Spring Data JPA

SQL

```
create table users (id integer(10), name varchar(255), password varchar(255),
email varchar(100), mobileNumber varchar(20), occupation varchar(255));
```

1. Create Entity

```
@Entity
@Table(name = "users")
public class User {

    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private Long id;

    private String name;
    private String password;
    private String email;

    @Column(name = "MOBILE_NUMBER")
    private String mobileNumber;
    private String occupation;

    public Long getId() {
        return id;
    }
    public void setId(Long id) {
        this.id = id;
    }
    public String getName() {
        return name;
    }
    public void setName(String name) {
        this.name = name;
    }
    public String getPassword() {
        return password;
    }
    public void setPassword(String password) {
        this.password = password;
    }
    public String getEmail() {
        return email;
    }
    public void setEmail(String email) {
        this.email = email;
```

```
    }
    public String getMobileNumber() {
        return mobileNumber;
    }
    public void setMobileNumber(String mobileNumber) {
        this.mobileNumber = mobileNumber;
    }
    public String getOccupation() {
        return occupation;
    }
    public void setOccupation(String occupation) {
        this.occupation = occupation;
    }
}
```

2. Create Repository

```
@Repository
public interface UserRepository extends CrudRepository<User, Long> {

    public List<User> findAll();
}
```

3. Open Test class and write

Create

```
    @Autowired
    UserRepository userRepository;

    @Test
    void testCreate() {
        User user = new User();
        user.setId((long)3);
        user.setName("Sharvil");
        user.setPassword("dude");
        user.setEmail("sharvil@gmail.com");
        user.setMobileNumber("9898989898");
        user.setOccupation("Student");

        userRepository.save(user);
    }
```

Note: MSSQL might not work with `GenerationType.AUTO`, change it to `GenerationType.IDENTITY`.

Read

```
@Test
    public void testGet() {
        if(userRepository.existsById(1)) {
            userRepository
                .findById(1)
                .ifPresent(
                    (e) -> {
                        assertNotNull(e);
                        assertEquals(e.getName(), "Sharvil");
                    }
                );
        }
    }
```

Update

```
    @Test
    void testUpdate() {
        if(userRepository.existsById(2)) {
            userRepository
                .findById(2)
                .ifPresent(
                    (e) -> {
                        e.setEmail("dude@gmail.com");
                        User user = userRepository.save(e);
                        assertNotNull(user);
                        assertEquals(user.getName(), "Sharvil");
                    }
                );
        }
    }
```

Delete

```
    @Test
    void testDelete() {
        if(userRepository.existsById(2)) {
            userRepository
                .findById(2)
                .ifPresent(
                    (e) -> {
                        userRepository.delete(e);
                        assertNull(e);
                    }
                );
        }
    }
```

Table Generator

```
create table employee(id int PRIMARY KEY, name varchar(20));
create table id_gen(gen_name varchar(255) PRIMARY KEY, gen_val int(20));
```

Auto increment

```
@Id
@GeneratedValue(strategy=GenerationType.IDENTITY)
private int id;
```

Table type strategy

Custom Table Generator

```
@Id
@TableGenerator(name="employee_gen", table="id_gen", pkColumnName="gen_name",
valueColumnName="gen_val", allocationSize=10)
@GeneratedValue(strategy=GenerationType.TABLE, generator="employee_gen")
private int id;
```

Custom ID Generator

Custom ID Generator class

```
public class CustomRandomGenerator implements IdentiferGenerator {

    @Override
    public Serializable generate(SessionImplementor arg0, Object arg1) throws
HibernateException {

        int id = 0;
        Random random = new Random();
        random.nextInt(100000);

        return id;
    }
}
```

Entity class

```
@Entity
public class Employee {
```

```
    // emp_id is custom name, Generator name

    @Id
    @GenericGenerator(name="emp_id", strategy="packagename.CustomRandomGenerator")
    @GeneratedValue(generator="emp_id")
    public int id;

    ....
}
```

Query Methods

Derived queries with the predicates `IsStartingWith, StartingWith, StartsWith, IsEndingWith, EndingWith, EndsWith, IsNotContaining, NotContaining, NotContains, IsContaining, Containing, Contains` the respective arguments for these queries will get sanitized. This means if the arguments actually contain characters recognized by LIKE as wildcards these will get escaped so they match only as literals. The escape character used can be configured by setting the escapeCharacter of the @EnableJpaRepositories annotation.

```
    @Test
    @Disabled
    public void testQueryMethod1() {
        Optional<User> user$ = userRepository.findByName("Deepen");
        user$.ifPresent(e -> {assertNotNull(e);});

        //user$.ifPresent(System.out::println);
    }

    @Test
    @Disabled
    public void testQueryMethod2() {
        //Optional<User> user$ = userRepository.findByNameAndEmail("Deepen",
"sharvil@gmail.com");
        //user$.ifPresent(e -> {assertNotNull(e);});

        //user$.ifPresent(e -> {System.out.println("e: "+e.getName());});

        Optional<List<User>> user$ = userRepository.findByNameAndEmail("Deepen",
"sharvil@gmail.com");
        user$.ifPresent(e1 -> {
            e1.forEach(
                e -> {
                    System.out.println("e: "+e.getName());
                }
            );
        });
    }
```

```
    @Test
    @Disabled
    public void testQueryMethod3() {

        Optional<List<User>> user$ =
userRepository.findByEmailAndPassword("sharvil@gmail.com", "dude");
        user$.ifPresent(e1 -> {
            e1.forEach(
                e -> {
                    System.out.println("e: "+e.getName());
                }
            );
        });
    }

    @Test
    public void testQueryMethod4() {

        Optional<User> user$ = userRepository.findByNameStartingWith("Dee");
        user$.ifPresent(

            e -> {
                System.out.println("e: "+e.getName());
            }

        );
    }
```

```
@Repository
public interface UserRepository extends CrudRepository<User, Integer> {

    public List<User> findAll();
    public Optional<User> findByName(String name);
    //public Optional<User> findByNameAndEmail(String name, String email);
    public Optional<List<User>> findByNameAndEmail(String name, String email);

    public Optional<List<User>> findByEmailAndPassword(String name, String email);
    public Optional<User> findByNameStartingWith(String name);
}
```

Continue

Paging and Sorting

Earlier versions had `PageRequest()` constructor and in newer version it is `PageRequest.of()`.

To enable Paging and Sorting do this:

```
public interface ProductRepository extends PagingAndSortingRepository<Product,
Integer> {

}
```

To use:

```
@Test
public void testFindAllPage() {
    Pageable pageable = new PageRequest(0, 1); // 0 is the page index, 1 is size
    Page<Product> results = repository.findAll(pageable);
    results.forEach(p -> System.out.println(p.getName()));
}
```

Order is means column name. Direction is default to ASC.

Sort

```
repository
    .findAll(Sort.by(Direction.DESC, "name", "price"))
    .forEach(p -> System.out.println(p.getName()));
```

Easier way,

```
Pageable sortedByName =
  PageRequest.of(0, 3, Sort.by("name"));

Pageable sortedByPriceDesc =
  PageRequest.of(0, 3, Sort.by("price").descending());

Pageable sortedByPriceDescNameAsc =
  PageRequest.of(0, 5, Sort.by("price").descending().and(Sort.by("name")));
```

JPQL

(OLD) Java Persistence API

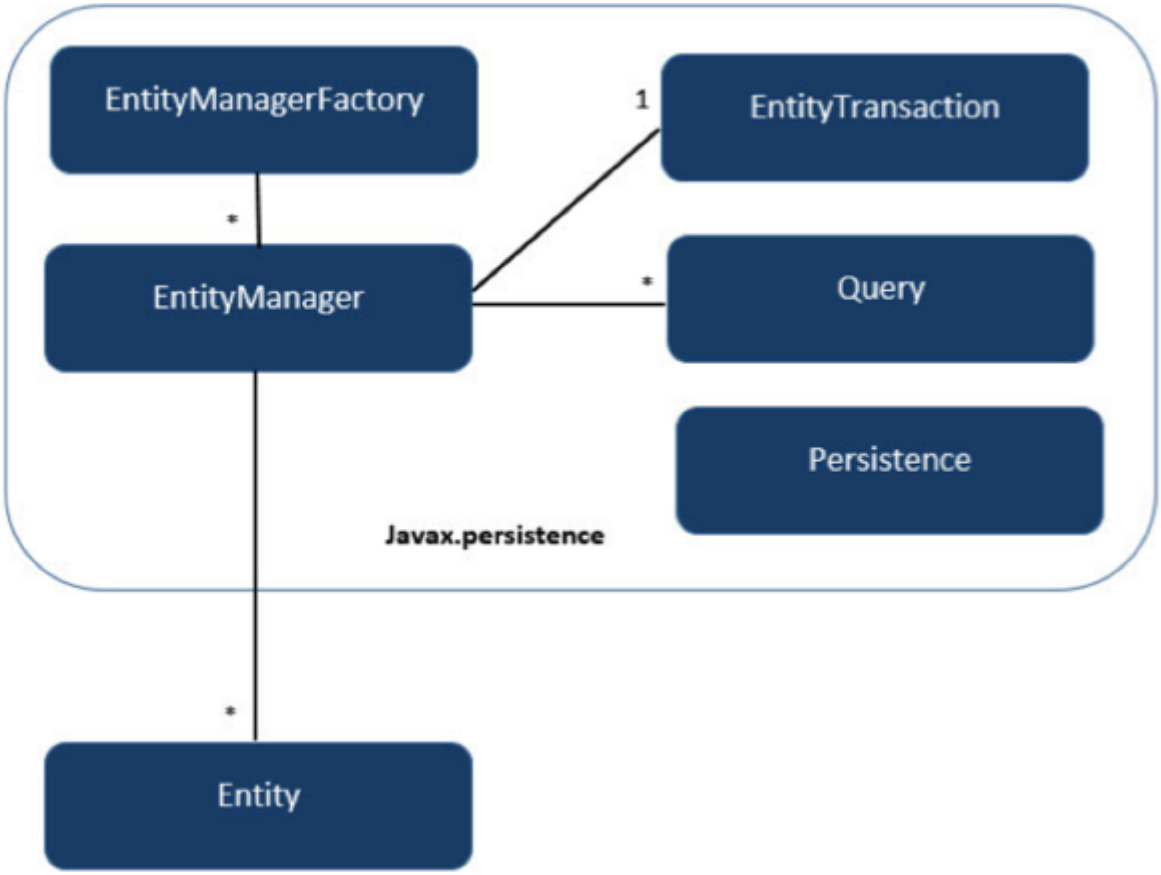
"The domain model has a class. The database has a table. They look pretty similar. It should be simple to convert one to the other automatically." This is a thought we've probably all had at one point or another while writing yet another Data Access Object (DAO) to convert Java Database Connectivity (JDBC) result sets into something object-oriented.

JPA History

Earlier versions of EJB, defined persistence layer combined with business logic layer using javax.ejb.EntityBean Interface.

- While introducing EJB 3.0, the persistence layer was separated and specified as JPA 1.0 (Java Persistence API). The specifications of this API were released along with the specifications of JAVA EE5 on May 11, 2006 using JSR 220.
- JPA 2.0 was released with the specifications of JAVA EE6 on December 10, 2009 as a part of Java Community Process JSR 317.
- JPA 2.1 was released with the specification of JAVA EE7 on April 22, 2013 using JSR 338.

JPA Relationships



- The relationship between EntityManagerFactory and EntityManager is one-to-many. It is a factory class to EntityManager instances.
- The relationship between EntityManager and EntityTransaction is one-to-one. For each EntityManager operation, there is an EntityTransaction instance.
- The relationship between EntityManager and Query is one-to-many. Many number of queries can execute using one EntityManager instance.
- The relationship between EntityManager and Entity is one-to-many. One EntityManager instance can manage multiple Entities.

> Previously, hibernate-annotations was released and versioned from hibernate core. But from version 3.5 and up it is included with hibernate core. And for some reason it was still released from 3.5.0 to 3.5.6 but you do not need it anymore.

JPA implementations have the choice of managing transactions themselves (RESOURCE_LOCAL), or having them managed by the application server's JTA implementation. In most cases, RESOURCE_LOCAL is fine. This would use basic JDBC-level transactions. The downside is that the transaction is local to the JPA persistence unit, so if you want a transaction that spans multiple persistence units (or other databases), then RESOURCE_LOCAL may not be good enough.