Spring Data JDBC - Reference Documentation

Jens Schauder, Jay Bryant, Mark Paluch – Version 1.0.1.RELEASE, 2018-10-15

Table of Contents

Preface

Spring Data JDBC offers a repository abstraction based on JDBC.

Project Metadata

- Version control: http://github.com/spring-projects/spring-data-jdbc
- Bugtracker: https://jira.spring.io/browse/DATAJDBC
- Release repository: https://repo.spring.io/libs-release
- Milestone repository: https://repo.spring.io/libs-milestone
- Snapshot repository: https://repo.spring.io/libs-snapshot

1. New & Noteworthy

This section covers the significant changes for each version.

1.1. What's New in Spring Data JDBC 1.0

- Basic support for `CrudRepository`.
- `@Query` support.
- MyBatis support.
- Id generation.

- Event support.
- Auditing.
- `CustomConversions`.

---

2. Dependencies

Due to the 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 to rely on the Spring Data Release Train BOM that we ship with the compatible versions defined. In a Maven project, you would declare this dependency in the `<dependencyManagement />` section of your POM, as follows:

Example 1. Using the Spring Data release train BOM

```
<dependencyManagement>
  <dependencies>
    <dependency>
      <groupId>org.springframework.data</groupId>
      <artifactId>spring-data-releasetrain</artifactId>
      <version>Lovelace-SR1</version>
      <scope>import</scope>
      <type>pom</type>
    </dependency>
  </dependencies>
</dependencyManagement>
```

The current release train version is `Lovelace-SR1`. The train names ascend alphabetically and the currently available trains 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`, and so on: Milestones
- `RC1`, `RC2`, and so on: Release candidates
- `RELEASE`: GA release
- `SR1`, `SR2`, and so on: Service releases

A working example of using the BOMs can be found in our Spring Data examples repository. With that in place, you can declare the Spring Data modules you would like to use without a version in the `<dependencies />` block, as follows:

Example 2. Declaring a dependency to a Spring Data module

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

2.1. Dependency Management with Spring Boot

Spring Boot selects a recent version of Spring Data modules for you. If you still want to upgrade to a newer version, configure the property `spring-data-releasetrain.version` to the train name and iteration you would like to use.

2.2. Spring Framework

The current version of Spring Data modules require Spring Framework in version 5.1.1.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.

3. Working with Spring Data Repositories

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

> *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. You should adapt the XML namespace declaration and the types to be extended to the equivalents of the particular

module that you use. “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, see the chapter on that module of this document.

3.1. Core concepts

The central interface in the Spring Data repository abstraction is `Repository`. 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*

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

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

  Optional<T> findById(ID primaryKey);   2

  Iterable<T> findAll();                 3

  long count();                          4

  void delete(T entity);                 5

  boolean existsById(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.

> We also provide persistence technology-specific abstractions, such as `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 such as `CrudRepository`.

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

Example 4. `PagingAndSortingRepository` *interface*

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

  Iterable<T> findAll(Sort sort);

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

To access the second page of `User` by a page size of 20, you could do something like the following:

```
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. The following list shows the interface definition for a derived count query:

Example 5. Derived Count Query

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

  long countByLastname(String lastname);
}
```

The following list shows the interface definition for a derived delete query:

Example 6. Derived Delete Query

```
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 should handle, as shown in the following example:

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

2. Declare query methods on the interface.

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

3. Set up Spring to create proxy instances for those interfaces, either with JavaConfig or with XML configuration.

   a. To use Java configuration, create a class similar to the following:

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

@EnableJpaRepositories
class Config {}
```

   b. To use XML configuration, define a bean similar to 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: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 use the repository abstraction for any other store, you need to change this to the appropriate namespace declaration of your store module. In other words, you should exchange `jpa` in favor of, for example, `mongodb`.

+ Also, note that the JavaConfig variant does not configure a package explicitly, because the package of the annotated class is used by default. To customize the package to scan, use one of the `basePackage…` attributes of the data-store-specific repository's `@Enable${store}Repositories`-annotation.

4. Inject the repository instance and use it, as shown in the following example:

```
class SomeClient {

  private final PersonRepository repository;

  SomeClient(PersonRepository repository) {
    this.repository = repository;
  }

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

The sections that follow explain each step in detail:

- Defining Repository Interfaces
- Defining Query Methods
- Creating Repository Instances
- Custom Implementations for Spring Data Repositories

3.3. Defining Repository Interfaces

First, 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 extends `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, copy the methods you want to expose from `CrudRepository` into your domain repository.

> Doing so lets you define your own abstractions on top of the provided Spring Data Repositories functionality.

The following example shows how to selectively expose CRUD methods (`findById` and `save`, in this case):

Example 7. Selectively exposing CRUD methods

```
@NoRepositoryBean
interface MyBaseRepository<T, ID extends Serializable> extends Repository<T, ID> {

  Optional<T> findById(ID id);

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

interface UserRepository extends MyBaseRepository<User, Long> {
  User findByEmailAddress(EmailAddress emailAddress);
}
```

In the prior example, you defined a common base interface for all your domain repositories and exposed `findById(…)` as well as `save(…)`.These methods are routed into the base repository implementation of the store of your choice provided by Spring Data (for example, if you use JPA, the implementation is `SimpleJpaRepository`), because they match the method signatures in `CrudRepository`. So the `UserRepository` can now save users, find individual users by ID, and trigger a query to find `Users` by email address.

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

3.3.2. Null Handling of Repository Methods

As of Spring Data 2.0, repository CRUD methods that return an individual aggregate instance use Java 8's `Optional` to indicate the potential absence of a value. Besides that, Spring Data supports returning the following wrapper types on query methods:

- `com.google.common.base.Optional`
- `scala.Option`
- `io.vavr.control.Option`
- `javaslang.control.Option` (deprecated as Javaslang is deprecated)

Alternatively, query methods can choose not to use a wrapper type at all. The absence of a query result is then indicated by returning `null`. Repository methods returning collections, collection alternatives, wrappers, and streams are guaranteed never to return `null` but rather the corresponding empty representation. See "Repository query return types" for details.

Nullability Annotations

You can express nullability constraints for repository methods by using Spring Framework's nullability annotations. They provide a tooling-friendly approach and opt-in `null` checks during runtime, as follows:

- `@NonNullApi`: Used on the package level to declare that the default behavior for parameters and return values is to not accept or produce `null` values.
- `@NonNull`: Used on a parameter or return value that must not be `null` (not needed on a parameter and return value where `@NonNullApi` applies).
- `@Nullable`: Used on a parameter or return value that can be `null`.

Spring annotations are meta-annotated with JSR 305 annotations (a dormant but widely spread JSR). JSR 305 meta-annotations let tooling vendors such as IDEA, Eclipse, and Kotlin provide null-safety support in a generic way, without having to hard-code support for Spring annotations. To enable runtime checking of nullability constraints for query methods, you need to activate non-nullability on the package level by using Spring's `@NonNullApi` in `package-info.java`, as shown in the following example:

Example 8. Declaring Non-nullability in `package-info.java`

```
@org.springframework.lang.NonNullApi
package com.acme;
```

Once non-null defaulting is in place, repository query method invocations get validated at runtime for nullability constraints. If a query execution result violates the defined constraint, an exception is thrown. This happens when the method would return `null` but is declared as non-nullable (the default with the annotation defined on the package the repository resides in). If you want to opt-in to nullable results again, selectively use `@Nullable` on individual methods. Using the result wrapper types mentioned at the start of this section continues to work as expected: An empty result is translated into the value that represents absence.

The following example shows a number of the techniques just described:

Example 9. Using different nullability constraints

```
package com.acme;                                                       1

import org.springframework.lang.Nullable;

interface UserRepository extends Repository<User, Long> {

  User getByEmailAddress(EmailAddress emailAddress);                    2

  @Nullable
  User findByEmailAddress(@Nullable EmailAddress emailAdress);          3

  Optional<User> findOptionalByEmailAddress(EmailAddress emailAddress); 4
}
```

1 The repository resides in a package (or sub-package) for which we have defined non-null behavior.

2 Throws an `EmptyResultDataAccessException` when the query executed does not produce a result. Throws an `IllegalArgumentException` when the `emailAddress` handed to the method is `null`.

3 Returns `null` when the query executed does not produce a result. Also accepts `null` as the value for `emailAddress`.

4 Returns `Optional.empty()` when the query executed does not produce a result. Throws an `IllegalArgumentException` when the `emailAddress` handed to the

method is `null`.

Nullability in Kotlin-based Repositories

Kotlin has the definition of nullability constraints baked into the language. Kotlin code compiles to bytecode, which does not express nullability constraints through method signatures but rather through compiled-in metadata. Make sure to include the `kotlin-reflect` JAR in your project to enable introspection of Kotlin's nullability constraints. Spring Data repositories use the language mechanism to define those constraints to apply the same runtime checks, as follows:

Example 10. Using nullability constraints on Kotlin repositories

```
interface UserRepository : Repository<User, String> {

  fun findByUsername(username: String): User      1

  fun findByFirstname(firstname: String?): User?  2
}
```

1 The method defines both the parameter and the result as non-nullable (the Kotlin default). The Kotlin compiler rejects method invocations that pass `null` to the method. If the query execution yields an empty result, an `EmptyResultDataAccessException` is thrown.

2 This method accepts `null` for the `firstname` parameter and returns `null` if the query execution does not produce a result.

3.3.3. Using Repositories with Multiple Spring Data Modules

Using a unique Spring Data module in your application makes things simple, because all repository interfaces in the defined scope are bound to the Spring Data module. Sometimes, applications require using more than one Spring Data module. In such cases, a repository definition must distinguish between persistence technologies. When it detects multiple repository factories on the class path, Spring Data enters strict repository configuration mode. Strict configuration uses details on the repository or the domain class to decide about Spring Data module binding for a repository definition:

1. If the repository definition extends the module-specific repository, then it is a valid candidate for the particular Spring Data module.

2. If the domain class is annotated with the module-specific type annotation, then it is a valid candidate for the particular Spring Data module. Spring Data modules accept either third-party annotations (such as JPA's `@Entity`) or provide their own annotations (such as `@Document` for Spring Data MongoDB and Spring Data Elasticsearch).

The following example shows a repository that uses module-specific interfaces (JPA in this case):

Example 11. Repository definitions using module-specific interfaces

```
interface MyRepository extends JpaRepository<User, Long> { }

@NoRepositoryBean
interface MyBaseRepository<T, ID extends Serializable> extends JpaRepository<T, ID> {
  …
}

interface UserRepository extends MyBaseRepository<User, Long> {
  …
}
```

`MyRepository` and `UserRepository` extend `JpaRepository` in their type hierarchy. They are valid candidates for the Spring Data JPA module.

The following example shows a repository that uses generic interfaces:

Example 12. Repository definitions using generic interfaces

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

@NoRepositoryBean
interface MyBaseRepository<T, ID extends Serializable> extends CrudRepository<T, ID> {
  …
}

interface AmbiguousUserRepository extends MyBaseRepository<User, Long> {
  …
}
```

`AmbiguousRepository` and `AmbiguousUserRepository` extend only `Repository` and `CrudRepository` in their type hierarchy. While this is perfectly fine when using a unique

Spring Data module, multiple modules cannot distinguish to which particular Spring Data these repositories should be bound.

The following example shows a repository that uses domain classes with annotations:

Example 13. Repository definitions using domain classes with annotations

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

@Entity
class Person {
  …
}

interface UserRepository extends Repository<User, Long> {
 …
}

@Document
class User {
  …
}
```

`PersonRepository` references `Person`, which is annotated with the JPA `@Entity` annotation, so this repository clearly belongs to Spring Data JPA. `UserRepository` references `User`, which is annotated with Spring Data MongoDB's `@Document` annotation.

The following bad example shows a repository that uses domain classes with mixed annotations:

Example 14. Repository definitions using domain classes with mixed annotations

```
interface JpaPersonRepository extends Repository<Person, Long> {
 …
}

interface MongoDBPersonRepository extends Repository<Person, Long> {
 …
}

@Entity
```

```
@Document
class Person {
  …
}
```

This example shows a domain class using both JPA and Spring Data MongoDB annotations. It defines two repositories, `JpaPersonRepository` and `MongoDBPersonRepository`. One is intended for JPA and the other for MongoDB usage. Spring Data is no longer able to tell the repositories apart, which leads to undefined behavior.

Repository type details and distinguishing domain class annotations are used for strict repository configuration to identify repository candidates for a particular Spring Data module. Using multiple persistence technology-specific annotations on the same domain type is possible and enables reuse of domain types across multiple persistence technologies. However, Spring Data can then no longer determine a unique module with which to bind the repository.

The last way to distinguish repositories is by scoping repository base packages. Base packages define the starting points for scanning for repository interface definitions, which implies having repository definitions located in the appropriate packages. By default, annotation-driven configuration uses the package of the configuration class. The base package in XML-based configuration is mandatory.

The following example shows annotation-driven configuration of base packages:

Example 15. Annotation-driven configuration of base packages

```
@EnableJpaRepositories(basePackages = "com.acme.repositories.jpa")
@EnableMongoRepositories(basePackages = "com.acme.repositories.mongo")
interface Configuration { }
```

3.4. Defining Query Methods

The repository proxy has two ways to derive a store-specific query from the method name:

- By deriving the query from the method name directly.
- By using a manually defined query.

Available options depend on the actual store. However, there must be a strategy that decides what actual query is created. The next section describes the available options.

3.4.1. Query Lookup Strategies

The following strategies are available for the repository infrastructure to resolve the query. With XML configuration, you can configure the strategy at the namespace through the `query-lookup-strategy` attribute. For Java configuration, you can use the `queryLookupStrategy` attribute of the `Enable${store}Repositories` annotation. 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. You can read more about query construction in "Query Creation".
- `USE_DECLARED_QUERY` tries to find a declared query and throws an exception if cannot 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, is 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`. The following example shows how to create a number of queries:

Example 16. Query creation from method names

```
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`, and `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 supports 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. Consider the following method signature:

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

Assume a `Person` has an `Address` with a `ZipCode`. In that case, the method creates the property traversal `x.address.zipCode`. The resolution algorithm starts by 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 continues building the tree down from there, splitting the tail up in the way just described. If the first split does not match, the algorithm moves 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, choose the wrong property, and 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 be as follows:

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

Because we treat the underscore character as a reserved character, we strongly advise following standard Java naming conventions (that is, not using underscores in property names but using camel case instead).

3.4.4. Special parameter handling

To handle parameters in your query, define method parameters as already seen in the preceding examples. Besides that, the infrastructure recognizes certain specific types like `Pageable` and `Sort`, to apply pagination and sorting to your queries dynamically. The following example demonstrates these features:

Example 17. 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 lets you 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), you can instead return a `Slice`. A `Slice` only knows about whether a next `Slice` is available, which might be sufficient when walking through a larger result set.

Sorting options are handled through the `Pageable` instance, too. If you only need sorting, add an `org.springframework.data.domain.Sort` parameter to your method. As you can see, returning a `List` is also possible. In this case, the additional metadata required to build the actual `Page` instance is not created (which, in turn, means that the additional count query that would have been necessary is not issued). Rather, it restricts the query to look up only the given range of entities.

> To find out how many pages you get for an entire query, you have to trigger an additional count query. By default, this query is derived from the query you actually trigger.

3.4.5. Limiting Query Results

The results of query methods can be limited by using the `first` or `top` keywords, which can be used interchangeably. An optional numeric value can be appended to `top` or `first` to specify the maximum result size to be returned. If the number is left out, a result size of 1 is assumed. The following example shows how to limit the query size:

Example 18. 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 with the `Optional` keyword is supported.

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

> Limiting the results in combination with dynamic sorting by using a `Sort` parameter lets you 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 wrapping the query results in a `Stream` data store-specific methods are used to perform the streaming, as shown in the following example:

Example 19. 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);
```

> A `Stream` potentially wraps underlying data store-specific resources and

must, therefore, be closed after usage. You can either manually close the `Stream` by using the `close()` method or by using a Java 7 `try-with-resources` block, as shown in the following example:

Example 20. Working with a `Stream<T>` result in a try-with-resources block

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

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

3.4.7. Async query results

Repository queries can be run asynchronously by using [Spring's asynchronous method execution capability](). This means the method returns immediately upon invocation while the actual query execution occurs in a task that has been submitted to a Spring `TaskExecutor`. Asynchronous query execution is different from reactive query execution and should not be mixed. Refer to store-specific documentation for more details on reactive support. The following example shows a number of asynchronous queries:

```
@Async
Future<User> findByFirstname(String firstname);               1

@Async
CompletableFuture<User> findOneByFirstname(String firstname); 2

@Async
ListenableFuture<User> findOneByLastname(String lastname);    3
```

1. Use `java.util.concurrent.Future` as the return type.
2. Use a Java 8 `java.util.concurrent.CompletableFuture` as the return type.
3. Use a `org.springframework.util.concurrent.ListenableFuture` as the return type.

3.5. Creating Repository Instances

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

3.5.1. XML configuration

Each Spring Data module includes a `repositories` element that lets you define a base package that Spring scans for you, as shown in the following example:

Example 21. Enabling Spring Data repositories via 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 have bean instances created for them. To do so, use `<include-filter />` and `<exclude-filter />` elements inside the `<repositories />` element. The semantics are exactly equivalent to the elements in Spring's context namespace. For details, see the Spring reference documentation for these elements.

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

Example 22. Using exclude-filter element

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

The preceding example excludes all interfaces ending in `SomeRepository` from being instantiated.

3.5.2. JavaConfig

The repository infrastructure can also be triggered by using a store-specific `@Enable${store}Repositories` annotation on a JavaConfig class. For an introduction into Java-based configuration of the Spring container, see JavaConfig in the Spring reference documentation.

A sample configuration to enable Spring Data repositories resembles the following:

Example 23. Sample annotation based repository configuration

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

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

The preceding example 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. See the sections covering the store-specific configuration.

3.5.3. Standalone usage

You can also use the repository infrastructure outside of a Spring container — for example, 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 24. Standalone usage of repository factory

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

3.6. Custom Implementations for Spring Data Repositories

This section covers repository customization and how fragments form a composite repository.

When a query method requires a different behavior or cannot be implemented by query derivation, then it is necessary to provide a custom implementation. Spring Data repositories let you provide custom repository code and integrate it with generic CRUD abstraction and query method functionality.

3.6.1. Customizing Individual Repositories

To enrich a repository with custom functionality, you must first define a fragment interface and an implementation for the custom functionality, as shown in the following example:

Example 25. Interface for custom repository functionality

```
interface CustomizedUserRepository {
  void someCustomMethod(User user);
}
```

Then you can let your repository interface additionally extend from the fragment interface, as shown in the following example:

Example 26. Implementation of custom repository functionality

```
class CustomizedUserRepositoryImpl implements CustomizedUserRepository {

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

> The most important part of the class name that corresponds to the fragment interface is the `Impl` postfix.

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

You can let your repository interface extend the fragment interface, as shown in the following example:

Example 27. Changes to your repository interface

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

  // Declare query methods here
}
```

Extending the fragment interface with your repository interface combines the CRUD and custom functionality and makes it available to clients.

Spring Data repositories are implemented by using fragments that form a repository composition. Fragments are the base repository, functional aspects (such as QueryDsl), and custom interfaces along with their implementation. Each time you add an interface to your repository interface, you enhance the composition by adding a fragment. The base repository and repository aspect implementations are provided by each Spring Data module.

The following example shows custom interfaces and their implementations:

Example 28. Fragments with their implementations

```
interface HumanRepository {
  void someHumanMethod(User user);
}

class HumanRepositoryImpl implements HumanRepository {

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

interface ContactRepository {

  void someContactMethod(User user);

  User anotherContactMethod(User user);
}

class ContactRepositoryImpl implements ContactRepository {

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

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

The following example shows the interface for a custom repository that extends `CrudRepository`:

Example 29. Changes to your repository interface

```
interface UserRepository extends CrudRepository<User, Long>, HumanRepository,
ContactRepository {

  // Declare query methods here
}
```

Repositories may be composed of multiple custom implementations that are imported in the order of their declaration. Custom implementations have a higher priority than the base implementation and repository aspects. This ordering lets you override base repository and aspect methods and resolves ambiguity if two fragments contribute the same method signature. Repository fragments are not limited to use in a single repository interface. Multiple repositories may use a fragment interface, letting you reuse customizations across different repositories.

The following example shows a repository fragment and its implementation:

Example 30. Fragments overriding `save(…)`

```
interface CustomizedSave<T> {
  <S extends T> S save(S entity);
}

class CustomizedSaveImpl<T> implements CustomizedSave<T> {

  public <S extends T> S save(S entity) {
    // Your custom implementation
  }
}
```

The following example shows a repository that uses the preceding repository fragment:

Example 31. Customized repository interfaces

```
interface UserRepository extends CrudRepository<User, Long>, CustomizedSave<User> {
}

interface PersonRepository extends CrudRepository<Person, Long>, CustomizedSave<Person> {
}
```

Configuration

If you use namespace configuration, the repository infrastructure tries to autodetect custom implementation fragments by scanning for classes below the package in which it found a repository. These classes need to follow the naming convention of appending the namespace element's `repository-impl-postfix` attribute to the fragment interface name. This postfix defaults to `Impl`. The following example shows a repository that uses the default postfix and a repository that sets a custom value for the postfix:

Example 32. Configuration example

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

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

The first configuration in the preceding example tries to look up a class called `com.acme.repository.CustomizedUserRepositoryImpl` to act as a custom repository implementation. The second example tries to lookup `com.acme.repository.CustomizedUserRepositoryMyPostfix`.

Resolution of Ambiguity

If multiple implementations with matching class names are found in different packages, Spring Data uses the bean names to identify which one to use.

Given the following two custom implementations for the `CustomizedUserRepository` shown earlier, the first implementation is used. Its bean name is `customizedUserRepositoryImpl`, which matches that of the fragment interface (`CustomizedUserRepository`) plus the postfix `Impl`.

Example 33. Resolution of amibiguous implementations

```
package com.acme.impl.one;

class CustomizedUserRepositoryImpl implements CustomizedUserRepository {

  // Your custom implementation
}
```

```
package com.acme.impl.two;

@Component("specialCustomImpl")
class CustomizedUserRepositoryImpl implements CustomizedUserRepository {

  // Your custom implementation
}
```

If you annotate the `UserRepository` interface with `@Component("specialCustom")`, the bean name plus `Impl` then matches the one defined for the repository implementation in `com.acme.impl.two`, and it is used instead of the first one.

Manual Wiring

If your custom implementation uses annotation-based configuration and autowiring only, the preceding approach shown works well, because it is treated as any other Spring bean. If your implementation fragment bean needs special wiring, you can declare the bean and name it according to the conventions described in the preceding section. The infrastructure then refers to the manually defined bean definition by name instead of creating one itself. The following example shows how to manually wire a custom implementation:

Example 34. Manual wiring of custom implementations

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

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

3.6.2. Customize the Base Repository

The approach described in the preceding section requires customization of each repository interfaces when you want to customize the base repository behavior so that all repositories are affected. To instead change behavior for all repositories, you can create an implementation that extends the persistence technology-specific repository base class. This class then acts as a custom base class for the repository proxies, as shown in the following example:

Example 35. Custom repository base class

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

  private final EntityManager entityManager;

  MyRepositoryImpl(JpaEntityInformation entityInformation,
                          EntityManager entityManager) {
    super(entityInformation, entityManager);

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

  @Transactional
  public <S extends T> S save(S entity) {
    // implementation goes here
```

```
    }
}
```

The class needs to have a constructor of the super class which the store-specific repository factory implementation uses. If the repository base class has multiple constructors, override the one taking an `EntityInformation` plus a store specific infrastructure object (such as an `EntityManager` or a template class).

The final step is to make the Spring Data infrastructure aware of the customized repository base class. In Java configuration, you can do so by using the `repositoryBaseClass` attribute of the `@Enable${store}Repositories` annotation, as shown in the following example:

Example 36. Configuring a custom repository base class using JavaConfig

```
@Configuration
@EnableJpaRepositories(repositoryBaseClass = MyRepositoryImpl.class)
class ApplicationConfiguration { … }
```

A corresponding attribute is available in the XML namespace, as shown in the following example:

Example 37. Configuring a custom repository base class using XML

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

3.7. Publishing Events from Aggregate Roots

Entities managed by repositories are aggregate roots. In a Domain-Driven Design application, these aggregate roots usually publish domain events. Spring Data provides an annotation called `@DomainEvents` that you can use on a method of your aggregate root to make that publication as easy as possible, as shown in the following example:

Example 38. Exposing domain events from an aggregate root

```
class AnAggregateRoot {

    @DomainEvents 1
    Collection<Object> domainEvents() {
        // … return events you want to get published here
    }

    @AfterDomainEventPublication 2
    void callbackMethod() {
       // … potentially clean up domain events list
    }
}
```

1 The method using `@DomainEvents` can return either a single event instance or a collection of events. It must not take any arguments.

2 After all events have been published, we have a method annotated with `@AfterDomainEventPublication`. It can be used to potentially clean the list of events to be published (among other uses).

The methods are called every time one of a Spring Data repository's `save(…)` methods is called.

3.8. 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.

3.8.1. Querydsl Extension

Querydsl is a framework that enables the construction of statically typed SQL-like queries through its fluent API.

Several Spring Data modules offer integration with Querydsl through `QuerydslPredicateExecutor`, as shown in the following example:

Example 39. QuerydslPredicateExecutor interface

```
public interface QuerydslPredicateExecutor<T> {

  Optional<T> findById(Predicate predicate);   1
```

```
  Iterable<T> findAll(Predicate predicate);    2

  long count(Predicate predicate);    3

  boolean exists(Predicate predicate);    4

  // … more functionality omitted.
}
```

1 Finds and returns a single entity matching the `Predicate`.

2 Finds and returns all entities matching the `Predicate`.

3 Returns the number of entities matching the `Predicate`.

4 Returns whether an entity that matches the `Predicate` exists.

To make use of Querydsl support, extend `QuerydslPredicateExecutor` on your repository interface, as shown in the following example

Example 40. Querydsl integration on repositories

```
interface UserRepository extends CrudRepository<User, Long>,
QuerydslPredicateExecutor<User> {
}
```

The preceding example lets you write typesafe queries using Querydsl `Predicate` instances, as shown in the following example:

```
Predicate predicate = user.firstname.equalsIgnoreCase("dave")
    .and(user.lastname.startsWithIgnoreCase("mathews"));

userRepository.findAll(predicate);
```

3.8.2. Web support

> This section contains the documentation for the Spring Data web support as it is implemented in the current (and later) versions of Spring Data Commons. As the newly introduced support changes many things, we kept the documentation of the former behavior in [web.legacy].

Spring Data modules that support the repository programming model ship with a variety of web support. The web related components require Spring MVC JARs to be on the classpath. Some of them even provide integration with Spring HATEOAS. In general, the integration support is enabled by using the `@EnableSpringDataWebSupport` annotation in your JavaConfig configuration class, as shown in the following example:

Example 41. 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 use XML configuration, register either `SpringDataWebConfiguration` or `HateoasAwareSpringDataWebConfiguration` as Spring beans, as shown in the following example (for `SpringDataWebConfiguration`):

Example 42. Enabling Spring Data web support in XML

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

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

Basic Web Support

The configuration shown in the previous section registers a few basic components:

- A `DomainClassConverter` to let Spring MVC 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` lets you use domain types in your Spring MVC controller method signatures directly, so that you need not manually lookup the instances through the repository, as shown in the following example:

Example 43. A Spring MVC controller using domain types in method signatures

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

  @RequestMapping("/{id}")
  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 `findById(…)` on the repository instance registered for the domain type.

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

HandlerMethodArgumentResolvers for Pageable and Sort

The configuration snippet shown in the previous section also registers a `PageableHandlerMethodArgumentResolver` as well as an instance of `SortHandlerMethodArgumentResolver`. The registration enables `Pageable` and `Sort` as valid controller method arguments, as shown in the following example:

Example 44. Using Pageable as controller method argument

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

```
  private final UserRepository repository;

  UserController(UserRepository repository) {
    this.repository = repository;
  }

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

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

The preceding method signature causes Spring MVC try to derive a `Pageable` instance from the request parameters by 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 — for example, `?sort=firstname&sort=lastname,asc`.

To customize this behavior, register a bean implementing the `PageableHandlerMethodArgumentResolverCustomizer` interface or the `SortHandlerMethodArgumentResolverCustomizer` interface, respectively. Its `customize()` method gets called, letting you change settings, as shown in the following example:

```
@Bean SortHandlerMethodArgumentResolverCustomizer sortCustomizer() {
    return s -> s.setPropertyDelimiter("<-->");
}
```

If setting the properties of an existing `MethodArgumentResolver` is not sufficient for your purpose, extend either `SpringDataWebConfiguration` or the HATEOAS-enabled equivalent, override the `pageableResolver()` or `sortResolver()` methods, and import your customized configuration file instead of using the `@Enable` annotation.

If 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}_`. The followig example shows the resulting method signature:

```
String showUsers(Model model,
      @Qualifier("thing1") Pageable first,
      @Qualifier("thing2") Pageable second) { … }
```

you have to populate `thing1_page` and `thing2_page` and so on.

The default `Pageable` passed into the method is equivalent to a `new PageRequest(0, 20)` but can be customized by using the `@PageableDefault` 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, called the `PagedResourcesAssembler`. The following example shows how to use a `PagedResourcesAssembler` as a controller method argument:

Example 45. 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 in the preceding example lets the `PagedResourcesAssembler` be used as a controller method argument. Calling `toResources(…)` on it has the following effects:

- The content of the `Page` becomes the content of the `PagedResources` instance.
- The `PagedResources` object gets a `PageMetadata` instance attached, and it is populated with information from the `Page` and the underlying `PageRequest`.
- The `PagedResources` may get `prev` and `next` links attached, depending on the page's state. The links point to the URI to which the method maps. The pagination parameters added to the method match the setup of the `PageableHandlerMethodArgumentResolver` to make sure the links can be resolved later.

Assume we have 30 Person instances in the database. You can now trigger a request (`GET http://localhost:8080/persons`) and see output similar to the following:

```
{ "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 picked up the default configuration to resolve the parameters into a `Pageable` for an upcoming request. This means that, if you change that configuration, the links 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, which overloads the `PagedResourcesAssembler.toResource(…)` method.

Web Databinding Support

Spring Data projections (described in [projections]) can be used to bind incoming request payloads by either using JSONPath expressions (requires Jayway JsonPath or XPath expressions (requires XmlBeam), as shown in the following example:

Example 46. HTTP payload binding using JSONPath or XPath expressions

```
@ProjectedPayload
public interface UserPayload {
```

```
    @XBRead("//firstname")
    @JsonPath("$..firstname")
    String getFirstname();

    @XBRead("/lastname")
    @JsonPath({ "$.lastname", "$.user.lastname" })
    String getLastname();
  }
```

The type shown in the preceding example can be used as a Spring MVC handler method argument or by using `ParameterizedTypeReference` on one of `RestTemplate`'s methods. The preceding method declarations would try to find `firstname` anywhere in the given document. The `lastname` XML lookup is performed on the top-level of the incoming document. The JSON variant of that tries a top-level `lastname` first but also tries `lastname` nested in a `user` sub-document if the former does not return a value. That way, changes in the structure of the source document can be mitigated easily without having clients calling the exposed methods (usually a drawback of class-based payload binding).

Nested projections are supported as described in [projections]. If the method returns a complex, non-interface type, a Jackson `ObjectMapper` is used to map the final value.

For Spring MVC, the necessary converters are registered automatically as soon as `@EnableSpringDataWebSupport` is active and the required dependencies are available on the classpath. For usage with `RestTemplate`, register a `ProjectingJackson2HttpMessageConverter` (JSON) or `XmlBeamHttpMessageConverter` manually.

For more information, see the web projection example in the canonical Spring Data Examples repository.

Querydsl Web Support

For those stores having QueryDSL integration, it is possible to derive queries from the attributes contained in a `Request` query string.

Consider the following query string:

```
?firstname=Dave&lastname=Matthews
```

Given the `User` object from previous examples, a query string can be resolved to the following value by using the `QuerydslPredicateArgumentResolver`.

```
QUser.user.firstname.eq("Dave").and(QUser.user.lastname.eq("Matthews"))
```

> The feature is automatically enabled, along with `@EnableSpringDataWebSupport`, when Querydsl is found on the classpath.

Adding a `@QuerydslPredicate` to the method signature provides a ready-to-use `Predicate`, which can be run by using the `QuerydslPredicateExecutor`.

> Type information is typically resolved from the method’s return type. Since that information does not necessarily match the domain type, it might be a good idea to use the `root` attribute of `QuerydslPredicate`.

The following exampe shows how to use `@QuerydslPredicate` in a method signature:

```
@Controller
class UserController {

  @Autowired UserRepository repository;

  @RequestMapping(value = "/", method = RequestMethod.GET)
  String index(Model model, @QuerydslPredicate(root = User.class) Predicate predicate,
 1
          Pageable pageable, @RequestParam MultiValueMap<String, String> parameters) {

    model.addAttribute("users", repository.findAll(predicate, pageable));

    return "index";
  }
}
```

1 Resolve query string arguments to matching `Predicate` for `User`.

The default binding is as follows:

- `Object` on simple properties as `eq`.

- `Object` on collection like properties as `contains`.
- `Collection` on simple properties as `in`.

Those bindings can be customized through the `bindings` attribute of `@QuerydslPredicate` or by making use of Java 8 `default methods` and adding the `QuerydslBinderCustomizer` method to the repository interface.

```
interface UserRepository extends CrudRepository<User, String>,
                                 QuerydslPredicateExecutor<User>,                1
                                 QuerydslBinderCustomizer<QUser> {               2

  @Override
  default void customize(QuerydslBindings bindings, QUser user) {

    bindings.bind(user.username).first((path, value) -> path.contains(value))    3
    bindings.bind(String.class)
      .first((StringPath path, String value) -> path.containsIgnoreCase(value)); 4
    bindings.excluding(user.password);                                           5
  }
}
```

1. `QuerydslPredicateExecutor` provides access to specific finder methods for `Predicate`.
2. `QuerydslBinderCustomizer` defined on the repository interface is automatically picked up and shortcuts `@QuerydslPredicate(bindings=…)`.
3. Define the binding for the `username` property to be a simple `contains` binding.
4. Define the default binding for `String` properties to be a case-insensitive `contains` match.
5. Exclude the `password` property from `Predicate` resolution.

3.8.3. Repository Populators

If you work with the Spring JDBC module, you are probably familiar with the support to populate a `DataSource` with 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 47. Data defined in JSON

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

You can 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, declare a populator similar to the following:

Example 48. 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:jackson2-populator locations="classpath:data.json" />

</beans>
```

The preceding declaration causes the `data.json` file to be read and deserialized by a Jackson `ObjectMapper`.

The type to which the JSON object is unmarshalled is determined by inspecting the `_class` attribute of the JSON document. The infrastructure eventually selects the appropriate repository to handle the object that was deserialized.

To instead use XML to define the data the repositories should be populated with, you can use the `unmarshaller-populator` element. You configure it to use one of the XML marshaller options available in Spring OXM. See the Spring reference documentation for details. The following example shows how to unmarshal a repository populator with JAXB:

Example 49. 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>
```

Reference Documentation

4. JDBC Repositories

This chapter points out the specialties for repository support for JDBC. This builds on the core repository support explained in Working with Spring Data Repositories. You should have a sound understanding of the basic concepts explained there.

4.1. Why Spring Data JDBC?

The main persistence API for relational databases in the Java world is certainly JPA, which has its own Spring Data module. Why is there another one?

JPA does a lot of things in order to help the developer. Among other things, it tracks changes to entities. It does lazy loading for you. It lets you map a wide array of object constructs to an equally wide array of database designs.

This is great and makes a lot of things really easy. Just take a look at a basic JPA tutorial. But it often gets really confusing as to why JPA does a certain thing. Also, things that are really simple conceptually get rather difficult with JPA.

Spring Data JDBC aims to be much simpler conceptually, by embracing the following design decisions:

- If you load an entity, SQL statements get executed. Once this is done, you have a completely loaded entity. No lazy loading or caching is done.
- If you save an entity, it gets saved. If you do not, it does not. There is no dirty tracking and no session.
- There is a simple model of how to map entities to tables. It probably only works for rather simple cases. If you do not like that, you should code your own strategy. Spring Data JDBC offers only very limited support for customizing the strategy with annotations.

4.2. Domain Driven Design and Relational Databases.

All Spring Data modules are inspired by the concepts of “repository”, “aggregate”, and “aggregate root” from Domain Driven Design. These are possibly even more important for Spring Data JDBC, because they are, to some extent, contrary to normal practice when working with relational databases.

An aggregate is a group of entities that is guaranteed to be consistent between atomic changes to it. A classic example is an `Order` with `OrderItems`. A property on `Order` (for example, `numberOfItems` is consistent with the actual number of `OrderItems`) remains consistent as changes are made.

References across aggregates are not guaranteed to be consistent at all times. They are guaranteed to become consistent eventually.

Each aggregate has exactly one aggregate root, which is one of the entities of the aggregate. The aggregate gets manipulated only through methods on that aggregate root. These are the atomic changes mentioned earlier.

A repository is an abstraction over a persistent store that looks like a collection of all the aggregates of a certain type. For Spring Data in general, this means you want to have one `Repository` per aggregate root. In addition, for Spring Data JDBC this means that all entities reachable from an aggregate root are considered to be part of that aggregate root. Spring Data JDBC assumes that only the aggregate has a foreign key to a table storing non-root entities of the aggregate and no other entity points toward non-root entities.

> In the current implementation, entities referenced from an aggregate root are deleted and recreated by Spring Data JDBC.

You can overwrite the repository methods with implementations that match your style of working and designing your database.

4.3. Annotation-based Configuration

The Spring Data JDBC repositories support can be activated by an annotation through Java configuration, as the following example shows:

Example 50. Spring Data JDBC repositories using Java configuration

```
@Configuration
@EnableJdbcRepositories
class ApplicationConfig {

  @Bean
  public DataSource dataSource() {

    EmbeddedDatabaseBuilder builder = new EmbeddedDatabaseBuilder();
    return builder.setType(EmbeddedDatabaseType.HSQL).build();
  }

}
```

The configuration class in the preceding example sets up an embedded HSQL database by using the `EmbeddedDatabaseBuilder` API of `spring-jdbc`. We activate Spring Data JDBC repositories by using the `@EnableJdbcRepositories`. If no base package is configured, it uses the package in which the configuration class resides.

4.4. Persisting Entities

Saving an aggregate can be performed with the `CrudRepository.save(…)` method. If the aggregate is new, this results in an insert for the aggregate root, followed by insert statements for all directly or indirectly referenced entities.

If the aggregate root is not new, all referenced entities get deleted, the aggregate root gets updated, and all referenced entities get inserted again. Note that whether an instance is new is part of the instance's state.

This approach has some obvious downsides. If only few of the referenced entities have been actually changed, the deletion and insertion is wasteful. While this process could and probably will be improved, there are certain limitations to what Spring Data JDBC can offer. It does not know the previous state of an aggregate. So any update process always has to take whatever it finds in the database and make sure it converts it to whatever is the state of the entity passed to the save method.

4.4.1. Object Mapping Fundamentals

This section covers the fundamentals of Spring Data object mapping, object creation, field and property access, mutability and immutability. Note, that this section only applies to Spring Data modules that do not use the object mapping of the underlying data store (like JPA). Also be sure to consult the store-specific sections for store-specific object mapping, like indexes, customizing column or field names or the like.

Core responsibility of the Spring Data object mapping is to create instances of domain objects and map the store-native data structures onto those. This means we need two fundamental steps:

1. Instance creation by using one of the constructors exposed.
2. Instance population to materialize all exposed properties.

Object creation

Spring Data automatically tries to detect a persistent entity's constructor to be used to materialize objects of that type. The resolution algorithm works as follows:

1. If there's a no-argument constructor, it will be used. Other constructors will be ignored.
2. If there's a single constructor taking arguments, it will be used.
3. If there are multiple constructors taking arguments, the one to be used by Spring Data will have to be annotated with `@PersistenceConstructor`.

The value resolution assumes constructor argument names to match the property names of the entity, i.e. the resolution will be performed as if the property was to be populated, including all customizations in mapping (different datastore column or field name etc.). This also requires either parameter names information available in the class file or an `@ConstructorProperties` annotation being present on the constructor.

The value resolution can be customized by using Spring Framework's `@Value` value annotation using a store-specific SpEL expression. Please consult the section on store specific mappings for further details.

Object creation internals

To avoid the overhead of reflection, Spring Data object creation uses a factory class generated at runtime by default, which will call the domain classes constructor directly. I.e. for this example type:

```
class Person {
  Person(String firstname, String lastname) { … }
}
```

we will create a factory class semantically equivalent to this one at runtime:

```
class PersonObjectInstantiator implements ObjectInstantiator {

  Object newInstance(Object... args) {
    return new Person((String) args[0], (String) args[1]);
  }
}
```

This gives us a roundabout 10% performance boost over reflection. For the domain class to be eligible for such optimization, it needs to adhere to a set of constraints:

- it must not be a private class
- it must not be a non-static inner class
- it must not be a CGLib proxy class
- the constructor to be used by Spring Data must not be private

If any of these criteria match, Spring Data will fall back to entity instantiation via reflection.

Property population

Once an instance of the entity has been created, Spring Data populates all remaining persistent properties of that class. Unless already populated by the entity's constructor (i.e. consumed through its constructor argument list), the identifier property will be populated first to allow the resolution of cyclic object references. After that, all non-transient properties that have not already been populated by the constructor are set on the entity instance. For that we use the following algorithm:

1. If the property is immutable but exposes a wither method (see below), we use the wither to create a new entity instance with the new property value.
2. If property access (i.e. access through getters and setters) is defined, we're invoking the setter method.
3. By default, we set the field value directly.

Property population internals

Similarly to our optimizations in object construction we also use Spring Data runtime generated accessor classes to interact with the entity instance.

```
class Person {

  private final Long id;
  private String firstname;
  private @AccessType(Type.PROPERTY) String lastname;

  Person() {
    this.id = null;
  }

  Person(Long id, String firstname, String lastname) {
    // Field assignments
  }

  Person withId(Long id) {
    return new Person(id, this.firstname, this.lastame);
  }

  void setLastname(String lastname) {
    this.lastname = lastname;
  }
}
```

Example 51. A generated Property Accessor

```
class PersonPropertyAccessor implements PersistentPropertyAccessor {

  private static final MethodHandle firstname;              2
```

```
  private Person person;                                    1

  public void setProperty(PersistentProperty property, Object value) {

    String name = property.getName();

    if ("firstname".equals(name)) {
      firstname.invoke(person, (String) value);             2
    } else if ("id".equals(name)) {
      this.person = person.withId((Long) value);            3
    } else if ("lastname".equals(name)) {
      this.person.setLastname((String) value);              4
    }
  }
}
```

1. PropertyAccessor's hold a mutable instance of the underlying object. This is, to enable mutations of otherwise immutable properties.
2. By default, Spring Data uses field-access to read and write property values. As per visibility rules of `private` fields, `MethodHandles` are used to interact with fields.
3. The class exposes a `withId(…)` method that's used to set the identifier, e.g. when an instance is inserted into the datastore and an identifier has been generated. Calling `withId(…)` creates a new `Person` object. All subsequent mutations will take place in the new instance leaving the previous untouched.
4. Using property-access allows direct method invocations without using `MethodHandles`.

This gives us a roundabout 25% performance boost over reflection. For the domain class to be eligible for such optimization, it needs to adhere to a set of constraints:

- Types must not reside in the default or under the `java` package.
- Types and their constructors must be `public`
- Types that are inner classes must be `static`.
- The used Java Runtime must allow for declaring classes in the originating `ClassLoader`. Java 9 and newer impose certain limitations.

By default, Spring Data attempts to use generated property accessors and falls back to reflection-based ones if a limitation is detected.

Let’s have a look at the following entity:

Example 52. A sample entity

```
class Person {

  private final @Id Long id;                                             1
  private final String firstname, lastname;                              2
  private final LocalDate birthday;
  private final int age; 3

  private String comment;                                                4
  private @AccessType(Type.PROPERTY) String remarks;                     5

  static Person of(String firstname, String lastname, LocalDate birthday) { 6

    return new Person(null, firstname, lastname, birthday,
      Period.between(birthday, LocalDate.now()).getYears());
  }

  Person(Long id, String firstname, String lastname, LocalDate birthday, int age) { 6

    this.id = id;
    this.firstname = firstname;
    this.lastname = lastname;
    this.birthday = birthday;
    this.age = age;
  }

  Person withId(Long id) {                                               1
    return new Person(id, this.firstname, this.lastname, this.birthday);
  }

  void setRemarks(String remarks) {                                      5
    this.remarks = remarks;
  }
}
```

1 The identifier property is final but set to `null` in the constructor. The class exposes a `withId(…)` method that’s used to set the identifier, e.g. when an instance is inserted into the datastore and an identifier has been generated. The original `Person` instance stays unchanged as a new one is created. The same pattern is usually applied for other properties that are store managed but might have to be changed for persistence operations.

2 The `firstname` and `lastname` properties are ordinary immutable properties potentially

exposed through getters.

3 The `age` property is an immutable but derived one from the `birthday` property. With the design shown, the database value will trump the defaulting as Spring Data uses the only declared constructor. Even if the intent is that the calculation should be preferred, it's important that this constructor also takes `age` as parameter (to potentially ignore it) as otherwise the property population step will attempt to set the age field and fail due to it being immutable and no wither being present.

4 The `comment` property is mutable is populated by setting its field directly.

5 The `remarks` properties are mutable and populated by setting the `comment` field directly or by invoking the setter method for

6 The class exposes a factory method and a constructor for object creation. The core idea here is to use factory methods instead of additional constructors to avoid the need for constructor disambiguation through `@PersistenceConstructor`. Instead, defaulting of properties is handled within the factory method.

General recommendations

- *Try to stick to immutable objects* — Immutable objects are straightforward to create as materializing an object is then a matter of calling its constructor only. Also, this avoids your domain objects to be littered with setter methods that allow client code to manipulate the objects state. If you need those, prefer to make them package protected so that they can only be invoked by a limited amount of co-located types. Constructor-only materialization is up to 30% faster than properties population.
- *Provide an all-args constructor* — Even if you cannot or don't want to model your entities as immutable values, there's still value in providing a constructor that takes all properties of the entity as arguments, including the mutable ones, as this allows the object mapping to skip the property population for optimal performance.
- *Use factory methods instead of overloaded constructors to avoid* `@PersistenceConstructor` — With an all-argument constructor needed for optimal performance, we usually want to expose more application use case specific constructors that omit things like auto-generated identifiers etc. It's an established pattern to rather use static factory methods to expose these variants of the all-args constructor.
- *Make sure you adhere to the constraints that allow the generated instantiator and property accessor classes to be used* —
- *For identifiers to be generated, still use a final field in combination with a wither method* —

- *Use Lombok to avoid boilerplate code* — As persistence operations usually require a constructor taking all arguments, their declaration becomes a tedious repetition of boilerplate parameter to field assignments that can best be avoided by using Lombok's `@AllArgsConstructor`.

Kotlin support

Spring Data adapts specifics of Kotlin to allow object creation and mutation.

Kotlin object creation

Kotlin classes are supported to be instantiated , all classes are immutable by default and require explicit property declarations to define mutable properties. Consider the following `data class Person`:

```
data class Person(val id: String, val name: String)
```

The class above compiles to a typical class with an explicit constructor. We can customize this class by adding another constructor and annotate it with `@PersistenceConstructor` to indicate a constructor preference:

```
data class Person(var id: String, val name: String) {

    @PersistenceConstructor
    constructor(id: String) : this(id, "unknown")
}
```

Kotlin supports parameter optionality by allowing default values to be used if a parameter is not provided. When Spring Data detects a constructor with parameter defaulting, then it leaves these parameters absent if the data store does not provide a value (or simply returns `null`) so Kotlin can apply parameter defaulting. Consider the following class that applies parameter defaulting for `name`

```
data class Person(var id: String, val name: String = "unknown")
```

Every time the `name` parameter is either not part of the result or its value is `null`, then the `name` defaults to `unknown`.

Property population of Kotlin data classes

In Kotlin, all classes are immutable by default and require explicit property declarations to define mutable properties. Consider the following `data class Person`:

```
data class Person(val id: String, val name: String)
```

This class is effectively immutable. It allows to create new instances as Kotlin generates a `copy(…)` method that creates new object instances copying all property values from the existing object and applying property values provided as arguments to the method.

4.4.2. Supported Types in Your Entity

The properties of the following types are currently supported:

- All primitive types and their boxed types (`int`, `float`, `Integer`, `Float`, and so on)
- Enums get mapped to their name.
- `String`
- `java.util.Date`, `java.time.LocalDate`, `java.time.LocalDateTime`, and `java.time.LocalTime`
- Anything your database driver accepts.
- References to other entities. They are considered a one-to-one relationship. It is optional for such entities to have an `id` attribute. The table of the referenced entity is expected to have an additional column named the same as the table of the referencing entity. You can change this name by implementing `NamingStrategy.getReverseColumnName(RelationalPersistentProperty property)`.
- `Set<some entity>` is considered a one-to-many relationship. The table of the referenced entity is expected to have an additional column named the same as the table of the referencing entity. You can change this name by implementing `NamingStrategy.getReverseColumnName(RelationalPersistentProperty property)`.
- `Map<simple type, some entity>` is considered a qualified one-to-many relationship. The table of the referenced entity is expected to have two additional columns: One named the same as the table of the referencing entity for the foreign key and one with the same

name and an additional `_key` suffix for the map key. You can change this behavior by implementing `NamingStrategy.getReverseColumnName(RelationalPersistentProperty property)` and `NamingStrategy.getKeyColumn(RelationalPersistentProperty property)`, respectively. Alternatively you may annotate the attribute with `@Column(value="your_column_name", keyColumn="your_key_column_name")`

- `List<some entity>` is mapped as a `Map<Integer, some entity>`.

The handling of referenced entities is limited. This is based on the idea of aggregate roots as described above. If you reference another entity, that entity is, by definition, part of your aggregate. So, if you remove the reference, the previously referenced entity gets deleted. This also means references are 1-1 or 1-n, but not n-1 or n-m.

If you have n-1 or n-m references, you are, by definition, dealing with two separate aggregates. References between those should be encoded as simple `id` values, which should map properly with Spring Data JDBC.

4.4.3. `NamingStrategy`

When you use the standard implementations of `CrudRepository` that Spring Data JDBC provides, they expect a certain table structure. You can tweak that by providing a [`NamingStrategy`] in your application context.

4.4.4. Entity State Detection Strategies

The following table describes the strategies that Spring Data JDBC offers for detecting whether an entity is new:

Table 2. Options for detection whether an entity is new in Spring Data JDBC

Id-Property inspection (the default)	By default, Spring Data JDBC inspects the identifier property of the given entity. If the identifier property is `null`, then the entity is assumed to be new. Otherwise, it is assumed to not be new.
Implementing `Persistable`	If an entity implements `Persistable`, Spring Data JDBC delegates the new detection to the `isNew(…)` method of the entity. See the [Javadoc] for details.
Implementing `EntityInformation`	You can customize the `EntityInformation` abstraction used in the `SimpleJdbcRepository` implementation by creating a subclass of `JdbcRepositoryFactory` and overriding the `getEntityInformation(…)` method. You then have to register the custom implementation of

	`JdbcRepositoryFactory` as a Spring bean. Note that this should rarely be necessary. See the Javadoc for details.

4.4.5. ID Generation

Spring Data JDBC uses the ID to identify entities. The ID of an entity must be annotated with Spring Data's `@Id` annotation.

When your data base has an auto-increment column for the ID column, the generated value gets set in the entity after inserting it into the database.

One important constraint is that, after saving an entity, the entity must not be new any more. Note that whether an entity is new is part of the entity's state. With auto-increment columns, this happens automatically, because the ID gets set by Spring Data with the value from the ID column. If you are not using auto-increment columns, you can use a `BeforeSave` listener, which sets the ID of the entity (covered later in this document).

4.5. Query Methods

This section offers some specific information about the implementation and use of Spring Data JDBC.

4.5.1. Query Lookup Strategies

The JDBC module supports defining a query manually only as a String in a `@Query` annotation. Deriving a query from the name of the method is currently not supported.

4.5.2. Using `@Query`

The following example shows how to use `@Query` to declare a query method:

Example 53. Declare a query method by using @Query

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

  @Query("select firstName, lastName from User u where u.emailAddress = :email")
  User findByEmailAddress(@Param("email") String email);
}
```

Spring fully supports Java 8's parameter name discovery based on the `-parameters` compiler flag. By using this flag in your build as an alternative to debug information, you can omit the `@Param` annotation for named parameters.

Spring Data JDBC supports only named parameters.

Custom `RowMapper`

You can configure which `RowMapper` to use, either by using the `@Query(rowMapperClass = ….)` or by registering a `RowMapperMap` bean and registering a `RowMapper` per method return type. The following example shows how to register `RowMappers`:

```
@Bean
RowMapperMap rowMappers() {
    return new ConfigurableRowMapperMap() //
        .register(Person.class, new PersonRowMapper()) //
        .register(Address.class, new AddressRowMapper());
}
```

When determining which `RowMapper` to use for a method, the following steps are followed, based on the return type of the method:

1. If the type is a simple type, no `RowMapper` is used.

   Instead, the query is expected to return a single row with a single column, and a conversion to the return type is applied to that value.

2. The entity classes in the `RowMapperMap` are iterated until one is found that is a superclass or interface of the return type in question. The `RowMapper` registered for that class is used.

   Iterating happens in the order of registration, so make sure to register more general types after specific ones.

If applicable, wrapper types such as collections or `Optional` are unwrapped. Thus, a return type of `Optional<Person>` uses the `Person` type in the preceding process.

Modifying Query

You can mark a query as being a modifying query by using the `@Modifying` on query method, as the following example shows:

```
@Modifying
@Query("UPDATE DUMMYENTITY SET name = :name WHERE id = :id")
boolean updateName(@Param("id") Long id, @Param("name") String name);
```

You can specify the following return types:

- `void`
- `int` (updated record count)
- `boolean` (whether a record was updated)

4.6. MyBatis Integration

For each operation in `CrudRepository`, Spring Data JDBC runs multiple statements. If there is a `SqlSessionFactory` in the application context, Spring Data checks, for each step, whether the `SessionFactory` offers a statement. If one is found, that statement (including its configured mapping to an entity) is used.

The name of the statement is constructed by concatenating the fully qualified name of the entity type with `Mapper.` and a `String` determining the kind of statement. For example, if an instance of `org.example.User` is to be inserted, Spring Data JDBC looks for a statement named `org.example.UserMapper.insert`.

When the statement is run, an instance of [`MyBatisContext`] gets passed as an argument, which makes various arguments available to the statement.

The following table describes the available MyBatis statements:

Name	Purpose	CrudRepository methods that might trigger this statement	Attributes available in the `MyBatisContext`

Name	Purpose	CrudRepository methods that might trigger this statement	Attributes available in the `MyBatisContext`
`insert`	Inserts a single entity. This also applies for entities referenced by the aggregate root.	`save`, `saveAll`.	`getInstance`: the instance to be saved `getDomainType`: The type of the entity to be saved. `get(<key>)`: ID of the referencing entity, where `<key>` is the name of the back reference column provided by the `NamingStrategy`.
`update`	Updates a single entity. This also applies for entities referenced by the aggregate root.	`save`, `saveAll`.	`getInstance`: The instance to be saved `getDomainType`: The type of the entity to be saved.
`delete`	Deletes a single entity.	`delete`, `deleteById`.	`getId`: The ID of the instance to be deleted `getDomainType`: The type of the entity to be deleted.

Name	Purpose	CrudRepository methods that might trigger this statement	Attributes available in the `MyBatisContext`
`deleteAll-<propertyPath>`	Deletes all entities referenced by any aggregate root of the type used as prefix with the given property path. Note that the type used for prefixing the statement name is the name of the aggregate root, not the one of the entity to be deleted.	`deleteAll`.	`getDomainType`: The types of the entities to be deleted.
`deleteAll`	Deletes all aggregate roots of the type used as the prefix	`deleteAll`.	`getDomainType`: The type of the entities to be deleted.
`delete-<propertyPath>`	Deletes all entities referenced by an aggregate root with the given propertyPath	`deleteById`.	`getId`: The ID of the aggregate root for which referenced entities are to be deleted. `getDomainType`: The type of the entities to be deleted.

Name	Purpose	CrudRepository methods that might trigger this statement	Attributes available in the `MyBatisContext`
`findById`	Selects an aggregate root by ID	`findById`.	`getId`: The ID of the entity to load. `getDomainType`: The type of the entity to load.
`findAll`	Select all aggregate roots	`findAll`.	`getDomainType`: The type of the entity to load.
`findAllById`	Select a set of aggregate roots by ID values	`findAllById`.	`getId`: A list of ID values of the entities to load. `getDomainType`: The type of the entity to load.
`findAllByProperty-<propertyName>`	Select a set of entities that is referenced by another entity. The type of the referencing entity is used for the prefix. The referenced entities type is used as the suffix.	All `find*` methods.	`getId`: The ID of the entity referencing the entities to be loaded. `getDomainType`: The type of the entity to load.
`count`	Count the number of aggregate root of the type used as prefix	`count`	`getDomainType`: The type of aggregate roots to count.

4.7. Events

Spring Data JDBC triggers events that get published to any matching `ApplicationListener` in the application context. For example, the following listener gets invoked before an aggregate gets saved:

```
@Bean
public ApplicationListener<BeforeSave> timeStampingSaveTime() {

    return event -> {

        Object entity = event.getEntity();
        if (entity instanceof Category) {
            Category category = (Category) entity;
            category.timeStamp();
        }
    };
}
```

The following table describes the available events:

Table 3. Available events

Event	When It Is Published
`BeforeDeleteEvent`	Before an aggregate root gets deleted.
`AfterDeleteEvent`	After an aggregate root gets deleted.
`BeforeSaveEvent`	Before an aggregate root gets saved (that is, inserted or updated but after the decision about whether if it gets updated or deleted was made). The event has a reference to an `AggregateChange` instance. The instance can be modified by adding or removing `DbAction` instances.
`AfterSaveEvent`	After an aggregate root gets saved (that is, inserted or updated).

Event	When It Is Published
AfterLoadEvent	After an aggregate root gets created from a database `ResultSet` and all its property get set.

4.8. Logging

Spring Data JDBC does little to no logging on its own. Instead, the mechanics of `JdbcTemplate` to issue SQL statements provide logging. Thus, if you want to inspect what SQL statements are executed, activate logging for Spring's `NamedParameterJdbcTemplate` or MyBatis.

4.9. 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` annotation so that default transaction configuration applies. For details, see the Javadoc of `SimpleJdbcRepository`. If you need to tweak transaction configuration for one of the methods declared in a repository, redeclare the method in your repository interface, as follows:

Example 54. Custom transaction configuration for CRUD

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

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

  // Further query method declarations
}
```

The preceding causes the `findAll()` method to be executed with a timeout of 10 seconds and without the `readOnly` flag.

Another way to alter transactional behavior is by using a facade or service implementation that typically covers more than one repository. Its purpose is to define transactional boundaries for non-CRUD operations. The following example shows how to create such a facade:

Example 55. Using a facade to define transactions for multiple repository calls

```
@Service
class UserManagementImpl implements UserManagement {

  private final UserRepository userRepository;
  private final RoleRepository roleRepository;

  @Autowired
  public UserManagementImpl(UserRepository userRepository,
    RoleRepository roleRepository) {
    this.userRepository = userRepository;
    this.roleRepository = roleRepository;
  }

  @Transactional
  public void addRoleToAllUsers(String roleName) {

    Role role = roleRepository.findByName(roleName);

    for (User user : userRepository.findAll()) {
      user.addRole(role);
      userRepository.save(user);
    }
}
```

The preceding example causes calls to `addRoleToAllUsers(…)` to run inside a transaction (participating in an existing one or creating a new one if none are already running). The transaction configuration for the repositories is neglected, as the outer transaction configuration determines the actual repository to be used. Note that you have to explicitly activate `<tx:annotation-driven />` or use `@EnableTransactionManagement` to get annotation-based configuration for facades working. Note that the preceding example assumes you use component scanning.

4.9.1. Transactional Query Methods

To let your query methods be transactional, use `@Transactional` at the repository interface you define, as the following example shows:

Example 56. Using @Transactional at query methods

```
@Transactional(readOnly = true)
public interface UserRepository extends CrudRepository<User, Long> {

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

```
  @Modifying
  @Transactional
  @Query("delete from User u where u.active = false")
  void deleteInactiveUsers();
}
```

Typically, you want the `readOnly` flag to be set to true, because most of the query methods only read data. In contrast to that, `deleteInactiveUsers()` uses the `@Modifying` annotation and overrides the transaction configuration. Thus, the method is with the `readOnly` flag set to `false`.

> It is definitely reasonable to use transactions for read-only queries, and we can mark them as such by setting the `readOnly` flag. This does not, however, act as a check that you do not trigger a manipulating query (although some databases reject `INSERT` and `UPDATE` statements inside a read-only transaction). Instead, the `readOnly` flag is propagated as a hint to the underlying JDBC driver for performance optimizations.

4.10. Auditing

4.10.1. Basics

Spring Data provides sophisticated support to transparently keep track of who created or changed an entity and when the change 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` and `@LastModifiedBy` to capture the user who created or modified the entity as well as `@CreatedDate` and `@LastModifiedDate` to capture when the change happened.

Example 57. An audited entity

```
class Customer {

  @CreatedBy
  private User user;
```

```
  @CreatedDate
  private DateTime createdDate;

  // … further properties omitted
}
```

As you can see, the annotations can be applied selectively, depending on which information you want to capture. The annotations capturing when changes were made can be used on properties of type Joda-Time, `DateTime`, legacy Java `Date` and `Calendar`, JDK8 date and time types, and `long` or `Long`.

Interface-based Auditing Metadata

In case you do not 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 is also a convenience base class, `AbstractAuditable`, which you can extend to avoid the need to manually implement the interface methods. Doing so 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 what type the properties annotated with `@CreatedBy` or `@LastModifiedBy` have to be.

The following example shows an implementation of the interface that uses Spring Security's `Authentication` object:

Example 58. 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 accesses the `Authentication` object provided by Spring Security and looks up the custom `UserDetails` instance that you have created in your `UserDetailsService` implementation. We assume here that you are exposing the domain user through the `UserDetails` implementation but that, based on the `Authentication` found, you could also look it up from anywhere.

4.11. JDBC Auditing

In order to activate auditing, add `@EnableJdbcAuditing` to your configuration, as the following example shows:

Example 59. Activating auditing with Java configuration

```
@Configuration
@EnableJdbcAuditing
class Config {

  @Bean
  public AuditorAware<AuditableUser> auditorProvider() {
    return new AuditorAwareImpl();
  }
}
```

If you expose a bean of type `AuditorAware` to the `ApplicationContext`, the auditing infrastructure automatically picks it up and uses it to determine the current user to be set on domain types. If you have multiple implementations registered in the `ApplicationContext`, you can select the one to be used by explicitly setting the `auditorAwareRef` attribute of `@EnableJdbcAuditing`.

Appendix

Appendix A: Frequently Asked Questions

Sorry. We have no frequently asked questions so far.

Appendix B: Glossary

AOP

Aspect-Oriented Programming

CRUD

Create, Read, Update, Delete - Basic persistence operations

Dependency Injection

Pattern to hand a component's dependency to the component from outside, freeing the component to lookup the dependent itself. For more information, see http://en.wikipedia.org/wiki/Dependency_Injection.

JPA

Java Persistence API

Spring

Java application framework — http://projects.spring.io/spring-framework

Appendix C: 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". The following table describes the attributes of the `<repositories />` element:

Table 4. Attributes

Name	Description

Name	Description
`base-package`	Defines the package to be scanned for repository interfaces that extend `*Repository` (the actual interface is determined by the specific Spring Data module) in auto-detection mode. All packages below the configured package are scanned, too. Wildcards are allowed.
`repository-impl-postfix`	Defines the postfix to autodetect custom repository implementations. Classes whose names end with the configured postfix are 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 search for a Properties file containing externally defined queries.
`consider-nested-repositories`	Whether nested repository interface definitions should be considered. Defaults to `false`.

Appendix D: Populators namespace reference

The <populator /> element

The `<populator />` element allows to populate the a data store via the Spring Data repository infrastructure.[1]

Table 5. Attributes

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

Appendix E: Repository query keywords

Supported query keywords

Spring Data JDBC does not support query derivation yet.

Appendix F: 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 types listed here might not be supported in a particular store.

> Geospatial types (such as `GeoResult`, `GeoResults`, and `GeoPage`) are available only for data stores that support geospatial queries.

Table 6. 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. If no result is found, `null` is returned. More than one result triggers an `IncorrectResultSizeDataAccessException`.
`Iterator<T>`	An `Iterator`.
`Collection<T>`	A `Collection`.
`List<T>`	A `List`.

Return type	Description
`Optional<T>`	A Java 8 or Guava `Optional`. Expects the query method to return one result at most. If no result is found, `Optional.empty()` or `Optional.absent()` is returned. More than one result triggers an `IncorrectResultSizeDataAccessException`.
`Option<T>`	Either a Scala or Javaslang `Option` type. Semantically the same behavior as Java 8's `Optional`, described earlier.
`Stream<T>`	A Java 8 `Stream`.
`Future<T>`	A `Future`. Expects a method to be annotated with `@Async` and requires Spring's asynchronous method execution capability to be enabled.
`CompletableFuture<T>`	A Java 8 `CompletableFuture`. Expects a method to be annotated with `@Async` and requires Spring's asynchronous method execution capability to be enabled.
`ListenableFuture`	A `org.springframework.util.concurrent.ListenableFuture`. Expects a method to be annotated with `@Async` and requires Spring's asynchronous method execution capability to be enabled.
`Slice`	A sized chunk of data with an indication of whether there is more data available. Requires a `Pageable` method parameter.
`Page<T>`	A `Slice` with additional information, such as the total number of results. Requires a `Pageable` method parameter.
`GeoResult<T>`	A result entry with additional information, such as the distance to a reference location.
`GeoResults<T>`	A list of `GeoResult<T>` with additional information, such as the average distance to a reference location.
`GeoPage<T>`	A `Page` with `GeoResult<T>`, such as the average distance to a reference location.

Return type	Description
`Mono<T>`	A Project Reactor `Mono` emitting zero or one element using reactive repositories. Expects the query method to return one result at most. If no result is found, `Mono.empty()` is returned. More than one result triggers an `IncorrectResultSizeDataAccessException`.
`Flux<T>`	A Project Reactor `Flux` emitting zero, one, or many elements using reactive repositories. Queries returning `Flux` can emit also an infinite number of elements.
`Single<T>`	A RxJava `Single` emitting a single element using reactive repositories. Expects the query method to return one result at most. If no result is found, `Mono.empty()` is returned. More than one result triggers an `IncorrectResultSizeDataAccessException`.
`Maybe<T>`	A RxJava `Maybe` emitting zero or one element using reactive repositories. Expects the query method to return one result at most. If no result is found, `Mono.empty()` is returned. More than one result triggers an `IncorrectResultSizeDataAccessException`.
`Flowable<T>`	A RxJava `Flowable` emitting zero, one, or many elements using reactive repositories. Queries returning `Flowable` can emit also an infinite number of elements.

1. see XML configuration

Version 1.0.1.RELEASE
Last updated 2018-10-15 10:39:32 MESZ