5. JPA Repositories

This chapter will point out the specialties for repository support for JPA. This builds on the core repository support explained in [Working with Spring Data Repositories](http://docs.spring.io/spring-data/jpa/docs/1.11.3.RELEASE/reference/html/" \l "repositories). So make sure you’ve got a sound understanding of the basic concepts explained there.

5.1. Introduction

5.1.1. 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 using the repositories element:

*Example 42. Setting up JPA repositories 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

http://www.springframework.org/schema/beans/spring-beans.xsd

http://www.springframework.org/schema/data/jpa

http://www.springframework.org/schema/data/jpa/spring-jpa.xsd">

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

</beans>

Using this element looks up Spring Data repositories as described in [Creating repository instances](http://docs.spring.io/spring-data/jpa/docs/1.11.3.RELEASE/reference/html/" \l "repositories.create-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 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 we will automatically lookup 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 and/or EntityManagerFactory beans have been configured. Default to a single defined PlatformTransactionManager inside the current ApplicationContext. |

Note that we require a PlatformTransactionManager bean named transactionManager to be present if no explicit transaction-manager-ref is defined.

5.1.2. Annotation based configuration

The Spring Data JPA repositories support cannot only be activated through an XML namespace but also using an annotation through JavaConfig.

*Example 43. 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() {

JpaTransactionManager txManager = new JpaTransactionManager();

txManager.setEntityManagerFactory(entityManagerFactory());

return txManager;

}

}

|  |  |
| --- | --- |
|  | It’s important to create LocalContainerEntityManagerFactoryBean and not EntityManagerFactorydirectly since the former also participates in exception translation mechanisms besides simply creating EntityManagerFactory. |

The just shown configuration class sets up an embedded HSQL database using the EmbeddedDatabaseBuilder API of spring-jdbc. We then set up a EntityManagerFactory and use Hibernate as sample persistence provider. The last infrastructure component declared here is the JpaTransactionManager. We finally activate Spring Data JPA repositories using the @EnableJpaRepositories annotation which essentially carries the same attributes as the XML namespace does. If no base package is configured it will use the one the configuration class resides in.

5.2. Persisting entities

5.2.1. Saving entities

Saving an entity can be performed via the CrudRepository.save(…)-Method. It will persist or merge the given entity using the underlying JPA EntityManager. If the entity has not been persisted yet Spring Data JPA will save the entity via a call to the entityManager.persist(…) method, otherwise the entityManager.merge(…) method will be called.

Entity state detection strategies

Spring Data JPA offers the following strategies to detect whether an entity is new or not:

|  |  |
| --- | --- |
| *Table 3. Options for detection whether an entity is new in Spring Data JPA* | |
| Id-Property inspection (**default**) | By default Spring Data JPA inspects the identifier property of the given entity. If the identifier property is null, then the entity will be assumed as new, otherwise as not new. |
| Implementing Persistable | If an entity implements Persistable, Spring Data JPA will delegate the new detection to the isNew(…) method of the entity. See the [JavaDoc](http://docs.spring.io/spring-data/data-commons/docs/current/api/index.html?org/springframework/data/domain/Persistable.html) for details. |
| Implementing EntityInformation | You can customize the EntityInformation abstraction used in the SimpleJpaRepositoryimplementation 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](http://docs.spring.io/spring-data/data-jpa/docs/current/api/index.html?org/springframework/data/jpa/repository/support/JpaRepositoryFactory.html) for details. |

5.3. Query methods

5.3.1. Query lookup strategies

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

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 NamedQueries](http://docs.spring.io/spring-data/jpa/docs/1.11.3.RELEASE/reference/html/#jpa.query-methods.named-queries) for more information) or rather annotate your query method with @Query (see [Using @Query](http://docs.spring.io/spring-data/jpa/docs/1.11.3.RELEASE/reference/html/#jpa.query-methods.at-query) for details).

5.3.2. Query creation

Generally the query creation mechanism for JPA works as described in [Query methods](http://docs.spring.io/spring-data/jpa/docs/1.11.3.RELEASE/reference/html/" \l "repositories.query-methods). Here’s a short example of what a JPA query method translates into:

*Example 44. Query creation from method names*

public interface UserRepository extends Repository<User, Long> {

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

}

We will create a query using the JPA criteria API from this but essentially this translates into the following query: select u from User u where u.emailAddress = ?1 and u.lastname = ?2. Spring Data JPA will do a property check and traverse nested properties as described in [Property expressions](http://docs.spring.io/spring-data/jpa/docs/1.11.3.RELEASE/reference/html/#repositories.query-methods.query-property-expressions). Here’s an overview of the keywords supported for JPA and what a method containing that keyword essentially translates to.

| *Table 4. Supported keywords inside method names* | | |
| --- | --- | --- |
| **Keyword** | **Sample** | **JPQL snippet** |
| And | findByLastnameAndFirstname | … where x.lastname = ?1 and x.firstname = ?2 |
| Or | findByLastnameOrFirstname | … where x.lastname = ?1 or x.firstname = ?2 |
| Is,Equals | findByFirstname,findByFirstnameIs,findByFirstnameEquals | … where x.firstname = ?1 |
| Between | findByStartDateBetween | … where x.startDate between ?1 and ?2 |
| LessThan | findByAgeLessThan | … where x.age < ?1 |
| LessThanEqual | findByAgeLessThanEqual | … where x.age <= ?1 |
| GreaterThan | findByAgeGreaterThan | … where x.age > ?1 |
| GreaterThanEqual | findByAgeGreaterThanEqual | … where x.age >= ?1 |
| After | findByStartDateAfter | … where x.startDate > ?1 |
| Before | findByStartDateBefore | … where x.startDate < ?1 |
| IsNull | findByAgeIsNull | … where x.age is null |
| IsNotNull,NotNull | findByAge(Is)NotNull | … where x.age not null |
| Like | findByFirstnameLike | … where x.firstname like ?1 |
| NotLike | findByFirstnameNotLike | … where x.firstname not like ?1 |
| StartingWith | findByFirstnameStartingWith | … where x.firstname like ?1 (parameter bound with 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> age) | … where x.age not in ?1 |
| True | findByActiveTrue() | … where x.active = true |
| False | findByActiveFalse() | … where x.active = false |
| IgnoreCase | findByFirstnameIgnoreCase | … where UPPER(x.firstame) = UPPER(?1) |

|  |  |
| --- | --- |
|  | In and NotIn also take any subclass of Collection as parameter as well as arrays or varargs. For other syntactical versions of the very same logical operator check [Repository query keywords](http://docs.spring.io/spring-data/jpa/docs/1.11.3.RELEASE/reference/html/#repository-query-keywords). |

5.3.3. Using JPA NamedQueries

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

XML named query definition

To use XML configuration simply add the necessary <named-query /> element to the orm.xml JPA configuration file located in 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 45. XML named query configuration*

<named-query name="User.findByLastname">

<query>select u from User u where u.lastname = ?1</query>

</named-query>

As you can see the query has a special name which will be used to resolve it at runtime.

Annotation configuration

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

*Example 46. 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 execution of these named queries all you need to do is to specify the UserRepository as follows:

*Example 47. Query method declaration in UserRepository*

public interface UserRepository extends JpaRepository<User, Long> {

List<User> findByLastname(String lastname);

User findByEmailAddress(String emailAddress);

}

Spring Data will try 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 example here would use the named queries defined above instead of trying to create a query from the method name.

5.3.4. 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 executes them you actually can bind them directly using the Spring Data JPA @Query annotation rather than annotating them to the domain class. This will free the domain class from persistence specific information and co-locate the query to the repository interface.

Queries annotated to the query method will take precedence over queries defined using @NamedQuery or named queries declared in orm.xml.

*Example 48. 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 execution mechanism for manually defined queries using @Query allows the definition of advanced LIKEexpressions inside the query definition.

*Example 49. 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 just shown sample LIKE delimiter character % is recognized and the query transformed into a valid JPQL query (removing the %). Upon query execution the parameter handed into the method call gets augmented with the previously recognized LIKE pattern.