Spring Data JPA

Simeon Monov, Hristo Karaperev

Overview

- Spring Data JPA is part of the larger Spring Data family
- It makes it easy to implement JPA-based (Java Persistence API) repositories
- Spring Data JPA aims to significantly improve the implementation of data access layers by reducing the effort to the amount that’s actually needed
- As a developer you write your repository interfaces using any number of techniques, and Spring will wire it up for you automatically

Core Concepts (1)

- The central interface in the Spring Data repository abstraction is Repository
- It takes the domain class to manage as well as the identifier type of the domain class as type arguments
- The CrudRepository and ListCrudRepository interfaces provide sophisticated CRUD functionality for the entity class that is being managed
- Spring also provide persistence technology-specific abstractions, such as JpaRepository or MongoRepository. Those interfaces extend CrudRepository and expose the capabilities of the underlying persistence technology

CrudRepository Interface

JAVA

```java
public interface CrudRepository<T, ID> extends Repository<T, ID> {

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

  Optional<T> findById(ID primaryKey); ❷

  Iterable<T> findAll();               ❸

  long count();                        ❹

  void delete(T entity);               ❺

  boolean existsById(ID primaryKey);   ❻

  // … more functionality omitted.
}
```

Core Concepts (2)

- Additional to the CrudRepository, there are PagingAndSortingRepository and ListPagingAndSortingRepository which add additional methods to ease paginated access to entities
- ListPagingAndSortingRepository offers equivalent methods, but returns a List where the PagingAndSortingRepository methods return an Iterable

PagingAndSortingRepository Interface

JAVA

```java
public interface PagingAndSortingRepository<T, ID>  {

  Iterable<T> findAll(Sort sort);

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

Defining Repository Interfaces (1)

- To define a repository interface, you first need to define a domain class-specific repository interface. The interface must extend Repository and be typed to the domain class and an ID type
- Extending one of the CRUD repository interfaces exposes a complete set of methods to manipulate your entities. If you prefer to be selective about the methods being exposed, copy the methods you want to expose from the CRUD repository into your domain repository. When doing so, you may change the return type of methods

Defining Repository Interfaces (2)

- If many repositories in your application should have the same set of methods you can define your own base interface to inherit from. Such an interface must be annotated with @NoRepositoryBean

Selectively exposing CRUD methods

JAVA

```java
@NoRepositoryBean
interface MyBaseRepository<T, ID> extends Repository<T, ID> {

  Optional<T> findById(ID id);

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

interface UserRepository extends MyBaseRepository<User, Long> {
  User findByEmailAddress(EmailAddress emailAddress);
}
```

Defining Repository Interfaces (3)

- Sometimes, applications require using more than one Spring Data module. In such cases, a repository definition must distinguish between persistence technologies
- When it detects multiple repository factories on the class path, Spring Data enters strict repository configuration mode. Strict configuration uses details on the repository or the domain class to decide about Spring Data module binding for a repository definition

Defining Repository Interfaces (4)

1) If the repository definition extends the module-specific repository, it is a valid candidate for the particular Spring Data module
2) If the domain class is annotated with the module-specific type annotation, it is a valid candidate for the particular Spring Data module. Spring Data modules accept either third-party annotations (such as JPA's @Entity) or provide their own annotations (such as @Document for Spring Data MongoDB and Spring Data Elasticsearch)

```java
interface MyRepository extends JpaRepository<User, Long> { }

@NoRepositoryBean
interface MyBaseRepository<T, ID> extends JpaRepository<T, ID> { … }

interface UserRepository extends MyBaseRepository<User, Long> { … }
```

`MyRepository` and `UserRepository` extend `JpaRepository` in their type hierarchy. They are valid candidates for the Spring Data JPA module.

```java
interface PersonRepository extends Repository<Person, Long> { … }

@Entity
class Person { … }

interface UserRepository extends Repository<User, Long> { … }

@Document
class User { … }
```

`PersonRepository` references `Person`, which is annotated with the JPA `@Entity` annotation, so this repository clearly belongs to Spring Data JPA. `UserRepository` references `User`, which is annotated with Spring Data MongoDB's `@Document` annotation.

Repository type details and distinguishing domain class annotations are used for strict repository configuration to identify repository candidates for a particular Spring Data module. Using multiple persistence technology-specific annotations on the same domain type is possible and enables reuse of domain types across multiple persistence technologies. However, Spring Data can then no longer determine a unique module with which to bind the repository

JAVA

```java
interface JpaPersonRepository extends Repository<Person, Long> { … }

interface MongoDBPersonRepository extends Repository<Person, Long> { … }

@Entity
@Document
class Person { … }
```

This example shows a domain class using both JPA and Spring Data MongoDB annotations. It defines two repositories, `JpaPersonRepository` and `MongoDBPersonRepository`. One is intended for JPA and the other for MongoDB usage. Spring Data is no longer able to tell the repositories apart, which leads to undefined behavior.

Persisting Entities (1)

- Saving an entity can be performed with the CrudRepository.save(…) method
- It persists or merges the given entity by using the underlying JPA EntityManager
- If the entity has not yet been persisted, Spring Data JPA saves the entity with a call to the entityManager.persist(…) method
- Otherwise, it calls the entityManager.merge(…) method

Persisting Entities (2)

- Spring Data JPA offers the following strategies to detect whether an entity is new or not
  1) Version-Property and Id-Property inspection (default): By default Spring Data JPA inspects first if there is a Version-property of non-primitive type. If there is, the entity is considered new if the value of that property is null. Without such a Version-property Spring Data JPA inspects the identifier property of the given entity. If the identifier property is null, then the entity is assumed to be new. Otherwise, it is assumed to be not new
  2) Implementing Persistable: If an entity implements Persistable, Spring Data JPA delegates the new detection to the isNew(…) method of the entity

Persisting Entities (3)

- Option 1 is not an option for entities that use manually assigned identifiers and no version attribute as with those the identifier will always be non-null. A common pattern in that scenario is to use a common base class with a transient flag defaulting to indicate a new instance and using JPA lifecycle callbacks to flip that flag on persistence operations

```java
@MappedSuperclass
public abstract class AbstractEntity<ID> implements Persistable<ID> {

  @Transient
  private boolean isNew = true; ❶

  @Override
  public boolean isNew() {
    return isNew; ❷
  }

  @PrePersist ❸
  @PostLoad
  void markNotNew() {
    this.isNew = false;
  }
```

Defining Query Methods

- The repository proxy has two ways to derive a store-specific query from the method name
  1) By deriving the query from the method name directly
  2) By using a manually defined query

Query Lookup Strategies

- For Java configuration, you can use the queryLookupStrategy attribute of the EnableJpaRepositories annotation
  - CREATE attempts to construct a store-specific query from the query method name.
  - USE_DECLARED_QUERY tries to find a declared query and throws an exception if it cannot find one
  - CREATE_IF_NOT_FOUND (the default) combines CREATE and USE_DECLARED_QUERY

Query Creation (1)

- Parsing query method names is divided into subject and predicate. The first part (find…By, exists…By) defines the subject of the query, the second part forms the predicate
- The first By acts as a delimiter to indicate the start of the actual criteria predicate. At a very basic level, you can define conditions on entity properties and concatenate them with And and Or

Query Creation (2)

Query creation from method names

JAVA

```java
interface PersonRepository extends Repository<Person, Long> {

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

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

Supported query method subject and predicate keywords

Table 1. Query subject keywords

Keyword	Description
`find…By` , `read…By` , `get…By` , `query…By` , `search…By` , `stream…By`	General query method returning typically the repository type, a `Collection` or `Streamable` subtype or a result wrapper such as `Page` , `GeoResults` or any other store-specific result wrapper. Can be used as `findBy…` , `findMyDomainTypeBy…` or in combination with additional keywords.
`exists…By`	Exists projection, returning typically a `boolean` result.
`count…By`	Count projection returning a numeric result.
`delete…By` , `remove…By`	Delete query method returning either no result (`void`) or the delete count.
`…First<number>…` , `…Top<number>…`	Limit the query results to the first `<number>` of results. This keyword can occur in any place of the subject between `find` (and the other keywords) and `by` .
`…Distinct…`	Use a distinct query to return only unique results. Consult the store-specific documentation whether that feature is supported. This keyword can occur in any place of the subject between `find` (and the other keywords) and `by` .

Table 2. Query predicate keywords

Logical keyword	Keyword expressions
`AND`	`And`
`OR`	`Or`
`AFTER`	`After`, `IsAfter`
`BEFORE`	`Before`, `IsBefore`
`CONTAINING`	`Containing`, `IsContaining`, `Contains`
`BETWEEN`	`Between`, `IsBetween`
`ENDING_WITH`	`EndingWith`, `IsEndingWith`, `EndsWith`
`EXISTS`	`Exists`
`FALSE`	`False`, `IsFalse`
`GREATER_THAN`	`GreaterThan`, `IsGreaterThan`
`GREATER_THAN_EQUALS`	`GreaterThanEqual`, `IsGreaterThanEqual`
`IN`	`In`, `IsIn`
`IS`	`Is`, `Equals`, (or no keyword)
`IS_EMPTY`	`IsEmpty`, `Empty`
`IS_NOT_EMPTY`	`IsNotEmpty`, `NotEmpty`
`IS_NOT_NULL`	`NotNull`, `IsNotNull`
`IS_NULL`	`Null`, `IsNull`
`LESS_THAN`	`LessThan`, `IsLessThan`
`LESS_THAN_EQUAL`	`LessThanEqual`, `IsLessThanEqual`
`LIKE`	`Like`, `IsLike`
`NEAR`	`Near`, `IsNear`
`NOT`	`Not`, `IsNot`
`NOT_IN`	`NotIn`, `IsNotIn`
`NOT_LIKE`	`NotLike`, `IsNotLike`
`REGEX`	`Regex`, `MatchesRegex`, `Matches`
`STARTING_WITH`	`StartingWith`, `IsStartingWith`, `StartsWith`
`TRUE`	`True`, `IsTrue`
`WITHIN`	`Within`, `IsWithin`

Table 3. Query predicate modifier keywords

Keyword	Description
`IgnoreCase`, `IgnoringCase`	Used with a predicate keyword for case-insensitive comparison.
`AllIgnoreCase`, `AllIgnoringCase`	Ignore case for all suitable properties. Used somewhere in the query method predicate.
`OrderBy…`	Specify a static sorting order followed by the property path and direction (e. g. `OrderByFirstnameAscLastnameDesc`).

Query Creation (3)

- Although getting a query derived from the method name is quite convenient, one might face the situation in which either the method name parser does not support the keyword one wants to use or the method name would get unnecessarily ugly. So you can annotate your query method with @Query

Example 5. Declare query at the query method using @Query

JAVA

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

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

Using Named Parameters

- By default, Spring Data JPA uses position-based parameter binding
- This makes query methods a little error-prone when refactoring regarding the parameter position
- To solve this issue, you can use @Param annotation to give a method parameter a concrete name and bind the name in the query

Example 13. Using named parameters

JAVA

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

  @Query("select u from User u where u.firstname = :firstname or u.lastname = :lastname")
  User findByLastnameOrFirstname(@Param("lastname") String lastname,
                                 @Param("firstname") String firstname);
}
```

Using SpEL Expressions

- Spring supports the usage of restricted SpEL template expressions in manually defined queries that are defined with @Query
- Spring Data JPA supports a variable called entityName. It inserts the entityName of the domain type associated with the given repository
- If the domain type has set the name property on the @Entity annotation, it is used. Otherwise, the simple class-name of the domain type is used

Example 14. Using SpEL expressions in repository query methods - entityName

JAVA

```java
@Entity
public class User {

  @Id
  @GeneratedValue
  Long id;

  String lastname;
}

public interface UserRepository extends JpaRepository<User,Long> {

  @Query("select u from #{#entityName} u where u.lastname = ?1")
  List<User> findByLastname(String lastname);
}
```

Projections

- Spring Data query methods usually return one or multiple instances of the aggregate root managed by the repository
- However, it might sometimes be desirable to create projections based on certain attributes of those types
- Spring Data allows modeling dedicated return types, to more selectively retrieve partial views of the managed aggregates

Interface-based Projections

- The easiest way to limit the result of the queries to only the name attributes is by declaring an interface that exposes accessor methods for the properties to be read

A projection interface to retrieve a subset of attributes

JAVA

```java
interface NamesOnly {

  String getFirstname();
  String getLastname();
}
```

the aggregate root

A repository using an interface based projection with a query method

JAVA

```java
interface PersonRepository extends Repository<Person, UUID> {

  Collection<NamesOnly> findByLastname(String lastname);
}
```

Interface-based Projections (2)

- Projections can be used recursively
- On method invocation, the address property of the target instance is obtained and wrapped into a projecting proxy
- A projection interface whose accessor methods all match properties of the target aggregate is considered to be a closed projection

A projection interface to retrieve a subset of attributes

JAVA

```
interface PersonSummary {

  String getFirstname();
  String getLastname();
  AddressSummary getAddress();

  interface AddressSummary {
    String getCity();
  }
}
```

A closed projection

JAVA

```
interface NamesOnly {

  String getFirstname();
  String getLastname();
}
```

Interface-based Projections (3)

- A projection interface using @Value is an open projection
- Accessor methods in projection interfaces can also be used to compute new values by using the @Value annotation
- The aggregate root backing the projection is available in the target variable

An Open Projection

JAVA

```java
interface NamesOnly {

  @Value("#{target.firstname + ' ' + target.lastname}")
  String getFullName();
  …
}
```

Interface-based Projections (4)

- For very simple expressions, one option might be to resort to default methods
- This approach requires you to be able to implement logic purely based on the other accessor methods exposed on the projection interface.
- A second, more flexible, option is to implement the custom logic in a Spring bean and then invoke that from the SpEL expression

A projection interface using a default method for custom logic

JAVA

```
interface NamesOnly {

  String getFirstname();
  String getLastname();

  default String getFullName() {
    return getFirstname().concat(" ").concat(getLastname());
  }
}
```

JAVA

```
@Component
class MyBean {

  String getFullName(Person person) {
    …
  }
}

interface NamesOnly {

  @Value("#{@myBean.getFullName(target)}")
  String getFullName();
  …
}
```

Class-based Projections

- Another way of defining projections is by using value type DTOs (Data Transfer Objects) that hold properties for the fields that are supposed to be retrieved
- These DTO types can be used in exactly the same way projection interfaces are used, except that no proxying happens and no nested projections can be applied

A projecting DTO

JAVA

```
record NamesOnly(String firstname, String lastname) {
}
```

Dynamic Projections

- You might want to select the type to be used at invocation time (which makes it dynamic). To apply dynamic projections, use a query method
- This way, the method can be used to obtain the aggregates as is or with a projection applied

A repository using a dynamic projection parameter

```java
interface PersonRepository extends Repository<Person, UUID> {

  <T> Collection<T> findByLastname(String lastname, Class<T> type);
}
```

Using a repository with dynamic projections

```java
void someMethod(PersonRepository people) {

  Collection<Person> aggregates =
    people.findByLastname("Matthews", Person.class);

  Collection<NamesOnly> aggregates =
    people.findByLastname("Matthews", NamesOnly.class);
}
```

Stored Procedures

- The JPA 2.1 specification introduced support for calling stored procedures by using the JPA criteria query API
- Spring introduced the @Procedure annotation for declaring stored procedure metadata on a repository method

```sql
/;
DROP procedure IF EXISTS plus1inout
/;
CREATE procedure plus1inout (IN arg int, OUT res int)
BEGIN ATOMIC
 set res = arg + 1;
END
/;
```

```java
@Procedure("plus1inout")
Integer explicitlyNamedPlus1inout(Integer arg);
```

```java
@Procedure(procedureName = "plus1inout")
Integer callPlus1inout(Integer arg);
```

```java
@Procedure
Integer plus1inout(@Param("arg") Integer arg);
```

Query by Example (1)

- Query by Example (QBE) is a user-friendly querying technique with a simple interface. It allows dynamic query creation and does not require you to write queries that contain field names
- The Query by Example API
  - Probe: The actual example of a domain object with populated fields
  - ExampleMatcher: The ExampleMatcher carries details on how to match particular fields. It can be reused across multiple Examples
  - Example: An Example consists of the probe and the ExampleMatcher. It is used to create the query
  - FetchableFluentQuery: A FetchableFluentQuery offers a fluent API, that allows further customization of a query derived from an Example

Query by Example (2)

- Query by Example is well suited for several use cases
  - Querying your data store with a set of static or dynamic constraints
  - Frequent refactoring of the domain objects without worrying about breaking existing queries
  - Working independently from the underlying data store API
- Query by Example also has several limitations
  - No support for nested or grouped property constraints, such as firstname = ?0 or (firstname = ?1 and lastname = ?2)
  - Only supports starts/contains/ends/regex matching for strings and exact matching for other property types

Query by Example (3)

- Examples can be built by either using the of factory method or by using ExampleMatcher
- Example is immutable
- You can run the example queries by using repositories. Let your repository interface extend QueryByExampleExecutor<T>

```java
Person person = new Person();
person.setFirstname("Dave");

Example<Person> example = Example.of(person);
```

```java
public interface QueryByExampleExecutor<T> {

  <S extends T> S findOne(Example<S> example);

  <S extends T> Iterable<S> findAll(Example<S> example);

  // … more functionality omitted.
}
```

Query by Example (4)

- Examples are not limited to default settings. You can specify your own defaults for string matching, null handling, and property-specific settings by using the ExampleMatcher

JAVA

```
Person person = new Person();                                    ❶
person.setFirstname("Dave");                                     ❷

ExampleMatcher matcher = ExampleMatcher.matching()               ❸
  .withIgnorePaths("lastname")                                   ❹
  .withIncludeNullValues()                                       ❺
  .withStringMatcher(StringMatcher.ENDING);                      ❻

Example<Person> example = Example.of(person, matcher); ❼
```

❶ Create a new instance of the domain object.

❷ Set properties.

❸ Create an `ExampleMatcher` to expect all values to match. It is usable at th is stage even without further configuration.

❹ Construct a new `ExampleMatcher` to ignore the `lastname` property path.

❺ Construct a new `ExampleMatcher` to ignore the `lastname` property path and to include null values.

❻ Construct a new `ExampleMatcher` to ignore the `lastname` property path, to include null values, and to perform suffix string matching.

❼ Create a new `Example` based on the domain object and the configured `Ex ampleMatcher`.

Query by Example (5)

- QueryByExampleExecutor offers one more method:
- <S extends T, R> R findBy(Example<S> example, Function<FluentQuery.FetchableFluentQuery<S>, R> queryFunction)
- It executes a query derived from an Example
- However, with the second argument, you can control aspects of that execution that you cannot dynamically control otherwise
- You do so by invoking the various methods of the FetchableFluentQuery in the second argument

Use the fluent API to get the last of potentially many results, ordered by lastname.

JAVA

```
Optional<Person> match = repository.findBy(example,
    q -> q
        .sortBy(Sort.by("lastname").descending())
        .first()
);
```

Transactionality (1)

- By default, methods inherited from CrudRepository inherit the transactional configuration from SimpleJpaRepository.
- For read operations, the transaction configuration readOnly flag is set to true.
- All others are configured with a plain @Transactional so that default transaction configuration applies
- Another way to alter transactional behaviour is to use a facade or service implementation that (typically) covers more than one repository
- Its purpose is to define transactional boundaries for non-CRUD operations

Transactionality (2)

- This example causes call to addRoleToAllUsers(…) to run inside a transaction
- The transaction configuration at the repositories is then neglected, as the outer transaction configuration determines the actual one used
- Note that you must activate @EnableTransactionManagement explicitly to get annotation-based configuration of facades to work

```java
@Service
public class UserManagementImpl implements UserManagement {

  private final UserRepository userRepository;
  private final RoleRepository roleRepository;

  public UserManagementImpl(UserRepository userRepository,
    RoleRepository roleRepository) {
    this.userRepository = userRepository;
    this.roleRepository = roleRepository;
  }

  @Transactional
  public void addRoleToAllUsers(String roleName) {

    Role role = roleRepository.findByName(roleName);

    for (User user : userRepository.findAll()) {
      user.addRole(role);
      userRepository.save(user);
    }
  }
}
```

Transactionality (3)

- Declared query methods (including default methods) do not get any transaction configuration applied by default
- To run those methods transactionally, use @Transactional at the repository interface you define
- Typically, you want the readOnly flag to be set to true, as most of the query methods only read data.
- In contrast to that, deleteInactiveUsers() makes use of the @Modifying annotation and overrides the transaction configuration. Thus, the method runs with the readOnly flag set to false

```java
@Transactional(readOnly = true)
interface UserRepository extends JpaRepository<User, Long> {

  List<User> findByLastname(String lastname);

  @Modifying
  @Transactional
  @Query("delete from User u where u.active = false")
  void deleteInactiveUsers();
}
```