2. JPA Repositories

Abstract

This chapter includes details of the JPA repository implementation.

2.1 Introduction

2.1.1 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 2.1. 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 Section 1.2.3, “Creating repository instances”. Beyond that it activates persistence exception translation for all beans annotated with `@Repository` to let exceptions being thrown by the JPA presistence 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 detailled control over the setup of the repositories:

Table 2.1. 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 single `EntityManagerFactory` configured 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.

2.1.2 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 2.2. 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 EntityManagerFactory entityManagerFactory() {

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

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

    return factory.getObject();
  }

  @Bean
  public PlatformTransactionManager transactionManager() {

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

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 eventually 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.

2.2 Persisting entities

2.2.1 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 2.2. Options for detection whether an entity is new in Spring Data JPA

Id-Property inspection (**default**)	By default Spring Data JPA inspects the Id-Property of the given Entity. If the Id-Property is `null`, then the entity will be assumed as new, otherwise as not new.
Implementing `Persistable`	If an entity implements the `Persistable` interface, Spring Data JPA will delegate the new-detection to the `isNew` - Method of the Entity. See the JavaDoc for

	details.
Implementing `EntityInformation`	One can customize the `EntityInformation` abstraction used in the `SimpleJpaRepository` implementation by creating a subclass of `JpaRepositoryFactory` and overriding the `getEntityInformation`-Method accordingly. One then has to register the custom implementation of `JpaRepositoryFactory` as a Spring bean. Note that this should be rarely necessary. See the JavaDoc for details.

2.3 Query methods

2.3.1 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 Section 2.3.3, “Using JPA NamedQueries” for more information) or rather annotate your query method with `@Query` (see Section 2.3.4, “Using @Query” for details).

2.3.2 Query creation

Generally the query creation mechanism for JPA works as described in Section 1.2, “Query methods”. Here's a short example of what a JPA query method translates into:

Example 2.3. 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 ???. Here's an overview of the keywords supported for JPA and what a method containing that keyword essentially translates to.

Table 2.3. 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`
`Between`	`findByStartDateBetween`	`… where x.startDate between 1? and ?2`
`LessThan`	`findByAgeLessThan`	`… where x.age < ?1`
`GreaterThan`	`findByAgeGreaterThan`	`… where x.age > ?1`
`After`	`findByStartDateAfter`	`… where x.startDate > ?1`
`Before`	`findByStartDateBefore`	`… where x.startDate < ?1`
`IsNull`	`findByAgeIsNull`	`… where x.age is null`
`IsNotNull,NotNull`	`findByAge(Is)NotNull`	`… where x.age not null`
`Like`	`findByFirstnameLike`	`… where x.firstname like ?1`

NotLike	findByFirstnameNotLike	… where x.firstname not like ?1
StartingWith	findByFirstnameStartingWith	… where x.firstname like ?1 (parameter bound with 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

Note

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 Appendix B, *Repository query keywords*.

2.3.3 Using JPA NamedQueries

Note

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 2.4. 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 2.5. 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 2.6. 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.

2.3.4 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 2.7. 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` allow the definition of advanced `LIKE` expressions inside the query definition.

Example 2.8. 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.

Native queries

The `@Query` annotation allows to execute native queries by setting the `nativeQuery` flag to true. Note, that we currently don't support execution of pagination or dynamic sorting for native queries as we'd have to manipulate the actual query declared and we cannot do this reliably for native SQL.

Example 2.9. 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 = ?0", nativeQuery = t
  User findByEmailAddress(String emailAddress);
}
```

2.3.5 Using named parameters

By default Spring Data JPA will use position based parameter binding as described in all the samples above. This makes query methods a little error prone to 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 2.10. Using named parameters

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

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

Note that the method parameters are switched according to the occurrence in the query defined.

2.3.6 Using SpEL expressions

As of Spring Data JPA Release 1.4 we support the usage of restricted SpEL template expressions in manually defined queries via `@Query`. Upon query execution these expressions are evaluated against a predefined set of variables. We support the following list of variables to be used in a manual query.

Table 2.4. Supported variables inside SpEL based query templates

Variable	Usage	Description
`entityName`	`select x from #{#entityName} x`	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 then it will be used. Otherwise the simple class-name of the domain type will be 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 with a manually defined query. In order not to have to state the actual entity name in the query string of a `@Query` annotation one can use the `#{#entityName}` Variable.

Example 2.11. 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);
}
```

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` will pick up potential future remappings of the `User` class to a different entity name (e.g. 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. In order not to have to 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.

Example 2.12. 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 example the interface `MappedTypeRepository` is the common parent interface for a few domain types extending `AbstractMappedType`. It also defines the generic method `findAllByAttribute(…)` which can be used on instances of the specialized repository interfaces. If you now invoke `findByAllAttribute(…)` on `ConcreteRepository` the query being executed will be `select t from ConcreteType t where t.attribute = ?1`.

2.3.7 Modifying queries

All the sections above describe how to declare queries to access a given entity or collection of entities. Of course you can add custom modifying behaviour by using facilities described in Section 1.3, “Custom implementations for Spring Data repositories”.

As this approach is feasible for comprehensive custom functionality, you can achieve the execution of modifying queries that actually only need parameter binding by annotating the query method with `@Modifying`:

Example 2.13. Declaring manipulating queries

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

This will trigger the query annotated to the method as 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 JavaDoc of `EntityManager.clear()` for details) since this will effectively drop all non-flushed changes still pending in the `EntityManager`. If you wish the `EntityManager` to be cleared automatically you can set `@Modifying` annotation's `clearAutomatically` attribute to `true`;

2.3.8 Applying query hints

To apply JPA `QueryHints` 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 addtional count query triggered when applying pagination.

Example 2.14. 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 just shown 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.

2.4 Specifications

JPA 2 introduces a criteria API that can be used to build queries programmatically. Writing a `criteria` you actually define the where-clause of a query for a domain class. Taking another step back these criteria can be regarded as 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` using the JPA criteria API. To support specifications you can extend your repository interface with the `JpaSpecificationExecutor` interface:

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

The additional interface carries methods that allow you to execute `Specifications` in a variety of ways.

For example, the `findAll` method will return all entities that match the specification:

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

The `Specification` interface is as follows:

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

Okay, so what is the typical use case? `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. Here's an example:

Example 2.15. Specifications for a Customer

```
public class CustomerSpecs {

  public static Specification<Customer> isLongTermCustomer() {
    return new Specification<Customer>() {
      public Predicate toPredicate(Root<Customer> root, CriteriaQuery<?> query,
            CriteriaBuilder builder) {

         LocalDate date = new LocalDate().minusYears(2);
         return builder.lessThan(root.get(Customer_.createdAt), date);
      }
    };
  }

  public static Specification<Customer> hasSalesOfMoreThan(MontaryAmount value)
    return new Specification<Customer>() {
      public Predicate toPredicate(Root<T> root, CriteriaQuery<?> query,
            CriteriaBuilder builder) {
```

```
         // build query here
      }
    };
  }
}
```

Admittedly the amount of boilerplate leaves room for improvement (that will hopefully be reduced by Java 8 closures) but the client side becomes much nicer as you will see below. The `Customer_` type is a metamodel type generated using the JPA Metamodel generator (see the Hibernate implementation's documentation for example). So the expression `Customer_.createdAt` is asuming the `Customer` having 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 2.16. Using a simple Specification

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

Okay, why not simply create a query for this kind of data access? You're right. 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 `Specifications` helper class we provide to build expressions like this:

Example 2.17. Combined Specifications

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

As you can see, `Specifications` offers some glue-code methods to chain and combine `Specifications`. Thus extending your data access layer is just a matter of creating new `Specification` implementations and combining them with ones already existing.

2.5 Transactionality

CRUD methods on repository instances are transactional by default. For reading 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 `Repository`. If you need to tweak transaction configuration for one of the methods declared in `Repository` simply redeclare the method in your repository interface as follows:

Example 2.18. Custom transaction configuration for CRUD

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

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

  // Further query method declarations
}
```

This will cause the `findAll()` method to be executed with a timeout of 10 seconds and without the `readOnly` flag.

Another possibility to alter transactional behaviour is using a facade or service implementation that typically covers more than one repository. Its purpose is to define transactional boundaries for non-CRUD operations:

Example 2.19. 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 will cause call to `addRoleToAllUsers(…)` to run inside a transaction (participating in an existing one or create a new one if none already running). The transaction configuration at the repositories will be neglected then as the outer transaction configuration determines the actual one used. Note that you will have to activate `<tx:annotation-driven />` explicitly to get annotation based configuration at facades working. The example above assumes you are using component scanning.

2.5.1 Transactional query methods

To allow your query methods to be transactional simply use `@Transactional` at the repository interface you define.

Example 2.20. 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 will want the `readOnly` flag set to true as most of the query methods will only read data. In contrast to that `deleteInactiveUsers()` makes use of the `@Modifying` annotation and overrides the transaction configuration. Thus the method will be executed with `readOnly` flag set to false.

Note

It's definitely reasonable to use transactions for read only queries and we can mark them as such by setting the `readOnly` flag. This will not, however, act as 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 instead is propagated as hint to the underlying JDBC driver for performance optimizations. Furthermore, Spring will perform some optimizations on the underlying JPA provider. E.g. 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).

2.6 Locking

To specify the lock mode to be used the `@Lock` annotation can be used on query methods:

Example 2.21. Defining lock metadata on query methods

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

  // Plain query method
  @Lock(LockModeType.READ)
  List<User> findByLastname(String lastname);
}
```

This method declaration will cause the query being triggered to be equipped with the `LockModeType` `READ`. You can also define locking for CRUD methods by redeclaring them in your repository interface and adding the `@Lock` annotation:

Example 2.22. Defining lock metadata on CRUD methods

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

  // Redeclaration of a CRUD method
  @Lock(LockModeType.READ);
  List<User> findAll();
}
```

2.7 Auditing

2.7.1 Basics

Spring Data provides sophisticated support to transparently keep track of who created or changed an entity and the point in time this happened. To benefit from that functionality you have to equip your entity classes with auditing metadata that can be defined either using annotations or by implementing an interface.

Annotation based auditing metadata

We provide `@CreatedBy`, `@LastModifiedBy` to capture the user who created or modified the entity as well as `@CreatedDate` and `@LastModifiedDate` to capture the point in time this happened.

Example 2.23. An audited entity

```
class Customer {
  @CreatedBy
  private User user;

  @CreatedDate
  private DateTime createdDate;

  // … further properties omitted
}
```

As you can see, the annotations can be applied selectively, depending on which information you'd like to capture. For the annotations capturing the points in time can be used on properties of type `org.joda.time.DateTime`, `java.util.Date` as well as `long`/`Long`.

Interface-based auditing metadata

In case you don't want to use annotations to define auditing metadata you can let your domain class implement the `Auditable` interface. It exposes setter methods for all of the auditing properties.

There's also a convenience base class `AbstractAuditable` which you can extend to avoid the need to manually implement the interface methods. Be aware that this increases the coupling of your domain classes to Spring Data which might be something you want to avoid. Usually the annotation based way of defining auditing metadata is preferred as it is less invasive and more flexible.

AuditorAware

In case you use either `@CreatedBy` or `@LastModifiedBy`, the auditing infrastructure somehow needs to become aware of the current principal. To do so, we provide an `AuditorAware<T>` SPI interface that you have to implement to tell the infrastructure who the current user or system interacting with the application is. The generic type `T` defines of what type the properties annotated with `@CreatedBy` or `@LastModifiedBy` have to be.

Here's an example implementation of the interface using Spring Security's `Authentication` object:

Example 2.24. Implementation of AuditorAware based on Spring Security

```
class SpringSecurityAuditorAware implements AuditorAware<User> {

  public User getCurrentAuditor() {

    Authentication authentication = SecurityContextHolder.getContext().getAuthen
    if (authentication == null || !authentication.isAuthenticated()) {
      return null;
    }

    return ((MyUserDetails) authentication.getPrincipal()).getUser();
  }
}
```

The implementation is accessing the `Authentication` object provided by Spring Security and looks up the custom `UserDetails` instance from it that you have created in your `UserDetailsService` implementation. We're assuming here that you are exposing the domain user through that `UserDetails` implementation but you could also look it up from anywhere based on the `Authentication` found.

2.7.2 General auditing configuration

Spring Data JPA ships with an entity listener that can be used to trigger capturing auditing information. So first you have to register the `AuditingEntityListener` inside your `orm.xml` to be used for all entities in your persistence contexts:

Note that the auditing feature requires `spring-aspects.jar` to be on the classpath.

Example 2.25. Auditing configuration orm.xml

```
<persistence-unit-metadata>
  <persistence-unit-defaults>
    <entity-listeners>
      <entity-listener class="….data.jpa.domain.support.AuditingEntityListener"
    </entity-listeners>
  </persistence-unit-defaults>
</persistence-unit-metadata>
```

Now activating auditing functionality is just a matter of adding the Spring Data JPA `auditing` namespace element to your configuration:

Example 2.26. Activating auditing in the Spring configuration

```
<jpa:auditing auditor-aware-ref="yourAuditorAwareBean" />
```

As you can see you have to provide a bean that implements the `AuditorAware` interface which looks as follows:

Example 2.27. `AuditorAware` interface

```
public interface AuditorAware<T, ID extends Serializable> {

    T getCurrentAuditor();
}
```

Usually you will have some kind of authentication component in your application that tracks the user currently working with the system. This component should be `AuditorAware` and thus allow seamless tracking of the auditor.

2.8 Miscellaneous

2.8.1 Merging persistence units

Spring supports having multiple persistence units out of the box. Sometimes, however, you might want to modularize your application but still make sure that all these modules run inside a single persistence unit at runtime. To do so Spring Data JPA offers a `PersistenceUnitManager` implementation that automatically merges persistence units based on their name.

Example 2.28. Using MergingPersistenceUnitmanager

```
<bean class="….LocalContainerEntityManagerFactoryBean">
  <property name="persistenceUnitManager">
    <bean class="….MergingPersistenceUnitManager" />
  </property
</bean>
```

2.8.2 Classpath scanning for @Entity classes and JPA mapping files

A plain JPA setup requires all annotation mapped entity classes listed in `orm.xml`. Same applies to XML mapping files. Spring Data JPA provides a `ClasspathScanningPersistenceUnitPostProcessor` that gets a base package configured and optionally takes a mapping filename pattern. It will then scan the given package for classes annotated with `@Entity` or `@MappedSuperclass` and also loads the configuration files matching the filename pattern and hands them to the JPA configuration. The PostProcessor has to be configured like this

Example 2.29. Using ClasspathScanningPersistenceUnitPostProcessor

```
<bean class="….LocalContainerEntityManagerFactoryBean">
```

```
  <property name="persistenceUnitPostProcessors">
    <list>
      <bean class="org.springframework.data.jpa.support.ClasspathScanningPersist
        <constructor-arg value="com.acme.domain" />
        <property name="mappingFileNamePattern" value="**/*Mapping.xml" />
      </bean>
    </list>
  </property>
</bean>
```

Note

As of Spring 3.1 a package to scan can be configured on the `LocalContainerEntityManagerFactoryBean` directly to enable classpath scanning for entity classes. See the JavaDoc for details.

2.8.3 CDI integration

Instances of the repository interfaces are usually created by a container, which Spring is the most natural choice when working with Spring Data. There's sophisticated support to easily set up Spring to create bean instances documented in Section 1.2.3, “Creating repository instances”. As of version 1.1.0 Spring Data JPA ships with a custom CDI extension that allows using the repository abstraction in CDI environments. The extension is part of the JAR so all you need to do to activate it is dropping the Spring Data JPA JAR into your classpath.

You can now set up the infrastructure by implementing a CDI `Producer` for the `EntityManagerFactory`:

```
class EntityManagerFactoryProducer {

  @Produces
  @ApplicationScoped
  public EntityManagerFactory createEntityManagerFactory() {
    return Persistence.createEntityManagerFactory("my-presistence-unit");
  }

  public void close(@Disposes EntityManagerFactory entityManagerFactory) {
    entityManagerFactory.close();
  }
}
```

The Spring Data JPA CDI extension will pick up all `EntityManagers` availables as CDI beans and create a proxy for a Spring Data repository whenever an bean of a repository type is requested by the container. Thus obtaining an instance of a Spring Data repository is a matter of declaring an `@Injected` property:

```
class RepositoryClient {
  @Inject
  PersonRepository repository;

  public void businessMethod() {

    List<Person> people = repository.findAll();
  }
}
```

Prev Up Next

1. Working with Spring Data Repositories Home Part II. Appendix