Java Enterprise Edition

ANUDIP FOUNDATION

Spring Data JPA

Objective: • Reader Hierarchy • Writer Hierarchy	**Materials Required:** 1. Eclipse IDE/IntelliJ/STS 2. Notepad 3. Gradle/Maven 4. Jdk1.8 or later
Theory:80mins	**Practical:40mins**
Total Duration: 120 mins	

Spring Data JPA

- Spring Data JPA Intro & Overview
- Core Concepts, @RepositoryRestResource
- Defining Query methods
- Query Creation
- Using JPA Named Queries
- Defining Repository Interfaces
- Creating Repository instances
- JPA Repositories
- Persisting Entities
- Transactions

JPA Repositories

This chapter points out the specialties for repository support for JPA. This builds on the core repository support explained in “Working with Spring Data Repositories”. Make sure you have a sound understanding of the basic concepts explained there.

1.1. Introduction

This section describes the basics of configuring Spring Data JPA through either:

- “Spring Namespace” (XML configuration)
- “Annotation-based Configuration” (Java configuration)

1.1.1. Spring Namespace

The JPA module of Spring Data contains a custom namespace that allows defining repository beans. It also contains certain features and element attributes that are special to JPA. Generally, the JPA repositories can be set up by using the repositories element, as shown in the following example:
Example 54. Setting up JPA repositories by using the namespace

```
<?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
    https://www.springframework.org/schema/beans/spring-beans.xsd
    http://www.springframework.org/schema/data/jpa
    https://www.springframework.org/schema/data/jpa/spring-jpa.xsd">

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

</beans>
```

Using the repositories element looks up Spring Data repositories as described in "Creating Repository Instances". Beyond that, it activates persistence exception translation for all beans annotated with @Repository, to let exceptions being thrown by the JPA persistence providers be converted into Spring's DataAccessException hierarchy.

Custom Namespace Attributes

Beyond the default attributes of the repositories element, the JPA namespace offers additional attributes to let you gain more detailed control over the setup of the repositories:

Table 2. Custom JPA-specific attributes of the repositories **element**

entity-manager-factory-ref	Explicitly wire the EntityManagerFactory to be used with the repositories being detected by the repositories element. Usually used if multiple EntityManagerFactory beans are used within the application. If not configured, Spring Data automatically looks up the EntityManagerFactory bean with the name entityManagerFactory in the ApplicationContext.
transaction-manager-ref	Explicitly wire the PlatformTransactionManager to be used with the repositories being detected by the repositories element. Usually only necessary if multiple transaction managers or EntityManagerFactory beans have been configured. Default to a single defined PlatformTransactionManager inside the current ApplicationContext.

Spring Data JPA requires a PlatformTransactionManager bean named transactionManager to be present if no explicit transaction-manager-ref is defined.

1.1.2. Annotation-based Configuration

The Spring Data JPA repositories support can be activated not only through an XML namespace but also by using an annotation through JavaConfig, as shown in the following example:

Example 55. Spring Data JPA repositories using JavaConfig

```
@Configuration
@EnableJpaRepositories
@EnableTransactionManagement
class ApplicationConfig {
 @Bean
 public DataSource dataSource() {

   EmbeddedDatabaseBuilder builder = new EmbeddedDatabaseBuilder();
   return builder.setType(EmbeddedDatabaseType.HSQL).build();
 }

 @Bean
 public LocalContainerEntityManagerFactoryBean entityManagerFactory() {

   HibernateJpaVendorAdapter vendorAdapter = new HibernateJpaVendorAdapter();
   vendorAdapter.setGenerateDdl(true);

   LocalContainerEntityManagerFactoryBean factory = new LocalContainerEntityManagerFactoryBean();
   factory.setJpaVendorAdapter(vendorAdapter);
   factory.setPackagesToScan("com.acme.domain");
   factory.setDataSource(dataSource());
   return factory;
 }

 @Bean
 public PlatformTransactionManager transactionManager(EntityManagerFactory
entityManagerFactory) {

   JpaTransactionManager txManager = new JpaTransactionManager();
   txManager.setEntityManagerFactory(entityManagerFactory);
   return txManager;
 }
}
```

You must create LocalContainerEntityManagerFactoryBean and not EntityManagerFactory directly, since the former also participates in exception translation mechanisms in addition to creating EntityManagerFactory.

The preceding configuration class sets up an embedded HSQL database by using the EmbeddedDatabaseBuilder API of spring-jdbc. Spring Data then sets up an EntityManagerFactory and uses Hibernate as the sample persistence provider. The last infrastructure component declared here is the JpaTransactionManager. Finally, the example activates Spring Data JPA repositories by using the @EnableJpaRepositories annotation, which essentially carries the same attributes as the XML namespace. If no base package is configured, it uses the one in which the configuration class resides.

1.1.3. Bootstrap Mode

By default, Spring Data JPA repositories are default Spring beans. They are singleton scoped and eagerly initialized. During startup, they already interact with the JPA EntityManager for verification and metadata analysis purposes. Spring Framework supports the initialization of the JPA EntityManagerFactory in a background thread because that process usually takes up a significant

amount of startup time in a Spring application. To make use of that background initialization effectively, we need to make sure that JPA repositories are initialized as late as possible.
As of Spring Data JPA 2.1 you can now configure a BootstrapMode (either via the @EnableJpaRepositories annotation or the XML namespace) that takes the following values:

- DEFAULT (default) — Repositories are instantiated eagerly unless explicitly annotated with @Lazy. The lazification only has effect if no client bean needs an instance of the repository as that will require the initialization of the repository bean.
- LAZY — Implicitly declares all repository beans lazy and also causes lazy initialization proxies to be created to be injected into client beans. That means, that repositories will not get instantiated if the client bean is simply storing the instance in a field and not making use of the repository during initialization. Repository instances will be initialized and verified upon first interaction with the repository.
- DEFERRED — Fundamentally the same mode of operation as LAZY, but triggering repository initialization in response to an ContextRefreshedEvent so that repositories are verified before the application has completely started.

Recommendations

If you're not using asynchronous JPA bootstrap stick with the default bootstrap mode.

In case you bootstrap JPA asynchronously, DEFERRED is a reasonable default as it will make sure the Spring Data JPA bootstrap only waits for the EntityManagerFactory setup if that itself takes longer than initializing all other application components. Still, it makes sure that repositories are properly initialized and validated before the application signals it's up.
LAZY is a decent choice for testing scenarios and local development. Once you are pretty sure that repositories can properly bootstrap, or in cases where you are testing other parts of the application, running verification for all repositories might unnecessarily increase the startup time. The same applies to local development in which you only access parts of the application that might need to have a single repository initialized.

1.2. Persisting Entities

This section describes how to persist (save) entities with Spring Data JPA.

1.2.1. Saving Entities

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.

Entity State-detection Strategies

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. See the JavaDoc for details.

3. Implementing EntityInformation: You can customize the EntityInformation abstraction used in the SimpleJpaRepository implementation by creating a subclass of JpaRepositoryFactory and overriding the getEntityInformation(…) method accordingly. You then have to register the custom implementation of JpaRepositoryFactory as a Spring bean. Note that this should be rarely necessary. See the JavaDoc for details.

Option 1 is not an option for entities that use manually assigned identifiers 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:

Example 56. A base class for entities with manually assigned identifiers

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

  @Transient
  private boolean isNew = true;

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

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

  // More code…
}
```

Declare a flag to hold the new state. Transient so that it’s not persisted to the database.

Return the flag in the implementation of Persistable.isNew() so that Spring Data repositories know whether to call EntityManager.persist() or ….merge().

Declare a method using JPA entity callbacks so that the flag is switched to indicate an existing entity after a repository call to save(…) or an instance creation by the persistence provider.

1.3. Query Methods

This section describes the various ways to create a query with Spring Data JPA.

1.3.1. Query Lookup Strategies

The JPA module supports defining a query manually as a String or having it being derived from the method name.

Derived queries with the predicates IsStartingWith, StartingWith, StartsWith, IsEndingWith, EndingWith, EndsWith, IsNotContaining, NotContaining, NotContains, IsContaining, Containing, Contains the respective arguments for these queries will get sanitized. This means if the arguments actually contain characters recognized by LIKE as wildcards these will get escaped so they match only as literals. The escape character used can be configured by setting the escapeCharacter of the @EnableJpaRepositories annotation. Compare with Using SpEL Expressions.

Declared Queries

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 either use JPA named queries through a naming convention (see Using JPA Named Queries for more information) or rather annotate your query method with @Query (see Using @Query for details).

1.3.2. Query Creation

Generally, the query creation mechanism for JPA works as described in "Query Methods". The following example shows what a JPA query method translates into:

Example 57. Query creation from method names

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

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

We 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. Spring Data JPA does a property check and traverses nested properties, as described in "Property Expressions".

The following table describes the keywords supported for JPA and what a method containing that keyword translates to:

Table 3. Supported keywords inside method names

Keyword	Sample	JPQL snippet
Distinct	findDistinctByLastnameAndFirstname	select distinct … where x.lastname = ?1 and x.firstname = ?2
And	findByLastnameAndFirstname	… where x.lastname = ?1 and x.firstname = ?2
Or	findByLastnameOrFirstname	… where x.lastname = ?1 or x.firstname = ?2

Table 3. Supported keywords inside method names

Keyword	Sample	JPQL snippet
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
Before	findByStartDateBefore	… where x.startDate < ?1
IsNull, Null	findByAge(Is)Null	… 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

Table 3. Supported keywords inside method names

Keyword	Sample	JPQL snippet
		bound with appended %)
EndingWith	findByFirstnameEndingWith	… where x.firstname like ?1 (parameter bound with prepended %)
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> ages)	… where x.age not in ?1
True	findByActiveTrue()	… where x.active = true
False	findByActiveFalse()	… where x.active = false
IgnoreCase	findByFirstnameIgnoreCase	… where UPPER(x.firstname) = UPPER(?1)

In and NotIn also take any subclass of Collection as a parameter as well as arrays or varargs. For other syntactical versions of the same logical operator, check "Repository query keywords".

1.3.3. Using JPA Named Queries

The examples use the <named-query /> element and @NamedQuery annotation. The queries for these configuration elements have to be defined in the JPA query language. Of course, you can use <named-native-query /> or @NamedNativeQuery too. These elements let you define the query in native SQL by losing the database platform independence.

XML Named Query Definition

To use XML configuration, add the necessary <named-query /> element to the orm.xml JPA configuration file located in the META-INF folder of your classpath. Automatic invocation of named queries is enabled by using some defined naming convention. For more details, see below.

Example. XML named query configuration

```
<named-query name="User.findByLastname">
  <query>select u from User u where u.lastname = ?1</query>
</named-query>
```

The query has a special name that is used to resolve it at runtime.

Annotation-based Configuration

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

Example. Annotation-based named query configuration

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

}
```

Declaring Interfaces

To allow these named queries, specify the UserRepository as follows:

Example. Query method declaration in UserRepository

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

  List<User> findByLastname(String lastname);

  User findByEmailAddress(String emailAddress);
}
```

Spring Data tries to resolve a call to these methods to a named query, starting with the simple name of the configured domain class, followed by the method name separated by a dot. So the preceding example would use the named queries defined earlier instead of trying to create a query from the method name.

1.3.4. 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 runs them, you can actually bind them directly by using the Spring Data JPA @Query annotation rather than annotating them to the domain class. This frees the domain class from persistence specific information and co-locates the query to the repository interface.
Queries annotated to the query method take precedence over queries defined using @NamedQuery or named queries declared in orm.xml.
The following example shows a query created with the @Query annotation:
Example 61. 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);
}
```

Using Advanced LIKE Expressions
The query running mechanism for manually defined queries created with @Query allows the definition of advanced LIKE expressions inside the query definition, as shown in the following example:
Example. Advanced like **expressions in @Query**

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

  @Query("select u from User u where u.firstname like %?1")
  List<User> findByFirstnameEndsWith(String firstname);
}
```

In the preceding example, the LIKE delimiter character (%) is recognized, and the query is transformed into a valid JPQL query (removing the %). Upon running the query, the parameter passed to the method call gets augmented with the previously recognized LIKE pattern.

Native Queries

The @Query annotation allows for running native queries by setting the nativeQuery flag to true, as shown in the following example:
Example. Declare a native query at the query method using @Query

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

  @Query(value = "SELECT * FROM USERS WHERE EMAIL_ADDRESS = ?1", nativeQuery =
true)
  User findByEmailAddress(String emailAddress);
}
```

Spring Data JPA does not currently support dynamic sorting for native queries, because it would have to manipulate the actual query declared, which it cannot do reliably for native SQL. You can, however, use native queries for pagination by specifying the count query yourself, as shown in the following example:

Example. Declare native count queries for pagination at the query method by using @Query

```
public interface UserRepository extends JpaRepository<User, Long> {
 @Query(value = "SELECT * FROM USERS WHERE LASTNAME = ?1",
   countQuery = "SELECT count(*) FROM USERS WHERE LASTNAME = ?1",
   nativeQuery = true)
 Page<User> findByLastname(String lastname, Pageable pageable);
}
```

A similar approach also works with named native queries, by adding the .count suffix to a copy of your query. You probably need to register a result set mapping for your count query, though.

1.3.5. Using Sort

Sorting can be done by either providing a PageRequest or by using Sort directly. The properties actually used within the Order instances of Sort need to match your domain model, which means they need to resolve to either a property or an alias used within the query. The JPQL defines this as a state field path expression.

Using any non-referenceable path expression leads to an Exception.

However, using Sort together with @Query lets you sneak in non-path-checked Order instances containing functions within the ORDER BY clause. This is possible because the Order is appended to the given query string. By default, Spring Data JPA rejects any Order instance containing function calls, but you can use JpaSort.unsafe to add potentially unsafe ordering.
The following example uses Sort and JpaSort, including an unsafe option on JpaSort:
Example 65. Using Sort **and** JpaSort

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

 @Query("select u from User u where u.lastname like ?1%")
 List<User> findByAndSort(String lastname, Sort sort);

 @Query("select u.id, LENGTH(u.firstname) as fn_len from User u where u.lastname like
?1%")
 List<Object[]> findByAsArrayAndSort(String lastname, Sort sort);
}

repo.findByAndSort("lannister", Sort.by("firstname"));
repo.findByAndSort("stark", Sort.by("LENGTH(firstname)"));
repo.findByAndSort("targaryen", JpaSort.unsafe("LENGTH(firstname)"));
repo.findByAsArrayAndSort("bolton", Sort.by("fn_len"));
```

Valid Sort expression pointing to property in domain model.

Invalid Sort containing function call. Throws Exception.

Valid Sort containing explicitly *unsafe* Order.

Valid Sort expression pointing to aliased function.

Using Named Parameters

By default, Spring Data JPA uses position-based parameter binding, as described in all the preceding examples. 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, as shown in the following example:
Example 66. Using named parameters

```
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);
}
```

The method parameters are switched according to their order in the defined query.

As of version 4, Spring fully supports Java 8’s parameter name discovery based on the -parameters compiler flag. By using this flag in your build as an alternative to debug information, you can omit the @Param annotation for named parameters.

Using SpEL Expressions

As of Spring Data JPA release 1.4, we support the usage of restricted SpEL template expressions in manually defined queries that are defined with @Query. Upon the query being run, these expressions are evaluated against a predefined set of variables. Spring Data JPA supports a variable called entityName. Its usage is select x from #{#entityName} x. It inserts the entityName of the domain type associated with the given repository. The entityName is resolved as follows: 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.
The following example demonstrates one use case for the #{#entityName} expression in a query string where you want to define a repository interface with a query method and a manually defined query:
Example 67. Using SpEL expressions in repository query methods - entityName

```
@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);
}
```

To avoid stating the actual entity name in the query string of a @Query annotation, you can use the #{#entityName} variable.

The entityName can be customized by using the @Entity annotation. Customizations in orm.xml are not supported for the SpEL expressions.

Of course, you could have just used User in the query declaration directly, but that would require you to change the query as well. The reference to #entityName picks up potential future remappings of the User class to a different entity name (for example, by using @Entity(name = "MyUser").
Another use case for the #{#entityName} expression in a query string is if you want to define a generic repository interface with specialized repository interfaces for a concrete domain type. To not repeat the definition of custom query methods on the concrete interfaces, you can use the entity name expression in the query string of the @Query annotation in the generic repository interface, as shown in the following example:
Example . Using SpEL expressions in repository query methods - entityName with inheritance

```
@MappedSuperclass
public abstract class AbstractMappedType {
  …
  String attribute
}

@Entity
public class ConcreteType extends AbstractMappedType { … }

@NoRepositoryBean
public interface MappedTypeRepository<T extends AbstractMappedType>
  extends Repository<T, Long> {

  @Query("select t from #{#entityName} t where t.attribute = ?1")
  List<T> findAllByAttribute(String attribute);
}

public interface ConcreteRepository
  extends MappedTypeRepository<ConcreteType> { … }
```

In the preceding example, the MappedTypeRepository interface is the common parent interface for a few domain types extending AbstractMappedType. It also defines the generic findAllByAttribute(…) method, which can be used on instances of the specialized repository interfaces. If you now invoke findByAllAttribute(…) on ConcreteRepository, the query becomes select t from ConcreteType t where t.attribute = ?1.
SpEL expressions to manipulate arguments may also be used to manipulate method arguments. In these SpEL expressions the entity name is not available, but the arguments are. They can be accessed by name or index as demonstrated in the following example.

Example . Using SpEL expressions in repository query methods - accessing arguments.

```
@Query("select u from User u where u.firstname = ?1 and u.firstname=?#{[0]} and
u.emailAddress = ?#{principal.emailAddress}")
List<User> findByFirstnameAndCurrentUserWithCustomQuery(String firstname);
```

For like-conditions one often wants to append % to the beginning or the end of a String valued parameter. This can be done by appending or prefixing a bind parameter marker or a SpEL expression with %. Again the following example demonstrates this.

Example . Using SpEL expressions in repository query methods - wildcard shortcut.

```
@Query("select u from User u where u.lastname like %:#{[0]}% and u.lastname like
%:lastname%")
List<User> findByLastnameWithSpelExpression(@Param("lastname") String lastname);
```

When using like-conditions with values that are coming from a not secure source the values should be sanitized so they can't contain any wildcards and thereby allow attackers to select more data than they should be able to. For this purpose the escape(String) method is made available in the SpEL context. It prefixes all instances of _ and % in the first argument with the single character from the second argument. In combination with the escape clause of the like expression available in JPQL and standard SQL this allows easy cleaning of bind parameters.
Example . Using SpEL expressions in repository query methods - sanitizing input values.

```
@Query("select u from User u where u.firstname like %?#{escape([0])}% escape
?#{escapeCharacter()}")
List<User> findContainingEscaped(String namePart);
```

Given this method declaration in a repository interface findContainingEscaped("Peter_") will find Peter_Parker but not Peter Parker. The escape character used can be configured by setting the escapeCharacter of the @EnableJpaRepositories annotation. Note that the method escape(String) available in the SpEL context will only escape the SQL and JPQL standard wildcards _ and %. If the underlying database or the JPA implementation supports additional wildcards these will not get escaped.

1.3.8. Modifying Queries

All the previous sections describe how to declare queries to access a given entity or collection of entities. You can add custom modifying behavior by using the custom method facilities described in "Custom Implementations for Spring Data Repositories". As this approach is feasible for comprehensive custom functionality, you can modify queries that only need parameter binding by annotating the query method with @Modifying, as shown in the following example:
Example. Declaring manipulating queries

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

Doing so triggers the query annotated to the method as an updating query instead of a selecting one. As the EntityManager might contain outdated entities after the execution of the modifying query, we do not automatically clear it (see the JavaDoc of EntityManager.clear() for details), since this effectively drops all non-flushed changes still pending in the EntityManager. If you wish the EntityManager to be cleared automatically, you can set the @Modifying annotation's clearAutomatically attribute to true.
The @Modifying annotation is only relevant in combination with the @Query annotation. Derived query methods or custom methods do not require this annotation.

Derived Delete Queries

Spring Data JPA also supports derived delete queries that let you avoid having to declare the JPQL query explicitly, as shown in the following example:

Example Using a derived delete query

```
interface UserRepository extends Repository<User, Long> {
 void deleteByRoleId(long roleId);

 @Modifying
 @Query("delete from User u where u.role.id = ?1")
 void deleteInBulkByRoleId(long roleId);
}
```

Although the deleteByRoleId(…) method looks like it basically produces the same result as the deleteInBulkByRoleId(…), there is an important difference between the two method declarations in terms of the way they are run. As the name suggests, the latter method issues a single JPQL query (the one defined in the annotation) against the database. This means even currently loaded instances of User do not see lifecycle callbacks invoked.
To make sure lifecycle queries are actually invoked, an invocation of deleteByRoleId(…) runs a query and then deletes the returned instances one by one, so that the persistence provider can actually invoke @PreRemove callbacks on those entities.
In fact, a derived delete query is a shortcut for running the query and then calling CrudRepository.delete(Iterable<User> users) on the result and keeping behavior in sync with the implementations of other delete(…) methods in CrudRepository.

1.3.9. Applying Query Hints

To apply JPA query hints to the queries declared in your repository interface, you can use the @QueryHints annotation. It takes an array of JPA @QueryHint annotations plus a boolean flag to potentially disable the hints applied to the additional count query triggered when applying pagination, as shown in the following example:
Example 74. Using QueryHints with a repository method

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

 @QueryHints(value = { @QueryHint(name = "name", value = "value")},
          forCounting = false)
 Page<User> findByLastname(String lastname, Pageable pageable);
}
```

The preceding declaration would apply the configured @QueryHint for that actually query but omit applying it to the count query triggered to calculate the total number of pages.

1.3.10. Configuring Fetch- and LoadGraphs

The JPA 2.1 specification introduced support for specifying Fetch- and LoadGraphs that we also support with the @EntityGraph annotation, which lets you reference a @NamedEntityGraph definition. You can use that annotation on an entity to configure the fetch plan of the resulting query. The type (Fetch or Load) of the fetching can be configured by using the type attribute on the @EntityGraph annotation. See the JPA 2.1 Spec 3.7.4 for further reference.
The following example shows how to define a named entity graph on an entity:

Example. Defining a named entity graph on an entity.

```
@Entity
@NamedEntityGraph(name = "GroupInfo.detail",
  attributeNodes = @NamedAttributeNode("members"))
public class GroupInfo {
  // default fetch mode is lazy.
  @ManyToMany
  List<GroupMember> members = new ArrayList<GroupMember>();

  …
}
```

The following example shows how to reference a named entity graph on a repository query method:

Example . Referencing a named entity graph definition on a repository query method.

```
@Repository
public interface GroupRepository extends CrudRepository<GroupInfo, String> {

 @EntityGraph(value = "GroupInfo.detail", type = EntityGraphType.LOAD)
 GroupInfo getByGroupName(String name);

}
```

It is also possible to define ad hoc entity graphs by using @EntityGraph. The provided attributePaths are translated into the according EntityGraph without needing to explicitly add @NamedEntityGraph to your domain types, as shown in the following example:
Example 77. Using AD-HOC entity graph definition on an repository query method.

```
@Repository
public interface GroupRepository extends CrudRepository<GroupInfo, String> {

 @EntityGraph(attributePaths = { "members" })
 GroupInfo getByGroupName(String name);

}
```

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.

Imagine a repository and aggregate root type such as the following example:

Example. A sample aggregate and repository

```
class Person {

 @Id UUID id;
 String firstname, lastname;
 Address address;

 static class Address {
   String zipCode, city, street;
```

```
  }
}

interface PersonRepository extends Repository<Person, UUID> {

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

Now imagine that we want to retrieve the person’s name attributes only. What means does Spring Data offer to achieve this? The rest of this chapter answers that question.

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, as shown in the following example:

Example A projection interface to retrieve a subset of attributes

```
interface NamesOnly {

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

The important bit here is that the properties defined here exactly match properties in the aggregate root. Doing so lets a query method be added as follows:

Example A repository using an interface based projection with a query method

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

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

The query execution engine creates proxy instances of that interface at runtime for each element returned and forwards calls to the exposed methods to the target object.

Projections can be used recursively. If you want to include some of the Address information as well, create a projection interface for that and return that interface from the declaration of getAddress(), as shown in the following example:
Example. A projection interface to retrieve a subset of attributes

```
interface PersonSummary {

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

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

On method invocation, the address property of the target instance is obtained and wrapped into a projecting proxy in turn.

Closed Projections

A projection interface whose accessor methods all match properties of the target aggregate is considered to be a closed projection. The following example (which we used earlier in this chapter, too) is a closed projection:

Example. A closed projection

```
interface NamesOnly {

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

If you use a closed projection, Spring Data can optimize the query execution, because we know about all the attributes that are needed to back the projection proxy. For more details on that, see the module-specific part of the reference documentation.

Open Projections

Accessor methods in projection interfaces can also be used to compute new values by using the @Value annotation, as shown in the following example:
Example. An Open Projection

```
interface NamesOnly {

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

The aggregate root backing the projection is available in the target variable. A projection interface using @Value is an open projection. Spring Data cannot apply query execution optimizations in this case, because the SpEL expression could use any attribute of the aggregate root.
The expressions used in @Value should not be too complex — you want to avoid programming in String variables. For very simple expressions, one option might be to resort to default methods (introduced in Java 8), as shown in the following example:
Example A projection interface using a default method for custom logic

```
interface NamesOnly {

  String getFirstname();
  String getLastname();

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

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, as shown in the following example:

Example. Sample Person object

```
@Component
class MyBean {

  String getFullName(Person person) {
    …
  }
}

interface NamesOnly {

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

Notice how the SpEL expression refers to myBean and invokes the getFullName(…) method and forwards the projection target as a method parameter. Methods backed by SpEL expression evaluation can also use method parameters, which can then be referred to from the expression. The method parameters are available through an Object array named args. The following example shows how to get a method parameter from the args array:

Example. Sample Person object

```
interface NamesOnly {

  @Value("#{args[0] + ' ' + target.firstname + '!'}")
  String getSalutation(String prefix);
}
```

Again, for more complex expressions, you should use a Spring bean and let the expression invoke a method, as described earlier.

Nullable Wrappers

Getters in projection interfaces can make use of nullable wrappers for improved null-safety. Currently supported wrapper types are:

- java.util.Optional
- com.google.common.base.Optional
- scala.Option
- io.vavr.control.Option

Example . A projection interface using nullable wrappers

```
interface NamesOnly {

  Optional<String> getFirstname();
}
```

If the underlying projection value is not null, then values are returned using the present-representation of the wrapper type. In case the backing value is null, then the getter method returns the empty representation of the used wrapper type.

Class-based Projections (DTOs)

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.

If the store optimizes the query execution by limiting the fields to be loaded, the fields to be loaded are determined from the parameter names of the constructor that is exposed.

The following example shows a projecting DTO:

Example . A projecting DTO

```
class NamesOnly {

  private final String firstname, lastname;

  NamesOnly(String firstname, String lastname) {

    this.firstname = firstname;
    this.lastname = lastname;
  }

  String getFirstname() {
    return this.firstname;
  }

  String getLastname() {
    return this.lastname;
  }

  // equals(…) and hashCode() implementations
}
```

Avoid boilerplate code for projection DTOs

You can dramatically simplify the code for a DTO by using Project Lombok, which provides an @Value annotation (not to be confused with Spring’s @Value annotation shown in the earlier interface examples). If you use Project Lombok’s @Value annotation, the sample DTO shown earlier would become the following:

```
@Value
class NamesOnly {
            String firstname, lastname;
}
```

Fields are private final by default, and the class exposes a constructor that takes all fields and automatically gets equals(…) and hashCode() methods implemented.

Dynamic Projections

So far, we have used the projection type as the return type or element type of a collection. However, 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 such as the one shown in the following example:

Example . A repository using a dynamic projection parameter

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

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

This way, the method can be used to obtain the aggregates as is or with a projection applied, as shown in the following example:

Example. Using a repository with dynamic projections

```
void someMethod(PersonRepository people) {

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

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

1.4. Stored Procedures

The JPA 2.1 specification introduced support for calling stored procedures by using the JPA criteria query API. We Introduced the @Procedure annotation for declaring stored procedure metadata on a repository method.
The examples to follow use the following stored procedure:

Example. The definition of the plus1inout **procedure in HSQL DB.**

```
/;
DROP procedure IF EXISTS plus1inout
/;
```

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

Metadata for stored procedures can be configured by using
the NamedStoredProcedureQuery annotation on an entity type.
Example. StoredProcedure metadata definitions on an entity.

```
@Entity
@NamedStoredProcedureQuery(name = "User.plus1", procedureName = "plus1inout",
parameters = {
  @StoredProcedureParameter(mode = ParameterMode.IN, name = "arg", type =
Integer.class),
  @StoredProcedureParameter(mode = ParameterMode.OUT, name = "res", type =
Integer.class) })
public class User {}
```

Note that @NamedStoredProcedureQuery has two different names for the stored procedure. name is the name JPA uses. procedureName is the name the stored procedure has in the database.
You can reference stored procedures from a repository method in multiple ways. The stored procedure to be called can either be defined directly by using the value or procedureName attribute of the @Procedure annotation. This refers directly to the stored procedure in the database and ignores any configuration via @NamedStoredProcedureQuery.
Alternatively you may specify the @NamedStoredProcedureQuery.name attribute as the @Procedure.name attribute. If neither value, procedureName nor name is configured, the name of the repository method is used as the name attribute.
The following example shows how to reference an explicitly mapped procedure:

Example Referencing explicitly mapped procedure with name "plus1inout" in database.

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

The following example is equivalent to the previous one but uses the procedureName alias:
Example. Referencing implicitly mapped procedure with name "plus1inout" in database via procedureName **alias.**

```
@Procedure(procedureName = "plus1inout")
Integer callPlus1InOut(Integer arg);
```

The following is again equivalent to the previous two but using the method name instead of an explicite annotation attribute.

Example. Referencing implicitly mapped named stored procedure "User.plus1" in EntityManager **by using the method name.**

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

The following example shows how to reference a stored procedure by referencing the @NamedStoredProcedureQuery.name attribute.

Example. Referencing explicitly mapped named stored procedure "User.plus1IO" in EntityManager**.**

```
@Procedure(name = "User.plus1IO")
Integer entityAnnotatedCustomNamedProcedurePlus1IO(@Param("arg") Integer arg);
```

If the stored procedure getting called has a single out parameter that parameter may be returned as the return value of the method. If there are multiple out parameters specified in a @NamedStoredProcedureQuery annotation those can be returned as a Map with the key being the parameter name given in the @NamedStoredProcedureQuery annotation.

1.5. Specifications

JPA 2 introduces a criteria API that you can use to build queries programmatically. By writing a criteria, you define the where clause of a query for a domain class. Taking another step back, these criteria can be regarded as a predicate over the entity that is described by the JPA criteria API constraints.
Spring Data JPA takes the concept of a specification from Eric Evans' book, "Domain Driven Design", following the same semantics and providing an API to define such specifications with the JPA criteria API. To support specifications, you can extend your repository interface with the JpaSpecificationExecutor interface, as follows:

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

The additional interface has methods that let you run specifications in a variety of ways. For example, the findAll method returns all entities that match the specification, as shown in the following example:

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

The Specification interface is defined as follows:

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

Specifications can easily be used to build an extensible set of predicates on top of an entity that then can be combined and used with JpaRepository without the need to declare a query (method) for every needed combination, as shown in the following example:
Example 97. Specifications for a Customer

```
public class CustomerSpecs {


  public static Specification<Customer> isLongTermCustomer() {
    return (root, query, builder) -> {
      LocalDate date = LocalDate.now().minusYears(2);
      return builder.lessThan(root.get(Customer_.createdAt), date);
    };
  }

  public static Specification<Customer> hasSalesOfMoreThan(MonetaryAmount value) {
    return (root, query, builder) -> {
      // build query here
    };
  }
}
```

The Customer_ type is a metamodel type generated using the JPA Metamodel generator (see the Hibernate implementation's documentation for an example). So the expression, Customer_.createdAt, assumes the Customer has a createdAt attribute of type Date. Besides that, we have expressed some criteria on a business requirement abstraction level and created executable Specifications. So a client might use a Specification as follows:

Example . Using a simple Specification

```
List<Customer> customers = customerRepository.findAll(isLongTermCustomer());
```

Why not create a query for this kind of data access? Using a single Specification does not gain a lot of benefit over a plain query declaration. The power of specifications really shines when you combine them to create new Specification objects. You can achieve this through the default methods of Specification we provide to build expressions similar to the following:

Example . Combined Specifications

```
MonetaryAmount amount = new MonetaryAmount(200.0, Currencies.DOLLAR);
List<Customer> customers = customerRepository.findAll(
  isLongTermCustomer().or(hasSalesOfMoreThan(amount)));
```

Specification offers some "glue-code" default methods to chain and combine Specification instances. These methods let you extend your data access layer by creating new Specification implementations and combining them with already existing implementations.

1.6. Query by Example

1.6.1. Introduction

This chapter provides an introduction to Query by Example and explains how to use it.

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. In fact, Query by Example does not require you to write queries by using store-specific query languages at all.

1.6.2. Usage

The Query by Example API consists of three parts:

- 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.

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.

Before getting started with Query by Example, you need to have a domain object. To get started, create an interface for your repository, as shown in the following example:

Example . Sample Person object

```
public class Person {

  @Id
  private String id;
  private String firstname;
  private String lastname;
  private Address address;

  // … getters and setters omitted
}
```

The preceding example shows a simple domain object. You can use it to create an Example. By default, fields having null values are ignored, and strings are matched by using the store specific defaults.

Inclusion of properties into a Query by Example criteria is based on nullability. Properties using primitive types (int, double, …) are always included unless ignoring the property path.

Examples can be built by either using the of factory method or by using ExampleMatcher. Example is immutable. The following listing shows a simple Example:
Example 101. Simple Example

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

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

Create a new instance of the domain object.

Set the properties to query.

Create the Example.

You can run the example queries by using repositories. To do so, let your repository interface extend QueryByExampleExecutor<T>. The following listing shows an excerpt from the QueryByExampleExecutor interface:
Example 102. The QueryByExampleExecutor

```
public interface QueryByExampleExecutor<T> {

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

```

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

 // … more functionality omitted.
}
```

1.6.3. Example Matchers

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, as shown in the following example:
Example . Example matcher with customized matching

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

ExampleMatcher matcher = ExampleMatcher.matching()
  .withIgnorePaths("lastname")
  .withIncludeNullValues()
  .withStringMatcherEnding();

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

Create a new instance of the domain object.

Set properties.

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

Construct a new ExampleMatcher to ignore the lastname property path.

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

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

Create a new Example based on the domain object and the configured ExampleMatcher.

By default, the ExampleMatcher expects all values set on the probe to match. If you want to get results matching any of the predicates defined implicitly, use ExampleMatcher.matchingAny().
You can specify behavior for individual properties (such as "firstname" and "lastname" or, for nested properties, "address.city"). You can tune it with matching options and case sensitivity, as shown in the following example:

Example . Configuring matcher options

```
ExampleMatcher matcher = ExampleMatcher.matching()
  .withMatcher("firstname", endsWith())
  .withMatcher("lastname", startsWith().ignoreCase());
}
```

Another way to configure matcher options is to use lambdas (introduced in Java 8). This approach creates a callback that asks the implementor to modify the matcher. You need not return the matcher, because configuration options are held within the matcher instance. The following example shows a matcher that uses lambdas:

Example . Configuring matcher options with lambdas

```
ExampleMatcher matcher = ExampleMatcher.matching()
  .withMatcher("firstname", match -> match.endsWith())
  .withMatcher("firstname", match -> match.startsWith());
}
```

Queries created by Example use a merged view of the configuration. Default matching settings can be set at the ExampleMatcher level, while individual settings can be applied to particular property paths. Settings that are set on ExampleMatcher are inherited by property path settings unless they are defined explicitly. Settings on a property patch have higher precedence than default settings. The following table describes the scope of the various ExampleMatcher settings:

Table 4. Scope of ExampleMatcher **settings**

Setting	Scope
Null-handling	ExampleMatcher
String matching	ExampleMatcher and property path
Ignoring properties	Property path
Case sensitivity	ExampleMatcher and property path
Value transformation	Property path

1.6.4. Running an Example

In Spring Data JPA, you can use Query by Example with Repositories, as shown in the following example:

Example 106. Query by Example using a Repository

```
public interface PersonRepository extends JpaRepository<Person, String> { … }

public class PersonService {

  @Autowired PersonRepository personRepository;

  public List<Person> findPeople(Person probe) {
    return personRepository.findAll(Example.of(probe));
  }
}
```

Currently, only SingularAttribute properties can be used for property matching.

The property specifier accepts property names (such as firstname and lastname). You can navigate by chaining properties together with dots (address.city). You can also tune it with matching options and case sensitivity.
The following table shows the various StringMatcher options that you can use and the result of using them on a field named firstname:

Table 5. StringMatcher **options**

Matching	**Logical result**
DEFAULT (case-sensitive)	firstname = ?0
DEFAULT (case-insensitive)	LOWER(firstname) = LOWER(?0)
EXACT (case-sensitive)	firstname = ?0
EXACT (case-insensitive)	LOWER(firstname) = LOWER(?0)
STARTING (case-sensitive)	firstname like ?0 + '%'
STARTING (case-insensitive)	LOWER(firstname) like LOWER(?0) + '%'
ENDING (case-sensitive)	firstname like '%' + ?0
ENDING (case-insensitive)	LOWER(firstname) like '%' + LOWER(?0)
CONTAINING (case-sensitive)	firstname like '%' + ?0 + '%'
CONTAINING (case-insensitive)	LOWER(firstname) like '%' + LOWER(?0) + '%'

1.7. Transactionality

By default, CRUD methods on repository instances are transactional. 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. For details, see JavaDoc of SimpleJpaRepository. If you need to tweak transaction configuration for one of the methods declared in a repository, redeclare the method in your repository interface, as follows:
Example . Custom transaction configuration for CRUD

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

  @Override
  @Transactional(timeout = 10)
  public List<User> findAll();

  // Further query method declarations
}
```

Doing so causes the findAll() method to run with a timeout of 10 seconds and without the readOnly flag.

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. The following example shows how to use such a facade for more than one repository:

Example . Using a facade to define transactions for multiple repository calls

```
@Service
class UserManagementImpl implements UserManagement {

  private final UserRepository userRepository;
  private final RoleRepository roleRepository;

  @Autowired
  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);
    }
}
```

This example causes call to addRoleToAllUsers(…) to run inside a transaction (participating in an existing one or creating a new one if none are already running). The transaction configuration at the repositories is then neglected, as the outer transaction configuration determines the actual one used. Note that you must activate <tx:annotation-driven /> or use @EnableTransactionManagement explicitly to get annotation-based configuration of facades to work. This example assumes you use component scanning.
Note that the call to save is not strictly necessary from a JPA point of view, but should still be there in order to stay consistent to the repository abstraction offered by Spring Data.

1.7.1. Transactional query methods

To let your query methods be transactional, use @Transactional at the repository interface you define, as shown in the following example:
Example 109. Using @Transactional at query methods

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

  List<User> findByLastname(String lastname);
  @Modifying
  @Transactional
  @Query("delete from User u where u.active = false")
  void deleteInactiveUsers();
}
```

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.

You can use transactions for read-only queries and mark them as such by setting the readOnly flag. Doing so does not, however, act as a check that you do not trigger a manipulating query (although some databases reject INSERT and UPDATE statements inside a read-only transaction). The readOnly flag is instead propagated as a hint to the underlying JDBC driver for performance optimizations. Furthermore, Spring performs some optimizations on the underlying JPA provider. For example, when used with Hibernate, the flush mode is set to NEVER when you configure a transaction as readOnly, which causes Hibernate to skip dirty checks (a noticeable improvement on large object trees).