Spring Data Commons - Reference Documentation

Oliver Gierke, Thomas Darimont, Christoph Strobl, Mark Pollack, Thomas Risberg

Version 1.10.1.RELEASE

2015-06-30

Table of Contents

Preface

The Spring Data Commons project applies core Spring concepts to the development of solutions using many relational and non-relational data stores.

Chapter 1. Project metadata

- Version control - http://github.com/spring-projects/spring-data-commons

- Bugtracker - https://jira.spring.io/browse/DATACMNS

- Release repository - https://repo.spring.io/libs-release

- Milestone repository - https://repo.spring.io/libs-milestone

- Snapshot repository - https://repo.spring.io/libs-snapshot

Reference documentation

Chapter 2. Dependencies

Due to different inception dates of individual Spring Data modules, most of them carry different major and minor version numbers. The easiest way to find compatible ones is by relying on the Spring Data Release Train BOM we ship with the compatible versions defined. In a Maven project you'd declare this dependency in the `<dependencyManagement />` section of your POM:

Example 1. Using the Spring Data release train BOM

```xml
<dependencyManagement>
  <dependencies>
    <dependency>
      <groupId>org.springframework.data</groupId>
      <artifactId>spring-data-releasetrain</artifactId>
      <version>${release-train}</version>
      <scope>import</scope>
      <type>pom</type>
    </dependency>
  <dependencies>
</dependencyManagement>
```

The current release train version is Fowler-BUILD-SNAPSHOT. The train names are ascending alphabetically and currently available ones are listed here. The version name follows the following pattern: `${name}-${release}` where release can be one of the following:

- `BUILD-SNAPSHOT` - current snapshots

- `M1`, `M2` etc. - milestones

- `RC1`, `RC2` etc. - release candidates

- `RELEASE` - GA release

- `SR1`, `SR2` etc. - service releases

A working example of using the BOMs can be found in our Spring Data examples repository.

If that's in place declare the Spring Data modules you'd like to use without a version in the `<dependencies />` block.

Example 2. Declaring a dependency to a Spring Data module

```xml
<dependencies>
  <dependency>
    <groupId>org.springframework.data</groupId>
    <artifactId>spring-data-jpa</artifactId>
  </dependency>
<dependencies>
```

2.1. Dependency management with Spring Boot

Spring Boot already selects a very recent version of Spring Data modules for you. In case you want to upgrade to a newer version nonetheless, simply configure the property `spring-data-releasetrain.version` to the train name and iteration you'd like to use.

2.2. Spring Framework

The current version of Spring Data modules require Spring Framework in version 4.0.9.RELEASE or better. The modules might also work with an older bugfix version of that minor version. However, using the most recent version within that generation is highly recommended.

Chapter 3. 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. Namespace reference covers XML configuration which is supported across all Spring Data modules supporting the repository API, Repository query keywords covers the query 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.

3.1. Core concepts

The central interface in Spring Data repository abstraction is `Repository` (probably not that much of a surprise). It takes 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.

Example 3. CrudRepository interface

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

    <S extends T> S save(S entity); <1>

    T findOne(ID primaryKey);      <2>

    Iterable<T> findAll();         <3>

    Long count();                  <4>

    void delete(T entity);         <5>

    boolean exists(ID primaryKey); <6>

    //    more functionality omitted.
}
```

① Saves the given entity.

② Returns the entity identified by the given id.

③ Returns all entities.

④ Returns the number of entities.

⑤ Deletes the given entity.

⑥ Indicates whether an entity with the given id exists.

NOTE We also provide persistence technology-specific abstractions like e.g. `JpaRepository` or `MongoRepository`. Those interfaces extend `CrudRepository` and expose the capabilities of the underlying persistence technology in addition to the rather generic persistence technology-agnostic interfaces like e.g. CrudRepository.

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

Example 4. PagingAndSortingRepository

```java
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:

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

In addition to query methods, query derivation for both count and delete queries, is available.

Example 5. Derived Count Query

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

  Long countByLastname(String lastname);
}
```

Example 6. Derived Delete Query

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

  Long deleteByLastname(String lastname);

  List<User> removeByLastname(String lastname);

}
```

3.2. 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 and ID type that it will handle.

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

2. Declare query methods on the interface.

```
interface PersonRepository extends Repository<User, Long> {
  List<Person> findByLastname(String lastname);
}
```

3. Set up Spring to create proxy instances for those interfaces. Either via JavaConfig:

```
import org.springframework.data.jpa.repository.config.EnableJpaRepositories;

@EnableJpaRepositories
class Config {}
```

or via XML configuration:

```
<?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>
```

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.

Also, note that the JavaConfig variant doesn't configure a package explictly as the package of the annotated class is used by default. To customize the package to scan use one of the basePackage attribute of the data-store specific repository @Enable   -annotation.

1. 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 in detail.

3.3. 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.

3.3.1. 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.

| NOTE | This allows you to define your own abstractions on top of the provided Spring Data Repositories functionality. |

Example 7. Selectively exposing CRUD methods

```
@NoRepositoryBean
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 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 ,e.g. in the case if JPA `SimpleJpaRepository`, 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.

NOTE	Note, that the intermediate repository interface is annotated with `@NoRepositoryBean`. Make sure you add that annotation to all repository interfaces that Spring Data should not create instances for at runtime.

3.4. 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 a manually defined query. Available options depend on the actual store. However, there's got to be a strategy that decides what actual query is created. Let's have a look at the available options.

3.4.1. 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 in case of XML configuration or via the `queryLookupStrategy` attribute of the Enable${store}Repositories annotation in case of Java config. Some strategies may not be supported for particular datastores.

- `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 Query creation.

- `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) 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.

3.4.2. 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`, `query By`, `count 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`.

Example 8. Query creation from method names

```java
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);
}
```

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 `String` instances, 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 Special parameter handling.

3.4.3. 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 a `Person` has an `Address` with a `ZipCode`. 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);
```

As we treat underscore as a reserved character we stongly advise to follow standard Java naming conventions (i.e. **not** using underscores in property names but camel case instead).

3.4.4. Special parameter handling

To handle parameters in 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.

Example 9. Using Pageable, Slice and Sort in query methods

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

Slice<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 an `org.springframework.data.domain.Pageable` instance to the query method to dynamically add paging to your statically defined query. A `Page` knows about the total number of elements and pages available. It does so by the infrastructure triggering a count query to calculate the overall number. As this might be expensive depending on the store used, `Slice` can be used as return instead. A `Slice` only knows about whether there's a next `Slice` available which might be just sufficient when walking thought a larger result set.

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. |

3.4.5. Limiting query results

The results of query methods can be limited via the keywords `first` or `top`, which can be used interchangeably. An optional numeric value can be appended to top/first to specify the maximum result size to be returned. If the number is left out, a result size of 1 is assumed.

Example 10. Limiting the result size of a query with `Top` *and* `First`

```
User findFirstByOrderByLastnameAsc();

User findTopByOrderByAgeDesc();

Page<User> queryFirst10ByLastname(String lastname, Pageable pageable);

Slice<User> findTop3ByLastname(String lastname, Pageable pageable);

List<User> findFirst10ByLastname(String lastname, Sort sort);

List<User> findTop10ByLastname(String lastname, Pageable pageable);
```

The limiting expressions also support the `Distinct` keyword. Also, for the queries limiting the result set to one instance, wrapping the result into an `Optional` is supported.

If pagination or slicing is applied to a limiting query pagination (and the calculation of the number of pages available) then it is applied within the limited result.

NOTE	Note that limiting the results in combination with dynamic sorting via a `Sort` parameter allows to express query methods for the 'K' smallest as well as for the 'K' biggest elements.

3.4.6. Streaming query results

The results of query methods can be processed incrementally by using a Java 8 `Stream<T>` as return type. Instead of simply wrapping the query results in a `Stream` data store specific methods are used to perform the streaming.

Example 11. Stream the result of a query with Java 8 `Stream<T>`

```
@Query("select u from User u")
Stream<User> findAllByCustomQueryAndStream();

Stream<User> readAllByFirstnameNotNull();

@Query("select u from User u")
Stream<User> streamAllPaged(Pageable pageable);
```

NOTE	A `Stream` potentially wraps underlying data store specific resources and must therefore be closed after usage. You can either manually close the `Stream` using the `close()` method or by using a Java 7 try-with-resources block.

Example 12. Working with a Stream<T> result in a try-with-resources block

```java
try (Stream<User> stream = repository.findAllByCustomQueryAndStream()) {
  stream.forEach(  );
}
```

NOTE   Not all Spring Data modules currently support Stream<T> as a return type.

3.5. Creating repository instances

In this section you create instances and bean definitions for the repository interfaces defined. One way to do so is using the Spring namespace that is shipped with each Spring Data module that supports the repository mechanism although we generally recommend to use the Java-Config style configuration.

3.5.1. XML configuration

Each Spring Data module includes a repositories element that allows you to simply define a base package that Spring scans for you.

Example 13. Enabling Spring Data repositories via XML

```xml
<?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 sub-packages for interfaces extending Repository or one of its sub-interfaces. 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 define a pattern of scanned packages.

Using filters

By default the infrastructure picks up every interface extending the persistence technology-specific `Repository` sub-interface 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:

Example 14. Using exclude-filter element

```xml
<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.

3.5.2. 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, see the reference documentation. [JavaConfig in the Spring reference documentation]

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

Example 15. Sample annotation based repository configuration

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

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

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.

3.5.3. Standalone usage

You can also use the repository infrastructure outside of a Spring container, e.g. in CDI environments. 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.

Example 16. Standalone usage of repository factory

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

3.6. 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.

3.6.1. 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.

Example 17. Interface for custom repository functionality

```
interface UserRepositoryCustom {
  public void someCustomMethod(User user);
}
```

Example 18. Implementation of custom repository functionality

```
class UserRepositoryImpl implements UserRepositoryCustom {

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

NOTE The most important bit for the class to be found is the `Impl` postfix of the name on it compared to the core repository interface (see below).

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 like a JdbTemplate, take part in aspects, and so on.

Example 19. Changes to the your basic repository interface

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

  // Declare query methods here
}
```

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

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`.

Example 20. 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 look up a class `com.acme.repository.UserRepositoryImpl` to act as custom repository implementation, whereas the second example will try to lookup

`com.acme.repository.UserRepositoryFooBar`.

Manual wiring

The approach just shown 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.

Example 21. Manual wiring of custom implementations

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

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

3.6.2. 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.

   Example 22. An interface declaring custom shared behavior

   ```
   @NoRepositoryBean
   public interface MyRepository<T, ID extends Serializable>
     extends PagingAndSortingRepository<T, ID> {

     void sharedCustomMethod(ID id);
   }
   ```

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

3. 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.

Example 23. Custom repository base class

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

  private final EntityManager entityManager;

  public MyRepositoryImpl(Class<T> domainClass, EntityManager entityManager) {
    super(domainClass, entityManager);

    // Keep the EntityManager around to used from the newly introduced methods.
    this.entityManager = entityManager;
  }

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

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` (as seen above) or move it outside of the configured `base-package`.

4. 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.

Example 24. Custom repository factory bean

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

  protected RepositoryFactorySupport createRepositoryFactory(EntityManager em) {
    return new MyRepositoryFactory(em);
  }

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

    private final EntityManager em;

    public MyRepositoryFactory(EntityManager em) {

      super(em);
      this.em = em;
    }

    protected Object getTargetRepository(RepositoryMetadata metadata) {
      return new MyRepositoryImpl<T, I>((Class<T>) metadata.getDomainClass(), em);
    }

    protected Class<?> getRepositoryBaseClass(RepositoryMetadata metadata) {
      return MyRepositoryImpl.class;
    }
  }
}
```

5. Finally, either declare beans of the custom factory directly or use the `factory-class` attribute of the Spring namespace or `@Enable` annotation to instruct the repository infrastructure to use your custom factory implementation.

Example 25. Using the custom factory with the namespace

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

Example 26. Using the custom factory with the `@Enable`   annotation

```
@EnableJpaRepositories(factoryClass = "com.acme.MyRepositoryFactoryBean")
class Config {}
```

3.7. 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.

3.7.1. Web support

> **NOTE** This section contains the documentation for the Spring Data web support as it is implemented as of Spring Data Commons in the 1.6 range. As it the newly introduced support changes quite a lot of things we kept the documentation of the former behavior in Legacy web support.

Spring Data modules ships with a variety of web support if the module supports the repository programming model. The web related stuff requires Spring MVC JARs on the classpath, some of them even provide integration with Spring HATEOAS [Spring HATEOAS - https://github.com/SpringSource/spring-hateoas]. In general, the integration support is enabled by using the `@EnableSpringDataWebSupport` annotation in your JavaConfig configuration class.

Example 27. Enabling Spring Data web support

```
@Configuration
@EnableWebMvc
@EnableSpringDataWebSupport
class WebConfiguration { }
```

The `@EnableSpringDataWebSupport` annotation registers a few components we will discuss in a bit. It will also detect Spring HATEOAS on the classpath and register integration components for it as well if present.

Alternatively, if you are using XML configuration, register either `SpringDataWebSupport` or `HateoasAwareSpringDataWebSupport` as Spring beans:

Example 28. Enabling Spring Data web support in XML

```xml
<bean class="org.springframework.data.web.config.SpringDataWebConfiguration" />

<!-- If you're using Spring HATEOAS as well register this one *instead* of the former
-->
<bean class=
"org.springframework.data.web.config.HateoasAwareSpringDataWebConfiguration" />
```

Basic web support

The configuration setup shown above will register a few basic components:

- A `DomainClassConverter` to enable Spring MVC to resolve instances of repository managed domain classes from request parameters or path variables.

- `HandlerMethodArgumentResolver` implementations to let Spring MVC resolve Pageable and Sort instances from request parameters.

DomainClassConverter

The `DomainClassConverter` allows you to use domain types in your Spring MVC controller method signatures directly, so that you don't have to manually lookup the instances via the repository:

Example 29. A Spring MVC controller using domain types in method signatures

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

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

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

As you can see the method receives a User instance directly and no further lookup is necessary. The instance can be resolved by letting Spring MVC convert the path variable into the id type of the domain class first and eventually access the instance through calling `findOne(  )` on the repository instance registered for the domain type.

NOTE	Currently the repository has to implement `CrudRepository` to be eligible to be discovered for conversion.

HandlerMethodArgumentResolvers for Pageable and Sort

The configuration snippet above also registers a `PageableHandlerMethodArgumentResolver` as well as an instance of `SortHandlerMethodArgumentResolver`. The registration enables `Pageable` and `Sort` being valid controller method arguments

Example 30. Using Pageable as controller method argument

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

  @Autowired UserRepository repository;

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

    model.addAttribute("users", repository.findAll(pageable));
    return "users";
  }
}
```

This method signature will cause Spring MVC try to derive a Pageable instance from the request parameters using the following default configuration:

Table 1. Request parameters evaluated for Pageable instances

`page`	Page you want to retrieve, 0 indexed and defaults to 0.
`size`	Size of the page you want to retrieve, defaults to 20.
`sort`	Properties that should be sorted by in the format `property,property(,ASC|DESC)`. Default sort direction is ascending. Use multiple `sort` parameters if you want to switch directions, e.g. `?sort=firstname&sort=lastname,asc`.

To customize this behavior extend either `SpringDataWebConfiguration` or the HATEOAS-enabled equivalent and override the `pageableResolver()` or `sortResolver()` methods and import your customized configuration file instead of using the `@Enable`-annotation.

In case you need multiple `Pageable` or `Sort` instances 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` etc.

The default `Pageable` handed into the method is equivalent to a `new PageRequest(0, 20)` but can be customized using the `@PageableDefaults` annotation on the `Pageable` parameter.

Hypermedia support for Pageables

Spring HATEOAS ships with a representation model class `PagedResources` that allows enriching the content of a `Page` instance with the necessary `Page` metadata as well as links to let the clients easily navigate the pages. The conversion of a Page to a `PagedResources` is done by an implementation of the Spring HATEOAS `ResourceAssembler` interface, the `PagedResourcesAssembler`.

Example 31. Using a PagedResourcesAssembler as controller method argument

```
@Controller
class PersonController {

  @Autowired PersonRepository repository;

  @RequestMapping(value = "/persons", method = RequestMethod.GET)
  HttpEntity<PagedResources<Person>> persons(Pageable pageable,
    PagedResourcesAssembler assembler) {

    Page<Person> persons = repository.findAll(pageable);
    return new ResponseEntity<>(assembler.toResources(persons), HttpStatus.OK);
  }
}
```

Enabling the configuration as shown above allows the `PagedResourcesAssembler` to be used as controller method argument. Calling `toResources(   )` on it will cause the following:

- The content of the `Page` will become the content of the `PagedResources` instance.

- The `PagedResources` will get a `PageMetadata` instance attached populated with information form the `Page` and the underlying `PageRequest`.

- The `PagedResources` gets `prev` and `next` links attached depending on the page's state. The links will

point to the URI the method invoked is mapped to. The pagination parameters added to the method will match the setup of the `PageableHandlerMethodArgumentResolver` to make sure the links can be resolved later on.

Assume we have 30 Person instances in the database. You can now trigger a request `GET` `http://localhost:8080/persons` and you'll see something similar to this:

```
{ "links" : [ { "rel" : "next",
                "href" : "http://localhost:8080/persons?page=1&size=20 }
  ],
  "content" : [
       // 20 Person instances rendered here
  ],
  "pageMetadata" : {
    "size" : 20,
    "totalElements" : 30,
    "totalPages" : 2,
    "number" : 0
  }
}
```

You see that the assembler produced the correct URI and also picks up the default configuration present to resolve the parameters into a `Pageable` for an upcoming request. This means, if you change that configuration, the links will automatically adhere to the change. By default the assembler points to the controller method it was invoked in but that can be customized by handing in a custom `Link` to be used as base to build the pagination links to overloads of the `PagedResourcesAssembler.toResource( )` method.

3.7.2. 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:

Example 32. Data defined in JSON

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

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:

Example 33. 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 locations="classpath:data.json" />

</beans>
```

This declaration causes the `data.json` file to be read and deserialized via 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.

Example 34. Declaring an unmarshalling repository populator (using JAXB)

```xml
<?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 locations="classpath:data.json"
    unmarshaller-ref="unmarshaller" />

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

</beans>
```

3.7.3. Legacy web support

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!");
    this.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.

```xml
<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.

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

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

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

Chapter 4. Auditing

4.1. Basics

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

4.1.1. Annotation based auditing metadata

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

Example 35. An audited entity

```java
class Customer {

  @CreatedBy
  private User user;

  @CreatedDate
  private DateTime createdDate;

  //    further properties omitted
}
```

As you can see, the annotations can be applied selectively, depending on which information you'd like to capture. For the annotations capturing the points in time can be used on properties of type JodaTimes `DateTime`, legacy Java `Date` and `Calendar`, JDK8 date/time types as well as `long`/`Long`.

4.1.2. 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.

4.1.3. AuditorAware

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

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

Example 36. Implementation of AuditorAware based on Spring Security

```
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();
  }
}
```

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.

Appendix

Appendix A: Namespace reference

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. [see XML configuration]

Table 2. 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 Query lookup strategies for details. Defaults to `create-if-not-found`.
`named-queries-location`	Defines the location to look for a Properties file containing externally defined queries.
`consider-nested-repositories`	Controls whether nested repository interface definitions should be considered. Defaults to `false`.

Appendix B: Populators namespace reference

The <populator /> element

The `<populator />` element allows to populate the a data store via the Spring Data repository infrastructure. [see XML configuration]

Table 3. Attributes

Name	Description
`locations`	Where to find the files to read the objects from the repository shall be populated with.

Appendix C: Repository query keywords

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 4. 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_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

Logical keyword	Keyword expressions
REGEX	Regex, MatchesRegex, Matches
STARTING_WITH	StartingWith, IsStartingWith, StartsWith
TRUE	True, IsTrue
WITHIN	Within, IsWithin

Logical keyword	Keyword expressions
REGEX	Regex, MatchesRegex, Matches
STARTING_WITH	StartingWith, IsStartingWith, StartsWith

Appendix D: Repository query return types

Supported query return types

The following table lists the return types generally supported by Spring Data repositories. However, consult the store-specific documentation for the exact list of supported return types, because some listed here might not be supported in a particular store.

NOTE	Geospatial types like (`GeoResult`, `GeoResults`, `GeoPage`) are only available for data stores that support geospatial queries.

Table 5. Query return types

Return type	Description
`void`	Denotes no return value.
Primitives	Java primitives.
Wrapper types	Java wrapper types.
`T`	An unique entity. Expects the query method to return one result at most. In case no result is found `null` is returned. More than one result will trigger an `IncorrectResultSizeDataAccessException`.
`Iterator<T>`	An `Iterator`.
`Collection<T>`	A `Collection`.
`List<T>`	A `List`.
`Optional<T>`	A Java 8 or Guava `Optional`. Expects the query method to return one result at most. In case no result is found `Optional.empty()`/`Optional.absent()` is returned. More than one result will trigger an `IncorrectResultSizeDataAccessException`.
`Stream<T>`	A Java 8 `Stream`.
`Slice`	A sized chunk of data with information whether there is more data available. Requires a `Pageable` method parameter.
`Page<T>`	A `Slice` with additional information, e.g. the total number of results. Requires a `Pageable` method parameter.
`GeoResult<T>`	A result entry with additional information, e.g. distance to a reference location.
`GeoResults<T>`	A list of `GeoResult<T>` with additional information, e.g. average distance to a reference location.
`GeoPage<T>`	A `Page` with `GeoResult<T>`, e.g. average distance to a reference location.