Spring Data JPA - Reference Documentation

Oliver Gierke

Preface

1. Project metadata

- Version control - git://github.com/SpringSource/spring-data-jpa.git
- Bugtracker - https://jira.springsource.org/browse/DATAJPA
- Release repository - http://repo.springsource.org/libs-release
- Milestone repository - http://repo.springsource.org/libs-milestone
- Snapshot repository - http://repo.springsource.org/libs-snapshot

Part I. Reference Documentation

Chapter 1. Repositories

1.1. Introduction

Implementing a data access layer of an application has been cumbersome for quite a while. Too much boilerplate code had to be written. Domain classes were anemic and not designed in a real object oriented or domain driven manner.

Using both of these technologies makes developers life a lot easier regarding rich domain model's persistence. Nevertheless the amount of boilerplate code to implement repositories especially is still quite high. So the goal of the repository abstraction of Spring Data is to reduce the effort to implement data access layers for various persistence stores significantly.

The following chapters will introduce the core concepts and interfaces of Spring Data repositories in general for detailled information on the specific features of a particular store consult the later chapters of this document.

Note

As this part of the documentation is pulled in from Spring Data Commons we have to decide for a particular module to be used as example. The configuration and code samples in this chapter are using the JPA module. Make sure you adapt e.g. the XML namespace declaration, types to be extended to the equivalents of the module you're actually using.

1.2. Core concepts

The central interface in Spring Data repository abstraction is `Repository` (probably not that much of a surprise). It is typeable to the domain class to manage as well as the id type of the domain class. This interface mainly acts as marker interface to capture the types to deal with and help us when discovering interfaces that extend this one. Beyond that there's `CrudRepository` which provides some sophisticated functionality around CRUD for the entity being managed.

Example 1.1. `CrudRepository` interface

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

    T save(T entity);                                                    ❶

    T findOne(ID primaryKey);                                            ❷

    Iterable<T> findAll();                                               ❸

    Long count();

    void delete(T entity);                                               ❹

    boolean exists(ID primaryKey);                                       ❺

    // … more functionality omitted.                                     ❻
}
```

❶ Saves the given entity.
❷ Returns the entity identified by the given id.

❸ Returns all entities.
❹ Returns the number of entities.
❺ Deletes the given entity.
❻ Returns whether an entity with the given id exists.

Usually we will have persistence technology specific sub-interfaces to include additional technology specific methods. We will now ship implementations for a variety of Spring Data modules that implement this interface.

On top of the `CrudRepository` there is a `PagingAndSortingRepository` abstraction that adds additional methods to ease paginated access to entities:

Example 1.2. PagingAndSortingRepository

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

    Iterable<T> findAll(Sort sort);

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

Accessing the second page of `User` by a page size of 20 you could simply do something like this:

```
PagingAndSortingRepository<User, Long> repository = // … get access to a bean
Page<User> users = repository.findAll(new PageRequest(1, 20));
```

1.3. Query methods

Next to standard CRUD functionality repositories are usually queries on the underlying datastore. With Spring Data declaring those queries becomes a four-step process:

1. Declare an interface extending `Repository` or one of its sub-interfaces and type it to the domain class it shall handle.

```
public interface PersonRepository extends Repository<User, Long> { … }
```

2. Declare query methods on the interface.

```
List<Person> findByLastname(String lastname);
```

3. Setup Spring to create proxy instances for those interfaces.

```
<?xml version="1.0" encoding="UTF-8"?>
<beans:beans xmlns:beans="http://www.springframework.org/schema/beans"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns="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">

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

</beans>
```

Note

Note that we use the JPA namespace here just by example. If you're using the repository abstraction for any other store you need to change this to the appropriate namespace declaration of your store module which should be exchanging `jpa` in favor of e.g. `mongodb`.

4. Get the repository instance injected and use it.

```
public class SomeClient {

  @Autowired
  private PersonRepository repository;

  public void doSomething() {
    List<Person> persons = repository.findByLastname("Matthews");
  }
```

At this stage we barely scratched the surface of what's possible with the repositories but the general approach should be clear. Let's go through each of these steps and figure out details and various options that you have at each stage.

1.3.1. Defining repository interfaces

As a very first step you define a domain class specific repository interface. It's got to extend `Repository` and be typed to the domain class and an ID type. If you want to expose CRUD methods for that domain type, extend `CrudRepository` instead of `Repository`.

1.3.1.1. Fine tuning repository definition

Usually you will have your repository interface extend `Repository`, `CrudRepository` or `PagingAndSortingRepository`. If you don't like extending Spring Data interfaces at all you can also annotate your repository interface with `@RepositoryDefinition`. Extending `CrudRepository` will expose a complete set of methods to manipulate your entities. If you would rather be selective about the methods being exposed, simply copy the ones you want to expose from `CrudRepository` into your domain repository.

Example 1.3. Selectively exposing CRUD methods

```
interface MyBaseRepository<T, ID extends Serializable> extends Repository<T, ID> {
  T findOne(ID id);
  T save(T entity);
}

interface UserRepository extends MyBaseRepository<User, Long> {

  User findByEmailAddress(EmailAddress emailAddress);
}
```

In the first step we define a common base interface for all our domain repositories and expose `findOne(…)` as well as `save(…)`.These methods will be routed into the base repository implementation of the store of your choice because they are matching the method signatures in `CrudRepository`. So our `UserRepository` will now be able to save users, find single ones by id as well as triggering a query to find `Users` by their email address.

1.3.2. Defining query methods

1.3.2.1. Query lookup strategies

The next thing we have to discuss is the definition of query methods. There are two main ways that the repository proxy is able to come up with the store specific query from the method name. The first option is to derive the query from the method name directly, the second is using some kind of additionally created query. What detailed options are available pretty much depends on the actual store, however, there's got to be some algorithm that decides what actual query is created.

There are three strategies available for the repository infrastructure to resolve the query. The strategy to be used can be configured at the namespace through the `query-lookup-strategy` attribute. However, It might be the case that some of the strategies are not supported for specific datastores. Here are your options:

CREATE

This strategy will try to construct a store specific query from the query method's name. The general approach is to remove a given set of well-known prefixes from the method name and parse the rest of the method. Read more about query construction in Section 1.3.2.2, “Query creation”.

USE_DECLARED_QUERY

This strategy tries to find a declared query which will be used for execution first. The query could be defined by an annotation somewhere or declared by other means. Please consult the documentation of the specific store to find out what options are available for that store. If the repository infrastructure does not find a declared query for the method at bootstrap time it will fail.

CREATE_IF_NOT_FOUND (default)

This strategy is actually a combination of `CREATE` and `USE_DECLARED_QUERY`. It will try to lookup a declared query first but create a custom method name based query if no declared query was found. This is the default lookup strategy and thus will be used if you don't configure anything explicitly. It allows quick query definition by method names but also custom tuning of these queries by introducing declared queries as needed.

1.3.2.2. Query creation

The query builder mechanism built into Spring Data repository infrastructure is useful to build constraining queries over entities of the repository. We will strip the prefixes `findBy`, `find`, `readBy`, `read`, `getBy` as well as `get` from the method and start parsing the rest of it. At a very basic level you can define conditions on entity properties and concatenate them with `AND` and `OR`.

Example 1.4. Query creation from method names

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

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

The actual result of parsing that method will of course depend on the persistence store we create the query for, however, there are some general things to notice. The expressions are usually property traversals combined with operators that can be concatenated. As you can see in the example you can combine property expressions

with And and Or. Beyond that you also get support for various operators like `Between`, `LessThan`, `GreaterThan`, `Like` for the property expressions. As the operators supported can vary from datastore to datastore please consult the according part of the reference documentation.

1.3.2.2.1. Property expressions

Property expressions can just refer to a direct property of the managed entity (as you just saw in the example above). On query creation time we already make sure that the parsed property is at a property of the managed domain class. However, you can also define constraints by traversing nested properties. Assume `Persons` have `Addresses` with `ZipCodes`. In that case a method name of

```
List<Person> findByAddressZipCode(ZipCode zipCode);
```

will create the property traversal `x.address.zipCode`. The resolution algorithm starts with interpreting the entire part (`AddressZipCode`) as property and checks the domain class for a property with that name (uncapitalized). If it succeeds it just uses that. If not it starts splitting up the source at the camel case parts from the right side into a head and a tail and tries to find the according property, e.g. `AddressZip` and `Code`. If we find a property with that head we take the tail and continue building the tree down from there. As in our case the first split does not match we move the split point to the left (`Address`, `ZipCode`).

Although this should work for most cases, there might be cases where the algorithm could select the wrong property. Suppose our `Person` class has an `addressZip` property as well. Then our algorithm would match in the first split round already and essentially choose the wrong property and finally fail (as the type of `addressZip` probably has no code property). To resolve this ambiguity you can use _ inside your method name to manually define traversal points. So our method name would end up like so:

```
List<Person> findByAddress_ZipCode(ZipCode zipCode);
```

1.3.2.3. Special parameter handling

To hand parameters to your query you simply define method parameters as already seen in the examples above. Besides that we will recognizes certain specific types to apply pagination and sorting to your queries dynamically.

Example 1.5. Using Pageable and Sort in query methods

```
Page<User> findByLastname(String lastname, Pageable pageable);

List<User> findByLastname(String lastname, Sort sort);

List<User> findByLastname(String lastname, Pageable pageable);
```

The first method allows you to pass a `Pageable` instance to the query method to dynamically add paging to your statically defined query. `Sorting` options are handed via the `Pageable` instance too. If you only need sorting, simply add a `Sort` parameter to your method. As you also can see, simply returning a `List` is possible as well. We will then not retrieve the additional metadata required to build the actual `Page` instance but rather simply restrict the query to lookup only the given range of entities.

Note

To find out how many pages you get for a query entirely we have to trigger an additional count

query. This will be derived from the query you actually trigger by default.

1.3.3. Creating repository instances

So now the question is how to create instances and bean definitions for the repository interfaces defined.

1.3.3.1. XML Configuration

The easiest way to do so is by using the Spring namespace that is shipped with each Spring Data module that supports the repository mechanism. Each of those includes a repositories element that allows you to simply define a base package that Spring will scan for you.

```
<?xml version="1.0" encoding="UTF-8"?>
<beans:beans xmlns:beans="http://www.springframework.org/schema/beans"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns="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">

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

</beans:beans>
```

In this case we instruct Spring to scan com.acme.repositories and all its sub packages for interfaces extending `Repository` or one of its sub-interfaces. For each interface found it will register the persistence technology specific `FactoryBean` to create the according proxies that handle invocations of the query methods. Each of these beans will be registered under a bean name that is derived from the interface name, so an interface of `UserRepository` would be registered under `userRepository`. The `base-package` attribute allows the use of wildcards, so that you can have a pattern of scanned packages.

Using filters

By default we will pick up every interface extending the persistence technology specific `Repository` sub-interface located underneath the configured base package and create a bean instance for it. However, you might want finer grained control over which interfaces bean instances get created for. To do this we support the use of `<include-filter />` and `<exclude-filter />` elements inside `<repositories />`. The semantics are exactly equivalent to the elements in Spring's context namespace. For details see Spring reference documentation on these elements.

E.g. to exclude certain interfaces from instantiation as repository, you could use the following configuration:

Example 1.6. Using exclude-filter element

```
<repositories base-package="com.acme.repositories">
  <context:exclude-filter type="regex" expression=".*SomeRepository" />
</repositories>
```

This would exclude all interfaces ending in `SomeRepository` from being instantiated.

1.3.3.2. JavaConfig

The repository infrastructure can also be triggered using a store-specific `@Enable${store}Repositories` annotation on a JavaConfig class. For an introduction into Java based configuration of the Spring container please have a look at the reference documentation.[1]

A sample configuration to enable Spring Data repositories would look something like this.

Example 1.7. Sample annotation based repository configuration

```
@Configuration
@EnableJpaRepositories("com.acme.repositories")
class ApplicationConfiguration {

  @Bean
  public EntityManagerFactory entityManagerFactory() {
    // …
  }
}
```

Note that the sample uses the JPA specific annotation which would have to be exchanged dependingon which store module you actually use. The same applies to the definition of the `EntityManagerFactory` bean. Please consult the sections covering the store-specific configuration.

1.3.3.3. Standalone usage

You can also use the repository infrastructure outside of a Spring container usage. You will still need to have some of the Spring libraries on your classpath but you can generally setup repositories programmatically as well. The Spring Data modules providing repository support ship a persistence technology specific `RepositoryFactory` that can be used as follows:

Example 1.8. Standalone usage of repository factory

```
RepositoryFactorySupport factory = … // Instantiate factory here
UserRepository repository = factory.getRepository(UserRepository.class);
```

1.4. Custom implementations

1.4.1. Adding behaviour to single repositories

Often it is necessary to provide a custom implementation for a few repository methods. Spring Data repositories easily allow you to provide custom repository code and integrate it with generic CRUD abstraction and query method functionality. To enrich a repository with custom functionality you have to define an interface and an implementation for that functionality first and let the repository interface you provided so far extend that custom interface.

[1] JavaConfig in the Spring reference documentation - http://static.springsource.org/spring/docs/3.1.x/spring-framework-reference/html/beans.html#beans-java

Example 1.9. Interface for custom repository functionality

```
interface UserRepositoryCustom {

  public void someCustomMethod(User user);
}
```

Example 1.10. Implementation of custom repository functionality

```
class UserRepositoryImpl implements UserRepositoryCustom {

  public void someCustomMethod(User user) {
    // Your custom implementation
  }
}
```

Note that the implementation itself does not depend on Spring Data and can be a regular Spring bean. So you can use standard dependency injection behaviour to inject references to other beans, take part in aspects and so on.

Example 1.11. Changes to the your basic repository interface

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

  // Declare query methods here
}
```

Let your standard repository interface extend the custom one. This makes CRUD and custom functionality available to clients.

Configuration

If you use namespace configuration the repository infrastructure tries to autodetect custom implementations by looking up classes in the package we found a repository using the naming conventions appending the namespace element's attribute `repository-impl-postfix` to the classname. This suffix defaults to `Impl`.

Example 1.12. Configuration example

```
<repositories base-package="com.acme.repository" />

<repositories base-package="com.acme.repository" repository-impl-postfix="FooBar" />
```

The first configuration example will try to lookup a class `com.acme.repository.UserRepositoryImpl` to act as custom repository implementation, where the second example will try to lookup `com.acme.repository.UserRepositoryFooBar`.

Manual wiring

The approach above works perfectly well if your custom implementation uses annotation based configuration and autowiring entirely as it will be treated as any other Spring bean. If your custom implementation bean needs some special wiring you simply declare the bean and name it after the conventions just described. We will then pick up the custom bean by name rather than creating an instance.

Example 1.13. Manual wiring of custom implementations (I)

```
<repositories base-package="com.acme.repository" />

<beans:bean id="userRepositoryImpl" class="…">
  <!-- further configuration -->
</beans:bean>
```

1.4.2. Adding custom behaviour to all repositories

In other cases you might want to add a single method to all of your repository interfaces. So the approach just shown is not feasible. The first step to achieve this is adding and intermediate interface to declare the shared behaviour

Example 1.14. An interface declaring custom shared behaviour

```
public interface MyRepository<T, ID extends Serializable>
  extends JpaRepository<T, ID> {

  void sharedCustomMethod(ID id);
}
```

Now your individual repository interfaces will extend this intermediate interface instead of the `Repository` interface to include the functionality declared. The second step is to create an implementation of this interface that extends the persistence technology specific repository base class which will then act as a custom base class for the repository proxies.

Note

The default behaviour of the Spring `<repositories />` namespace is to provide an implementation for all interfaces that fall under the `base-package`. This means that if left in it's current state, an implementation instance of `MyRepository` will be created by Spring. This is of course not desired as it is just supposed to act as an intermediary between `Repository` and the actual repository interfaces you want to define for each entity. To exclude an interface extending `Repository` from being instantiated as a repository instance it can either be annotate it with `@NoRepositoryBean` or moved out side of the configured `base-package`.

Example 1.15. Custom repository base class

```
public class MyRepositoryImpl<T, ID extends Serializable>
  extends SimpleJpaRepository<T, ID> implements MyRepository<T, ID> {

  private EntityManager entityManager;
```

```
  // There are two constructors to choose from, either can be used.
  public MyRepositoryImpl(Class<T> domainClass, EntityManager entityManager) {
    super(domainClass, entityManager);

    // This is the recommended method for accessing inherited class dependencies.
    this.entityManager = entityManager;
  }

  public void sharedCustomMethod(ID id) {
    // implementation goes here
  }
}
```

The last step is to create a custom repository factory to replace the default `RepositoryFactoryBean` that will in turn produce a custom `RepositoryFactory`. The new repository factory will then provide your `MyRepositoryImpl` as the implementation of any interfaces that extend the `Repository` interface, replacing the `SimpleJpaRepository` implementation you just extended.

Example 1.16. Custom repository factory bean

```
public class MyRepositoryFactoryBean<R extends JpaRepository<T, I>, T, I extends Serializable>
  extends JpaRepositoryFactoryBean<R, T, I> {

  protected RepositoryFactorySupport createRepositoryFactory(EntityManager entityManager) {

    return new MyRepositoryFactory(entityManager);
  }

  private static class MyRepositoryFactory<T, I extends Serializable> extends JpaRepositoryFactory {

    private EntityManager entityManager;

    public MyRepositoryFactory(EntityManager entityManager) {
      super(entityManager);

      this.entityManager = entityManager;
    }

    protected Object getTargetRepository(RepositoryMetadata metadata) {

      return new MyRepositoryImpl<T, I>((Class<T>) metadata.getDomainClass(), entityManager);
    }

    protected Class<?> getRepositoryBaseClass(RepositoryMetadata metadata) {

      // The RepositoryMetadata can be safely ignored, it is used by the JpaRepositoryFactory
      //to check for QueryDslJpaRepository's which is out of scope.
      return MyRepository.class;
    }
  }
}
```

Finally you can either declare beans of the custom factory directly or use the `factory-class` attribute of the Spring namespace to tell the repository infrastructure to use your custom factory implementation.

Example 1.17. Using the custom factory with the namespace

```
<repositories base-package="com.acme.repository"
  factory-class="com.acme.MyRepositoryFactoryBean" />
```

1.5. Extensions

This chapter documents a set of Spring Data extensions that enable Spring Data usage in a variety of contexts. Currently most of the integration is targeted towards Spring MVC.

1.5.1. Domain class web binding for Spring MVC

Given you are developing a Spring MVC web applications you typically have to resolve domain class ids from URLs. By default it's your task to transform that request parameter or URL part into the domain class to hand it layers below then or execute business logic on the entities directly. This should look something like this:

```
@Controller
@RequestMapping("/users")
public class UserController {

  private final UserRepository userRepository;

  public UserController(UserRepository userRepository) {
    userRepository = userRepository;
  }

  @RequestMapping("/{id}")
  public String showUserForm(@PathVariable("id") Long id, Model model) {

    // Do null check for id
    User user = userRepository.findOne(id);
    // Do null check for user
    // Populate model
    return "user";
  }
}
```

First you pretty much have to declare a repository dependency for each controller to lookup the entity managed by the controller or repository respectively. Beyond that looking up the entity is boilerplate as well as it's always a `findOne(…)` call. Fortunately Spring provides means to register custom converting components that allow conversion between a `String` value to an arbitrary type.

PropertyEditors

For versions up to Spring 3.0 simple Java `PropertyEditors` had to be used. Thus, we offer a `DomainClassPropertyEditorRegistrar`, that will look up all Spring Data repositories registered in the `ApplicationContext` and register a custom `PropertyEditor` for the managed domain class

```
<bean class="….web.servlet.mvc.annotation.AnnotationMethodHandlerAdapter">
  <property name="webBindingInitializer">
    <bean class="….web.bind.support.ConfigurableWebBindingInitializer">
      <property name="propertyEditorRegistrars">
        <bean class="org.springframework.data.repository.support.DomainClassPropertyEditorRegistrar" />
      </property>
    </bean>
  </property>
</bean>
```

If you have configured Spring MVC like this you can turn your controller into the following that reduces a lot of the clutter and boilerplate.

```
@Controller
@RequestMapping("/users")
public class UserController {

  @RequestMapping("/{id}")
  public String showUserForm(@PathVariable("id") User user, Model model) {
```

```
    // Do null check for user
    // Populate model
    return "userForm";
  }
}
```

ConversionService

As of Spring 3.0 the `PropertyEditor` support is superseeded by a new conversion infrstructure that leaves all the drawbacks of `PropertyEditors` behind and uses a stateless X to Y conversion approach. We now ship with a `DomainClassConverter` that pretty much mimics the behaviour of `DomainClassPropertyEditorRegistrar`. To register the converter you have to declare `ConversionServiceFactoryBean`, register the converter and tell the Spring MVC namespace to use the configured conversion service:

```
<mvc:annotation-driven conversion-service="conversionService" />

<bean id="conversionService" class="….context.support.ConversionServiceFactoryBean">
  <property name="converters">
    <list>
      <bean class="org.springframework.data.repository.support.DomainClassConverter">
        <constructor-arg ref="conversionService" />
      </bean>
    </list>
  </property>
</bean>
```

1.5.2. Web pagination

```
@Controller
@RequestMapping("/users")
public class UserController {

  // DI code omitted

  @RequestMapping
  public String showUsers(Model model, HttpServletRequest request) {

    int page = Integer.parseInt(request.getParameter("page"));
    int pageSize = Integer.parseInt(request.getParameter("pageSize"));
    model.addAttribute("users", userService.getUsers(pageable));
    return "users";
  }
}
```

As you can see the naive approach requires the method to contain an `HttpServletRequest` parameter that has to be parsed manually. We even omitted an appropriate failure handling which would make the code even more verbose. The bottom line is that the controller actually shouldn't have to handle the functionality of extracting pagination information from the request. So we include a `PageableArgumentResolver` that will do the work for you.

```
<bean class="….web.servlet.mvc.annotation.AnnotationMethodHandlerAdapter">
  <property name="customArgumentResolvers">
    <list>
      <bean class="org.springframework.data.web.PageableArgumentResolver" />
    </list>
  </property>
</bean>
```

This configuration allows you to simplify controllers down to something like this:

```
@Controller
@RequestMapping("/users")
```

```
public class UserController {

  @RequestMapping
  public String showUsers(Model model, Pageable pageable) {

    model.addAttribute("users", userDao.readAll(pageable));
    return "users";
  }
}
```

The `PageableArgumentResolver` will automatically resolve request parameters to build a `PageRequest` instance. By default it will expect the following structure for the request parameters:

Table 1.1. Request parameters evaluated by `PageableArgumentResolver`

`page`	The page you want to retrieve
`page.size`	The size of the page you want to retrieve
`page.sort`	The property that should be sorted by
`page.sort.dir`	The direction that should be used for sorting

In case you need multiple `Pageable`s to be resolved from the request (for multiple tables e.g.) you can use Spring's `@Qualifier` annotation to distinguish one from another. The request parameters then have to be prefixed with `${qualifier}_`. So a method signature like this:

```
public String showUsers(Model model,
      @Qualifier("foo") Pageable first,
      @Qualifier("bar") Pageable second) { … }
```

you'd have to populate `foo_page` and `bar_page` and the according subproperties.

Defaulting

The `PageableArgumentResolver` will use a `PageRequest` with the first page and a page size of 10 by default and will use that in case it can't resolve a `PageRequest` from the request (because of missing parameters e.g.). You can configure a global default on the bean declaration directly. In case you might need controller method specific defaults for the `Pageable` simply annotate the method parameter with `@PageableDefaults` and specify page and page size as annotation attributes:

```
public String showUsers(Model model,
  @PageableDefaults(pageNumber = 0, value = 30) Pageable pageable) { … }
```

1.5.3. Repository populators

If you have been working with the JDBC module of Spring you're probably familiar with the support to populate a DataSource using SQL scripts. A similar abstraction is available on the repositories level although we don't use SQL as data definition language as we need to be store independent of course. Thus the populators support XML (through Spring's OXM abstraction) and JSON (through Jackson) to define data for the repositories to be populated with.

Assume you have a file `data.json` with the following content:

Example 1.18. Data defined in JSON

```
[ { "_class" : "com.acme.Person",
 "firstname" : "Dave",
  "lastname" : "Matthews" },
  { "_class" : "com.acme.Person",
 "firstname" : "Carter",
  "lastname" : "Beauford" } ]
```

You can easily populate you repositories by using the populator elements of the repository namespace provided in Spring Data Commons. To get the just shown data be populated to your `PersonRepository` all you need to do is the following:

Example 1.19. Declaring a Jackson repository populator

```
<?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:repository="http://www.springframework.org/schema/data/repository"
  xsi:schemaLocation="http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans.xsd
    http://www.springframework.org/schema/data/repository
    http://www.springframework.org/schema/data/repository/spring-repository.xsd">

  <repository:jackson-populator location="classpath:data.json" />

</beans>
```

This declaration causes the data.json file being read, deserialized by a Jackson `ObjectMapper`. The type the JSON object will be unmarshalled to will be determined by inspecting the `_class` attribute of the JSON document. We will eventually select the appropriate repository being able to handle the object just deserialized.

To rather use XML to define the repositories shall be populated with you can use the unmarshaller-populator you hand one of the marshaller options Spring OXM provides you with.

Example 1.20. Declaring an unmarshalling repository populator (using JAXB)

```
<?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:repository="http://www.springframework.org/schema/data/repository"
  xmlns:oxm="http://www.springframework.org/schema/oxm"
  xsi:schemaLocation="http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans.xsd
    http://www.springframework.org/schema/data/repository
    http://www.springframework.org/schema/data/repository/spring-repository.xsd
    http://www.springframework.org/schema/oxm
    http://www.springframework.org/schema/oxm/spring-oxm.xsd">

  <repository:unmarshaller-populator location="classpath:data.json" unmarshaller-ref="unmarshaller" />

  <oxm:jaxb2-marshaller contextPath="com.acme" />

</beans>
```

Chapter 2. JPA Repositories

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 1.3.3, “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`.

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 LocalContainerEntityManagerFactoryBean();
    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. Query methods

2.2.1. 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 2.2.3, “Using JPA NamedQueries” for more information) or rather annotate your query method with `@Query` (see Section 2.2.4, “Using @Query” for details).

2.2.2. Query creation

Generally the query creation mechanism for JPA works as described in Section 1.3, “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 Section 1.3.2.2.1, “Property expressions”. Here's an overview of the keywords supported for JPA and what a method containing that keyword essentially translates to.

Table 2.2. 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`

Keyword	Sample	JPQL snippet
NotLike	findByFirstnameNotLike	… where x.firstname not like ?1
StartingWith	findByFirstnameStartingWith	… where x.firstname like ?1 (parameter bound with prepended %)
ngWith	findByFirstnameEndingWith	… where x.firstname like ?1 (parameter bound with appended %)
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*.

2.2.3. 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.

2.2.4. 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);
}
```

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 2.8. 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 = true)
  User findByEmailAddress(String emailAddress);
}
```

2.2.5. 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 2.9. 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);
}
```

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

2.2.6. 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 1.4, “Custom implementations”. 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 2.10. 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 automatically

clear it (see JavaDoc of `EntityManager.clear()` for details). This will effectively drop all non-flushed changes still pending in the `EntityManager`. If you don't wish the `EntityManager` to be cleared automatically you can set `@Modifying` annotation's `clearAutomatically` attribute to `false`;

2.2.7. 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 2.11. 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.

2.3. 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>, JpaSpecificationExecutor {
 …
}
```

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

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

```
List<T> readAll(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 2.12. Specifications for a Customer

```
public class CustomerSpecs {

  public static Specification<Customer> isLongTermCustomer() {
    return new Specification<Customer>() {
      Predicate toPredicate(Root<T> 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>() {
      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. 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 2.13. 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 2.14. Combined Specifications

```
MonetaryAmount amount = new MonetaryAmount(200.0, Currencies.DOLLAR);
List<Customer> customers = customerRepository.readAll(
  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.

2.4. 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 2.15. 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 2.16. 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.readAll()) {
      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.

2.4.1. Transactional query methods

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

Example 2.17. 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).

2.5. Locking

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

Example 2.18. 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 2.19. Defining lock metadata on CRUD methods

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

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

2.6. Auditing

Most applications will require some form of auditability to track when an entity was created or modified and by whom. Spring Data JPA provides facilities to add this audit information to an entity transparently by AOP means. To take part in this functionality your domain classes must implement a more advanced interface:

Example 2.20. `Auditable` interface

```
public interface Auditable<U, ID extends Serializable>
        extends Persistable<ID> {

    U getCreatedBy();

    void setCreatedBy(U createdBy);

    DateTime getCreatedDate();

    void setCreated(Date creationDate);

    U getLastModifiedBy();

    void setLastModifiedBy(U lastModifiedBy);

    DateTime getLastModifiedDate();

    void setLastModified(Date lastModifiedDate);
}
```

As you can see the modifying entity itself only has to be an entity. Mostly this will be some sort of User entity, so we chose U as parameter type.

Note

To minimize boilerplate code Spring Data JPA offers `AbstractPersistable` and `AbstractAuditable` base classes that implement and pre-configure entities. Thus you can decide to only implement the interface or enjoy more sophisticated support by extending the base class.

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:

Example 2.21. 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 2.22. 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 2.23. `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.

2.7. Miscellaneous

2.7.1. 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 2.24. Using MergingPersistenceUnitmanager

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

2.7.2. 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 2.25. Using ClasspathScanningPersistenceUnitPostProcessor

```
<bean class="….LocalContainerEntityManagerFactoryBean">
  <property name="persistenceUnitPostProcessors">
    <list>
      <bean class="org.springframework.data.jpa.support.ClasspathScanningPersistenceUnitPostProcessor">
        <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.

2.7.3. 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 1.3.3, “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();
  }
```

```
}
```

Part II. Appendix

Appendix A. Namespace reference

A.1. The `<repositories />` element

The `<repositories />` triggers the setup of the Spring Data repository infrastructure. The most important attribute is `base-package` which defines the package to scan for Spring Data repository interfaces.[1]

Table A.1. Attributes

Name	Description
`base-package`	Defines the package to be used to be scanned for repository interfaces extending `*Repository` (actual interface is determined by specific Spring Data module) in auto detection mode. All packages below the configured package will be scanned, too. Wildcards are also allowed.
`repository-impl-postfix`	Defines the postfix to autodetect custom repository implementations. Classes whose names end with the configured postfix will be considered as candidates. Defaults to `Impl`.
`query-lookup-strategy`	Determines the strategy to be used to create finder queries. See Section 1.3.2.1, “Query lookup strategies” for details. Defaults to `create-if-not-found`.

[1] see Section 1.3.3.1, “XML Configuration”

Appendix B. Repository query keywords

B.1. Supported query keywords

The following table lists the keywords generally supported by the Spring data repository query derivation mechanism. However consult the store specific documentation for the exact list of supported keywords as some of the ones listed here might not be supported in a particular store.

Table B.1. Query keywords

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

Appendix C. Frequently asked questions

C.1. Common

C.1.1.

I'd like to get more detailed logging information on what methods are called inside `JpaRepository`, e.g. How can I gain them?

You can make use of `CustomizableTraceInterceptor` provided by Spring:

```
<bean id="customizableTraceInterceptor" class="
  org.springframework.aop.interceptor.CustomizableTraceInterceptor">
  <property name="enterMessage" value="Entering $[methodName]($[arguments])"/>
  <property name="exitMessage" value="Leaving $[methodName](): $[returnValue]"/>
</bean>

<aop:config>
  <aop:advisor advice-ref="customizableTraceInterceptor"
    pointcut="execution(public * org.sfw.data.jpa.repository.JpaRepository+.*(..))"/>
</aop:config>
```

C.2. Infrastructure

C.2.1.

Currently I have implemented a repository layer based on `HibernateDaoSupport`. I create a `SessionFactory` by using Spring's `AnnotationSessionFactoryBean`. How do I get Spring Data repositories working in this environment?

You have to replace `AnnotationSessionFactoryBean` with the `LocalContainerEntityManagerFactoryBean`. Supposed you have registered it under `entityManagerFactory` you can reference it in you repositories based on `HibernateDaoSupport` as follows:

Example C.1. Looking up a SessionFactory from an HibernateEntityManagerFactory

```
<bean class="com.acme.YourDaoBasedOnHibernateDaoSupport">
  <property name="sessionFactory">
    <bean factory-bean="entityManagerFactory"
      factory-method="getSessionFactory" />
  </property>
</bean>
```

C.3. Auditing

C.3.1.

I want to use Spring Data JPA auditing capabilities but have my database already set up to set modification and creation date on entities. How to prevent Spring Data from setting the date programmatically.

Just use the `set-dates` attribute of the `auditing` namespace element to false.

Glossary

A

AOP	Aspect oriented programming

C

Commons DBCP	Commons DataBase Connection Pools - Library of the Apache foundation offering pooling implementations of the `DataSource` interface.
CRUD	Create, Read, Update, Delete - Basic persistence operations

D

DAO	Data Access Object - Pattern to separate persisting logic from the object to be persisted
Dependency Injection	Pattern to hand a component's dependency to the component from outside, freeing the component to lookup the dependant itself. For more information see http://en.wikipedia.org/wiki/Dependency_Injection.

E

EclipseLink	Object relational mapper implementing JPA - http://www.eclipselink.org

H

Hibernate	Object relational mapper implementing JPA - http://www.hibernate.org

J

JPA	Java Persistence Api

S

Spring	Java application framework - http://www.springframework.org