

Spring Data (JPA Focused)

“Code with Passion!”



Topics

- What is and Why Spring Data?
- Spring Data JPA
- Spring Data Repository interfaces
- Step by step of building Spring Data JPA application
- Paging and Sorting
- Query generation strategies

What is & Why Spring Data?

Data is new frontier in Computing

- The amount of data is exploding
 - > Enterprise
 - > Social networking
 - > Personal data
- The types of data are varying
 - > Relational data (table-driven & structured) is not the only data type
 - > Most data is now unstructured
- Data management is a new strategic tool in business

Types of Data Stores

- Relational
 - > JDBC, JPA
- Document store
 - > MongoDB, CouchDB
- Column oriented
 - > HBase, Cassandra
- Graph
 - > Neo4j, OrientDB
- Data grids
 - > GemFire, Coherence
- Key value
 - > Redis, Riak
- Big data
 - > Hadoop

What is Spring Data?

- Spring Data is an umbrella project which aims
 - > to provide familiar and consistent Spring-based programming model for new data stores
 - > while retaining data store-specific features and capabilities
- Sub-projects
 - > Spring Data JPA
 - > Spring Data MongoDB
 - > Spring Data Neo4J
 - > Spring Data Redis
 - > Spring Data Solr
 - > Spring for Hadoop
 - > Spring Data Gemfire
 - > Spring Data REST
 - > Spring Data JDBC extensions

Spring Data JPA

Features of Spring Data JPA

- Sophisticated support to build repositories based on Spring and JPA
- Support for Querydsl predicates and thus type-safe JPA queries
- Transparent auditing of domain class
- Pagination support, dynamic query execution, ability to integrate custom data access code
- Validation of @Query annotated queries at bootstrap time
- Support for XML based entity mapping
- JavaConfig based repository configuration by introducing @EnableJpaRepositories.

Classes you need in “Plain” Spring JPA

- Domain classes
- Repository (Dao) class for each domain class
 - > Interface
 - > Implementation
- Service class
 - > Uses the repository class

Pain Points of “Plain” Spring JPA

- Too much boiler plate coding
 - > You have to write same boiler plate code over and over
- Not type safe
 - > JPQL is not type safe
- You have to deal with pagination and sorting yourself
 - > More boiler plate code for pagination and sorting
- Code is not portable to other data storage
 - > What if I want to use MongoDB?

Players in Spring Data JPA

- Domain class
- Repository (Dao) class for each domain class
 - > Interface
 - > No implementation class need to be written - implementation classes are generated by Spring Data during runtime
- Service
 - > Uses the repository class

Spring Data Repository Interfaces

Why Spring Data Repository Interfaces?

- Provided by the **Spring Data Commons project** and extended by the data store specific sub projects
- Provides consistent data access layer over various data stores
 - > Enhances the portability of the code
- No need to write data store specific implementation
 - > The data store specific implementation is provided by the container during runtime

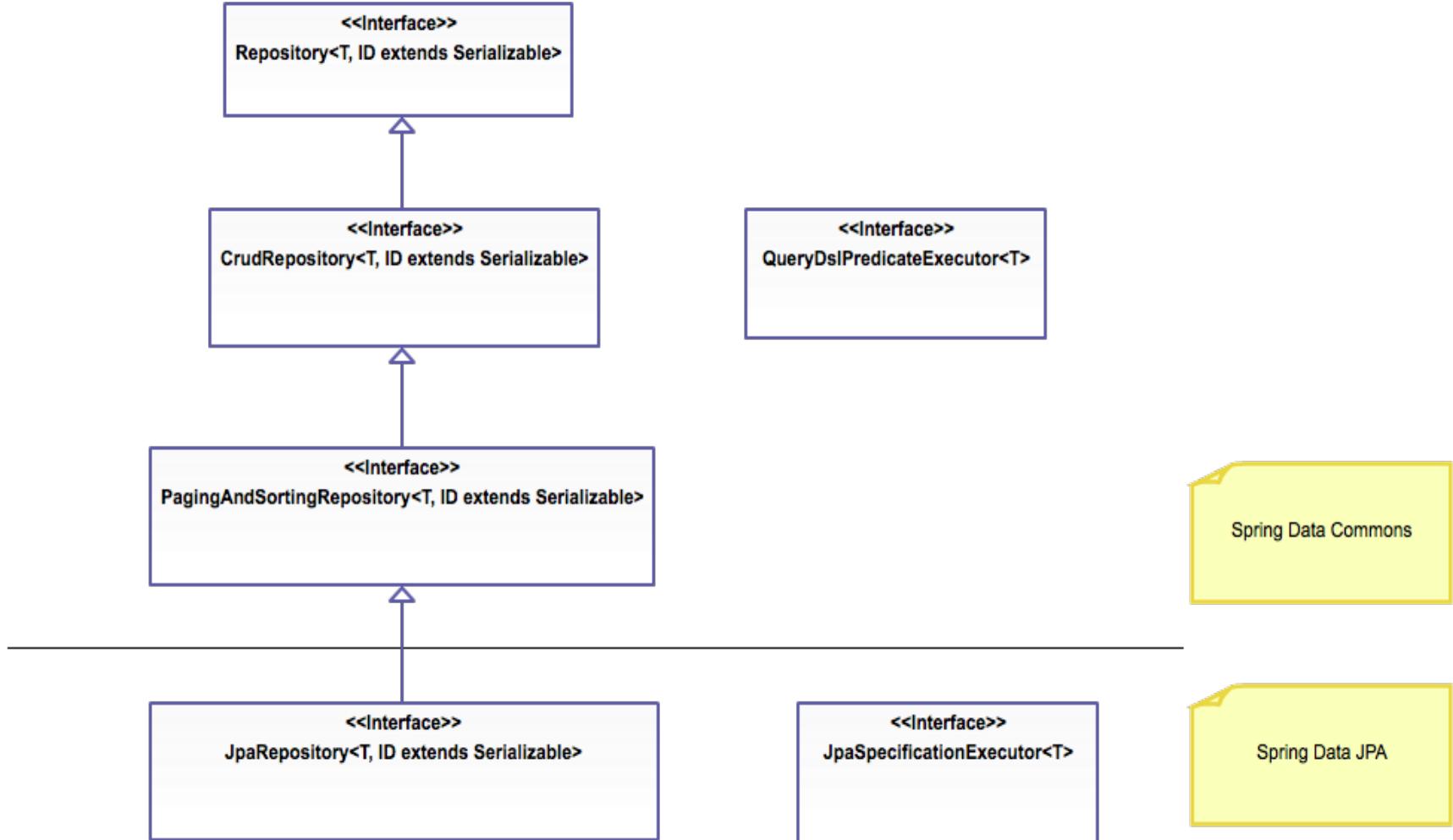
Common Repository Interfaces

- `Repository<T, ID extends Serializable>`
 - > T - the type of the managed entity
 - > ID - the type of the entity's id
- `CrudRepository<T, ID extends Serializable>`
 - > Provides CRUD methods for the managed entity
 - > `save(S entity)`, `findOne(ID id)`, `findAll()`, `count()`, `delete(ID id)`,
`deleteAll()`, `exists(ID id)`
- `PagingAndSortingRepository<T, ID extends Serializable>`
 - > Supports pagination and sorting
 - > `findAll(Pageable pageable)`, `findAll(Sort sort)`

Data Store specific Interfaces

- JpaRepository<T, ID extends Serializable>
 - > Provides JPA specific methods such as flushing the persistence context and delete record in a batch
- MongoRepository<T, ID extends Serializable>
 - > Mongo specific Repository interface

Repository Interface Hierarchy



Which Interface to use?

- If you need JPA specific features, use `JpaRepository<T, ID extends Serializable>`
- Otherwise, use `CrudRepository<T, ID extends Serializable>` or `PagingAndSortingRepository<T, ID extends Serializable>`
 - > Enhanced portability of code

Step by Step Building Spring Data JPA Application

Spring Data JPA Query Methods

- Spring Data JPA also allows you to define other query methods by simply declaring their method signature
 - > findByXxx(aaa)
 - > findByXxxAndYyy(aaa, bbb)
 - > findByXxxOrYyy(aaa, bbb)
 - > findByXxxStartingWith(aaa)
 - > findByXxxNot(aaa)
 - > findByXxxIn(Collection<E> aaa)
 - > ...

Steps for Spring Data JPA Application

- Step #1: Create a repository interface
 - > Extend one of the repository interfaces, most likely *JpaRepository* interface
- Step #2: Add custom query methods
 - > To the created repository interface
 - > *findByLastnameAndFirstname()*, for example
- Step #3: Inject the repository interface to another component (for example, a Service object)
 - > The service object will use implementation that is provided automatically by Spring Data

Step #1: Create Custom Repository Interface

```
package com.jpassion;  
  
import org.springframework.data.jpa.repository.JpaRepository  
  
// By extending JpaRepository, CustomerRepository inherits several methods  
// for working with Customer persistence, including methods for saving, deleting,  
// and finding Customer entities.  
public interface CustomerRepository extends JpaRepository<Customer, Long> {  
}
```

Step #2: Add Custom Query Method

```
package com.jpassion;

import java.util.List;
import org.springframework.data.jpa.repository.JpaRepository

// By extending JpaRepository, CustomerRepository inherits several methods
// for working with Customer persistence, including methods for saving, deleting,
// and finding Customer entities.
public interface CustomerRepository extends JpaRepository<Customer, Long> {

    // Define custom query methods by simply declaring their method signature
    List<Customer> findByLastName(String lastName);
}
```

Step #3: Inject Repository object

```
@service
public class CustomerServiceImpl implements CustomerService {
    private final CustomerRepository customerRepository;

    @Autowired
    public CustomerServiceImpl(CustomerRepository customerRepository) {
        this.customerRepository = customerRepository;
    }

}
```

Lab:

Exercise 1: Simple Spring Data JPA Application

Exercise 2: Converting Spring JPA application to use Spring Data JPA

4944_spring4_data_jpa.zip



Paging and Sorting

Paging and Sorting

- PagingAndSortingRepository interface definition

```
public interface PagingAndSortingRepository<T, ID extends Serializable>
    extends CrudRepository<T, ID> {
```

```
    Iterable<T> findAll(Sort sort);
    Page<T> findAll(Pageable pageable);
}
```

additional methods to retrieve entities using the pagination and sorting abstraction

- If you want paging and sorting, use PagingAndSortingRepository interface

```
// Example
public interface CustomerRepository
    extends PagingAndSortingRepository<Customer, Long> {
```

```
}
```

PageRequest Class

- *PageRequest* - is an implementation of *Pageable* interface

```
public PageRequest(int page,  
                   int size,  
                   Sort.Direction direction,  
                   String... properties)
```

one of several
constructor methods

- Example of PageRequest

```
final PageRequest pageRequest1 =  
    new PageRequest(0, 3, Direction.ASC, "lastname", "emailAddress");  
Page<Customer> page = customerRepository.findAll(pageRequest1);  
customers = page.getContent();  
  
final PageRequest pageRequest2 =  
    new PageRequest(0, 2,  
                    new Sort(new Order(  
                        Direction.ASC, "lastname"), new Order(Direction.DESC, "firstname")));  
page = customerRepository.findAll(pageRequest2);  
customers = page.getContent();
```

Lab:

Exercise 3: Paging and Sorting
4944_spring4_data_jpa.zip



Query Generation Strategy

Query Strategies

- #1: Query generation from method names
- #2: Query generation from @Query annotations
- #3: Query generation from Named queries

#1: Query Generation from Method Names

- Examples
 - > `findById(Long id)`, `findNameById(Long id)`,
 - > `findByNameOrDescription(String name, String description);`
 - > `findDistinctByName(String name)`
 - > `findTopByName(String name)`, `findTop1ByName(String name)`,
`findFirstByName(String name)`, `findFirst1ByName(String name)`
 - > `findFirst3ByNameOrderByAsc(String name)`
 - > `findByNameContaining(String name)`

#2: Query generation from `@Query` annotation

- Use it to provide your own query implementation
- Supports both JPQL and native SQL

```
// JPQL  
{@Query("SELECT t FROM Product t WHERE t.name = ?1")  
public List<Product> findByName_JPQL(String name);}
```

```
// SQL  
{@Query(value = "SELECT * FROM Product t WHERE t.name = ?1", nativeQuery=true)  
public List<Product> findByName_SQL(String name);}
```

#3: Query generation from Named queries

- You can specify named queries with Spring Data JPA by using annotations, properties file or the orm.xml file

```
// Entity class
@Entity
@NamedQuery(name = "Product.findByName_NamedQuery",
             query = "FROM Product t WHERE t.name = ?1"
)
public class Product {

}

// Repository interface
public interface ProductRepository extends JpaRepository<Product, Long> {

    // Examples of query generation using Named query
    List<Product> findByName_NamedQuery(String name);
}

// Perform query operation
products = productRepository.findByName_NamedQuery("product1");
```

Lab:

Exercise 4: Query Generation
4944_spring4_data_jpa.zip



Other features

Transaction Support

- Repository classes are transactional by default
 - > Methods are `@Transactional` by default
 - > Reads are `readOnly`
- You can override it

Code with Passion!

