spring by Pivotal

Spring Data JDBC - Reference Documentation

Jens Schauder, Jay Bryant, Mark Paluch, Bastian Wilhelm · – Version 2.1.15, 2021-11-12

Preface

The Spring Data JDBC project applies core Spring concepts to the development of solutions that use JDBC databases aligned with Domain-driven design principles. We provide a “template” as a high-level abstraction for storing and querying aggregates.

This document is the reference guide for Spring Data JDBC Support. It explains the concepts and semantics and syntax..

This section provides some basic introduction. The rest of the document refers only to Spring Data JDBC features and assumes the user is familiar with SQL and Spring concepts.

1. Learning Spring

Spring Data uses Spring framework’s core functionality, including:

- IoC container

- type conversion system
- expression language
- JMX integration
- DAO exception hierarchy.

While you need not know the Spring APIs, understanding the concepts behind them is important. At a minimum, the idea behind Inversion of Control (IoC) should be familiar, and you should be familiar with whatever IoC container you choose to use.

The core functionality of the JDBC Aggregate support can be used directly, with no need to invoke the IoC services of the Spring Container. This is much like `JdbcTemplate`, which can be used "'standalone'" without any other services of the Spring container. To leverage all the features of Spring Data JDBC, such as the repository support, you need to configure some parts of the library to use Spring.

To learn more about Spring, you can refer to the comprehensive documentation that explains the Spring Framework in detail. There are a lot of articles, blog entries, and books on the subject. See the Spring framework home page for more information.

---

2. Requirements

The Spring Data JDBC binaries require JDK level 8.0 and above and Spring Framework 5.3.13 and above.

In terms of databases, Spring Data JDBC requires a dialect to abstract common SQL functionality over vendor-specific flavours. Spring Data JDBC includes direct support for the following databases:

- DB2
- H2
- HSQLDB
- MariaDB
- Microsoft SQL Server
- MySQL
- Oracle

- Postgres

If you use a different database then your application won't startup. The dialect section contains further detail on how to proceed in such case.

---

3. Additional Help Resources

Learning a new framework is not always straightforward. In this section, we try to provide what we think is an easy-to-follow guide for starting with the Spring Data JDBC module. However, if you encounter issues or you need advice, feel free to use one of the following links:

Community Forum

Spring Data on Stack Overflow is a tag for all Spring Data (not just Document) users to share information and help each other. Note that registration is needed only for posting.

Professional Support

Professional, from-the-source support, with guaranteed response time, is available from Pivotal Sofware, Inc., the company behind Spring Data and Spring.

---

4. Following Development

For information on the Spring Data JDBC source code repository, nightly builds, and snapshot artifacts, see the Spring Data JDBC homepage. You can help make Spring Data best serve the needs of the Spring community by interacting with developers through the Community on Stack Overflow. If you encounter a bug or want to suggest an improvement, please create a ticket on the Spring Data issue tracker. To stay up to date with the latest news and announcements in the Spring eco system, subscribe to the Spring Community Portal. You can also follow the Spring blog or the project team on Twitter (SpringData).

---

5. Project Metadata

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

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

6. New & Noteworthy

This section covers the significant changes for each version.

6.1. What's New in Spring Data JDBC 2.1

- Dialect for Oracle databases.
- Support for `@Value` in persistence constructors.

6.2. What's New in Spring Data JDBC 2.0

- Optimistic Locking support.
- Support for `PagingAndSortingRepository`.
- Query Derivation.
- Full Support for H2.
- All SQL identifiers know get quoted by default.
- Missing columns no longer cause exceptions.

6.3. What's New in Spring Data JDBC 1.1

- `@Embedded` entities support.
- Store `byte[]` as `BINARY`.
- Dedicated `insert` method in the `JdbcAggregateTemplate`.
- Read only property support.

6.4. What's New in Spring Data JDBC 1.0

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

- Id generation.
- Event support.
- Auditing.
- `CustomConversions` .

7. Dependencies

Due to the different inception dates of individual Spring Data modules, most of them carr different major and minor version numbers. The easiest way to find compatible ones is t 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

```xml
<dependencyManagement>
  <dependencies>
    <dependency>
      <groupId>org.springframework.data</groupId>
      <artifactId>spring-data-bom</artifactId>
      <version>2020.0.15</version>
      <scope>import</scope>
      <type>pom</type>
    </dependency>
  </dependencies>
</dependencyManagement>
```

The current release train version is `2020.0.15` . The train version uses calver with the pattern `YYYY.MINOR.MICRO` . The version name follows `${calver}` for GA releases and service releases and the following pattern for all other versions: `${calver}-${modifier}` , where `modifier` can be one of the following:

- `SNAPSHOT` : Current snapshots
- `M1` , `M2` , and so on: Milestones
- `RC1` , `RC2` , and so on: Release candidates

You can find a working example of using the BOMs 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

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

7.1. 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, set the `spring-data-releasetrain.version` property to the train version and iteration you would like to use.

7.2. Spring Framework

The current version of Spring Data modules require Spring Framework 5.3.13 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.

---

8. 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 that support 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.

8.1. 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` interface provides sophisticated CRUD functionality for the entity class that is being managed.

Example 3. `CrudRepository` Interface

```java
public interface CrudRepository<T, ID> 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

```java
public interface PagingAndSortingRepository<T, ID> 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:

```java
PagingAndSortingRepository<User, Long> repository = // … get access to a bean
Page<User> users = repository.findAll(PageRequest.of(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

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

  long countByLastname(String lastname);
}
```

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

Example 6. Derived Delete Query

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

  long deleteByLastname(String lastname);

  List<User> removeByLastname(String lastname);
}
```

8.2. 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:

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

2. Declare query methods on the interface.

```java
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:

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

@EnableJpaRepositories
class Config { … }
```

   b. To use XML configuration, define a bean similar to the following:

```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:jpa="http://www.springframework.org/schema/data/jpa"
   xsi:schemaLocation="http://www.springframework.org/schema/beans
     https://www.springframework.org/schema/beans/spring-beans.xsd
     http://www.springframework.org/schema/data/jpa
     https://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:

```java
```

```
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

8.3. Defining Repository Interfaces

To define a repository interface, you first need to 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`.

8.3.1. 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 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.

8.3.2. 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, it is a valid candidate for the particular Spring Data module.
2. If the domain class is annotated with the module-specific type annotation, 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 8. Repository definitions using module-specific interfaces

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

@NoRepositoryBean
interface MyBaseRepository<T, ID> 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 9. Repository definitions using generic interfaces

```java
```

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

@NoRepositoryBean
interface MyBaseRepository<T, ID> 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 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 10. Repository definitions using domain classes with annotations

```
JAVA
Explain
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 11. Repository definitions using domain classes with mixed annotations

```
JAVA
Explain
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 12. Annotation-driven configuration of base packages

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

8.4. 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.

8.4.1. 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 it cannot find one. The query can be defined by an annotation somewhere or declared by other means. See 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` (the 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.

8.4.2. Query Creation

The query builder mechanism built into the Spring Data repository infrastructure is useful for building constraining queries over entities of the repository.

The following example shows how to create a number of queries:

Example 13. Query creation from method names

```
interface PersonRepository extends Repository<Person, 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);
}
```

Parsing query method names is divided into subject and predicate. The first part (`find…By` , `exists…By`) defines the subject of the query, the second part forms the predicate. The introducing clause (subject) can contain further expressions. Any text between `find` (or other introducing keywords) and `By` is considered to be descriptive unless using one of the result-limiting keywords such as a `Distinct` to set a distinct flag on the query to be created or `Top` / `First` to limit query results.

The appendix contains the full list of query method subject keywords and query method predicate keywords including sorting and letter-casing modifiers. However, the first `By` acts as a delimiter to indicate the start of the actual criteria predicate. At a very basic level, you can define conditions on entity properties and concatenate them with `And` and `Or` .

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".

8.4.3. 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 prope a property of the managed domain class. However, you can also define constraints by traversing nested properties. Consider the following method signature:

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

Assume a `Person` has an `Address` with a `ZipCode` . In that case, the method creates the `x.address.zipCode` property traversal. 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:

```java
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).

8.4.4. 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 14. Using `Pageable`, `Slice`, and `Sort` in query methods

```java
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);
```

> APIs taking `Sort` and `Pageable` expect non-`null` values to be handed into methods. If you do not want to apply any sorting or pagination, use `Sort.unsorted()` and `Pageable.unpaged()`.

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` knows only 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 need only 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.

Paging and Sorting

You can define simple sorting expressions by using property names. You can concatenate expressions to collect multiple criteria into one expression.

Example 15. Defining sort expressions

```java
Sort sort = Sort.by("firstname").ascending()
  .and(Sort.by("lastname").descending());
```

For a more type-safe way to define sort expressions, start with the type for which to define the sort expression and use method references to define the properties on which to sort.

Example 16. Defining sort expressions by using the type-safe API

```java
TypedSort<Person> person = Sort.sort(Person.class);

Sort sort = person.by(Person::getFirstname).ascending()
  .and(person.by(Person::getLastname).descending());
```

`TypedSort.by(…)` makes use of runtime proxies by (typically) using CGlib, which may interfere with native image compilation when using tools such as Graal VM Native.

If your store implementation supports Querydsl, you can also use the generated metamodel types to define sort expressions:

Example 17. Defining sort expressions by using the Querydsl API

```java
QSort sort = QSort.by(QPerson.firstname.asc())
  .and(QSort.by(QPerson.lastname.desc()));
```

8.4.5. Limiting Query Results

You can limit the results of query methods by using the `first` or `top` keywords, which you can use interchangeably. You can append an optional numeric value 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`

```java
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 for datastores that support distinct queries. Also, for the queries that limit 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 available pages), 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.

8.4.6. Repository Methods Returning Collections or Iterables

Query methods that return multiple results can use standard Java `Iterable`, `List`, and `Set`. Beyond that, we support returning Spring Data's `Streamable`, a custom extension of `Iterable`, as well as collection types provided by Vavr. Refer to the appendix explaining all possible query method return types.

Using Streamable as Query Method Return Type

You can use `Streamable` as alternative to `Iterable` or any collection type. It provides convenience methods to access a non-parallel `Stream` (missing from `Iterable`) and the a... to directly `….filter(…)` and `….map(…)` over the elements and concatenate the `Streamable` to others:

Example 19. Using Streamable to combine query method results

Explain JAVA

```
interface PersonRepository extends Repository<Person, Long> {
  Streamable<Person> findByFirstnameContaining(String firstname);
  Streamable<Person> findByLastnameContaining(String lastname);
}

Streamable<Person> result = repository.findByFirstnameContaining("av")
  .and(repository.findByLastnameContaining("ea"));
```

Returning Custom Streamable Wrapper Types

Providing dedicated wrapper types for collections is a commonly used pattern to provide an API for a query result that returns multiple elements. Usually, these types are used by invoking a repository method returning a collection-like type and creating an instance of the wrapper type manually. You can avoid that additional step as Spring Data lets you use these wrapper types as query method return types if they meet the following criteria:

1. The type implements `Streamable`.
2. The type exposes either a constructor or a static factory method named `of(…)` or `valueOf(…)` that takes `Streamable` as an argument.

The following listing shows an example:

```java
class Product {                                          1
  MonetaryAmount getPrice() { … }
}

@RequiredArgsConstructor(staticName = "of")
class Products implements Streamable<Product> {          2

  private final Streamable<Product> streamable;

  public MonetaryAmount getTotal() {                     3
    return streamable.stream()
      .map(Priced::getPrice)
      .reduce(Money.of(0), MonetaryAmount::add);
  }


  @Override
  public Iterator<Product> iterator() {                  4
    return streamable.iterator();
  }
}

interface ProductRepository implements Repository<Product, Long> {
  Products findAllByDescriptionContaining(String text);  5
}
```

1 A `Product` entity that exposes API to access the product's price.

2 A wrapper type for a `Streamable<Product>` that can be constructed by using `Products.of(…)` (factory method created with the Lombok annotation). A standard constructor taking the `Streamable<Product>` will do as well.

3 The wrapper type exposes an additional API, calculating new values on the `Streamable<Product>`.

4 Implement the `Streamable` interface and delegate to the actual result.

5 That wrapper type `Products` can be used directly as a query method return type. You do not need to return `Streamable<Product>` and manually wrap it after the query in the repository client.

Support for Vavr Collections

Vavr is a library that embraces functional programming concepts in Java. It ships with a custom set of collection types that you can use as query method return types, as the following table shows:

Vavr collection type	Used Vavr implementation type	Valid Java source types
`io.vavr.collection.Seq`	`io.vavr.collection.List`	`java.util.Iterable`
`io.vavr.collection.Set`	`io.vavr.collection.LinkedHashSet`	`java.util.Iterable`
`io.vavr.collection.Map`	`io.vavr.collection.LinkedHashMap`	`java.util.Map`

You can use the types in the first column (or subtypes thereof) as query method return types and get the types in the second column used as implementation type, depending on the Java type of the actual query result (third column). Alternatively, you can declare `Traversable` (the Vavr `Iterable` equivalent), and we then derive the implementation class from the actual return value. That is, a `java.util.List` is turned into a Vavr `List` or `Seq`, a `java.util.Set` becomes a Vavr `LinkedHashSet` `Set`, and so on.

8.4.7. 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`

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, respectively, neither to accept nor to 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 used 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 20. Declaring Non-nullability in `package-info.java`

```
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 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 in which the repository resides). 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 21. Using different nullability constraints

```java
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 does not produce a result.
Throws an `IllegalArgumentException` when the `emailAddress` handed to the method is `null`.

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

4 Returns `Optional.empty()` when the query 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 22. Using nullability constraints on Kotlin repositories

```kotlin
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 yields an empty result, an `EmptyResultDataAccessException` is thrown.

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

8.4.8. Streaming Query Results

You can process the results of query methods incrementally by using a Java 8 `Stream<T>` as the 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 23. Stream the result of a query with Java 8 `Stream<T>`

```java
@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 24. Working with a `Stream<T>` result in a `try-with-resources` block

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

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

8.4.9. Asynchronous Query Results

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

```java
@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.

8.5. Creating Repository Instances

This section covers how to 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.

8.5.1. XML Configuration

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

Example 25. Enabling Spring Data repositories via XML

```xml
<?xml version="1.0" encoding="UTF-8"?>
<beans:beans xmlns:beans="http://www.springframework.org/schema/beans"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns="http://www.springframework.org/schema/data/jpa"
  xsi:schemaLocation="http://www.springframework.org/schema/beans
    https://www.springframework.org/schema/beans/spring-beans.xsd
    http://www.springframework.org/schema/data/jpa
    https://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`. Bean names for nested repository interfaces are prefixed with their enclosing type name. 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 that extends 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 26. Using exclude-filter element

```xml
<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.

8.5.2. Java Configuration

You can also trigger the repository infrastructure by using a store-specific `@Enable${store}Repositories` annotation on a Java configuration class. For an introduction to 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 27. Sample annotation-based repository configuration

```java
@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.

8.5.3. 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 with a persistence technology-specific `RepositoryFactory` that you can use, as follows:

Example 28. Standalone usage of the repository factory

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

JAVA

8.6. 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, you need 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.

8.6.1. 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 follows:

Example 29. Interface for custom repository functionality

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

Example 30. Implementation of custom repository functionality

```java
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.

Then you can let your repository interface extend the fragment interface, as follows:

Example 31. Changes to your repository interface

```java
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 implementations. 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 32. Fragments with their implementations

Explain JAVA

```
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 33. Changes to your repository interface

```java
interface UserRepository extends CrudRepository<User, Long>, HumanRepository, ContactRepos

  // 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 34. Fragments overriding `save(…)`

```java
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 35. Customized repository interfaces

```java
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 36. Configuration example

```xml
<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 look up `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 37. Resolution of ambiguous implementations

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

class CustomizedUserRepositoryImpl implements CustomizedUserRepository {

  // Your custom implementation
}
```

```java
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 38. Manual wiring of custom implementations

```xml
```

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

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

8.6.2. 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 repositor affected. To instead change behavior for all repositories, you can create an implementatio 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 39. Custom repository base class

JAVA

```
class MyRepositoryImpl<T, ID>
  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 40. 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 41. Configuring a custom repository base class using XML

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

8.7. 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 42. Exposing domain events from an aggregate root

```
Explain JAVA
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 that uses `@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`.
You can use it 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(…)`, `saveAll(…)`, `delete(…)` or `deleteAll(…)` methods are called.

8.8. 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.

8.8.1. 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 the following example shows:

Example 43. QuerydslPredicateExecutor interface

```
JAVA
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 use the Querydsl support, extend `QuerydslPredicateExecutor` on your repository interface, as the following example shows:

Example 44. Querydsl integration on repositories

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

The preceding example lets you write type-safe queries by using Querydsl `Predicate` instances, as the following example shows:

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

userRepository.findAll(predicate);
```

8.8.2. Web support

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 the following example shows:

Example 45. Enabling Spring Data web support

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

The `@EnableSpringDataWebSupport` annotation registers a few components. We discuss the later in this section. It also detects Spring HATEOAS on the classpath and registers integration components (if present) for it as well.

Alternatively, if you use XML configuration, register either `SpringDataWebConfiguration` or `HateoasAwareSpringDataWebConfiguration` as Spring beans, as the following example shows (for `SpringDataWebConfiguration`):

Example 46. Enabling Spring Data web support in XML

```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 Using the `DomainClassConverter` Class 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.
- Jackson Modules to de-/serialize types like `Point` and `Distance`, or store specific ones, depending on the Spring Data Module used.

Using the `DomainClassConverter` Class

The `DomainClassConverter` class 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 the following example shows:

Example 47. A Spring MVC controller using domain types in method signatures

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

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

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

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 the following example shows:

Example 48. Using Pageable as a controller method argument

```java
@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)(,IgnoreCase)`. The default sort direction is case-sensitive ascending. Use multiple `sort` parameters if you want to switch direction or case sensitivity — for example, `?sort=firstname&sort=lastname,asc&sort=city,ignorecase`.

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

```java
@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 mu 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 following example shows the resulting method signature:

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

You have to populate `thing1_page` , `thing2_page` , and so on.

The default `Pageable` passed into the method is equivalent to a `PageRequest.of(0, 20)` , but you can customize it 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 49. Using a PagedResourcesAssembler as controller method argument

```java
```

```
@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:

JAVASCRIPT Explain

```
{ "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
```

```
    }
  }
```

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 you can customize that by passing a custom `Link` to be used as base to build the pagination links, which overloads the `PagedResourcesAssembler.toResource(…)` method.

Spring Data Jackson Modules

The core module, and some of the store specific ones, ship with a set of Jackson Modules for types, like `org.springframework.data.geo.Distance` and `org.springframework.data.geo.Point`, used by the Spring Data domain.
Those Modules are imported once web support is enabled and `com.fasterxml.jackson.databind.ObjectMapper` is available.

During initialization `SpringDataJacksonModules`, like the `SpringDataJacksonConfiguration`, get picked up by the infrastructure, so that the declared `com.fasterxml.jackson.databind.Module`s are made available to the Jackson `ObjectMapper`.

Data binding mixins for the following domain types are registered by the common infrastructure.

```
org.springframework.data.geo.Distance
org.springframework.data.geo.Point
org.springframework.data.geo.Box
org.springframework.data.geo.Circle
org.springframework.data.geo.Polygon
```

The individual module may provide additional `SpringDataJacksonModules`.
Please refer to the store specific section for more details.

Web Databinding Support

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

Example 50. HTTP payload binding using JSONPath or XPath expressions

```java
@ProjectedPayload
public interface UserPayload {

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

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

You can use the type shown in the preceding example as a Spring MVC handler method argument or by using `ParameterizedTypeReference` on one of methods of the `RestTemplate`. 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 that have QueryDSL integration, you can derive queries from the attributes contained in a `Request` query string.

Consider the following query string:

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

TEXT

Given the `User` object from the previous examples, you can resolve a query string to the following value by using the `QuerydslPredicateArgumentResolver`, as follows:

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

TEXT

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 you can 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 example shows how to use `@QuerydslPredicate` in a method signature:

Explain

```
@Controller
class UserController {
```

JAVA

```
  @Autowired UserRepository repository;

  @RequestMapping(value = "/", method = RequestMethod.GET)
  String index(Model model, @QuerydslPredicate(root = User.class) Predicate predicate,
          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`.

You can customize those bindings through the `bindings` attribute of `@QuerydslPredicate` or by making use of Java 8 `default methods` and adding the `QuerydslBinderCustomizer` method to the repository interface, as follows:

```
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.

8.8.3. Repository Populators

If you work with the Spring JDBC module, you are probably familiar with the support for populating 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 called `data.json` with the following content:

Example 51. Data defined in JSON

```javascript
[ { "_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 52. Declaring a Jackson repository populator

```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"
  xsi:schemaLocation="http://www.springframework.org/schema/beans
    https://www.springframework.org/schema/beans/spring-beans.xsd
    http://www.springframework.org/schema/data/repository
    https://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 Jacks… `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 unmarshall a repository populator with JAXB:

Example 53. Declaring an unmarshalling repository populator (using JAXB)

Explain 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
    https://www.springframework.org/schema/beans/spring-beans.xsd
    http://www.springframework.org/schema/data/repository
    https://www.springframework.org/schema/data/repository/spring-repository.xsd
    http://www.springframework.org/schema/oxm
    https://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

9. JDBC Repositories

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

9.1. 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 run. 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.

9.2. 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.

9.3. Getting Started

An easy way to bootstrap setting up a working environment is to create a Spring-based project in STS or from Spring Initializr.

First, you need to set up a running database server. Refer to your vendor documentation on how to configure your database for JDBC access.

To create a Spring project in STS:

1. Go to File → New → Spring Template Project → Simple Spring Utility Project, and press Yes when prompted. Then enter a project and a package name, such as `org.spring.jdbc.example` .
2. Add the following to the `pom.xml` files `dependencies` element:

```xml
<dependencies>

  <!-- other dependency elements omitted -->

  <dependency>
    <groupId>org.springframework.data</groupId>
    <artifactId>spring-data-jdbc</artifactId>
    <version>2.1.15</version>
  </dependency>

</dependencies>
```

3. Change the version of Spring in the pom.xml to be

```xml
<spring.framework.version>5.3.13</spring.framework.version>
```

4. Add the following location of the Spring Milestone repository for Maven to your `pom.xml` such that it is at the same level of your `<dependencies/>` element:

```xml
<repositories>
  <repository>
    <id>spring-milestone</id>
    <name>Spring Maven MILESTONE Repository</name>
    <url>https://repo.spring.io/libs-milestone</url>
  </repository>
</repositories>
```

The repository is also browseable here.

9.4. Examples Repository

There is a GitHub repository with several examples that you can download and play around with to get a feel for how the library works.

9.5. Annotation-based Configuration

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

Example 54. Spring Data JDBC repositories using Java configuration

Explain JAVA

```
@Configuration
@EnableJdbcRepositories                                                    1
class ApplicationConfig extends AbstractJdbcConfiguration {                2

    @Bean
    public DataSource dataSource() {                                       3

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

    @Bean
    NamedParameterJdbcOperations namedParameterJdbcOperations(DataSource dataSource) {  4
        return new NamedParameterJdbcTemplate(dataSource);
    }

    @Bean
    TransactionManager transactionManager(DataSource dataSource) {         5
        return new DataSourceTransactionManager(dataSource);
    }
}
```

1 `@EnableJdbcRepositories` creates implementations for interfaces derived from `Repository`

2 `AbstractJdbcConfiguration` provides various default beans required by Spring Data JDBC

3. Creates a `DataSource` connecting to a database.
   This is required by the following two bean methods.
4. Creates the `NamedParameterJdbcOperations` used by Spring Data JDBC to access the database.
5. Spring Data JDBC utilizes the transaction management provided by Spring JDBC.

The configuration class in the preceding example sets up an embedded HSQL database by using the `EmbeddedDatabaseBuilder` API of `spring-jdbc`. The `DataSource` is then used to set up `NamedParameterJdbcOperations` and a `TransactionManager`. We finally 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. Extending `AbstractJdbcConfiguration` ensures various beans get registered. Overwriting its methods can be used to customize the setup (see below).

This configuration can be further simplified by using Spring Boot. With Spring Boot a `DataSource` is sufficient once the starter `spring-boot-starter-data-jdbc` is included in the dependencies. Everything else is done by Spring Boot.

There are a couple of things one might want to customize in this setup.

9.5.1. Dialects

Spring Data JDBC uses implementations of the interface `Dialect` to encapsulate behavior that is specific to a database or its JDBC driver. By default, the `AbstractJdbcConfiguration` tries to determine the database in use and register the correct `Dialect`. This behavior can be changed by overwriting `jdbcDialect(NamedParameterJdbcOperations)`.

If you use a database for which no dialect is available, then your application won't startup. In that case, you'll have to ask your vendor to provide a `Dialect` implementation. Alternatively, you can:

1. Implement your own `Dialect`.
2. Implement a `JdbcDialectProvider` returning the `Dialect`.
3. Register the provider by creating a `spring.factories` resource under `META-INF` and perform the registration by adding a line `org.springframework.data.jdbc.repository.config.DialectResolver$JdbcDialectProvider=<fully qualified name of your JdbcDialectProvider>`

9.6. 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.

9.6.1. 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 is a single constructor, it is used.
2. If there are multiple constructors and exactly one is annotated with `@PersistenceConstructor`, it is used.
3. If there's a no-argument constructor, it is used. Other constructors will be ignored.

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:

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

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

```java
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 `with…` method (see below), we use the `with…` method 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. If the property is mutable we set the field directly.
4. If the property is immutable we're using the constructor to be used by persistence operations (see Object creation) to create a copy of the instance.
5. 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.

```java
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 55. 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 56. A sample entity

```
JAVA
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, this.age);
  }

  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.
The wither method is optional as the persistence constructor (see 6) is effectively a copy constructor and setting the property will be translated into creating a fresh instance with the new identifier value applied.

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 `with…` method 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 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 an all-arguments persistence constructor (preferred) or a `with…` 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` :

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

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

```kotlin
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`

```kotlin
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`:

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

This class is effectively immutable. It allows creating 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.

9.6.2. 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`
- Arrays and Collections of the types mentioned above can be mapped to columns of array type if your database supports that.
- Anything your database driver accepts.
- References to other entities. They are considered a one-to-one relationship, or an embedded type. It is optional for one-to-one relationship 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(PersistentPropertyPathExtension path)`. Embedded entities do not need an `id`. If one is present it gets ignored.
- `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(PersistentPropertyPathExtension path)`.
- `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(PersistentPropertyPathExtension path)` and `NamingStrategy.getKeyColumn(RelationalPersistentProperty property)`, respectively. Alternatively you may annotate the attribute with `@MappedCollection(idColumn="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.

9.6.3. Custom converters

Custom converters can be registered, for types that are not supported by default, by inheriting your configuration from `AbstractJdbcConfiguration` and overwriting the method `jdbcCustomConversions()`.

JAVA

```
@Configuration
public class DataJdbcConfiguration extends AbstractJdbcConfiguration {

    @Override
    public JdbcCustomConversions jdbcCustomConversions() {

      return new JdbcCustomConversions(Collections.singletonList(TimestampTzToDateConverter

    }

    @ReadingConverter
    enum TimestampTzToDateConverter implements Converter<TIMESTAMPTZ, Date> {

        INSTANCE;

        @Override
        public Date convert(TIMESTAMPTZ source) {
            //...
        }
```

```
    }
}
```

The constructor of `JdbcCustomConversions` accepts a list of `org.springframework.core.convert.converter.Converter`.

Converters should be annotated with `@ReadingConverter` or `@WritingConverter` in order to control their applicability to only reading from or to writing to the database.

`TIMESTAMPTZ` in the example is a database specific data type that needs conversion into something more suitable for a domain model.

JdbcValue

Value conversion uses `JdbcValue` to enrich values propagated to JDBC operations with a `java.sql.Types` type. Register a custom write converter if you need to specify a JDBC-specific type instead of using type derivation. This converter should convert the value to `JdbcValue` which has a field for the value and for the actual `JDBCType`.

9.6.4. `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.

9.6.5. `Custom table names`

When the NamingStrategy does not matching on your database table names, you can customize the names with the `@Table` annotation. The element `value` of this annotation provides the custom table name. The following example maps the `MyEntity` class to the `CUSTOM_TABLE_NAME` table in the database:

JAVA

```
@Table("CUSTOM_TABLE_NAME")
public class MyEntity {
    @Id
    Integer id;
```

```
    String name;
}
```

9.6.6. Custom column names

When the NamingStrategy does not matching on your database column names, you can customize the names with the `@Column` annotation. The element `value` of this annotation provides the custom column name. The following example maps the `name` property of the `MyEntity` class to the `CUSTOM_COLUMN_NAME` column in the database:

```java
public class MyEntity {
    @Id
    Integer id;

    @Column("CUSTOM_COLUMN_NAME")
    String name;
}
```

The `@MappedCollection` annotation can be used on a reference type (one-to-one relationship) or on Sets, Lists, and Maps (one-to-many relationship). `idColumn` element of the annotation provides a custom name for the foreign key column referencing the id column in the other table. In the following example the corresponding table for the `MySubEntity` class has a `NAME` column, and the `CUSTOM_MY_ENTITY_ID_COLUMN_NAME` column of the `MyEntity` id for relationship reasons:

```java
public class MyEntity {
    @Id
    Integer id;

    @MappedCollection(idColumn = "CUSTOM_MY_ENTITY_ID_COLUMN_NAME")
    Set<MySubEntity> subEntities;
}

public class MySubEntity {
```

```
    String name;
}
```

When using `List` and `Map` you must have an additional column for the position of a dataset in the `List` or the key value of the entity in the `Map`. This additional column name may be customized with the `keyColumn` Element of the `@MappedCollection` annotation:

```java
public class MyEntity {
    @Id
    Integer id;

    @MappedCollection(idColumn = "CUSTOM_COLUMN_NAME", keyColumn = "CUSTOM_KEY_COLUMN_NAME"
    List<MySubEntity> name;
}

public class MySubEntity {
    String name;
}
```

9.6.7. Embedded entities

Embedded entities are used to have value objects in your java data model, even if there is only one table in your database. In the following example you see, that `MyEntity` is mapped with the `@Embedded` annotation. The consequence of this is, that in the database a table `my_entity` with the two columns `id` and `name` (from the `EmbeddedEntity` class) is expected.

However, if the `name` column is actually `null` within the result set, the entire property `embeddedEntity` will be set to null according to the `onEmpty` of `@Embedded`, which `null`s objects when all nested properties are `null`.
Opposite to this behavior `USE_EMPTY` tries to create a new instance using either a default constructor or one that accepts nullable parameter values from the result set.

Example 57. Sample Code of embedding objects

```java
```

```
public class MyEntity {

    @Id
    Integer id;

    @Embedded(onEmpty = USE_NULL)  1
    EmbeddedEntity embeddedEntity;
}

public class EmbeddedEntity {
    String name;
}
```

1 Null s `embeddedEntity` if `name` in `null` .
Use `USE_EMPTY` to instantiate `embeddedEntity` with a potential `null` value for the `name` property.

If you need a value object multiple times in an entity, this can be achieved with the optional `prefix` element of the `@Embedded` annotation. This element represents a prefix and is prepend for each column name in the embedded object.

Make use of the shortcuts `@Embedded.Nullable` & `@Embedded.Empty` for `@Embedded(onEmpty = USE_NULL)` and `@Embedded(onEmpty = USE_EMPTY)` to reduce verbosity and simultaneously set JSR-305 `@javax.annotation.Nonnull` accordingly.

Explain JAVA

```
public class MyEntity {

    @Id
    Integer id;

    @Embedded.Nullable  1
    EmbeddedEntity embeddedEntity;
}
```

1 Shortcut for `@Embedded(onEmpty = USE_NULL)` .

Embedded entities containing a `Collection` or a `Map` will always be considered non empty since they will at least contain the empty collection or map. Such an entity will therefore never be `null` even when using @Embedded(onEmpty = USE_NULL).

9.6.8. Entity State Detection Strategies

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

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

`@Id` -Property inspection (the default)	By default, Spring Data inspects the identifier property of the given entity. If the identifier property is `null` or `0` in case of primitive types, then the entity is assumed to be new. Otherwise, it is assumed to not be new.
`@Version` -Property inspection	If a property annotated with `@Version` is present and `null` , or in case of a version property of primitive type `0` the entity is considered new. If the version property is present but has a different value, the entity is considered to not be new. If no version property is present Spring Data falls back to inspection of the identifier property.
Implementing `Persistable`	If an entity implements `Persistable` , Spring Data delegates the new detection to the `isNew(…)` method of the entity. See the Javadoc for details. *Note: Properties of `Persistable` will get detected and persisted if you use `AccessType.PROPERTY` . To avoid that, use `@Transient` .*
Providing a custom `EntityInformation` implementation	You can customize the `EntityInformation` abstraction used in the repository base implementation by creating a subclass of the module specific repository factory and overriding the `getEntityInformation(…)` method. You then have to register the custom implementation of module specific repository factory as a Spring bean. Note that this should rarely be necessary.

9.6.9. 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).

9.6.10. Optimistic Locking

Spring Data JDBC supports optimistic locking by means of a numeric attribute that is annotated with `@Version` on the aggregate root. Whenever Spring Data JDBC saves an aggregate with such a version attribute two things happen: The update statement for the aggregate root will contain a where clause checking that the version stored in the database is actually unchanged. If this isn't the case an `OptimisticLockingFailureException` will be thrown. Also the version attribute gets increased both in the entity and in the database so a concurrent action will notice the change and throw an `OptimisticLockingFailureException` if applicable as described above.

This process also applies to inserting new aggregates, where a `null` or `0` version indicates a new instance and the increased instance afterwards marks the instance as not new anymore, making this work rather nicely with cases where the id is generated during object construction for example when UUIDs are used.

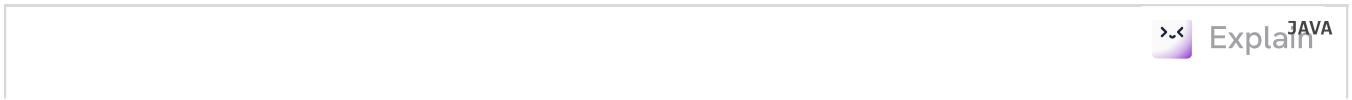
During deletes the version check also applies but no version is increased.

9.7. Query Methods

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

Most of the data access operations you usually trigger on a repository result in a query being run against the databases. Defining such a query is a matter of declaring a method on the repository interface, as the following example shows:

Example 58. PersonRepository with query methods



```
interface PersonRepository extends PagingAndSortingRepository<Person, String> {

  List<Person> findByFirstname(String firstname);                          1

  List<Person> findByFirstnameOrderByLastname(String firstname, Pageable pageable);  2

  Person findByFirstnameAndLastname(String firstname, String lastname);    3

  Person findFirstByLastname(String lastname);                             4

  @Query("SELECT * FROM person WHERE lastname = :lastname")
  List<Person> findByLastname(String lastname);                            5
}
```

1 The method shows a query for all people with the given `lastname`.
The query is derived by parsing the method name for constraints that can be concatenated with `And` and `Or`.
Thus, the method name results in a query expression of `SELECT … FROM person WHERE firstname = :firstname`.

2 Use `Pageable` to pass offset and sorting parameters to the database.

3 Find a single entity for the given criteria.
It completes with `IncorrectResultSizeDataAccessException` on non-unique results.

4 In contrast to <3>, the first entity is always emitted even if the query yields more result documents.

5 The `findByLastname` method shows a query for all people with the given last name.

The following table shows the keywords that are supported for query methods:

Table 3. Supported keywords for query methods

Keyword	Sample	Logical result
`After`	`findByBirthdateAfter(Date date)`	`birthdate > date`
`GreaterThan`	`findByAgeGreaterThan(int age)`	`age > age`
`GreaterThanE`	`findByAgeGreaterThanEqual(in`	`age >= age`

Keyword	Sample	Logical result
qual	t age)	
Before	findByBirthdateBefore(Date date)	birthdate < date
LessThan	findByAgeLessThan(int age)	age < age
LessThanEqual	findByAgeLessThanEqual(int age)	age <= age
Between	findByAgeBetween(int from, int to)	age BETWEEN from AND to
NotBetween	findByAgeNotBetween(int from, int to)	age NOT BETWEEN from AND to
In	findByAgeIn(Collection<Integer> ages)	age IN (age1, age2, ageN)
NotIn	findByAgeNotIn(Collection ages)	age NOT IN (age1, age2, ageN)
IsNotNull , NotNull	findByFirstnameNotNull()	firstname IS NOT NULL
IsNull , Null	findByFirstnameNull()	firstname IS NULL
Like , StartingWith , EndingWith	findByFirstnameLike(String name)	firstname LIKE name
NotLike , IsNotLike	findByFirstnameNotLike(String name)	firstname NOT LIKE name
Containing on String	findByFirstnameContaining(String name)	firstname LIKE '%' name +'%'

Keyword	Sample	Logical result
`NotContaining` on String	`findByFirstnameNotContaining(String name)`	`firstname NOT LIKE '%' name +'%'`
(No keyword)	`findByFirstname(String name)`	`firstname = name`
`Not`	`findByFirstnameNot(String name)`	`firstname != name`
`IsTrue`, `True`	`findByActiveIsTrue()`	`active IS TRUE`
`IsFalse`, `False`	`findByActiveIsFalse()`	`active IS FALSE`

Query derivation is limited to properties that can be used in a `WHERE` clause without using joins.

9.7.1. Query Lookup Strategies

The JDBC module supports defining a query manually as a String in a `@Query` annotation or as named query in a property file.

Deriving a query from the name of the method is is currently limited to simple properties, that means properties present in the aggregate root directly. Also, only select queries are supported by this approach.

9.7.2. Using `@Query`

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

Example 59. 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);
}
```

For converting the query result into entities the same `RowMapper` is used by default as for the queries Spring Data JDBC generates itself. The query you provide must match the format the `RowMapper` expects. Columns for all properties that are used in the constructor of an entity must be provided. Columns for properties that get set via setter, wither or field access are optional. Properties that don't have a matching column in the result will not be set. The q used for populating the aggregate root, embedded entities and one-to-one relationships including arrays of primitive types which get stored and loaded as SQL-array-types. Separate queries are generated for maps, lists, sets and arrays of entities.

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.

9.7.3. Named Queries

If no query is given in an annotation as described in the previous section Spring Data JDBC will try to locate a named query. There are two ways how the name of the query can be determined. The default is to take the *domain class* of the query, i.e. the aggregate root of the repository, take its simple name and append the name of the method separated by a `.`. Alternatively the `@Query` annotation has a `name` attribute which can be used to specify the name of a query to be looked up.

Named queries are expected to be provided in the property file `META-INF/jdbc-named-queries.properties` on the classpath.

The location of that file may be changed by setting a value to `@EnableJdbcRepositories.namedQueriesLocation`.

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 `DefaultQueryMappingConfiguration`:

```java
@Bean
QueryMappingConfiguration rowMappers() {
  return new DefaultQueryMappingConfiguration()
    .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 `QueryMappingConfiguration` 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.

> Using a custom `RowMapper` through `QueryMappingConfiguration`, `@Query(rowMapperClass=…)`, or a custom `ResultSetExtractor` disables Entity Callbacks and Lifecycle Events as the result mapping can issue its own events/callbacks if needed.

Modifying Query

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

```java
@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)

9.8. MyBatis Integration

The CRUD operations and query methods can be delegated to MyBatis. This section describes how to configure Spring Data JDBC to integrate with MyBatis and which conventions to follow to hand over the running of the queries as well as the mapping to the library.

9.8.1. Configuration

The easiest way to properly plug MyBatis into Spring Data JDBC is by importing `MyBatisJdbcConfiguration` into you application configuration:

```java
@Configuration
@EnableJdbcRepositories
@Import(MyBatisJdbcConfiguration.class)
class Application {

  @Bean
  SqlSessionFactoryBean sqlSessionFactoryBean() {
    // Configure MyBatis here
  }
}
```

As you can see, all you need to declare is a `SqlSessionFactoryBean` as `MyBatisJdbcConfiguration` relies on a `SqlSession` bean to be available in the `ApplicationContext` eventually.

9.8.2. Usage conventions

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`**
`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`.

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

Name	Purpose	CrudRepository methods that might trigger this statement	Attributes available in the `MyBatisContext`
`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.
`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	All `find*` methods. If no query is defined for `findAllByPath`	`getId` : The ID of the entity referencing the entities to be loaded.

Name	Purpose	CrudRepository methods that might trigger this statement	Attributes available in the `MyBatisContext`
	referencing entity is used for the prefix. The referenced entities type is used as the suffix. *This method is deprecated. Use `findAllByPath` instead*		`getDomainType` : The type of the entity to load.
`findAllByPath-<propertyPath>`	Select a set of entities that is referenced by another entity via a property path.	All `find*` methods.	`getIdentifier` : The `Identifier` holding the id of the aggregate root plus the keys and list indexes of all path elements. `getDomainType` : The type of the entity to load.
`findAllSorted`	Select all aggregate roots, sorted	`findAll(Sort)` .	`getSort` : The sorting specification.
`findAllPaged`	Select a page of aggregate roots, optionally sorted	`findAll(Page)` .	`getPageable` : The paging specification.
`count`	Count the number of aggregate root of the type used as prefix	`count`	`getDomainType` : The type of aggregate roots to count.

9.9. Lifecycle Events

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

```java
@Bean
public ApplicationListener<BeforeSaveEvent<Object>> loggingSaves() {

    return event -> {

        Object entity = event.getEntity();
        LOG.info("{} is getting saved.", entity);
    };
}
```

If you want to handle events only for a specific domain type you may derive your listener from `AbstractRelationalEventListener` and overwrite one or more of the `onXxx` methods, where `Xxx` stands for an event type. Callback methods will only get invoked for events related to the domain type and their subtypes so you don't require further casting.

```java
public class PersonLoadListener extends AbstractRelationalEventListener<Person> {

    @Override
    protected void onAfterLoad(AfterLoadEvent<Person> personLoad) {
        LOG.info(personLoad.getEntity());
    }
}
```

The following table describes the available events:

Table 4. Available events

Event	When It Is Published

Event	When It Is Published
`BeforeDeleteEvent`	Before an aggregate root gets deleted.
`AfterDeleteEvent`	After an aggregate root gets deleted.
`BeforeConvertEvent`	Before an aggregate root gets converted into a plan for executing SQL statements, but after the decision was made if the aggregate is new or not, i.e. if an update or an insert is in order. This is the correct event if you want to set an id programmatically.
`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).
`AfterSaveEvent`	After an aggregate root gets saved (that is, inserted or updated).
`AfterLoadEvent`	After an aggregate root gets created from a database `ResultSet` and all its properties get set.

Lifecycle events depend on an `ApplicationEventMulticaster`, which in case of the `SimpleApplicationEventMulticaster` can be configured with a `TaskExecutor`, and therefore gives no guarantees when an Event is processed.

9.9.1. Store-specific EntityCallbacks

Spring Data JDBC uses the `EntityCallback` API for its auditing support and reacts on the following callbacks:

Table 5. Available Callbacks

`EntityCallback`	**When It Is Published**
`BeforeDeleteCallback`	Before an aggregate root gets deleted.
`AfterDeleteCallback`	After an aggregate root gets deleted.
`BeforeConvertCallback`	Before an aggregate root gets converted into a plan for executing SQL statements, but after the decision was made if the aggregate is new or not, i.e. if an update or an insert is in order. This is the correct callback if you want to set an id programmatically.
`BeforeSaveCallback`	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).
`AfterSaveCallback`	After an aggregate root gets saved (that is, inserted or updated).
`AfterLoadCallback`	After an aggregate root gets created from a database `ResultSet` and all its property get set.

9.10. Entity Callbacks

The Spring Data infrastructure provides hooks for modifying an entity before and after certain methods are invoked. Those so called `EntityCallback` instances provide a convenient way to check and potentially modify an entity in a callback fashioned style.
An `EntityCallback` looks pretty much like a specialized `ApplicationListener`. Some Spring Data modules publish store specific events (such as `BeforeSaveEvent`) that allow modifying the given entity. In some cases, such as when working with immutable types, these events can cause trouble. Also, event publishing relies on `ApplicationEventMulticaster`. If configuring that with an asynchronous `TaskExecutor` it can lead to unpredictable outcomes, as event processing can be forked onto a Thread.

Entity callbacks provide integration points with both synchronous and reactive APIs to guarantee in-order execution at well-defined checkpoints within the processing chain,

returning a potentially modified entity or an reactive wrapper type.
Entity callbacks are typically separated by API type. This separation means that a synchronous API considers only synchronous entity callbacks and a reactive implementation considers only reactive entity callbacks.

The Entity Callback API has been introduced with Spring Data Commons 2.2. It is the recommended way of applying entity modifications. Existing store specific `ApplicationEvents` are still published **before** the invoking potentially registered `EntityCallback` instances.

9.10.1. Implementing Entity Callbacks

An `EntityCallback` is directly associated with its domain type through its generic type argument. Each Spring Data module typically ships with a set of predefined `EntityCallback` interfaces covering the entity lifecycle.

Example 60. Anatomy of an `EntityCallback`

```java
@FunctionalInterface
public interface BeforeSaveCallback<T> extends EntityCallback<T> {

    /**
     * Entity callback method invoked before a domain object is saved.
     * Can return either the same or a modified instance.
     *
     * @return the domain object to be persisted.
     */
    T onBeforeSave(T entity <2>, String collection <3>);  1
}
```

1 `BeforeSaveCallback` specific method to be called before an entity is saved. Returns a potentially modifed instance.

2 The entity right before persisting.

3 A number of store specific arguments like the *collection* the entity is persisted to.

Example 61. Anatomy of a reactive `EntityCallback`

```java
@FunctionalInterface
public interface ReactiveBeforeSaveCallback<T> extends EntityCallback<T> {

    /**
     * Entity callback method invoked on subscription, before a domain object is saved.
     * The returned Publisher can emit either the same or a modified instance.
     *
     * @return Publisher emitting the domain object to be persisted.
     */
    Publisher<T> onBeforeSave(T entity <2>, String collection <3>);  1
}
```

1 `BeforeSaveCallback` specific method to be called on subscription, before an entity is saved. Emits a potentially modifed instance.

2 The entity right before persisting.

3 A number of store specific arguments like the *collection* the entity is persisted to.

Optional entity callback parameters are defined by the implementing Spring Data module and inferred from call site of `EntityCallback.callback()`.

Implement the interface suiting your application needs like shown in the example below:

Example 62. Example `BeforeSaveCallback`

```java
class DefaultingEntityCallback implements BeforeSaveCallback<Person>, Ordered {       2

    @Override
    public Object onBeforeSave(Person entity, String collection) {                   1

        if(collection == "user") {
            return // ...
        }
```

```
        return // ...
    }

    @Override
    public int getOrder() {
        return 100;                                                    2
    }
}
```

1 Callback implementation according to your requirements.

2 Potentially order the entity callback if multiple ones for the same domain type ex
Ordering follows lowest precedence.

9.10.2. Registering Entity Callbacks

`EntityCallback` beans are picked up by the store specific implementations in case they are registered in the `ApplicationContext` . Most template APIs already implement `ApplicationContextAware` and therefore have access to the `ApplicationContext`

The following example explains a collection of valid entity callback registrations:

Example 63. Example `EntityCallback` Bean registration

JAVA

```
@Order(1)                                                              1
@Component
class First implements BeforeSaveCallback<Person> {

    @Override
    public Person onBeforeSave(Person person) {
        return // ...
    }
}

@Component
class DefaultingEntityCallback implements BeforeSaveCallback<Person>,
                                                        Ordered {   2

    @Override
    public Object onBeforeSave(Person entity, String collection) {
        // ...
    }
```

```
    @Override
    public int getOrder() {
        return 100;                                                    2
    }
}

@Configuration
public class EntityCallbackConfiguration {

    @Bean
    BeforeSaveCallback<Person> unorderedLambdaReceiverCallback() {     3
        return (BeforeSaveCallback<Person>) it -> // ...
    }
}

@Component
class UserCallbacks implements BeforeConvertCallback<User>,
                                    BeforeSaveCallback<User> {     4

    @Override
    public Person onBeforeConvert(User user) {
        return // ...
    }

    @Override
    public Person onBeforeSave(User user) {
        return // ...
    }
}
```

1 `BeforeSaveCallback` receiving its order from the `@Order` annotation.

2 `BeforeSaveCallback` receiving its order via the `Ordered` interface implementation.

3 `BeforeSaveCallback` using a lambda expression. Unordered by default and invoked last. Note that callbacks implemented by a lambda expression do not expose typing information hence invoking these with a non-assignable entity affects the callback throughput. Use a `class` or `enum` to enable type filtering for the callback bean.

4 Combine multiple entity callback interfaces in a single implementation class.

9.11. Custom Conversions

Spring Data JDBC allows registration of custom converters to influence how values are mapped in the database. Currently, converters are only applied on property-level.

9.11.1. Writing a Property by Using a Registered Spring Converter

The following example shows an implementation of a `Converter` that converts from a `Boolean` object to a `String` value:

```java
import org.springframework.core.convert.converter.Converter;

@WritingConverter
public class BooleanToStringConverter implements Converter<Boolean, String> {

    @Override
    public String convert(Boolean source) {
        return source != null && source ? "T" : "F";
    }
}
```

There are a couple of things to notice here: `Boolean` and `String` are both simple types hence Spring Data requires a hint in which direction this converter should apply (reading or writing). By annotating this converter with `@WritingConverter` you instruct Spring Data to write every `Boolean` property as `String` in the database.

9.11.2. Reading by Using a Spring Converter

The following example shows an implementation of a `Converter` that converts from a `String` to a `Boolean` value:

```java
@ReadingConverter
public class StringToBooleanConverter implements Converter<String, Boolean> {

    @Override
    public Boolean convert(String source) {
        return source != null && source.equalsIgnoreCase("T") ? Boolean.TRUE : Boolean.FALSE;
    }
}
```

There are a couple of things to notice here: `String` and `Boolean` are both simple types hence Spring Data requires a hint in which direction this converter should apply (reading or writing).

By annotating this converter with `@ReadingConverter` you instruct Spring Data to convert every `String` value from the database that should be assigned to a `Boolean` property.

9.11.3. Registering Spring Converters with the `JdbcConverter`

```java
class MyJdbcConfiguration extends AbstractJdbcConfiguration {

    // …

    @Overwrite
    @Bean
    public JdbcCustomConversions jdbcCustomConversions() {
        return new JdbcCustomConversions(Arrays.asList(new BooleanToStringConverter(), new Strin
    }
}
```

The following example of a Spring `Converter` implementation converts from a `String` to a custom `Email` value object:

```java
@ReadingConverter
public class EmailReadConverter implements Converter<String, Email> {

  public Email convert(String source) {
    return Email.valueOf(source);
  }
}
```

If you write a `Converter` whose source and target type are native types, we cannot determine whether we should consider it as a reading or a writing converter. Registering the converter instance as both might lead to unwanted results. For example, a `Converter<String, Long>` is ambiguous, although it probably does not make sense to try to convert all `String` instances into `Long` instances when writing. To let you force the infrastructure to register a converter for only one way, we provide `@ReadingConverter` and `@WritingConverter` annotations to be used in the converter implementation.

Converters are subject to explicit registration as instances are not picked up from a classpath or container scan to avoid unwanted registration with a conversion service and the side effects

resulting from such a registration. Converters are registered with `CustomConversions` as the central facility that allows registration and querying for registered converters based on source- and target type.

`CustomConversions` ships with a pre-defined set of converter registrations:

- JSR-310 Converters for conversion between `java.time`, `java.util.Date` and `String` types.
- Deprecated: Joda Time Converters for conversion between `org.joda.time`, JSR-310, and `java.util.Date`.
- Deprecated: ThreeTenBackport Converters for conversion between `org.joda.time`, JSR-310 and `java.util.Date`.

Default converters for local temporal types (e.g. `LocalDateTime` to `java.util.Date`) rely on system-default timezone settings to convert between those types. You can override the default converter, by registering your own converter.

Converter Disambiguation

Generally, we inspect the `Converter` implementations for the source and target types they convert from and to. Depending on whether one of those is a type the underlying data access API can handle natively, we register the converter instance as a reading or a writing converter. The following examples show a writing- and a read converter (note the difference is in the order of the qualifiers on `Converter`):

```java
// Write converter as only the target type is one that can be handled natively
class MyConverter implements Converter<Person, String> { … }

// Read converter as only the source type is one that can be handled natively
class MyConverter implements Converter<String, Person> { … }
```

9.12. 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 run, activate logging for Spring's `NamedParameterJdbcTemplate` or MyBatis.

9.13. 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 64. Custom transaction configuration for CRUD

```java
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 run 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 65. Using a facade to define transactions for multiple repository calls

```java
@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.

9.13.1. Transactional Query Methods

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

Example 66. Using @Transactional at query methods

JAVA

```
@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 hint to the underlying JDBC driver for performance optimizations.

9.14. Auditing

9.14.1. 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. Additionally, auditing has to be enabled either through Annotation configuration or XML configuration to register the required infrastructure components. Please refer to the store-specific section for configuration samples.

> Applications that only track creation and modification dates do not need to specify an `AuditorAware`.

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 67. An audited entity

JAVA Explain

```
class Customer {

  @CreatedBy
  private User user;

  @CreatedDate
  private Instant createdDate;

  // … further properties omitted
}
```

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

Auditing metadata does not necessarily need to live in the root level entity but can be added to an embedded one (depending on the actual store in use), as shown in the snipped below.

Example 68. Audit metadata in embedded entity

```
class Customer {

  private AuditMetadata auditingMetadata;

  // … further properties omitted
}

class AuditMetadata {

  @CreatedBy
  private User user;

  @CreatedDate
  private Instant createdDate;

}
```

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.

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 69. Implementation of `AuditorAware` based on Spring Security

```java
class SpringSecurityAuditorAware implements AuditorAware<User> {

  @Override
  public Optional<User> getCurrentAuditor() {

    return Optional.ofNullable(SecurityContextHolder.getContext())
            .map(SecurityContext::getAuthentication)
            .filter(Authentication::isAuthenticated)
            .map(Authentication::getPrincipal)
            .map(User.class::cast);
  }
}
```

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.

ReactiveAuditorAware

When using reactive infrastructure you might want to make use of contextual information to provide `@CreatedBy` or `@LastModifiedBy` information. We provide an `ReactiveAuditorAware<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 reactive Spring Security's `Authentication` object:

Example 70. Implementation of `ReactiveAuditorAware` based on Spring Security

```
class SpringSecurityAuditorAware implements ReactiveAuditorAware<User> {

  @Override
  public Mono<User> getCurrentAuditor() {

    return ReactiveSecurityContextHolder.getContext()
                .map(SecurityContext::getAuthentication)
                .filter(Authentication::isAuthenticated)
                .map(Authentication::getPrincipal)
                .map(User.class::cast);
  }
}
```

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.

9.15. JDBC Auditing

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

Example 71. 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 se domain types. If you have multiple implementations registered in the `ApplicationContext` can select the one to be used by explicitly setting the `auditorAwareRef` attribute of `@EnableJdbcAuditing`.

Appendix

Appendix A: 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 https://en.wikipedia.org/wiki/Dependency_Injection.

JPA

Java Persistence API

Spring

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

Appendix B: 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 6. Attributes

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 C: Populators namespace reference

The <populator /> element

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

Table 7. Attributes

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

Appendix D: Repository query keywords

Supported query method subject keywords

The following table lists the subject keywords generally supported by the Spring Data repository query derivation mechanism to express the predicate. Consult the store-specific documentation for the exact list of supported keywords, because some keywords listed here might not be supported in a particular store.

Table 8. Query subject keywords

Keyword	Description
`find…By` , `read…By` , `get…By` , `query…By` , `search…By` , `stream…By`	General query method returning typically the repository type, a `Collection` or `Streamable` subtype or a result wrapper such as `Page` , `GeoResults` or any other store-specific result wrapper. Can be used as `findBy…` , `findMyDomainTypeBy…` or in combination with additional keywords.
`exists…By`	Exists projection, returning typically a `boolean` result.
`count…By`	Count projection returning a numeric result.

Keyword	Description
`delete…By , remove… By`	Delete query method returning either no result (`void`) or the delete count.
`…First<number>… , … Top<number>…`	Limit the query results to the first `<number>` of results. This keyword can occur in any place of the subject between `find` (and the other keywords) and `by` .
`…Distinct…`	Use a distinct query to return only unique results. Consult the store-specific documentation whether that feature is supported. This keyword can occur in any place of the subject between `find` (and the other keywords) and `by` .

Supported query method predicate keywords and modifiers

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

Table 9. Query predicate keywords

Logical keyword	Keyword expressions
`AND`	`And`
`OR`	`Or`
`AFTER`	`After , IsAfter`
`BEFORE`	`Before , IsBefore`
`CONTAINING`	`Containing , IsContaining , Contains`
`BETWEEN`	`Between , IsBetween`
`ENDING_WITH`	`EndingWith , IsEndingWith , EndsWith`

Logical keyword	Keyword expressions
EXISTS	Exists
FALSE	False , IsFalse
GREATER_THAN	GreaterThan , IsGreaterThan
GREATER_THAN_EQUALS	GreaterThanEqual , IsGreaterThanEqual
IN	In , IsIn
IS	Is , Equals , (or no keyword)
IS_EMPTY	IsEmpty , Empty
IS_NOT_EMPTY	IsNotEmpty , NotEmpty
IS_NOT_NULL	NotNull , IsNotNull
IS_NULL	Null , IsNull
LESS_THAN	LessThan , IsLessThan
LESS_THAN_EQUAL	LessThanEqual , IsLessThanEqual
LIKE	Like , IsLike
NEAR	Near , IsNear
NOT	Not , IsNot
NOT_IN	NotIn , IsNotIn
NOT_LIKE	NotLike , IsNotLike
REGEX	Regex , MatchesRegex , Matches
STARTING_WITH	StartingWith , IsStartingWith , StartsWith
TRUE	True , IsTrue
WITHIN	Within , IsWithin

In addition to filter predicates, the following list of modifiers is supported:

Table 10. Query predicate modifier keywords

Keyword	Description
`IgnoreCase`, `IgnoringCase`	Used with a predicate keyword for case-insensitive comparison.
`AllIgnoreCase`, `AllIgnoringCase`	Ignore case for all suitable properties. Used somewhere in the query method predicate.
`OrderBy…`	Specify a static sorting order followed by the property path and direction (e. g. `OrderByFirstnameAscLastnameDesc`).

Appendix E: 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. Some store modules may define their own result wrapper types.

Table 11. Query return types

Return type	Description
`void`	Denotes no return value.

Return type	Description
Primitives	Java primitives.
Wrapper types	Java wrapper types.
`T`	A 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`.
`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 Vavr `Option` type. Semantically the same behavior as Java 8's `Optional`, described earlier.
`Stream<T>`	A Java 8 `Stream`.
`Streamable<T>`	A convenience extension of `Iterable` that directy exposes methods to stream, map and filter results, concatenate them etc.
Types that implement `Streamable` and take a `Streamable` constructor or factory method argument	Types that expose a constructor or `….of(…)` / `….valueOf(…)` factory method taking a `Streamable` as argument. See Returning Custom Streamable Wrapper Types for details.
Vavr `Seq`, `List`, `Map`, `Set`	Vavr collection types. See Support for Vavr Collections for details.

Return type	Description
`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`. Expect method to be annotated with `@Async` and requires Spring's asynchronous method execution capability to be enabled.
`Slice<T>`	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.
`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.

Return type	Description
	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 2.1.15
Last updated 2021-11-12 09:59:26 +0100