Spring Data JPA - Reference Documentation

1.3.3.RELEASE

OliverGierkeSenior ConsultantSpringSource - a division of VMwareogierke@vmware.com

Table of Contents

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

1. Working with Spring Data Repositories

The goal of Spring Data repository abstraction is to significantly reduce the amount of boilerplate code required to implement data access layers for various persistence stores.

Important

Spring Data repository documentation and your module

This chapter explains the core concepts and interfaces of Spring Data repositories. The information in this chapter is pulled from the Spring Data Commons module. It uses the configuration and code samples for the Java Persistence API (JPA) module. Adapt the XML namespace declaration and the types to be extended to the equivalents of the particular module that you are using. Appendix A, *Namespace reference* covers XML configuration which is supported across all Spring Data modules supporting the repository API, Appendix B, *Repository query keywords* covers the query method method keywords supported by the repository abstraction in general. For detailed information on the specific features of your module, consult the chapter on that module of this document.

1.1 Core concepts

The central interface in Spring Data repository abstraction is `Repository` (probably not that much of a surprise). It takes the the domain class to manage as well as the id type of the domain class as type arguments. This interface acts primarily as a marker interface to capture the types to work with and to help you to discover interfaces that extend this one. The `CrudRepository` provides sophisticated CRUD functionality for the entity class that is being managed.

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

    <S extends T> S save(S 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 Indicates whether an entity with the given id exists.

Example 1.1 `CrudRepository` interface

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 `CrudRepository`.

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

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

  Iterable<T> findAll(Sort sort);

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

Example 1.2 PagingAndSortingRepository

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));
```

1.2 Query methods

Standard CRUD functionality repositories usually have 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 subinterfaces and type it to the domain class that it will handle.

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

2. Declare query methods on the interface.

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

3. Set up 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

The JPA namespace is used in this example. If you are 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, for example, `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");
  }
}
```

The sections that follow explain each step.

Defining repository interfaces

As a first step you define a domain class-specific repository interface. The interface must 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`.

Fine-tuning repository definition

Typically, your repository interface will extend `Repository`, `CrudRepository` or `PagingAndSortingRepository`. Alternatively, if you do not want to extend Spring Data interfaces, you can also annotate your repository interface with `@RepositoryDefinition`. Extending `CrudRepository` exposes a complete set of methods to manipulate your entities. If you prefer to be selective about the methods being exposed, simply copy the ones you want to expose from `CrudRepository` into your domain repository.

```
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);
}
```

Example 1.3 Selectively exposing CRUD methods

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

Defining query methods

The repository proxy has two ways to derive a store-specific query from the method name. It can derive the query from the method name directly, or by using an additionally created query. Available options depend on the actual store. However, there's got to be an strategy that decides what actual query is created. Let's have a look at the available options.

Query lookup strategies

The following strategies are available for the repository infrastructure to resolve the query. You can configure the strategy at the namespace through the `query-lookup-strategy` attribute. Some strategies may not be supported for particular datastores.

CREATE

`CREATE` attempts to construct a store-specific query from the query method 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 the section called “Query creation”.

USE_DECLARED_QUERY

`USE_DECLARED_QUERY` tries to find a declared query and will throw an exception in case it can't find one. The query can be defined by an annotation somewhere or declared by other means. Consult the documentation of the specific store to find available options for that store. If the repository infrastructure does not find a declared query for the method at bootstrap time, it fails.

CREATE_IF_NOT_FOUND (default)

`CREATE_IF_NOT_FOUND` combines `CREATE` and `USE_DECLARED_QUERY`. It looks up a declared query first, and if no declared query is found, it creates a custom method name-based query. This is the default lookup strategy and thus will be used if you do not configure anything explicitly. It allows quick query definition by method names but also custom-tuning of these queries by introducing declared queries as needed.

Query creation

The query builder mechanism built into Spring Data repository infrastructure is useful for building constraining queries over entities of the repository. The mechanism strips the prefixes `find…By`, `read…By`, and `get…By` from the method and starts parsing the rest of it. The introducing clause can contain further expressions such as a `Distinct` to set a distinct flag on the query to be created. However, the first `By` acts as delimiter to indicate the start of the actual criteria. At a very basic level you can define conditions on entity properties and concatenate them with `And` and `Or` .

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

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

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

  // Enabling ignoring case for an individual property
  List<Person> findByLastnameIgnoreCase(String lastname);
  // Enabling ignoring case for all suitable properties
  List<Person> findByLastnameAndFirstnameAllIgnoreCase(String lastname, String firstname);

  // Enabling static ORDER BY for a query
  List<Person> findByLastnameOrderByFirstnameAsc(String lastname);
  List<Person> findByLastnameOrderByFirstnameDesc(String lastname);
}
```

Example 1.4 Query creation from method names

The actual result of parsing the method depends on the persistence store for which you create the query. However, there are some general things to notice.

- The expressions are usually property traversals combined with operators that can be concatenated. You can combine property expressions with `AND` and `OR`. You also get support for operators such as `Between`, `LessThan`, `GreaterThan`, `Like` for the property expressions. The supported operators can vary by datastore, so consult the appropriate part of your reference documentation.
- The method parser supports setting an `IgnoreCase` flag for individual properties, for example,`findByLastnameIgnoreCase(…)`) or for all properties of a type that support ignoring case (usually `Strings`, for example, `findByLastnameAndFirstnameAllIgnoreCase(…)`). Whether ignoring cases is supported may vary by store, so consult the relevant sections in the reference documentation for the store-specific query method.
- You can apply static ordering by appending an `OrderBy` clause to the query method that references a property and by providing a sorting direction (`Asc` or `Desc`). To create a query method that supports dynamic sorting, see the section called “Special parameter handling”.

Property expressions

Property expressions can refer only to a direct property of the managed entity, as shown in the preceding example. At query creation time you already make sure that the parsed property is 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);
```

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

Although this should work for most cases, it is possible for the algorithm to select the wrong property. Suppose the `Person` class has an `addressZip` property as well. The 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);
```

Special parameter handling

To handle parameters to your query you simply define method parameters as already seen in the examples above. Besides that the infrastructure will recognize certain specific types like `Pageable` and `Sort` to apply pagination and sorting to your queries dynamically.

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

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

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

Example 1.5 Using Pageable and Sort in query methods

The first method allows you to pass an `org.springframework.data.domain.Pageable` instance to the query method to dynamically add paging to your statically defined query. Sorting options are handled through the `Pageable` instance too. If you only need sorting, simply add an `org.springframework.data.domain.Sort` parameter to your method. As you also can see, simply returning a `List` is possible as well. In this case the additional metadata required to build the actual `Page` instance will not be created (which in turn means that the additional count query that would have been necessary not being issued) but rather simply restricts the query to look up only the given range of entities.

Note

To find out how many pages you get for a query entirely you have to trigger an additional count query. By default this query will be derived from the query you actually trigger.

Creating repository instances

In this section you create instances and bean definitions for the repository interfaces defined. 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.

XML configuration

Each Spring Data module includes a repositories element that allows you to simply define a base package that Spring scans 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 the preceding example, Spring is instructed to scan com.acme.repositories and all its subpackages for interfaces extending `Repository` or one of its subinterfaces. For each interface found, the infrastructure registers the persistence technology-specific `FactoryBean` to create the appropriate proxies that handle invocations of the query methods. Each bean is 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 wildcards, so that you can have a pattern of scanned packages.

Using filters

By default the infrastructure picks up every interface extending the persistence technology-specific `Repository` subinterface located under the configured base package and creates a bean instance for it. However, you might want more fine-grained control over which interfaces bean instances get created for. To do this you use `<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.

For example, to exclude certain interfaces from instantiation as repository, you could use the following configuration:

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

This example excludes all interfaces ending in `SomeRepository` from being instantiated.
Example 1.6 Using exclude-filter element

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, see the reference documentation.[2]

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

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

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

Example 1.7 Sample annotation based repository configuration

Note

The sample uses the JPA-specific annotation, which you would change according to the store module you actually use. The same applies to the definition of the `EntityManagerFactory` bean. Consult the sections covering the store-specific configuration.

Standalone usage

You can also use the repository infrastructure outside of a Spring container. You still need some Spring libraries in your classpath, but generally you can set up repositories programmatically as well. The Spring Data modules that provide repository support ship a persistence technology-specific `RepositoryFactory` that you can use as follows.

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

Example 1.8 Standalone usage of repository factory

1.3 Custom implementations for Spring Data 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.

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

Adding custom behavior to single repositories

To enrich a repository with custom functionality you first define an interface and an implementation for the custom functionality. Use the repository interface you provided to extend the custom interface.

```
interface UserRepositoryCustom {

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

Example 1.9 Interface for custom repository functionality

```
class UserRepositoryImpl implements UserRepositoryCustom {

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

Note

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

Example 1.10 Implementation of custom repository functionality

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

  // Declare query methods here
}
```

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

Example 1.11 Changes to the your basic repository interface

Configuration

If you use namespace configuration, the repository infrastructure tries to autodetect custom implementations by scanning for classes below the package we found a repository in. These classes need to follow the naming convention of appending the namespace element's attribute `repository-impl-postfix` to the found repository interface name. This postfix defaults to `Impl`.

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

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

Example 1.12 Configuration example

The first configuration example will try to look up 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 preceding approach works well if your custom implementation uses annotation-based configuration and autowiring only, as it will be treated as any other Spring bean. If your custom implementation bean

needs special wiring, you simply declare the bean and name it after the conventions just described. The infrastructure will then refer to the manually defined bean definition by name instead of creating one itself.

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

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

Example 1.13 Manual wiring of custom implementations (I)

Adding custom behavior to all repositories

The preceding approach is not feasible when you want to add a single method to all your repository interfaces.

1. To add custom behavior to all repositories, you first add an intermediate interface to declare the shared behavior.

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

     void sharedCustomMethod(ID id);
   }
   ```

   Example 1.14 An interface declaring custom shared behavior

   Now your individual repository interfaces will extend this intermediate interface instead of the `Repository` interface to include the functionality declared.

2. Next, create an implementation of the intermediate interface that extends the persistence technology-specific repository base class. This class will then act as a custom base class for the repository proxies.

   ```
   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
     }
   }
   ```

   Example 1.15 Custom repository base class

   The default behavior 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 its 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 that extends `Repository` from being instantiated as a repository instance, you can either annotate it with `@NoRepositoryBean` or move it outside of the configured `base-package`.

3. Then 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.

```
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;
    }
  }
}
```

Example 1.16 Custom repository factory bean

4. Finally, 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.

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

Example 1.17 Using the custom factory with the namespace

1.4 Spring Data extensions

This section 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.

Domain class web binding for Spring MVC

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

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

  private final UserRepository userRepository;

  @Autowired
  public UserController(UserRepository userRepository) {
    Assert.notNull(repository, "Repository must not be null!");
    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

    model.addAttribute("user", user);
    return "user";
  }
}
```

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

PropertyEditors

For Spring versions before 3.0 simple Java `PropertyEditors` had to be used. To integrate with that, Spring Data offers a `DomainClassPropertyEditorRegistrar`, which looks up all Spring Data repositories registered in the `ApplicationContext` and registers 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 as in the preceding example, you can configure your controller as follows, which 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) {

    model.addAttribute("user", user);
    return "userForm";
  }
}
```

ConversionService

In Spring 3.0 and later the `PropertyEditor` support is superseded by a new conversion infrastructure that eliminates the drawbacks of `PropertyEditors` and uses a stateless `X` to `Y` conversion approach. Spring Data now ships with a `DomainClassConverter` that mimics the behavior of `DomainClassPropertyEditorRegistrar`. To configure, simply declare a bean instance and pipe the `ConversionService` being used into its constructor:

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

<bean class="org.springframework.data.repository.support.DomainClassConverter">
  <constructor-arg ref="conversionService" />
</bean>
```

If you are using JavaConfig, you can simply extend Spring MVC's `WebMvcConfigurationSupport` and hand the `FormatingConversionService` that the configuration superclass provides into the `DomainClassConverter` instance you create.

```
class WebConfiguration extends WebMvcConfigurationSupport {

  // Other configuration omitted

  @Bean
  public DomainClassConverter<?> domainClassConverter() {
    return new DomainClassConverter<FormattingConversionService>(mvcConversionService());
  }
}
```

Web pagination

When working with pagination in the web layer you usually have to write a lot of boilerplate code yourself to extract the necessary metadata from the request. The less desirable approach shown in the example below requires the method to contain an `HttpServletRequest` parameter that has to be parsed manually. This example also omits appropriate failure handling, which would make the code even more verbose.

```
@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"));

    Pageable pageable = new PageRequest(page, pageSize);

    model.addAttribute("users", userService.getUsers(pageable));
    return "users";
  }
}
```

The bottom line is that the controller should not have to handle the functionality of extracting pagination information from the request. So Spring includes 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", userRepository.findAll(pageable));
    return "users";
  }
}
```

The `PageableArgumentResolver` automatically resolves request parameters to build a `PageRequest` instance. By default it expects the following structure for the request parameters.

Table 1.1. Request parameters evaluated by `PageableArgumentResolver`

`page`	Page you want to retrieve.
`page.size`	Size of the page you want to retrieve.
`page.sort`	Property that should be sorted by.
`page.sort.dir`	Direction that should be used for sorting.

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

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

you have to populate `foo_page` and `bar_page` and the related subproperties.

Configuring a global default on bean declaration

The `PageableArgumentResolver` will use a `PageRequest` with the first page and a page size of 10 by default. It will use that value if it cannot resolve a `PageRequest` from the request (because of missing parameters, for example). You can configure a global default on the bean declaration directly. If you might need controller method specific defaults for the `Pageable`, annotate the method parameter with `@PageableDefaults` and specify page (through `pageNumber`), page size (through `value`), `sort` (list of properties to sort by), and `sortDir` (the direction to sort by) as annotation attributes:

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

Repository populators

If you work with the Spring JDBC module, you probably are familiar with the support to populate a `DataSource` using SQL scripts. A similar abstraction is available on the repositories level, although it does not use SQL as the data definition language because it must be store-independent. Thus the populators support XML (through Spring's OXM abstraction) and JSON (through Jackson) to define data with which to populate the repositories.

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

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

Example 1.18 Data defined in JSON

You can easily populate your repositories by using the populator elements of the repository namespace provided in Spring Data Commons. To populate the preceding data to your `PersonRepository` , do the following:

```
<?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>
```

Example 1.19 Declaring a Jackson repository populator

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

To rather use XML to define the data the repositories shall be populated with, you can use the `unmarshaller-populator` element. You configure it to use one of the XML marshaller options Spring OXM provides you with. See the Spring reference documentation for details.

```
<?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>
```

Example 1.20 Declaring an unmarshalling repository populator (using JAXB)

2. JPA Repositories

This chapter includes details of the JPA repository implementation.

2.1 Introduction

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:

```
<?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>
```

Example 2.1 Setting up JPA repositories using the namespace

Using this element looks up Spring Data repositories as described in the section called “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`.

Annotation based configuration

The Spring Data JPA repositories support cannot only be activated through an XML namespace but also using an annotation through 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;
  }
}
```

Example 2.2 Spring Data JPA repositories using JavaConfig

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

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 the section called “Using JPA NamedQueries” for more information) or rather annotate your query method with `@Query` (see the section called “Using @Query” for details).

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:

```
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.

Example 2.3 Query creation from method names

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`

Keyword	Sample	JPQL snippet
`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*.

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.

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

Example 2.4 XML named query configuration

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.

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

}
```

Example 2.5 Annotation based named query configuration

Declaring interfaces

To allow execution of these named queries all you need to do is to specify the `UserRepository` as follows:

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

  List<User> findByLastname(String lastname);

  User findByEmailAddress(String emailAddress);
}
```

Example 2.6 Query method declaration in UserRepository

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.

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`.

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

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

Example 2.7 Declare query at the query method using `@Query`

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.

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

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

Example 2.8 Advanced `LIKE` expressions in `@Query`

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.

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

  @Query(value = "SELECT FROM USERS WHERE EMAIL_ADDRESS = ?0", nativeQuery = true)
  User findByEmailAddress(String emailAddress);
}
```

Example 2.9 Declare a native query at the query method using `@Query`

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:

```
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.
Example 2.10 Using named parameters

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`:

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

Example 2.11 Declaring manipulating queries

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`;

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.

```
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.

Example 2.12 Using QueryHints with a repository method

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 `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:

```
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
      }
    };
  }
}
```

Example 2.13 Specifications for a Customer

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:

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

Example 2.14 Using a simple Specification

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:

```
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.

Example 2.15 Combined Specifications

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:

```
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.

Example 2.16 Custom transaction configuration for CRUD

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:

```
@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.

Example 2.17 Using a facade to define transactions for multiple repository calls

Transactional query methods

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

```
@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.

Example 2.18 Using @Transactional at query methods

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:

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

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

Example 2.19 Defining lock metadata on query methods

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:

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

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

Example 2.20 Defining lock metadata on CRUD methods

2.6 Auditing

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.

```
class Customer {

  @CreatedBy
  private User user;

  @CreatedDate
  private DateTime createdDate;

  // … further properties omitted
}
```

Example 2.21 An audited entity

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:

```
class SpringSecurityAuditorAware implements AuditorAware<User> {

  public User getCurrentAuditor() {

    Authentication authentication =
 SecurityContextHolder.getContext().getAuthentication();

    if (authentication == null || !authentication.isAuthenticated()) {
      return null;
    }

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

Example 2.22 Implementation of AuditorAware based on Spring Security

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.

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:

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

Example 2.23 Auditing configuration orm.xml

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

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

Example 2.24 Activating auditing in the Spring configuration

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

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

    T getCurrentAuditor();
}
```

Example 2.25 `AuditorAware` interface

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

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.

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

Example 2.26 Using MergingPersistenceUnitmanager

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

```
<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>
```

Example 2.27 Using ClasspathScanningPersistenceUnitPostProcessor

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.

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 the section called “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 /> element 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 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 the section called “Query lookup strategies” for details. Defaults to create-if-not-found.

[1] see the section called “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, because some listed here might not be supported in a particular store.

Table B.1. Query keywords

Logical keyword	Keyword expressions
`AND`	`And`
`OR`	`Or`
`AFTER`	`After`, `IsAfter`
`BEFORE`	`Before`, `IsBefore`
`CONTAINING`	`Containing`, `IsContaining`, `Contains`
`BETWEEN`	`Between`, `IsBetween`
`ENDING_WITH`	`EndingWith`, `IsEndingWith`, `EndsWith`
`EXISTS`	`Exists`
`FALSE`	`False`, `IsFalse`
`GREATER_THAN`	`GreaterThan`, `IsGreaterThan`
`GREATER_THAN_EQUAL`	`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`

Logical keyword	Keyword expressions
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.springframework.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:

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

Example C.1 Looking up a SessionFactory from an HibernateEntityManagerFactory

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