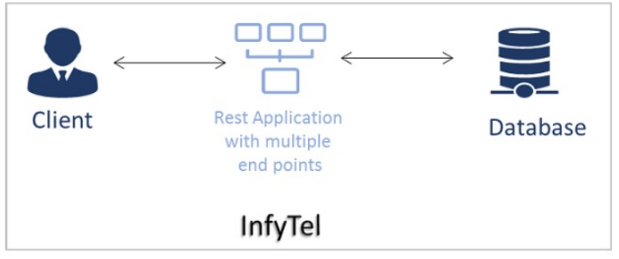
Spring data JPA with Boot

Prelodge

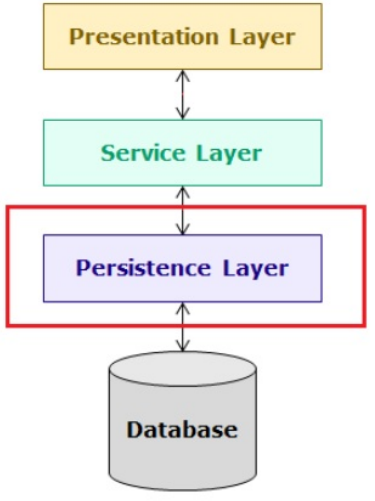
For implementing the Persistence Layer of an enterprise application Spring uses many approaches. A developer can choose appropriate Spring modules such as Spring JDBC, Spring ORM, or Spring Data JPA for implementing the data access layer of an enterprise application depending on the repository type. Spring simplifies transactions by allowing the developer to implement transactions in a declarative way using simple annotations and configurations. For the more fine-grained transactions, Spring also supports programmatic transactions.

In this course, we will focus on how to implement the data access layer of an enterprise application by using Spring Data JPA with Spring Boot.



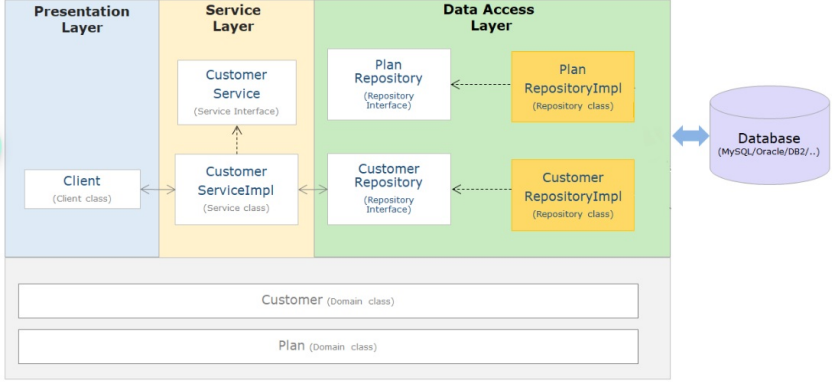
InfyTel is built as a three-tier application that consists of

- Presentation Layer
- Service/Business Layer
- Persistence Layer



Features supported by appl^n

- Add Customer
- Edit Customer Details
- Search Customer
- View all Customer
- Remove Customer



Introduction to Spring Data JPA

Consider the InfyTel application. The Admin of the InfyTel application has to maintain its Customer and Plan details. In this course, we will see how to implement the persistence layer of the InfyTel application using Spring Data JPA.

- Add Customer - Adding customer records
- Remove Customer - Deleting customer records

The following are the steps to implement the previous requirements of the InfyTel application.

Step 1: Define an entity class Customer.java to represent customer details.

Step 2: Add an appropriate database connector dependency in pom.xml.

Step 3: Create a data access layer with an interface CustomerDAO and a repository class CustomerDAOImpl to implement the interface.

Step 4: Create a service layer with an interface CustomerService and CustomerServiceImpl class to implement the interface

Step 5: Create a presentation layer with client class to access database operations and display details on the console

Step 6: Create a table Customer in the database.

```
create table customer(
phone_no bigint primary key,
name varchar(50),
age integer,
gender char(10),
address varchar(50),
plan_id integer
);
```

⇒ Limitations of JDBC API

Let us understand the limitations of JDBC API

- A developer needs to open and close the connection.
- A developer has to create, prepare, and execute the statement and also maintain the resultset.
- A developer needs to specify the SQL statement(s), prepare, and execute the statement.
- A developer has to set up a loop for iterating through the result (if any).
- A developer has to take care of exceptions and handle transactions.

The above limitations can be solved with the technologies Spring ORM.

→ ORM (object relational mapping)

Object Relational Mapping (ORM) is a technique or design pattern, which maps object models with the relational model.

- In Programming languages like Java, the related information or the data will be persisted in the form of hierarchical and interrelated objects.
- In the relational database, the data is persisted as table format or relations.

The greatest challenge in integrating the concepts of RDBMS and OOP is a mapping of the Java objects to databases. When object and relational paradigms work with each other, a lot of technical and conceptual difficulties arise, as mapping of an object to a table may not be possible in all the contexts. Storing and retrieving Java objects using a Relational database exposes a paradigm mismatch called "Object-Relational Impedance Mismatch". These differences are because of perception, style, and patterns involved in both the paradigms that lead to the following paradigm mismatches:

- Granularity: Mismatch between the number of classes in the object model and the number of tables in the relational model.
- Inheritance or Subtype: Inheritance is an object-oriented paradigm that is not available in RDBMS.
- Associations: In object-oriented programming, the association is represented using reference variables, whereas, in the relational model foreign keys are used for associating two tables.
- Identity: In Java, object equality is determined by the "==" operator or "equals()" method, whereas in RDBMS, uses the primary key to uniquely identify the records.
- Data Navigation: In Java, the dot(.) operator is used to travel through the object network, whereas, in RDBMS join operation is used to move between related records.

What is ORM?

Object Relational Mapping (ORM) is a technique or design pattern, which maps object models with the relational model. It has the following features:

- It resolves the object-relational impedance mismatch by mapping
  - Java classes to tables in the database
  - Instance variables to columns
  - Objects to rows in the table
- It helps the developer to get rid of SQL queries. They can concentrate on the business logic and work with the object model which leads to faster development of the application.
- It is database independent. All database vendors provide support for ORM. Hence, the application becomes portable without worrying about the underlying database.

Benifits of ORM

- ORM provides a programmatic approach to implement database operations.
- ORM maps Java objects to the relational database tables in an easier way based on simple configuration.
- Supports simple query approaches like HQL(Hibernate Query language) and JPQL (Java Persistence Query Language)
- Supports object-oriented concepts such as inheritance, mapping, etc.

Java Persistance API (JPA) uses ORM to provide :-

- Defines an API for mapping the object model with the relational model
- Defines an API for performing CRUD operations
- Standardizes ORM features and functionalities in Java.
- Provides an object query language called Java Persistence Query Language (JPQL) for querying the database.
- Provides Criteria API to fetch data over an object graph.

Configuration Required :-

@Repositoy:

It indicates the repository class(POJO class) in the persistence layer. Repository class is defined using **@Repository** annotation at the class level. This annotation is a specialization of **@Component** annotation for the persistence layer.

Benefits of the @Repository:

- Enables the Spring Framework auto scanning support to create a repository bean without the explicit bean definition in the configuration file.
- Causes Spring exception translation(translates checked exceptions into unchecked exceptions).

JPA needs an Entity class to perform all the CRUD operations because Entities can represent fine-grained persistent object and they are not remotely accessible components.

Entity Class:

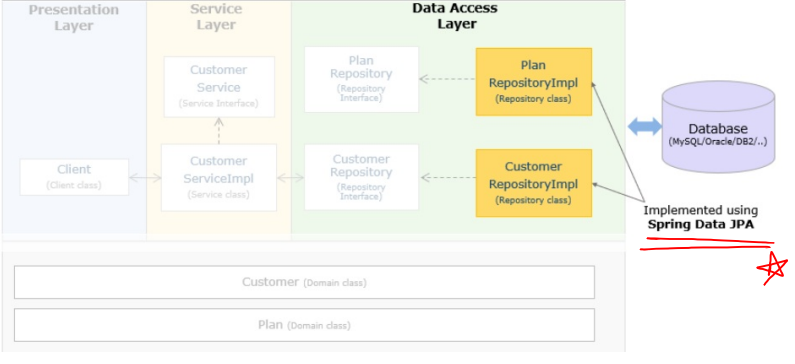
An Entity class is a class in Java that is mapped to a database table in a relational database.

```java
package com.infytel.entity;
import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.Id;
@Entity
public class Customer {

    @Id
    @Column(name = "phone_no")
    private Long phoneNumber;
    private String name;
    private Integer age;
    private Character gender;
    private String address;
    @Column(name = "plan_id")
    private Integer planId;
    //constructors
    //getters and setters
}
```

@Entity :- declare as entity class
@Id :- " " primary key
@Column :- declare mapping of a column in database
@Table :- declare mapping of a entity class to a table name
@Temporal :- converts the date & time values from java objects to compatible database type
@Transient :- It will not persisted to database

Why Spring data JPA ?



⇒ Spring data JPA overcome these limitations of Spring ORM JPA (with spring boot)

- The Programmer has to write the code to perform common database operations(CRUD operations) in repository class implementation.
- A developer can define the required database access methods in the repository implementation class in his/her own way, which leads to inconsistency in the data access layer implementation.

What is Spring data JPA

Spring data is high level project from spring whose objective is to unify and ease of access to SQL & noSQL databases.
It removes DAO (repository) impliment" entirely
It helps to impliment persistance layer and reducing the effort.

Spring data commons provide basic interfaces to support :-

- CRUD operations
- Sorting of data
- Pagination
- Specific abstraction
- JpaRepository interface to support JPA
- MongoRepository interface to support MongoDB

Introduction to Spring boot.

Spring is lightweight framework to develop enterprises applications :-
but it has difficult configurations and project dependency management
Thats why spring boot is developed :-

Spring Boot is a framework built on the top Spring framework that helps developers build Spring-based applications quickly and easily. The main goal of Spring Boot is to quickly create Spring-based applications without requiring developers to write the same boilerplate configuration again and again.

1. Spring Boot is an opinionated framework

Spring Boot forms opinions. It means that Spring Boot has some sensible defaults which you can use to quickly build your application.

default web container of Springboot is Tomcat.

2. Spring Boot is customizable

Though Spring Boot has its defaults, you can easily customize it at any time during your development based on your needs.

The main Spring Boot features are as follows:

1. Starter Dependencies
2. Automatic Configuration
3. Spring Boot Actuator
4. Easy-to-use Embedded Servlet Container Support

Note :-

The @SpringBootApplication annotation indicates that it is a configuration class and also triggers auto-configuration and component scanning. It is a combination of following annotations with their default attributes.

- @EnableAutoConfiguration – This annotation enables auto-configuration for Spring boot application which automatically configures our application based on the dependencies that a developer has already added.
- @ComponentScan – This enables the Spring bean dependency injection feature by using @Autowired annotation. All application components which are annotated with @Component, @Service, @Repository or @Controller are automatically registered as Spring Beans. These beans can be injected by using @Autowired annotation.
- @Configuration – This enables Java based configurations for Spring boot application.

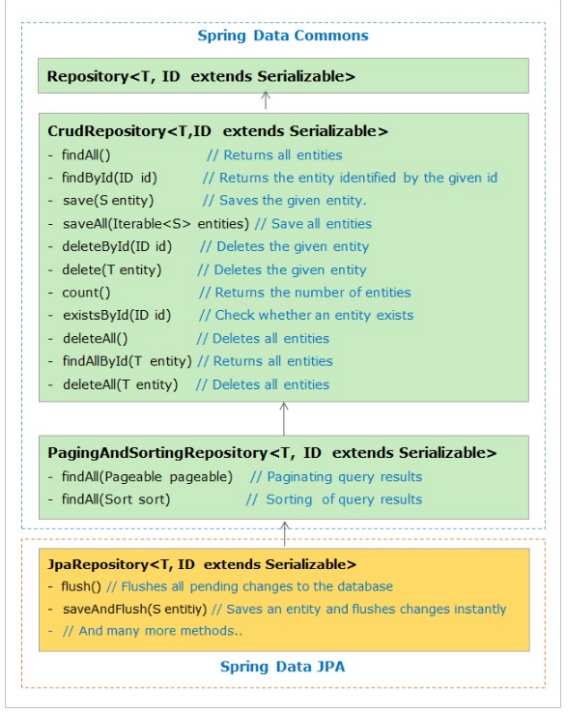
⇒ Spring data JPA CRUD

entity class, primary key datatype

```
@Repository
public interface CustomerRepository extends JpaRepository<Customer, Long> {

}
```

only need to extend the JpaRepository interface



⇒ Paging and Sorting

```
public interface PagingAndSortingRepository<T, ID extends Serializable> extends CrudRepository<T, ID> {
            Iterable<T> findAll(Sort sort);
            Page<T> findAll(Pageable pageable);
}
```

A Page object provides the data for the requested page as well as additional information like total result count, page index, and so on.

Steps to create pagination

Step 1

```
public interface CustomerRepository extends JpaRepository<Customer, Long> {

}
```

Step 2

```
//First argument "0" indicates first page and second argument 4 represents number of records.
Pageable pageable = PageRequest.of(0, 4);
```

In client code :-

Step 3

```
Page<Customer> customers = customerRepository.findAll(pageable);
```

Steps to create sorting

Step 1

```
public interface CustomerRepository extends JpaRepository<Customer, Long> {

}
```

In client code :-

Step 2

```
customerRepository.findAll(Sort.by(Sort.Direction.ASC,"name"));
```

In the Sort() method used above,

- The first parameter specifies the order of sorting i.e. is ascending order.
- The second parameter specifies the field value for sorting.

Demo to create paginat^n & sorting

Customer service interface

```
public interface CustomerService {

    public void insertCustomer(CustomerDTO Customer) ;
    Page<Customer> findAll(Pageable page);
    List<Customer> findAll(Sort sort);

}
```

Implimentation of same

Repository that extends JpaRepository

```
@Service("customerService")
public class CustomerServiceImpl implements CustomerService{
      @Autowired
      private CustomerRepository  repository;

    public void insertCustomer(CustomerDTO customer) {
        repository.saveAndFlush(CustomerDTO.prepareCustomerEntity(customer));
    }

    @Override
    public Page<Customer> findAll(Pageable page) {
        return repository.findAll(page);

    }
    @Override
    public List<Customer> findAll(Sort sort) {
        return repository.findAll(sort);
    }

}
```

⇒ Query Approach

previously we had only
↳ findAll
↳ findById(ID id)

1) But now,

- Query creation based on the method name.
- Query creation using @NamedQuery: JPA named queries through a naming convention.
- Query creation using @Query: annotate the query method with @Query.

Syntax

```
//method name where in <Op> is optional, it can be Gt,Lt,Ne,Between,Like etc..
findBy <DataMember><Op>
```

// method name

Ex:-

```
interface CustomerRepository extends JpaRepository<Customer, Long>{
        Customer findByAddress(String address); // method declared
}
```

// a programmer needs to write declaration only and spring will take care of implementation using keywords which are findby and address.

example

```
public interface CustomerRepository extends JpaRepository<Customer, Integer> {

    // Query record based on email
    // Equivalent JPQL: select c from Customer c where c.email=?
    Customer findByEmail(String email);

    // Query records based on LastName is like the provided last name
    // select c from Customer c where c.lastName LIKE CONCAT('%',?,'%')
    List<Customer> findByLastNameLike(String lastname);

    // Query records based on email or contact number
    // select c from Customer c where c.email=? or c.contactNumber=?
    List<Customer> findByEmailOrContactNumber(String email, String number);

    // Query records based on FirstName and city details. Following query creates the property traversal for city as Customer.address.city
    // select c from Customer c  where c.firstName=? and c.address.city=?
    List<Customer>  findByFirstNameAndAddress_City(String fname, String city);

    // Query records based on Last name and order by ascending based on first name
    // select c from Customer c where c.lastName=? order by c.firstName
    List<Customer> findByLastNameOrderByFirstNameAsc(String lastname);

    // Query records based on specified list of cities
    //select c from Customer c where c.address.city in ?1
    List<Customer> findByAddress_CityIn(Collection<String> cities);

    // Query records based if customer is active
    //select c from Customer c where c.active = true
    List<Customer> findByActiveTrue();

    // Query records based on creditPoints >= specified value
    //select c from Customer c where c.creditPoints >=?1
    List<Customer> findByCreditPointsGreaterThanEqual(int points) ;

    // Query records based on creditpoints between specified values
    //select c from Customer c where c.creditPoints between ?1 and ?2
    List<Customer> findByCreditPointsBetween(int point1, int point2)
}
```

Note: precedence order:-
1) @Query
2) @NameQuery
3) findBy methods

2) @NamedQuery

```
@Entity
//name starts with the entity class name followed by the method name separated by a dot.
@NamedQuery(name = "Customer.findByAddress", query = "select c from Customer c where c.address = ?1")
public class Customer {

    @Id
    @Column(name = "phone_no")
    private Long phoneNumber;
    private String name;
    private Integer age;
    private Character gender;
    private String address;
--------
}
```

```
public interface CustomerRepository extends JpaRepository<Customer, Long>{
    Customer findByAddress(String address);

}
```

Note :- @NamedQuery is valid for small no. of queries

3)

@Query annotation can be used to specify query details at repository interface methods instead of specifying at entity class. This will also reduce the entity class from persistence related information.

@Query is used to write flexible queries to fetch data.

```
public interface CustomerRepository extends JpaRepository<Customer, Long>{
//Query string is in JPQL
    @Query("select cus from Customer cus where cus.address = ?1")
    Customer findByAddress(String address);

}
```

Note:

JPQL:

JPQL is an object-oriented query language that is used to perform database operations on persistent entities. Instead of a database table, JPQL uses the entity object model to operate the SQL queries. Here, the role of JPA is to transform JPQL into SQL. Thus, it provides an easy platform for developers to handle SQL tasks.

- perform join operations
- update and delete data in a bulk
- perform an aggregate function with sorting and grouping clauses
- provide both single and multiple value result types

1)
```
Query query = em.createQuery("Select c.name from CustomerEntity c");
```

2)
```
Query query = em.createQuery( "update CustomerEntity SET planId=5 where phoneNumber=7022713766");
query.executeUpdate();
```

3)
```
Query query = em.createQuery( "delete from CustomerEntity where phoneNumber=7022713766");
query.executeUpdate();
```

⇒ Spring Transactions

When we implement transaction using JDBC API, it has few limitations like:

- Transaction related code is mixed with application code, hence it is difficult to maintain the code.
- Requires a lot of code modification to migrate between local and global transactions in an application.

Let us look at how the Spring transaction helps to overcome these limitations.

The Spring framework provides a common transaction API irrespective of underlying transaction technologies such as JDBC, Hibernate, JPA, and JTA.

- Spring supports both local and global transactions using a consistent approach with minimal modifications through configuration.
- Spring supports both declarative and programmatic transaction approaches.

Type	Definition	Implementation
Declarative Transaction	Spring applies the required transaction to the application code in a declarative way using a simple configuration. Spring internally uses Spring AOP to provide transaction logic to the application code.	• Using pure XML configuration • Using @Transactional annotation approach
Programmatic Transaction	The Required transaction is achieved by adding transaction-related code in the application code. This approach is used for more fine level control of transaction requirements. This mixes the transaction functionality with the business logic and hence it is recommended to use only based on the need.	• Using the TransactionTemplate (adopts the same approach as JdbcTemplate) • Using a PlatformTransactionManager implementation.

recomended

@Transactional annotation offers ease-of-use to define the transaction requirement at method, interface, or class level.

Through declarative transaction management, update (Customer customer) method can be executed in transaction scope using Spring.

```
public class CustomerServiceImpl {
    ----------------------
    @Transactional
    public void update(Customer customer) {
        //Method to update the current plan in Customer table
        customerDAO.update(customer);
        //Method to update the new plan details in Plan table
        planDAO.updatePlan(customer.getPlan());
    }

}
```

TransactionManager	Data Access Technique
DataSourceTransactionManager	JDBC
HibernateTransactionManager	Hibernate
JpaTransactionManager	JPA
JtaTransactionManager	Distributed transaction

```
public interface CustomerRepository extends JpaRepository<Customer,Long>{
@Override
@Transactional(readOnly = true)
public List<Customer> findAll();
// Further query method declarations
}
```

⇒ Update Operation in Spring data JPA

Yes, it can execute modifying queries such as update, delete or insert operations using @Query annotation along with @Transactional and @Modifying annotation at query method.

Example: Interface with a method to update the name of a customer based on the customer's address.

```
public interface CustomerRepository extends JpaRepository<Customer, Long> {
      @Transactional
      @Modifying(clearAutomatically = true)
      @Query("update Customer c set c.name = ?1 where c.address = ?2")
      void update(String name, String address);
}
```

@Modifying: This annotation will trigger @Query annotation to be used for an update operation instead of a query operation.

@Modifying(clearAutomatically = true): After executing modifying query, EntityManager might contain unrequired entities. It is a good practice to clear the EntityManager cache automatically by setting @Modifying annotation's clearAutomatically attribute to true.

@Transactional: Spring Data provided CRUD methods on repository instances that support transactional by default with read operation and by setting readOnly flag to true. Here, @Query is used for an update operation, and hence we need to override default readOnly behavior to read/write by explicitly annotating a method with @Transactional.

// safe way to use @Modifying

```
@Modifying(clearAutomatically=true, flushAutomatically=true)
```

Note :-

@Transactional annotation supports the following attributes:

Transaction isolation: The degree of isolation of this transaction with other transactions.

Transaction propagation: Defines the scope of the transaction.

Read-only status: A read-only transaction will not modify any data. Read-only transactions can be useful for optimization in some cases.

Transaction timeout: How long a transaction can run before timing out.

The default @Transactional settings are as follows:

@Transactional Attribute	Default values
Propagation	PROPOGATION_REQUIRED
Isolation level	ISOLATION_DEFAULT
Transaction Type	Read/Write
Timeout	Based on the underlying database settings
Rollback condition	Rollback on runtime exception and no rollback on checked exception

⇒ Custom Repository

Example: Let us consider a customer search scenario wherein customer details needs to be fetched based on name and address or gender or age.

Steps :-

The steps to implement the custom repository with a method to Retrieve customer records based on search criteria are as follows:

Step 1: Define an interface and an implementation for the custom functionality.

```
public interface ICustomerRepository {

    public List<Customer> searchCustomer(String name, String addr, Character gender, Integer age);
}
```

Step 2: Implement this interface using repository class as shown below:

```
public class CustomerRepositoryImpl implements ICustomerRepository{

    private EntityManagerFactory emf;

    @Autowired
    public void setEntityManagerFactory(EntityManagerFactory emf) {
        this.emf = emf;
    }

    @Override
    public List<Customer> searchCustomer(String name, String address, Character gender, Integer age) {
        EntityManager em = emf.createEntityManager();
        CriteriaBuilder builder = em.getCriteriaBuilder();

        CriteriaQuery<Customer> query = builder.createQuery(Customer.class);
        Root<Customer> root = query.from(Customer.class);

        Predicate cName = builder.equal(root.get("name"), name);
        Predicate cAddress = builder.equal(root.get("address"), address);

        Predicate exp1 = builder.and(cName, cAddress);

        Predicate cGender = builder.equal(root.get("gender"), gender);
        Predicate cAge = builder.equal(root.get("age"), age);

        Predicate exp2 = builder.or(cGender,cAge);

        query.where(builder.or(exp1, exp2));

        return em.createQuery(query.select(root)).getResultList();
    }
```

Step 3: Define repository interface extending the custom repository interface as shown below:

```
public interface CustomerRepository extends JpaRepository<Customer, Long>,ICustomerRepository{

}
```