503111

JAVA TECHNOLOGY

CHAPTER 7: Spring Data JPA

Outline

- Spring Data Overview
- Let's get started with plain JPA + Spring
- Refactoring to Spring Data JPA
- Spring Data repository abstraction
- Advanced topics (Specification/ QueryDSL)
- JPA Spring Configurations

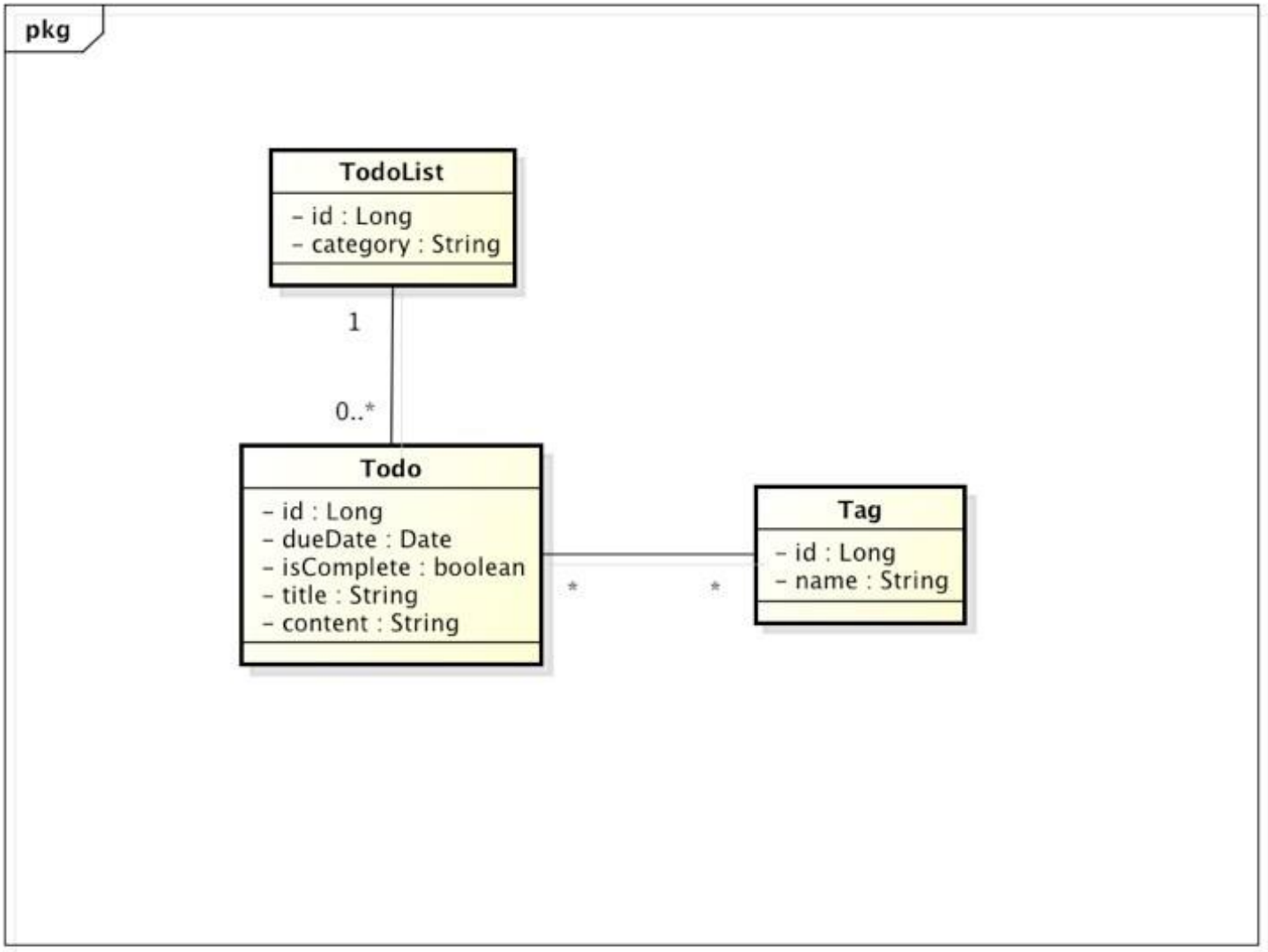
Spring Data Overview

- Abstracts away basic data management concepts
- Support for RDB, NoSQL: Graph, Key-Value and Map-Reduce types.
- Current implementations
  - JPA/JDBC
  - Hadoop
  - GemFire
  - REST
  - Redis/Riak
  - MongoDB
  - Neo4j
  - Blob (AWS S3, Rackspace, Azure,...)
- Plan for HBase and Cassandra

Let's get started

- Plain JPA + Spring

Overview of the Example



The Domain Entities

```
@Entity
public class TodoList {
     @Id
     @GeneratedValue(strategy = GenerationType.AUTO)
     private Long id;

     private String category;

// … methods omitted
}
```

```
@Entity
public class Tag {
     @Id
     @GeneratedValue(strategy = GenerationType.AUTO)
     private Long id;

     @ManyToMany
     private List<Todo> todos;

     private String name;

// … methods omitted
}
```

```
@Entity
public class Todo {

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

     @ManyToOne
     private TodoList todoList;

     @Temporal(TemporalType.DATE)
     private Date dueDate;

     private Boolean isComplete;

     private String title;


     @ManyToMany(cascade = { CascadeType.ALL },mappedBy="todos" )
     private List<Tag> tags;

   // … methods omitted
}
```

The DAO

```
@Repository
public class TodoDAO {

  @PersistenceContext
  private EntityManager em;

  @Override
  @Transactional
  public Todo save(Todo todo) {

    if (todo.getId() == null) {
      em.persist(todo);
      return todo;
    } else {
      return em.merge(todo);
    }
  }

  @Override
  public List<Todo> findByTodoList(TodoList todoList) {

    TypedQuery query = em.createQuery("select t from Todo a where t.todoList = ?1", Todo.class);
    query.setParameter(1, todoList);

    return query.getResultList();
  }

}
```

The Domain Service

```
@Service
@Transactional(readOnly = true)
class TodoListServiceImpl implements TodoListService {

  @Autowired
  private TodoDAO todoDAO;

  @Override
  @Transactional
  public Todo save(Todo todo) {
      return todoDAO.save(todo);
  }

  @Override
  public List<Todo> findTodos(TodoList todoList) {
      return todoDAO.findByTodoList(todoList);
  }


}
```

Pains

- A lot of code in the persistent framework and DAOs.
  1. Create a GenericDAO interface to define generic CRUD.
  2. Implement GenericDAO interface to an Abstract GenericDAO.
  3. Define DAO interface for each Domain entity (e.g., TodoDao).
  4. Implement each DAO interface and extend the Abstract GenericDAO for specific DAO (e.g., HibernateTodoDao)
- Still some duplicated Code in concrete DAOs.
- JPQL is not type-safe query.
- Pagination need to handle yourself, and integrated from MVC to persistent layer.
- If hybrid databases (ex, MySQL + MongoDB) are required for the system, it's not easy to have similar design concept in the architecture.

Refactoring to Spring Data JPA

- Refactoring..... Refactoring..... Refactoring.....

The Repository (DAO)

```
@Transactional(readOnly = true)
public interface TodoRepository extends JpaRepository<Todo, Long> {

  List<Todo> findByTodoList(TodoList todoList);
}
```

The Service

```
@Service
@Transactional(readOnly = true)
class TodoListServiceImpl implements TodoListService {

  @Autowired
  private TodoRepository repository;


  @Override
  @Transactional
  public Todo save(Todo todo) {
      return repository.save(todo);
  }

  @Override
  public List<Todo> findTodos(TodoList todoList) {
      return repository.findByTodoList(todoList);
  }

}
```

Summary of the refactoring

- 4 easy steps

1.Declare an interface extending the specific Repository sub-interface and type it to the domain class it shall handle.

```
public interface TodoRepository extends JpaRepository<Todo, Long> { … }
```

2.Declare query methods on the repository interface.

```
List<Todo> findByTodoList(TodoList todoList);
```

3.Setup Spring configuration.

```
<beans:beans xmlns:beans="http://www.springframework.org/schema/beans"
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xmlns="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">

    <repositories base-package="com.twjug.repositories" />

</beans>
```

4.Get the repository instance injected and use it.

```
public class SomeClient {

    @Autowired
    private TodoRepository repository;

    public void doSomething() {
        List<Todo> todos = repository.findByTodoList(todoList);
    }
```

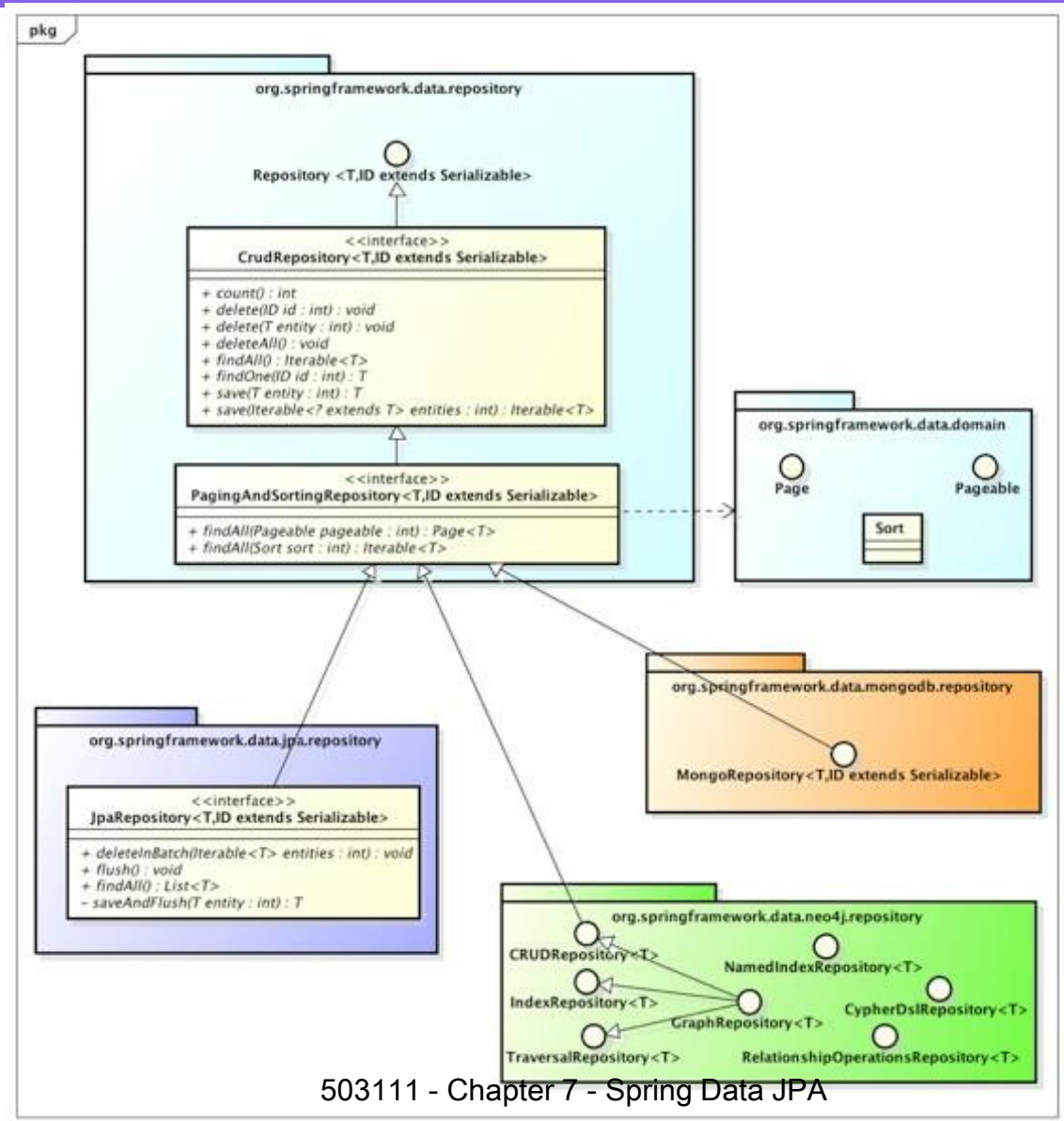
Spring Data Repository Abstraction

- The goal of the repository abstraction of Spring Data is to reduce the effort to implement data access layers for various persistence stores significantly.
- The central marker interface
  - Repository<T, ID extends Serializable>
- Borrow the concept from Domain Driven Design.

Domain Driven Design

- DDD main approaches
  - Entity
  - Value Object
  - Aggregate
  - **Repository**

Abstraction Class Diagram



Query Lookup Strategy

```
<jpa:repositories base-package="com.twjug.repositories"
 query-lookup-strategy="create-if-not-found"/>
```

- create
- use-declared-query
- create-if-not-found

Strategy: CREATE

- Split your query methods by prefix and property 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
Between	findByStartDateBetween	… where x.startDate between 1? and ?2
LessThan	findByAgeLessThan	… where x.age < ?1
GreaterThan	findByAgeGreaterThan	… where x.age > ?1
After	findByStartDateAfter	… where x.startDate > ?1
Before	findByStartDateBefore	… where x.startDate < ?1
IsNull	findByAgeIsNull	… where x.age is null
IsNotNull,NotNull	findByAge(Is)NotNull	… where x.age not null
Like	findByFirstnameLike	… where x.firstname like ?1
NotLike	findByFirstnameNotLike	… where x.firstname not like ?1
StartingWith	findByFirstnameStartingWith	… where x.firstname like ?1 (parameter bound with prepended %)
ngWith	findByFirstnameEndingWith	… where x.firstname like ?1 (parameter bound with appended %)
Containing	findByFirstnameContaining	… where x.firstname like ?1 (parameter bound wrapped in %)
OrderBy	findByAgeOrderByLastnameDesc	… where x.age = ?1 order by x.lastname desc
Not	findByLastnameNot	… where x.lastname <> ?1
In	findByAgeIn(Collection<Age> ages)	… where x.age in ?1
NotIn	findByAgeNotIn(Collection<Age> age)	… where x.age not in ?1
True	findByActiveTrue()	… where x.active = true
False	findByActiveFalse()	… where x.active = false

Strategy: CREATE

- Property expressions: traversal

```
Ex, if you want to traverse: x.address.zipCode
```

```
You can...
List<Person> findByAddressZipCode(ZipCode zipCode);
```

```
Better way
List<Person> findByAddress_ZipCode(ZipCode zipCode);
```

Strategy: USE_DECLARED_QUERY

- @NamedQuery

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

}
```

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

  List<User> findByLastname(String lastname);

  User findByEmailAddress(String emailAddress);
}
```

Strategy: USE_DECLARED_QUERY

- @Query

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

  @Query("select u from User u where u.emailAddress = ?1")
  User findByEmailAddress(String emailAddress);
}
```

From Spring Data JPA 1.1.0, native SQL is allowed to be executed.
Just set nativeQuery flag to true.
But, not support pagination and dynamic sort

Strategy: USE_DECLARED_QUERY

- @Modify

```
@Modifying
@Query("update User u set u.firstname = ?1 where u.lastname = ?2")
int setFixedFirstnameFor(String firstname, String lastname);
```

```
@Modifying
@Query("update User u set u.firstname = :firstName where u.lastname = :lastName")
int setFixedFirstnameFor(@Param("firstName") String firstname, @Param("lastName") String lastname);
```

Advanced Topics

- JPA Specification
- QueryDSL

Before Using JPA Specification

```
todoRepository.findUnComplete();
todoRepository.findUnoverdue();
todoRepository.findUncompleteAndUnoverdue();
todoRepository.findOOOXXXX(zz);
……
……
```

DDD: The Specification

Problem: Business rules often do not fit the responsibility of any of the obvious Entities or Value Objects, and their variety and combinations can overwhelm the basic meaning of the domain object. But moving the rules out of the domain layer is even worse, since the domain code no longer expresses the model.

Solution: Create explicit predicate-like Value Objects for specialized purposes. A Specification is a predicate that determines if an object does or does not satisfy some criteria.

Spring Data JPA Specification API

```
public interface Specification<T> {
  Predicate toPredicate(Root<T> root, CriteriaQuery<?> query,
            CriteriaBuilder builder);
}
```

Then, you can use one Repository method for all kind of queries which defined by Specification

```
List<T> readAll(Specification<T> spec);
```

Examples for JPA Specification

- Code
- JPA Metamodel generation with Maven

Overview of Querydsl

- Querydsl is a framework which enables the construction of type-safe SQL-like queries for multiple backends in Java.
- It supports JPA, JDO, SQL, Lucene, Hibernate Search, Mongodb, Java Collections, Scala。

Querydsl

- As an alternative to JPA Criteria API

JPA Criteria API

```
{
    CriteriaQuery query = builder.createQuery();
    Root<Person> men = query.from( Person.class );
    Root<Person> women = query.from( Person.class );
    Predicate menRestriction = builder.and(
        builder.equal( men.get( Person_.gender ), Gender.MALE ),
        builder.equal( men.get( Person_.relationshipStatus ),
            RelationshipStatus.SINGLE ));
    Predicate womenRestriction = builder.and(
        builder.equal( women.get( Person_.gender ), Gender.FEMALE ),
        builder.equal( women.get( Person_.relationshipStatus ),
            RelationshipStatus.SINGLE ));
    query.where( builder.and( menRestriction, womenRestriction ) );
    ......
}
```

Querydsl

```
{
    JPAQuery query = new JPAQuery(em);
    QPerson men = new QPerson("men");
    QPerson women = new QPerson("women");
    query.from(men, women).where(
      men.gender.eq(Gender.MALE),
      men.relationshipStatus.eq(RelationshipStatus.SINGLE),
      women.gender.eq(Gender.FEMALE),
      women.relationshipStatus.eq(RelationshipStatus.SINGLE));
}
```

Querydsl Example

- Code
- Maven integration

Configurations

- Before Spring 3.1, you need
  - META-INF/persistence.xml
  - orm.xml
  - Spring configuration

Configurations in Spring 3.1

```
<!-- Activate Spring Data JPA repository support -->
     <jpa:repositories  base-package="com.humus.domain.repo"  />



<!-- Declare a JPA entityManagerFactory -->
     <bean  id="entityManagerFactory"  class="org.springframework.orm.jpa.LocalContainerEntityManagerFactoryBean">
        <property name="dataSource" ref="dataSource" />
        <property name="jpaVendorAdapter">
            <bean   class="org.springframework.orm.jpa.vendor.HibernateJpaVendorAdapter">
                <property name="showSql" value="true" />
            </bean>
         </property>
         <property name="jpaProperties">
             <props>
                 <prop  key="hibernate.dialect">org.hibernate.dialect.H2Dialect</prop>
                 <prop key="hibernate.hbm2ddl.auto">create</prop>
             </props>
          </property>

          <property name="packagesToScan">
                  <list>
                      <value>com.humus.domain.entity</value>
                  </list>
          </property>
    </bean>
```

No persistence.xml any more~