@Test
	public void cross_join() {
		Query query = em.createQuery("Select c, s from Course c,
Student s");
		List<Object[]> resultList = query.getResultList();
		logger.info("Results Size -> {}", resultList.size());
		for (Object[] result : resultList) {
			logger.info("Course{} Student{}", result[0], result[1]);
		}
	}
```

11. Queries using Java API - Criteria Queries

Step 54 - Criteria Query - Retrieving all courses

In JPQL, the queries are very similar to SQL. Some java developers feel it complex to write JPQL queries. Instead of writing a query, why not write a java api for doing JPQL stuff also? There comes CriteriaAPI.

Step 55 - Criteria Query - Courses like 100 Steps

Step 56 - Criteria Query - Courses without Students

Step 57 - Criteria Query - Using Joins

```
@RunWith(SpringRunner.class)
@SpringBootTest(classes = AdvancedJpaApplication.class)
public class CriteriaQueryTest {

      private Logger logger = LoggerFactory.getLogger(this.getClass());

      @Autowired
      EntityManager em;

      @Test
      public void all_courses() {
            // "Select c From Course c" --> to be done in java

            /*
             * 1. Use Criteria Builder to create a Criteria Query
returning the expected
             * result object.
             *
             * 2. To build a Criteria Query we also define the expected
result
             */
            CriteriaBuilder cb = em.getCriteriaBuilder();
            CriteriaQuery<Course> cq = cb.createQuery(Course.class);

            // 2. Define roots for tables which are involved in the query
            Root<Course> courseRoot = cq.from(Course.class);
```

```
           // 3. Define Predicates etc using Criteria Builder

           // 4. Add Predicates etc to the Criteria Query

           // 5. Build the TypedQuery using the entity manager and
criteria query
           TypedQuery<Course> query =
em.createQuery(cq.select(courseRoot));

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

           logger.info("Typed Query -> {}", resultList);
           // [Course[JPA in 50 Steps], Course[Spring in 50 Steps],
Course[Spring
           // Boot in 100 Steps]]
     }

     @Test
     public void all_courses_having_100Steps() {
           // "Select c From Course c where name like '%100 Steps' "

           // 1. Use Criteria Builder to create a Criteria Query
returning the
           // expected result object
           CriteriaBuilder cb = em.getCriteriaBuilder();
           CriteriaQuery<Course> cq = cb.createQuery(Course.class);

           // 2. Define roots for tables which are involved in the query
           Root<Course> courseRoot = cq.from(Course.class);

           // 3. Define Predicates etc using Criteria Builder
           Predicate like100Steps = cb.like(courseRoot.get("name"), "%100
Steps");

           // 4. Add Predicates etc to the Criteria Query
           cq.where(like100Steps);

           // 5. Build the TypedQuery using the entity manager and
criteria query
           TypedQuery<Course> query =
em.createQuery(cq.select(courseRoot));

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

           logger.info("Typed Query -> {}", resultList);
           // [Course[Spring Boot in 100 Steps]]
     }

     @Test
     public void all_courses_without_students() {
           // "Select c From Course c where c.students is empty"
```

```
        // 1. Use Criteria Builder to create a Criteria Query
returning the
        // expected result object
        CriteriaBuilder cb = em.getCriteriaBuilder();
        CriteriaQuery<Course> cq = cb.createQuery(Course.class);

        // 2. Define roots for tables which are involved in the query
        Root<Course> courseRoot = cq.from(Course.class);

        // 3. Define Predicates etc using Criteria Builder
        Predicate studentsIsEmpty =
cb.isEmpty(courseRoot.get("students"));

        // 4. Add Predicates etc to the Criteria Query
        cq.where(studentsIsEmpty);

        // 5. Build the TypedQuery using the entity manager and
criteria query
        TypedQuery<Course> query =
em.createQuery(cq.select(courseRoot));

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

        logger.info("Typed Query -> {}", resultList);
        // [Course[Spring in 50 Steps]]
    }

    @Test
    public void join() {
        // "Select c From Course c join c.students s"

        // 1. Use Criteria Builder to create a Criteria Query
returning the
        // expected result object
        CriteriaBuilder cb = em.getCriteriaBuilder();
        CriteriaQuery<Course> cq = cb.createQuery(Course.class);

        // 2. Define roots for tables which are involved in the query
        Root<Course> courseRoot = cq.from(Course.class);

        // 3. Define Predicates etc using Criteria Builder
        Join<Object, Object> join = courseRoot.join("students");

        // 4. Add Predicates etc to the Criteria Query

        // 5. Build the TypedQuery using the entity manager and
criteria query
        TypedQuery<Course> query =
em.createQuery(cq.select(courseRoot));
```

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

           logger.info("Typed Query -> {}", resultList);
           // [Course[JPA in 50 Steps], Course[JPA in 50 Steps],
Course[JPA in 50
           // Steps], Course[Spring Boot in 100 Steps]]
      }

      @Test
      public void left_join() {
           // "Select c From Course c left join c.students s"

           // 1. Use Criteria Builder to create a Criteria Query
returning the
           // expected result object
           CriteriaBuilder cb = em.getCriteriaBuilder();
           CriteriaQuery<Course> cq = cb.createQuery(Course.class);

           // 2. Define roots for tables which are involved in the query
           Root<Course> courseRoot = cq.from(Course.class);

           // 3. Define Predicates etc using Criteria Builder
           Join<Object, Object> join = courseRoot.join("students",
JoinType.LEFT);

           // 4. Add Predicates etc to the Criteria Query

           // 5. Build the TypedQuery using the entity manager and
criteria query
           TypedQuery<Course> query =
em.createQuery(cq.select(courseRoot));

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

           logger.info("Typed Query -> {}", resultList);
           // [Course[JPA in 50 Steps], Course[JPA in 50 Steps],
Course[JPA in 50
           // Steps], Course[Spring in 50 Steps], Course[Spring Boot in
100 Steps]]
      }

}
```

12. Transaction Management

Step 58 - Introduction to Transaction Management

13. Spring Data JPA & Spring Data REST

Step 64 - Introduction to Spring Data JPA

CourseRepository.java and StudentRepository.java have exactly the same code apart from the fact that they are just managing the different entities.

The other problem is the proliferation of data stores. Earlier there were just relational data bases, but now we have varieties of big data databases.

Spring data aims to provide simple abstraction to be able to access any kind of data. Spring Data JPA is the JPA specific implementation of Spring Data.

```
public interface CourseSpringDataRepository extends JpaRepository<Course,
Long> {

}
```

JpaRepository<Course, Long> → Arguments are : Name of the class, data type of the primary key

Step 65 - Testing the Spring Data JPA Repository with findById.
Step 66 - Spring Data JPA Repository - CRUD Methods
Step 67 - Sorting using Spring Data JPA Repository
Step 68 - Pagination using Spring Data JPA Repository
CourseSpringDataRepositoryTest.java

```
package com.personal.Kunj.jpa.advancedjpa.repository;

import static org.junit.Assert.assertFalse;
import static org.junit.Assert.assertTrue;
import java.util.Optional;
import org.junit.Test;
import org.junit.runner.RunWith;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.context.SpringBootTest;
import org.springframework.data.domain.Page;
import org.springframework.data.domain.PageRequest;
import org.springframework.data.domain.Pageable;
import org.springframework.data.domain.Sort;
import org.springframework.test.context.junit4.SpringRunner;
import com.personal.Kunj.jpa.advancedjpa.AdvancedJpaApplication;
import com.personal.Kunj.jpa.advancedjpa.entity.Course;

@RunWith(SpringRunner.class)
@SpringBootTest(classes = AdvancedJpaApplication.class)
public class CourseSpringDataRepositoryTest {

    private Logger logger = LoggerFactory.getLogger(this.getClass());

    @Autowired
    CourseSpringDataRepository repository;

    @Test
    public void findById_CoursePresent() {
        /*
         * Spring data JPA returns Optional back.
repository.findById(10001L) will not
         * return a Course object, rather it will return an Optional.
         *
         * Optional provides a way to check if course exists or not.
         *
         * Optional eliminates the need for a null value. Suppose we
pass a course id
         * which is not present then courseOptional will be a proper
object but it would
         * not contain a course so isPresent() will return false.
         */

        Optional<Course> courseOptional = repository.findById(10001L);
        assertTrue(courseOptional.isPresent());
    }

    @Test
    public void findById_CourseNotPresent() {
        Optional<Course> courseOptional = repository.findById(20001L);
        assertFalse(courseOptional.isPresent());
    }

    @Test
    public void playingAroundWithSpringDataRepository() {

        // Same method repository.save(course) is doing save as well
as update.

        // Course course = new Course("Microservices in 100 Steps");
        // repository.save(course);

        // course.setName("Microservices in 100 Steps - Updated");
        // repository.save(course);
        logger.info("Courses -> {} ", repository.findAll());
        logger.info("Count -> {} ", repository.count());
    }
    @Test
    public void sort() {
        // Sort criteria can be added by doing .sort().
        Sort sort = new Sort(Sort.Direction.ASC, "name");
       logger.info("Sorted Courses -> {} ", repository.findAll(sort));

    }

    @Test
    public void pagination() {
        // Want to divide the result in the pages of 3 result
        PageRequest pageRequest = PageRequest.of(0, 3);
        Page<Course> firstPage = repository.findAll(pageRequest);
        logger.info("First Page -> {} ", firstPage);

        // To get the second page data
        Pageable secondPageable = firstPage.nextPageable();
        Page<Course> secondPage = repository.findAll(secondPageable);
        logger.info("Second Page -> {} ", secondPage.getContent());

    }

}
```

Step 69 - Custom Queries using Spring Data JPA Repository

CourseSpringDataRepository.java

```
package com.personal.Kunj.jpa.advancedjpa.repository;

import java.util.List;

import org.springframework.data.jpa.repository.JpaRepository;
import org.springframework.data.jpa.repository.Query;

import com.personal.Kunj.jpa.advancedjpa.entity.Course;

public interface CourseSpringDataRepository extends JpaRepository<Course,
Long> {
    // Defining custom methods
    // Methods can start with find/retrieve/query for select statements
    List<Course> findByNameAndId(String name, Long id);

    List<Course> findByName(String name);

    List<Course> countByName(String name);

    List<Course> findByNameOrderByIdDesc(String name);

    List<Course> deleteByName(String name);
```

```
      @Query("Select  c  From Course c where name like '%100 Steps'")
      List<Course> courseWith100StepsInName();

      @Query(value = "Select  *  From Course c where name like '%100
Steps'", nativeQuery = true)
      List<Course> courseWith100StepsInNameUsingNativeQuery();

      @Query(name = "query_get_100_Step_courses")
      List<Course> courseWith100StepsInNameUsingNamedQuery();
}
```

CourseSpringDataRepositoryTest.java

```
@Test
      public void findUsingName() {
            logger.info("FindByName -> {} ", repository.findByName("JPA in
50 Steps"));
      }

      @Test
      public void findUsingStudentsName() {
            logger.info("findUsingStudentsName -> {} ",
repository.findByName("Ranga"));
      }
```

Step 70 - Spring Data JPA REST

I want to expose restful services around CourseSpringDataRepository.java to be able to operate on the course.

Either you can use your typical Spring MVC to create restful web services or you can use Spring Data JPA Rest.

To use Spring Data JPA Rest add the below dependency to pom.xml.

```
        <dependency>
                <groupId>org.springframework.boot</groupId>
                <artifactId>spring-boot-starter-data-rest</artifactId>
        </dependency>
```

There will be lot of mapping created after the above dependency addition.

Q.) How to make use of the above added dependency?
Ans: Annotate CourseSpringDataRepository.java with @RepositoryRestResource(path = "course"). All the resources will be exposed on /course url.

Spring Data JPA Rest is not recommended for production.

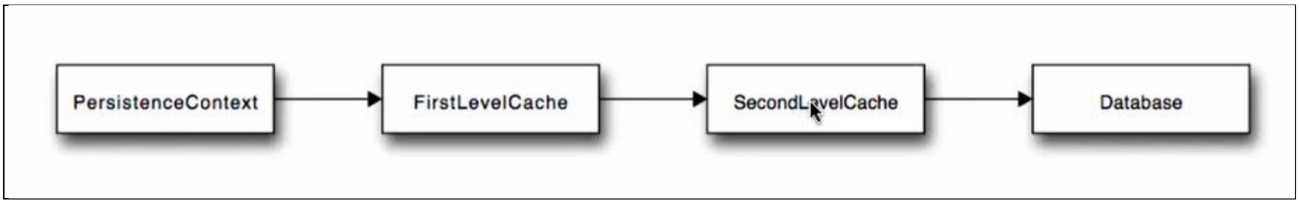
14. Caching with Hibernate & JPA

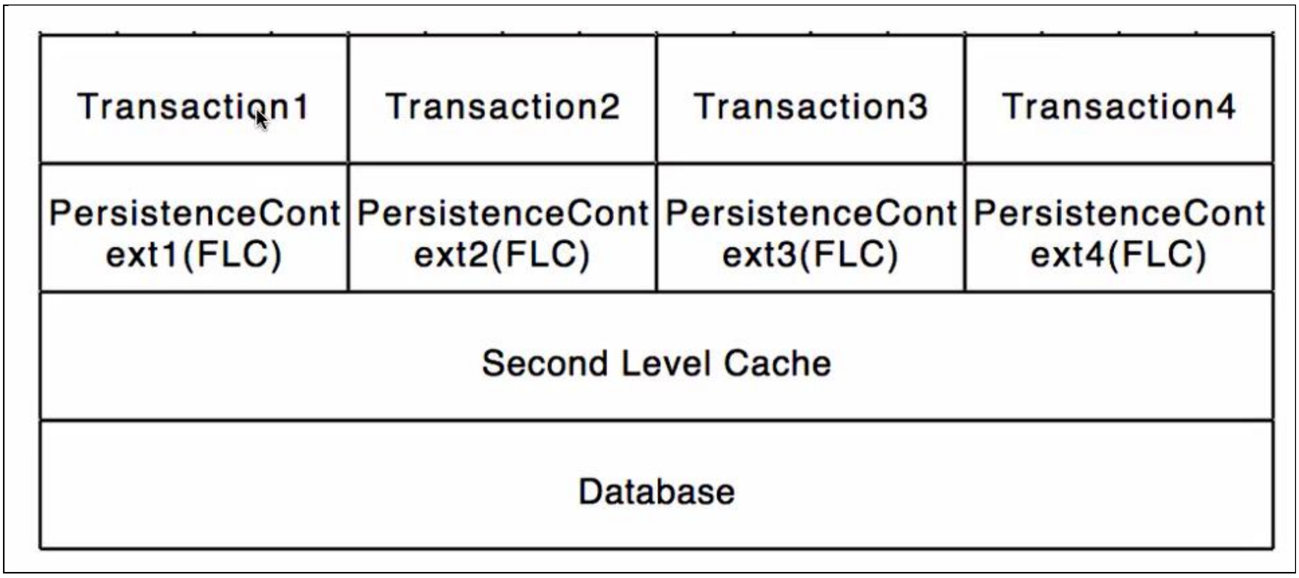
Step 71 - Introduction to Caching

Cache only the data that does not frequently changes.

In Hibernate, we have 2 levels of cache.

    First level cache
    Second level cache





Transaction1	Transaction2	Transaction3	Transaction4
PersistenceContext1(FLC)	PersistenceContext2(FLC)	PersistenceContext3(FLC)	PersistenceContext4(FLC)
Second Level Cache			
Database			

In a typical application, there are multiple transactions going on in parallel.

Here there are 4 transactions running. Each transaction is associated with PeristenceContext of its own where all the entities that are being modified during that transaction are tracked.
Let us say during transaction 1 I am retrieving the details of the same course again and again. In this case hibernate will not go the database all the 3 times. The first time it will go to the DB, retrieve the details, and it will have it inside the persistenceContext. The next time you ask for the same course details, hibernate will return it back from the persistenceContext, it will not talk to the DB. It will pull the data from the cache and return it back. Here First Level cache comes into picture. First level cache is within the boundary of a single transaction.

The second level cache comes into picture across multiple transactions. Let us say you have one instance of an application deployed on an application server and multiple users using this application. Irrespective of the users, the list of states/countries are same (these things generally do not change with user). We can store such info in second level cache. Second level cache will typically store the common info for all the users of an application. Let us say the application has just launched up and there is a request for list of countries, transaction 1 goes to the PersistenceContext and we would go to the second level cache, if details are not in the second level cache , we will go to the DB, get the details, put in the second level cache and from then on any of the request from any of the transaction thereafter the data will be retrieved from the second level cache.

Step 72 - Hibernate and JPA Caching - First Level Cache

If you want to make best use of first level cache then the boundary of the transaction should start with the service method. Starting with the service method, all the calls to the data layer should be within the scope of the single transaction then the first level cache will be efficient.

CourseRepositoryTest.java

```
     @Test
     @Transactional
     public void findById_firstLevelCacheDemo() {

          Course course = repository.findById(10001L);
          logger.info("First Course Retrieved {}", course);

          /*
           * It will not fire a separate DB query if the method is
annotataed with
           *
           * @Transactional.
           */
          Course course1 = repository.findById(10001L);
          logger.info("First Course Retrieved again {}", course1);

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

          assertEquals("JPA in 50 Steps", course1.getName());
     }
```

```
Hibernate:
    select
        course0_.id as id1_0_0_,
        course0_.created_date as created_2_0_0_,
        course0_.last_updated_date as last_upd3_0_0_,
        course0_.fullname as fullname4_0_0_
    from
        course course0_
    where
        course0_.id=?
2018-09-14 20:48:12.581 TRACE 3624 --- [           main]
o.h.type.descriptor.sql.BasicBinder      : binding parameter [1]
as [BIGINT] - [10001]
```

```
2018-09-14 20:48:12.600 TRACE 3624 --- [           main]
o.h.type.descriptor.sql.BasicExtractor   : extracted value
([created_2_0_0_] : [TIMESTAMP]) - [2018-09-14T20:48:06.766]
2018-09-14 20:48:12.600 TRACE 3624 --- [           main]
o.h.type.descriptor.sql.BasicExtractor   : extracted value
([last_upd3_0_0_] : [TIMESTAMP]) - [2018-09-14T20:48:06.766]
2018-09-14 20:48:12.601 TRACE 3624 --- [           main]
o.h.type.descriptor.sql.BasicExtractor   : extracted value
([fullname4_0_0_] : [VARCHAR]) - [JPA in 50 Steps]
2018-09-14 20:48:12.608 TRACE 3624 --- [           main]
org.hibernate.type.CollectionType        : Created collection
wrapper:
[com.personal.Kunj.jpa.advancedjpa.entity.Course.reviews#10001]
2018-09-14 20:48:12.609 TRACE 3624 --- [           main]
org.hibernate.type.CollectionType        : Created collection
wrapper:
[com.personal.Kunj.jpa.advancedjpa.entity.Course.students#10001]
2018-09-14 20:48:12.609  INFO 3624 --- [           main]
c.p.K.j.a.r.CourseRepositoryTest         : First Course Retrieved
Course [id=10001, name=JPA in 50 Steps]
2018-09-14 20:48:12.609  INFO 3624 --- [           main]
c.p.K.j.a.r.CourseRepositoryTest         : First Course Retrieved
again Course [id=10001, name=JPA in 50 Steps]
```

Step 73 - Hibernate and JPA Caching - Basics of Second Level Cache with EhCache

First level cache was active by default.
2nd level cache needs configuration. You cannot cache all the data at the second level cache bcz hibernate does not know what data is going to change. You need to tell hibernate about the data which is not going to change between multiple transactions. What is the data which is common to multiple users? One you specify that, we can use 2nd level cache framework for ex: ehCache to cache all the common data.

Pom.xml

```
		<dependency>
			<groupId>org.hibernate</groupId>
			<artifactId>hibernate-ehcache</artifactId>
		</dependency>
```

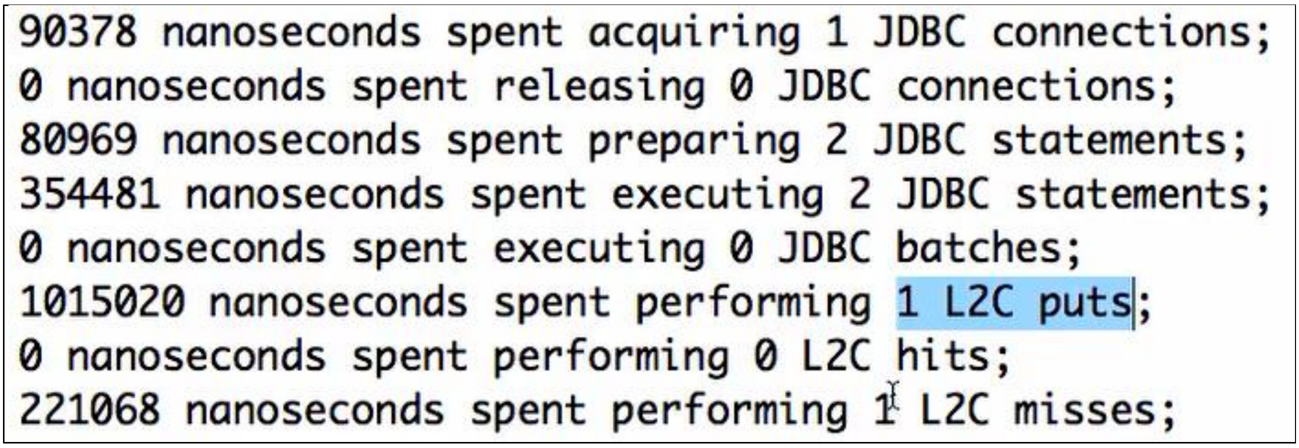
Now, configure the cache.
Application.properties

```
# Second Level Cache - Ehcache

#1. enable second level cache
spring.jpa.properties.hibernate.cache.use_second_level_cache=true

#2. specify the caching framework - EhCache
spring.jpa.properties.hibernate.cache.region.factory_class=org.hib
ernate.cache.ehcache.EhCacheRegionFactory

#3. Only cache what I tell to cache.
```

```
spring.jpa.properties.javax.persistence.sharedCache.mode=ENABLE_SE
LECTIVE

logging.level.net.sf.ehcache=debug

#4. What data to cache?
# By enabling caching on entities
```

Step 73 - Hibernate and JPA Caching - Basics of Second Level Cache with EhCache

Let us pick course entity for caching.



```
90378 nanoseconds spent acquiring 1 JDBC connections;
0 nanoseconds spent releasing 0 JDBC connections;
80969 nanoseconds spent preparing 2 JDBC statements;
354481 nanoseconds spent executing 2 JDBC statements;
0 nanoseconds spent executing 0 JDBC batches;
1015020 nanoseconds spent performing 1 L2C puts;
0 nanoseconds spent performing 0 L2C hits;
221068 nanoseconds spent performing 1 L2C misses;
```

```
@Entity
@Cacheable
public class Course {
//…………
}
```

15. Hibernate & JPA Tips

Step 75 - Hibernate Tips - Hibernate Soft Deletes - @SQLDelete and @Where

EntityManager.remove() will delete a record completely from the database. You will not have any history of it. In some of the circumstances you will want to keep history of the rows. We can do this by implementing soft delete. Soft delete is done by adding a column to the database to track whether it is deleted or not.

In Course.java we will use another attribute 'isDeleted' . If a row is deleted, we will update this indicator to 'true'. Whenever we create a new data in table, this indicator will be 'false'.

Data.sql

```
insert into course(id, name, created_date, last_updated_date,is_deleted)
values(10001,'JPA in 50 Steps', sysdate(), sysdate(),false);
insert into course(id, name, created_date, last_updated_date,is_deleted)
values(10002,'Spring in 50 Steps', sysdate(), sysdate(),false);
insert into course(id, name, created_date, last_updated_date,is_deleted)
values(10003,'Spring Boot in 100 Steps', sysdate(), sysdate(),false);
```

Now we will want to set isDeleted column true whenever we delete a row.

```
@Entity
@Cacheable
// On the delete of a row I want this SQL to be called
@SQLDelete(sql = "update course set is_deleted=true where id=?")
public class Course {

// ….
}
```

If we execute the below test at this point in time, the test fails but we can see the update statement fired for the DB.

JUnit test failure message:
java.lang.AssertionError: expected null, but was:<Course [id=10002, name=Spring in 50 Steps]>

CourseRepositoryTest.java

```
@Test
    // To reset the database status
    @DirtiesContext
    public void deleteById_basic() {
        repository.deleteById(10002L); // (1)
        assertNull(repository.findById(10002L)); // (2)
    }
```

```
Hibernate:
    update
        course
    set
```

```
        is_deleted=true
    where
        id=?
```

In deleteById_basic() method we are deleting a course (1). But when we try to retrieve the data in (2), the row is coming back bcz the row is still in the database.

If we look at the query fired to DB by (2);

```
Hibernate:
    select
        course0_.id as id1_0_0_,
        course0_.created_date as created_2_0_0_,
        course0_.is_deleted as is_delet3_0_0_,
        course0_.last_updated_date as last_upd4_0_0_,
        course0_.name as name5_0_0_
    from
        course course0_
    where
        course0_.id=?
```

This query is not using the fact that now there is a ‘is_Deleted’ column and if the ‘is_Deleted’ column has the value ‘true’ then I would not need to fetch that row bcz that is not an active row.

Now we have to tell Course entity to add a specific condition to all the select query. This can be done by adding @Where to the entity.

```
@Entity
@Cacheable
// On the delete of a row I want this SQL to be called
@SQLDelete(sql = "update course set is_deleted=true where id=?")
@Where(clause = "is_deleted = false")
public class Course {
// ………….
}
```

Now unit test succeeds.
In console we can see the changes select query to include the where clause specified on the entity.

```
Hibernate:
    select
        course0_.id as id1_0_0_,
        course0_.created_date as created_2_0_0_,
        course0_.is_deleted as is_delet3_0_0_,
        course0_.last_updated_date as last_upd4_0_0_,
        course0_.name as name5_0_0_
    from
        course course0_
    where
        course0_.id=?
        and (
            course0_.is_deleted = 0
        )
```

Step 76 - Hibernate Soft Deletes - Part 2

2 Caveats associated with soft delete.

(1)
If you run this unit test,
NativeQueriesTest.java

```
@Test
	public void native_queries_basic() {
		Query query = em.createNativeQuery("SELECT * FROM COURSE",
Course.class);
		List resultList = query.getResultList();
		logger.info("SELECT * FROM COURSE  -> {}", resultList);
		//SELECT * FROM COURSE  -> [Course[Web Services in 100 Steps],
Course[JPA in 50 Steps - Updated], Course[Spring in 50 Steps],
Course[Spring Boot in 100 Steps]]
	}
```

```
Hibernate:
    SELECT
        *
    FROM
        COURSE
```

We should note that the @Where condition that we added in the previous step does not apply to the native query. So yiu yourself have to add the where clause like the below:

```
Query query = em.createNativeQuery("SELECT * FROM COURSE where
is_Deleted=0", Course.class);
```

(2) Hibernate does not know what is happening inside @Where(clause = "is_deleted = false")
Whenever you are deleting a course entity we provided a where clause and this where clause is just appended to the query. Hibernate does not know that is_Deleted column is being set to false.

CourseRepository.java

```
public void deleteById(Long id) {
		Course course = findById(id);
		em.remove(course);
	}
```

What is happening when we try to remove the course in the above method?
Bcz there is @SQLDelete(sql = "update course set is_deleted=true where id=?") annotation on course, The sql inside this is getting fired. Is_Deleted is being set to true. If you look at the attribute isDeleted inside Course.java, the value will still be false bcz hibernate has no idea about the fact that you are actually setting inDeleted to false bcz that is done in a query @Where(clause = "is_deleted = false").

The solution to the above will be to set isDeleted attribute on the course to true whever we call the `em.remove(course).` But that is a little bit risky thing to do bcz that would mean whenever we try to delete a course using the EntityManager we will need to remember that course.isDeleted has to be set to true otherwise if any other thing is trying to retrieve the course in that specific transaction it would get the non updated course entity.

Or the other option is to use one of the entity life cycle methods. Whenever a row of a specific entity is deleted there is a method that gets fired (method annotated with @PreRemove).

Course.java

```
@PreRemove
	private void preRemove() {
		LOGGER.info("Setting isDeleted to True");
		this.isDeleted = true;
	}
```

See in the log that this method is being called.

Console:

```
c.p.Kunj.jpa.advancedjpa.entity.Course    : Setting isDeleted to
True
```

Step 77 - JPA Entity Life Cycle Methods

The important annotations related to life cycle methods in an entity are:
@PostLoad → If you mark a method in an entity with this annotation, it will be called as soon as the entity is retrieved and loaded. If there is a select query fired and that specific entity is being loaded , this specific method on this entity would be called.
@PostPersist → Method is called after the entity is persisted.
@PostRemove
@PostUpdate
@PrePersist
@PreRemove
@PreUpdate

Step 78 - Using Embedded and Embeddable with JPA

Let us say we have an address for an object. There is one address for a student.

In this step we will look at the scenarios where we would like address's fields to be directly present in the student. I do not want to create a relationship between student and address.

If we want address to be embedded in student entity, we need to add @Embeddable on the address. We need to annotate address var in student class with @Embedded.

Not just for entity, even for embedded object you need a default constructor.

```
@Entity
public class Student {

	@Id
	@GeneratedValue
```

```
    private Long id;

    @Column(nullable = false)
    private String name;

    @Embedded
    private Address address;

    @OneToOne(fetch = FetchType.LAZY)
    private Passport passport;

    @ManyToMany
    // JoinColumn for this entity is Student_id.
    // InverseJoinColumn is course_id
    @JoinTable(name = "STUDENT_COURSE", joinColumns = @JoinColumn(name
= "STUDENT_ID"), inverseJoinColumns = @JoinColumn(name = "COURSE_ID"))

    private List<Course> courses = new ArrayList<>();

// ……………………………
}

@Embeddable
public class Address {

    private String line1;
    private String line2;
    private String city;

    protected Address() {
    }

    public Address(String line1, String line2, String city) {
        super();
        this.line1 = line1;
        this.line2 = line2;
        this.city = city;
    }

}
```

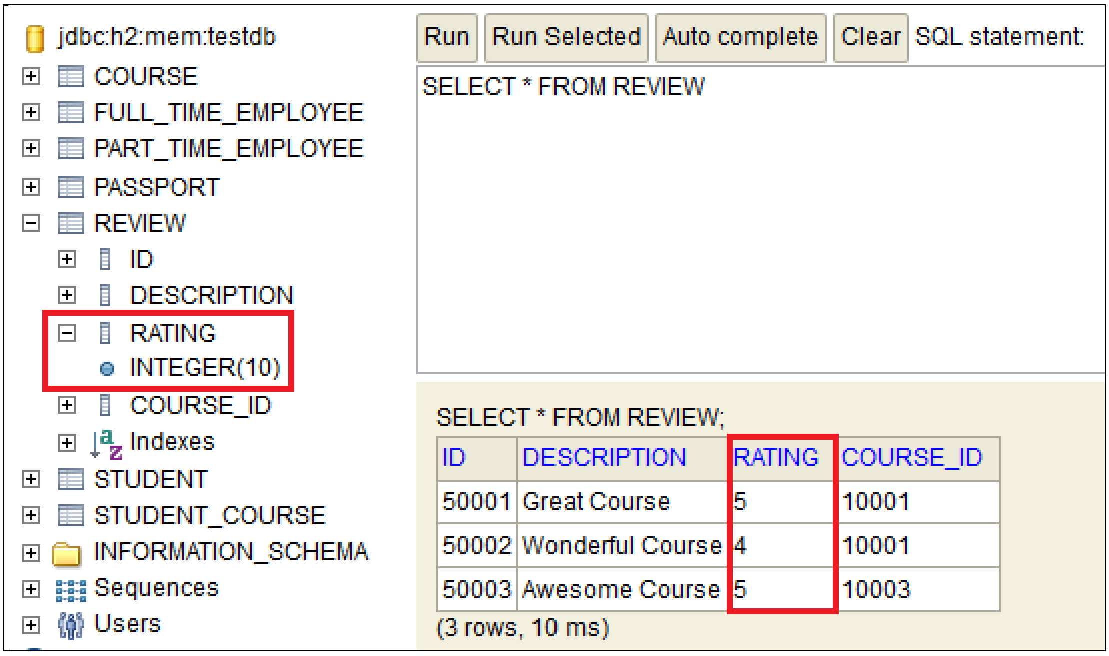
StudentRepositoryTest.java

```
    @Test
    @Transactional
    public void setAddressDetails() {
        Student student = em.find(Student.class, 20001L);
        student.setAddress(new Address("No 101", "Some Street",
"Hyderabad"));
        em.flush();
    }
```

Step 79 - Using Enums with JPA

In Review .java, we declared field review as String. This is not good practice as there are only 5 ratings possible. There is a chance that bad data gets stored in review entity

```
@Entity
public class Review {
	@Id
	@GeneratedValue
	private Long id;

	// @Enumerated --> To say that rating is an ENUM
	// By default a numeric column will be created that accepts ORDINAL
	(ie 1,2,3,4,5). 1 for the first enum constant and 2 for the 2nd and
	so on
	@Enumerated
	private ReviewRating rating;

	private String description;

	@ManyToOne
	private Course course;
// ..
}
```



```
SELECT * FROM REVIEW
```

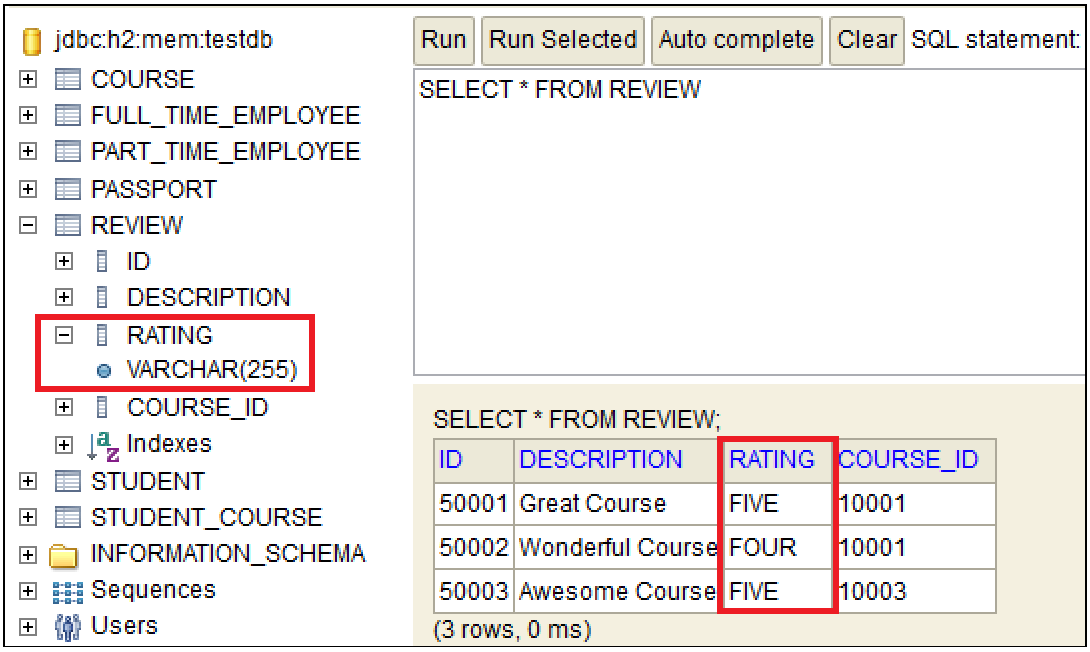
SELECT * FROM REVIEW;

ID	DESCRIPTION	RATING	COURSE_ID
50001	Great Course	5	10001
50002	Wonderful Course	4	10001
50003	Awesome Course	5	10003

(3 rows, 10 ms)

If we are changing ordinals then it is better to make it as string bcz the ordinal values will keep on changing if we insert constant in enum.

```
	@Enumerated(EnumType.STRING)
	private ReviewRating rating;
```



```
SELECT * FROM REVIEW
```

```
SELECT * FROM REVIEW;
```

ID	DESCRIPTION	RATING	COURSE_ID
50001	Great Course	FIVE	10001
50002	Wonderful Course	FOUR	10001
50003	Awesome Course	FIVE	10003

(3 rows, 0 ms)

Step 80 - JPA Tip - Be cautious with toString method implementations

Let us say we define toString() of Course.java as :

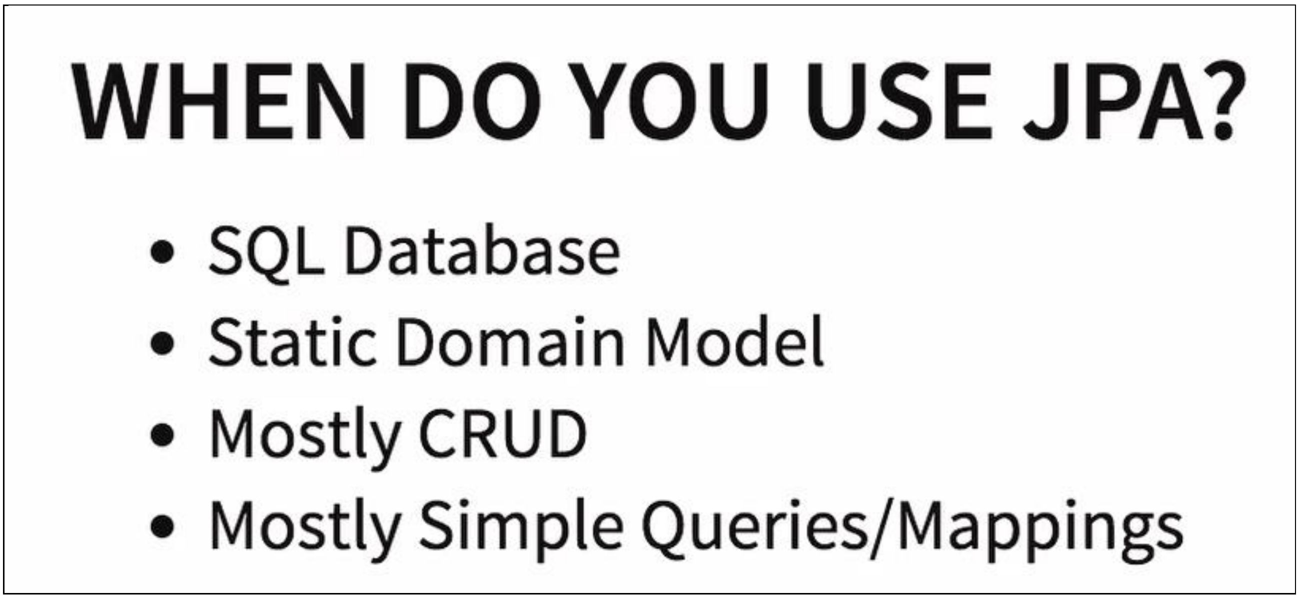
```
@Override
	public String toString() {
		return String.format("Course[%s] Review[%s]", name, reviews);
	}
```

```
In CouseRepository.java a developer logs course,
```

```
public Course findById(Long id) {
		Course course = em.find(Course.class, id);
		logger.info("Course -> {}");
		return course;
	}
```

The logger in findById(Long id) will print both Course and review details. 2 select queries will be fired to the database to fetch course as well as review details. But we were asking just for course details.

Step 81 - JPA Tip - When do you use JPA



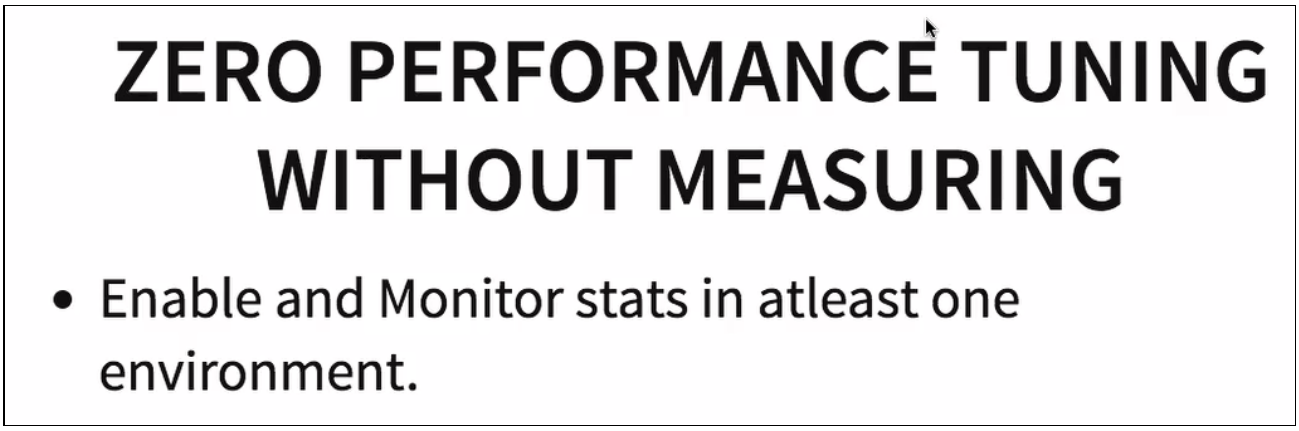
WHEN DO YOU USE JPA?

- SQL Database
- Static Domain Model
- Mostly CRUD
- Mostly Simple Queries/Mappings

If you have a batch oriented application, do not go for JPA.

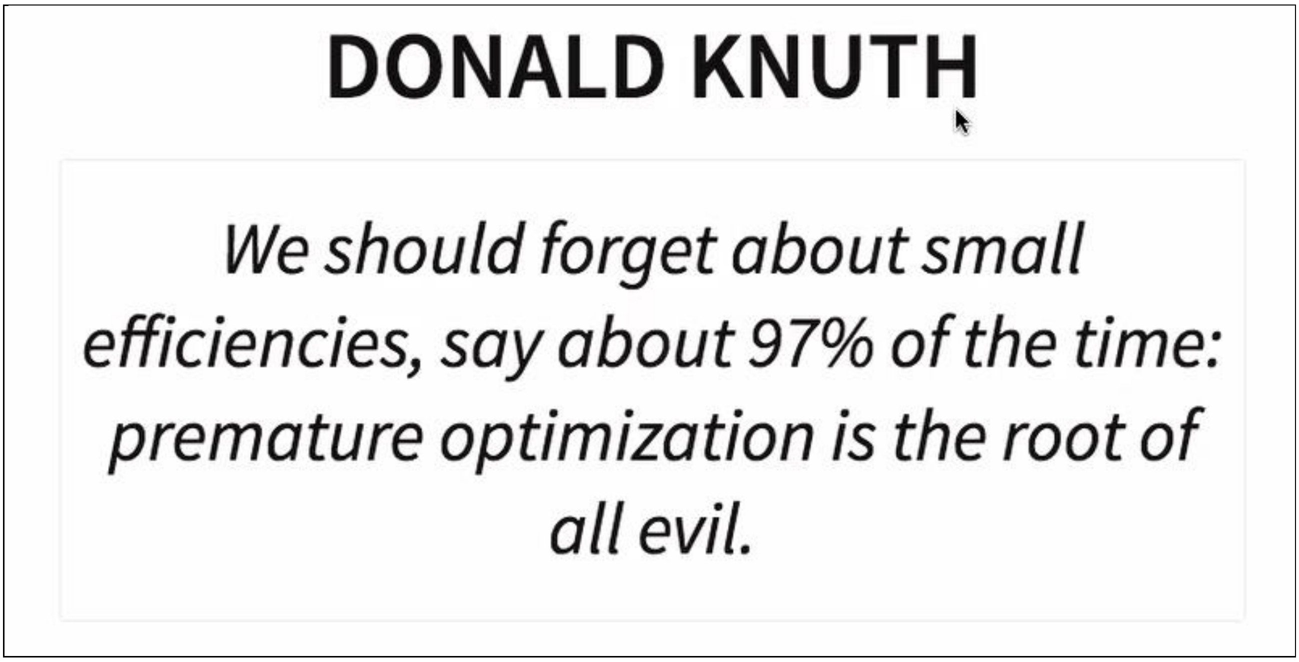
16. Performance Tuning Tips with Hibernate & JPA

Step 82 - Performance Tuning - Measure before Tuning



ZERO PERFORMANCE TUNING WITHOUT MEASURING

- Enable and Monitor stats in atleast one environment.

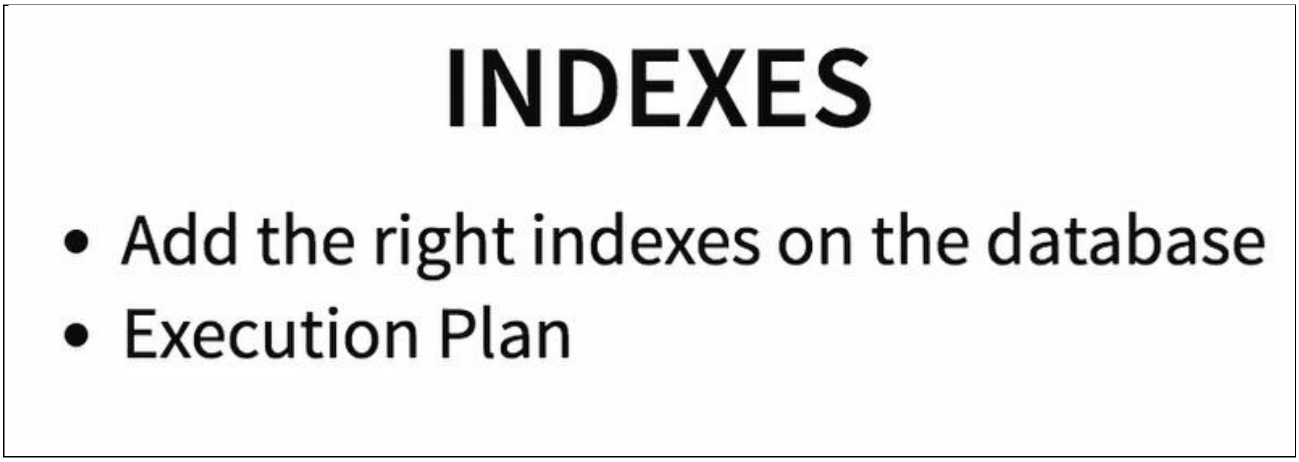


DONALD KNUTH

We should forget about small efficiencies, say about 97% of the time: premature optimization is the root of all evil.

```
#Turn Statistics on
# "generate_statistics" will tell no of queries fired
spring.jpa.properties.hibernate.generate_statistics=true
# keep logging level for "org.hibernate.stat" as debug
logging.level.org.hibernate.stat=trace
```

Step 83 - Performance Tuning – Indexes

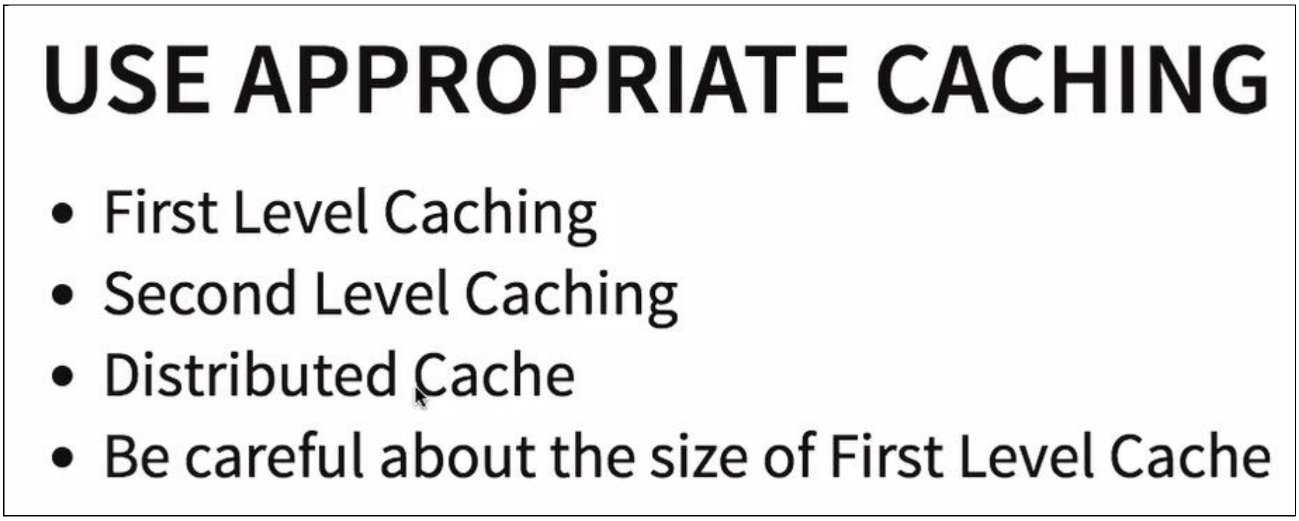


INDEXES

- Add the right indexes on the database
- Execution Plan

Any relational database need to have right indexes on table.
One of the ways to identify the right indexes is to look at the execution plans of your queries. For ex if students are often searched using name, then create index for the name in the student table.

Step 84 - Performance Tuning - Use Appropriate Caching



USE APPROPRIATE CACHING

- First Level Caching
- Second Level Caching
- Distributed Cache
- Be careful about the size of First Level Cache

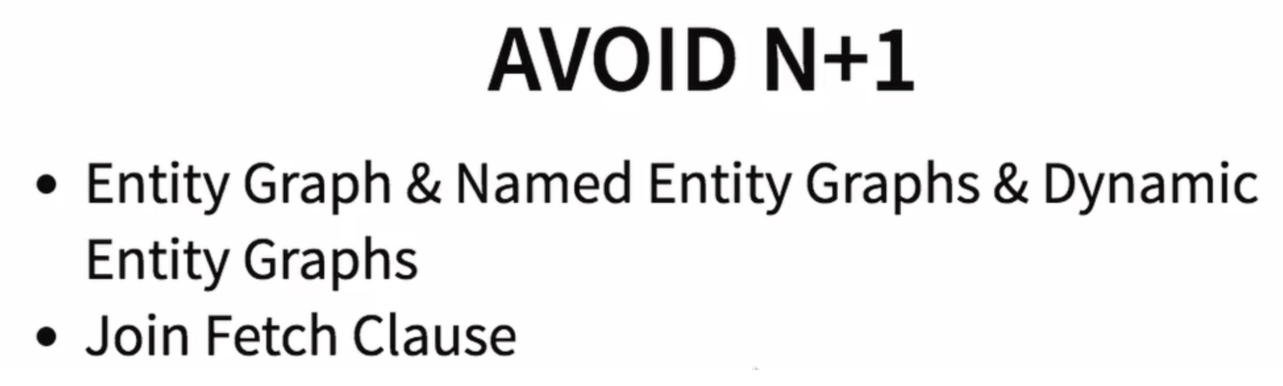
First level cache is within a single transaction and is automatically enabled. Make sure that you do not make first level cache too big. If you are storing 1000’s of entities in 1st level cache, regularly clear using the entitymanager. If the size of 1st level cache grows then searching through it will also become inefficient.

Second level caching is useful to make sure that different transaction on the same server or the same instance of the application can share the common data. For ex things like country and state (drop down values) which will be common for all the users.

The second level cache is specific to an instance of the application, but if you are running a lot of applications in parallel. You are expecting a huge amount of load and one application is not sufficient to handle that. In that case you would be distributing your load among multiple application instances. The distributed cache is useful to cache things across all the multiple instances. A good example of distributed cache is hazzlecast.

Step 85 - Performance Tuning - Eager vs Lazy Fetch
Depending on the situation, any of the fetch might be a good choice. We have to evaluate it by our use cases.

Step 86 - Performance Tuning - Avoid N+1 Problems



AVOID N+1

- Entity Graph & Named Entity Graphs & Dynamic Entity Graphs
- Join Fetch Clause

```java
@NamedQueries(value = { @NamedQuery(name = "query_get_all_courses", query
= "Select  c  From Course c"),
           @NamedQuery(name = "query_get_all_courses_join_fetch", query =
"Select  c  From Course c JOIN FETCH c.students s"),
           @NamedQuery(name = "query_get_100_Step_courses", query =
"Select  c  From Course c where name like '%100 Steps'") })
@Entity
@Cacheable
// On the delete of a row I want this SQL to be called
@SQLDelete(sql = "update course set is_deleted=true where id=?")
@Where(clause = "is_deleted = false")
public class Course {
// …
}
```

PerformanceTuningTest.java

```java
package com.personal.Kunj.jpa.advancedjpa.repository;

import java.util.List;

import javax.persistence.EntityGraph;
import javax.persistence.EntityManager;
import javax.persistence.Subgraph;
import javax.transaction.Transactional;

import org.junit.Test;
import org.junit.runner.RunWith;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.context.SpringBootTest;
import org.springframework.test.context.junit4.SpringRunner;

import com.personal.Kunj.jpa.advancedjpa.AdvancedJpaApplication;
import com.personal.Kunj.jpa.advancedjpa.entity.Course;

@RunWith(SpringRunner.class)
@SpringBootTest(classes = AdvancedJpaApplication.class)
public class PerformanceTuningTest {

     private Logger logger = LoggerFactory.getLogger(this.getClass());

     @Autowired
     EntityManager em;

     @Test
     @Transactional
     public void creatingNPlusOneProblem() {
        List<Course> courses =
em.createNamedQuery("query_get_all_courses",
Course.class).getResultList();
        /*
         * 4 queries will be fired. One for courses and 3 for the
students for
         * individual courses. this is N+1 problem.
         */
        /*
         * One solution is to make the students eager fetch in
Course.java. but the
         * problem is that anytime you want to retrieve a course, all
the students of
         * the course will be retrieved
         */
        for (Course course : courses) {
            logger.info("Course -> {} Students -> {}", course,
course.getStudents());
        }
    }

    /*
     * Second option to solves the N+1 problem.
     *
     * In this we will not change the course entity but when we are
retrieving
     * course and student entity together, we can add a hint in that
method.
     *
     * Just one JDBC statement will be fired in this case.
     */
    @Test
    @Transactional
    public void solvingNPlusOneProblem_EntityGraph() {

        EntityGraph<Course> entityGraph =
em.createEntityGraph(Course.class);
        Subgraph<Object> subGraph =
entityGraph.addSubgraph("students");

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

        for (Course course : courses) {
            logger.info("Course -> {} Students -> {}", course,
course.getStudents());
        }
    }
    // Third option to solve N+1 problem.
    @Test
    @Transactional
    public void solvingNPlusOneProblem_JoinFetch() {
        List<Course> courses =
em.createNamedQuery("query_get_all_courses_join_fetch",
Course.class).getResultList();
        for (Course course : courses) {
            logger.info("Course -> {} Students -> {}", course,
course.getStudents());
        }
    }

}
```

17. Few more FAQ

FAQ 5 - How to connect to a different database with Spring Boot

FAQ 6 - Approach to design great applications with JPA
Start thinking from the perspective of your database then jump into creating your entities and relationships.

FAQ 7 - Good Practices for developing JPA Applications
(1)
Fields in the entities to be private.
Member variables in the component to be private.
@Autowired
Private EntityManager entityManager;

(2) Use in-memory database for unit test
(3) Use data.sql to initialize your data for testing
(4) Use assert in your course