SPRING DATA JPA

Instructor:

Table Content

Learning Goals

- **After the session, attendees will be able to:**

Understand Spring Data JPA Framework and its core technologies.

Section 1

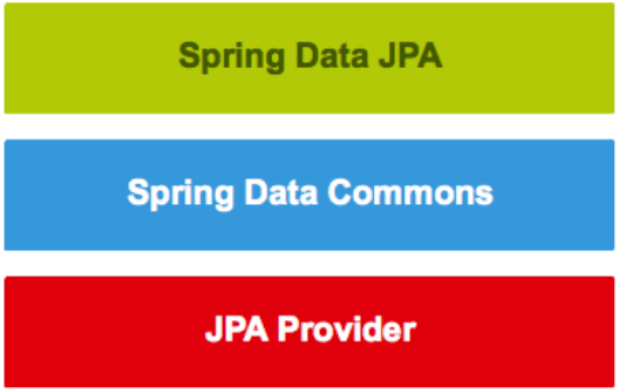
INTRODUCTION

Introduction

- **Spring Data** is a module of Spring Framework. The goal of Spring Data repository abstraction is to significantly **reduce the amount of boilerplate code required** to implement *data access layers* for various persistence stores.
- Java Persistence API (JPA) is Java's standard API specification for object-relational mapping. Spring Data JPA is a part of Spring Data and it supports **Hibernate 5**, **OpenJPA 2.4**, and **EclipseLink 2.6.1**.

What Spring Data JPA?

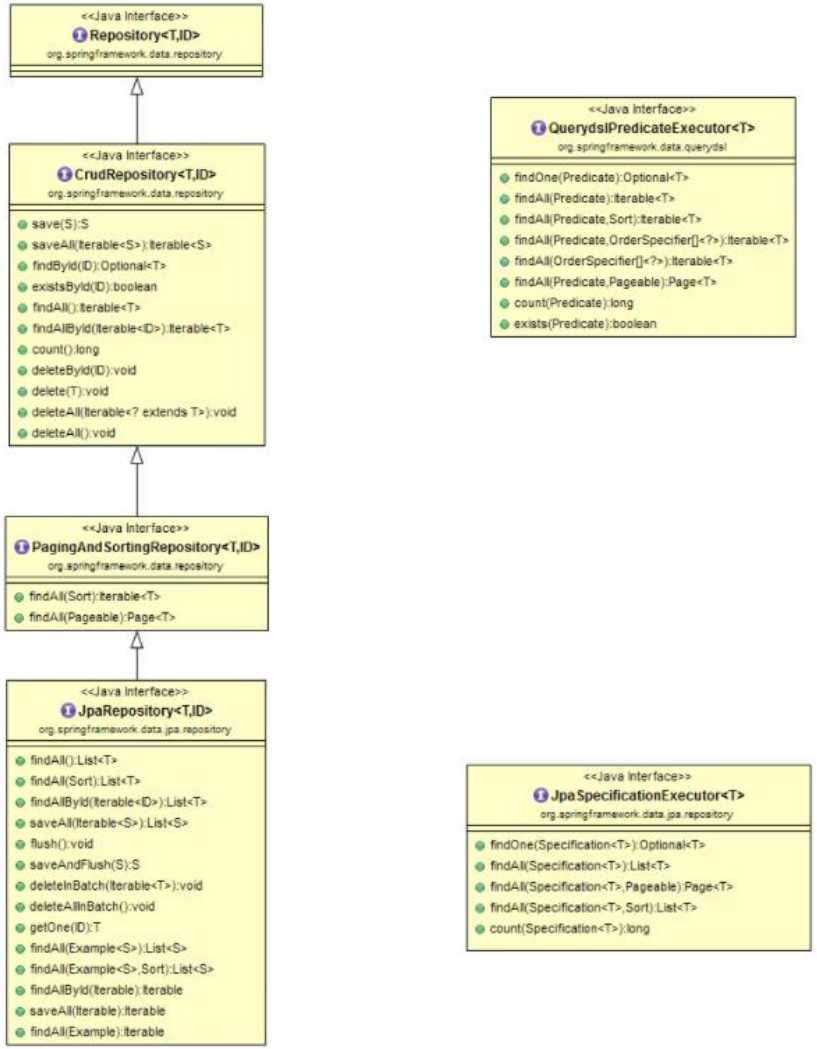
- **Spring Data JPA** is NOT a JPA provider.
- It is a library/framework that adds an extra layer of abstraction on the top of our JPA provider (like Hibernate).
- If we decide to use Spring Data JPA, the repository layer of our application contains three layers that are described in the following:



- ✓ *Spring Data JPA* provides support for creating JPA repositories by extending the Spring Data repository interfaces.
- ✓ *Spring Data Commons* provides the infrastructure that is shared by the datastore-specific Spring Data projects.
- ✓ *The JPA Provider* (like hibernate) implements the Java Persistence API.

Repositories/interfaces

❖ Spring Data Commons and Spring Data JPA



Repositories/interfaces

- It contains technology-neutral **repository interfaces** as well as a **metadata model** for persisting Java classes.
- Spring Data Commons project provides the following interfaces:
  - ✓ **Repository**<T, ID extends Serializable> interface
  - ✓ **CrudRepository**<T, ID extends Serializable> interface
  - ✓ **PagingAndSortingRepository**<T, ID extends Serializable> interface
  - ✓ **QueryDslPredicateExecutor** interface

Section 2

SPRING DATA JPA INTERFACE

Repository interface

- The `Repository<T, ID extends Serializable>` interface is a marker interface that has two purposes:
  - ✓ It captures the type of the managed entity and the type of the entity's id.
  - ✓ It helps the Spring container to discover the "concrete" repository interfaces during classpath scanning.
  - ✓ Let's look at the source code of the Repository interface.

```
package org.springframework.data.repository;

import org.springframework.stereotype.Indexed;

@Indexed
public interface Repository<T, ID> {

}
```

CrudRepository interface

- The *`CrudRepository<T, ID extends Serializable>` interface* provides CRUD operations for the managed entity.
- Let's look at the methods/APIs that the *CrudRepository* interface provides:

```
package org.springframework.data.repository;

import java.util.Optional;

@NoRepositoryBean
public interface CrudRepository < T, ID > extends Repository < T, ID > {

    <S extends T > S save(S entity);

    <S extends T > Iterable < S > saveAll(Iterable < S > entities);

    Optional < T > findById(ID id);

    boolean existsById(ID id);

    Iterable < T > findAll();

    Iterable < T > findAllById(Iterable < ID > ids);

    long count();

    void deleteById(ID id);

    void delete(T entity);

    void deleteAll();
}
```

CrudRepository interface

- Let's look at the usage of each method with description.
  - ✓ *long count()* - Returns the number of entities available.
  - ✓ *void delete(T entity)* - Deletes a given entity.
  - ✓ *void deleteAll()* - Deletes all entities managed by the repository.
  - ✓ *void deleteAll(Iterable<? extends T> entities)* - Deletes the given entities.
  - ✓ *void deleteById(ID id)* - Deletes the entity with the given id.
  - ✓ *boolean existsById(ID id)* - Returns whether an entity with the given id exists.
  - ✓ *Iterable findAll()* - Returns all instances of the type.
  - ✓ *Iterable findAllById(Iterable ids)* - Returns all instances of the type with the given IDs.
  - ✓ *Optional findById(ID id)* - Retrieves an entity by its id.
  - ✓ *save(S entity)* - Saves a given entity.
  - ✓ *Iterable saveAll(Iterable entities)* - Saves all given entities.

PagingAndSortingRepository inteface

- The *PagingAndSortingRepository<T, ID extends Serializable> interface* is an extension of ***CrudRepository*** to provide additional methods to retrieve entities using the pagination and sorting abstraction.

```
package org.springframework.data.repository;

import org.springframework.data.domain.Page;
import org.springframework.data.domain.Pageable;
import org.springframework.data.domain.Sort;

@NoRepositoryBean
public interface PagingAndSortingRepository < T, ID > extends CrudRepository < T, ID > {

    /**
     * Returns all entities sorted by the given options.
     *
     * @param sort
     * @return all entities sorted by the given options
     */
    Iterable < T > findAll(Sort sort);

    /**
     * Returns a {@link Page} of entities meeting the paging restriction provided in the
                                                                    {@code Pageable} object.
     *
     * @param pageable
     * @return a page of entities
     */
    Page < T > findAll(Pageable pageable);
}
```

QueryDslPredicateExecutor interface

- The *QueryDslPredicateExecutor* interface is not a “repository interface”. It declares the methods that are used to retrieve entities from the database by using *QueryDsl* Predicate objects.
- Let's look at the methods/APIs that the *QueryDslPredicateExecutor* interface provides:

```
package org.springframework.data.querydsl;

import java.util.Optional;

import org.springframework.data.domain.Page;
import org.springframework.data.domain.Pageable;
import org.springframework.data.domain.Sort;

import com.querydsl.core.types.OrderSpecifier;
import com.querydsl.core.types.Predicate;

public interface QuerydslPredicateExecutor < T > {

    Optional < T > findOne(Predicate predicate);

    Iterable < T > findAll(Predicate predicate);

    Iterable < T > findAll(Predicate predicate, Sort sort);

    Iterable < T > findAll(Predicate predicate, OrderSpecifier << ? > ...orders);

    Iterable < T > findAll(OrderSpecifier << ? > ...orders);

    Page < T > findAll(Predicate predicate, Pageable pageable);

    long count(Predicate predicate);

    boolean exists(Predicate predicate);
}
```

Spring Data JPA Interfaces

- *Spring Data JPA* module deals with enhanced support for JPA based data access layers.
- Spring Data JPA project provides the following interfaces:
  - ✓ `JpaRepository<T, ID extends Serializable> interface`
  - ✓ `JpaSpecificationExecutor interface`
- **JpaRepository interface:**
  - ✓ The *`JpaRepository<T, ID extends Serializable>`* `interface` is a JPA specific repository interface that combines the methods declared by the common repository interfaces behind a single interface.
  - ✓ Let's look at the methods/APIs that the *JpaRepository* interface provides:

```
package org.springframework.data.jpa.repository;

import java.util.List;

import javax.persistence.EntityManager;

import org.springframework.data.domain.Example;
import org.springframework.data.domain.Sort;
import org.springframework.data.repository.NoRepositoryBean;
import org.springframework.data.repository.PagingAndSortingRepository;
import org.springframework.data.repository.query.QueryByExampleExecutor;

@NoRepositoryBean
public interface JpaRepository < T, ID > extends PagingAndSortingRepository < T, ID > ,
                                                 QueryByExampleExecutor < T > {

    List < T > findAll();
```

JpaRepository interface

```
    List < T > findAll(Sort sort);

    List < T > findAllById(Iterable < ID > ids);

    <S extends T > List < S > saveAll(Iterable < S > entities);

    void flush();

    <S extends T > S saveAndFlush(S entity);

    void deleteInBatch(Iterable < T > entities);

    void deleteAllInBatch();

    T getOne(ID id);

    @Override <
    S extends T > List < S > findAll(Example < S > example);

    @Override <
    S extends T > List < S > findAll(Example < S > example, Sort sort);
}
```

JpaSpecificationExecutor interface

- The *JpaSpecificationExecutor* interface is not a “repository interface”. It declares the methods that are used to retrieve entities from the database by using Specification objects that use the JPA criteria API.
- Let's look at the methods/APIs that the *JpaSpecificationExecutor* interface provides:

```
package org.springframework.data.jpa.repository;

import java.util.List;
import java.util.Optional;

import org.springframework.data.domain.Page;
import org.springframework.data.domain.Pageable;
import org.springframework.data.domain.Sort;
import org.springframework.data.jpa.domain.Specification;
import org.springframework.lang.Nullable;

public interface JpaSpecificationExecutor<T> {

 Optional<T> findOne(@Nullable Specification<T> spec);

 List<T> findAll(@Nullable Specification<T> spec);

 Page<T> findAll(@Nullable Specification<T> spec, Pageable pageable);

 List<T> findAll(@Nullable Specification<T> spec, Sort sort);

 long count(@Nullable Specification<T> spec);
}
```

Pagination and Sorting

- Example to access our *Product*s, we'll need a *ProductRepository*:

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

    List<Product> findAllByPrice(double price, Pageable pageable);
}
```

- ✓ Create or obtain a *PageRequest* object, which is an implementation of the *Pageable* interface
- ✓ Pass the *PageRequest* object as an argument to the repository method we intend to use
- ✓ We can create a *PageRequest* object by passing in the requested page number and the page size. Here **the page count starts at zero:**

```
Pageable firstPageWithTwoElements = PageRequest.of(0, 2);

Pageable secondPageWithFiveElements = PageRequest.of(1, 5);
```

Pagination and Sorting

- Similarly, to just have our query results sorted, we can simply pass an instance of *Sort* to the method:

```
Page<Product> allProductsSortedByName =
                        productRepository.findAll(Sort.by("name"));
```

- What if we want to **both sort and page our data?**

```
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")));
```

Section 3

HOW TO USE SPRING DATA JPA INTERFACES

How to Use Spring Data JPA interfaces

- *(1) Create a repository interface and extend one of the repository interfaces provided by Spring Data.*

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

}
```

- *(2) Add custom query methods to the created repository interface (if we need them that is).*

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

    long deleteByLastname(String lastname);

    List<User> removeByLastname(String lastname);

    long countByLastname(String lastname);
}
```

How to Use Spring Data JPA interfaces

- *(3) Set up Spring to create proxy instances for those interfaces, either with JavaConfig or with XML configuration.*
  - ✓ To use Java configuration, create a class similar to the following:

```
import org.springframework.data.jpa.repository.config.EnableJpaRepositories;

@EnableJpaRepositories
public class Config {}
```

  - ✓ To use XML configuration, define a bean similar to the following:

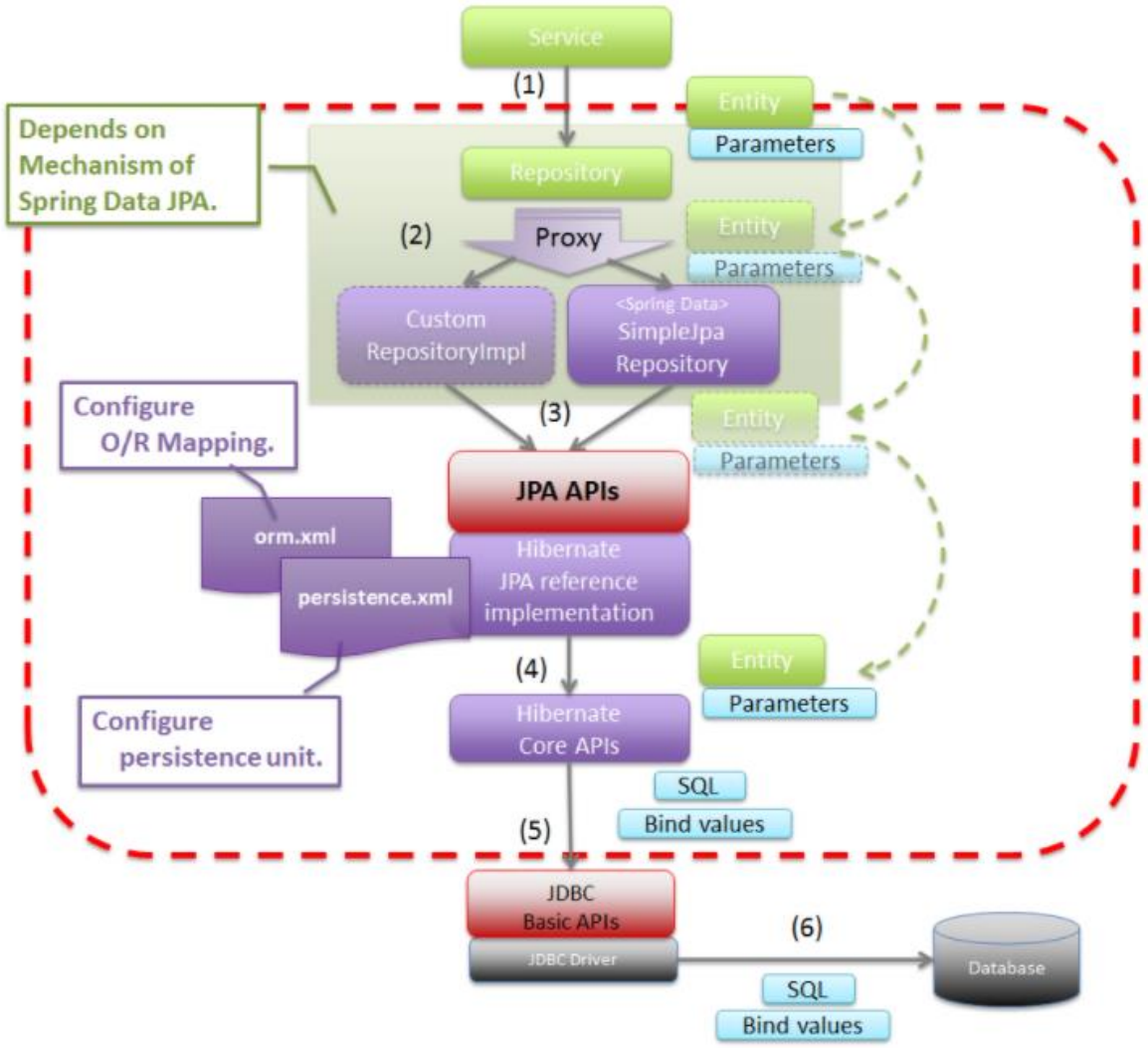
```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xmlns:jpa="http://www.springframework.org/schema/data/jpa"
   xsi:schemaLocation="http://www.springframework.org/schema/beans
     http://www.springframework.org/schema/beans/spring-beans.xsd
     http://www.springframework.org/schema/data/jpa
     http://www.springframework.org/schema/data/jpa/spring-jpa.xsd">

   <jpa:repositories base-package="com.acme.repositories"/>

</beans>
```

How to Use Spring Data JPA interfaces

- *(4) Inject the repository interface to another component and use the implementation that is provided automatically by Spring.*

```
@Service
public class CustomerServiceImpl implements CustomerService {

    @Autowired
    private CustomerRepository customerRepository;

    @Override
    @Transactional
    public List < Customer > getCustomers() {
        return customerRepository.findAll();
    }

    @Override
    @Transactional
    public void saveCustomer(Customer theCustomer) {
        customerRepository.save(theCustomer);
    }

    @Override
    @Transactional
    public Customer getCustomer(int id) throws ResourceNotFoundException {
        return customerRepository.findById(id).orElseThrow(
            () - > new ResourceNotFoundException(id));
    }

    @Override
    @Transactional
    public void deleteCustomer(int theId) {
        customerRepository.deleteById(theId);
    }
}
```

Basic Spring Data JPA Flow



Section 4

QUERY METHODS

Query lookup strategies

- The JPA module supports defining a *query manually as String* or have it being derived from the method name.
- **Query creation:**
  - ✓ The query builder mechanism built into Spring Data repository infrastructure is useful for **building constraining queries over entities** of the repository.
  - ✓ The mechanism strips the prefixes *find…By*, *read…By*, *query…By*, *count…By*, and *get…By* from the method and starts parsing the rest of it.
  - ✓ You can define **conditions** on entity properties and concatenate them with And and Or.
  - ✓ Examples: **Query creation from method names**

```
public interface PersonRepository extends Repository<User, Long> {

    List<Person> findByEmailAddressAndLastname(EmailAddress emailAddress,
                                               String lastname);

    // Enables the distinct flag for the query
    List<Person> findDistinctPeopleByLastnameOrFirstname(String lastname,
                                               String firstname);
```

Query creation

- Examples: **Query creation from method names**

```
    List<Person> findPeopleDistinctByLastnameOrFirstname(String lastname,
                                                         String firstname);

    // Enabling ignoring case for an individual property
    List<Person> findByLastnameIgnoreCase(String lastname);

    // Enabling ignoring case for all suitable properties
    List<Person> findByLastnameAndFirstnameAllIgnoreCase(String lastname,
                                                         String firstname);

    // Enabling static ORDER BY for a query
    List<Person> findByLastnameOrderByFirstnameAsc(String lastname);

    List<Person> findByLastnameOrderByFirstnameDesc(String lastname);
}
```

Query creation

❖ Special parameter handling:

- ✓ Besides that the infrastructure will recognize certain specific types like **Pageable** and **Sort** to apply pagination and sorting to your queries dynamically.

```
Page<User> findByLastname(String lastname, Pageable pageable);

List<User> findByLastname(String lastname, Sort sort);

List<User> findByLastname(String lastname, Pageable pageable);
```

Query creation

- **Query generated**:
  - ✓ Query creation from method names

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

    List<User> findByEmailAddressAndLastname(String emailAddress,
                        String lastname);
    }
```

  - ✓ We will create a query using the JPA criteria API from this but essentially this translates into the following query:

```
select u from User u where u.emailAddress = ?1
                                        and u.lastname = ?2
```

Query creation

❖ Supported keywords inside method names

Keyword	Sample	JPQL snippet
And	findByLastnameAndFirstname	… where x.lastname = ?1 and x.firstname = ?2
Or	findByLastnameOrFirstname	… where x.lastname = ?1 or x.firstname = ?2
Is,Equals	findByFirstname,findByFirstnameIs, findByFirstnameEquals	… where x.firstname = 1?
Between	findByStartDateBetween	… where x.startDate between 1? and ?2
LessThan	findByAgeLessThan	… where x.age < ?1
LessThanEqual	findByAgeLessThanEqual	… where x.age <= ?1
GreaterThan	findByAgeGreaterThan	… where x.age > ?1
GreaterThanEqual	findByAgeGreaterThanEqual	… where x.age >= ?1
After	findByStartDateAfter	… where x.startDate > ?1
…	…	…

Using JPA NamedQueries

❖ Annotation configuration

- ✓ Annotation configuration has the advantage of not needing another configuration file to be edited, probably lowering maintenance costs.
- ✓ You pay for that benefit by the need to recompile your domain class for every new query declaration.
- ✓ **Annotation based named query configuration**

```
@Entity
@Table(name = "USERS")
@NamedQuery(name = "User.findByEmailAddress",
  query = "select u from User u where u.emailAddress = ?1")
public class User {

}
```

Using JPA NamedQueries

Annotation configuration

- ✓ **Declaring interfaces:**
- ✓ *To allow execution of these named queries all you need to do is to specify the UserRepository as follows:*

```
public interface UserRepository extends
                                JpaRepository<User, Long> {
    List<User> findByLastname(String lastname);

    User findByEmailAddress(String emailAddress);

}
```

Using @Query

- Using **named queries** to declare queries for entities is a valid approach and works fine for a **small number of queries**.
- As the queries themselves are tied to the Java method that executes them you actually can bind them directly using the Spring Data JPA **@Query** annotation rather than annotating them to the domain class.
- This will free the domain class from *persistence specific information* and *co-locate the query* to the repository interface.
- **Declare query at the query method using @Query**

```
public interface UserRepository extends
                            JpaRepository<User, Long> {
    @Query("SELECT u FROM User u WHERE u.emailAddress = ?1")
    User findByEmailAddress(String emailAddress);

}
```

SUMMARY

1. Introduction
2. Spring Data JPA Interface
3. How to Use Spring Data JPA interfaces
4. Query methods

Thank you