Spring Data MongoDB - Reference Documentation

Mark Pollack, Thomas Risberg, Oliver Gierke, Costin Leau, Jon Brisbin, Thomas Darimont, Christoph Strobl, Mark Paluch

Version 2.0.2.RELEASE, 2017-11-27

Table of Contents

Preface

The Spring Data MongoDB project applies core Spring concepts to the development of solutions using the MongoDB document style data store. We provide a "template" as a high-level abstraction for storing and querying documents. You will notice similarities to the JDBC support in the Spring Framework.

This document is the reference guide for Spring Data - Document Support. It explains Document module concepts and semantics and the syntax for various store namespaces.

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

Chapter 1. Knowing Spring

Spring Data uses Spring framework's core functionality, such as the IoC container, type conversion system, expression language, JMX integration, and portable DAO exception hierarchy. While it is not important to know the Spring APIs, understanding the concepts behind them is. At a minimum, the idea behind IoC should be familiar for whatever IoC container you choose to use.

The core functionality of the MongoDB 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 MongoDB, such as the repository support, you will need to configure some parts of the library using Spring.

To learn more about Spring, you can refer to the comprehensive (and sometimes disarming) documentation that explains in detail the Spring Framework. There are a lot of articles, blog entries and books on the matter - take a look at the Spring framework home page for more information.

Chapter 2. Knowing NoSQL and Document databases

NoSQL stores have taken the storage world by storm. It is a vast domain with a plethora of solutions, terms and patterns (to make things worse even the term itself has multiple meanings). While some of the principles are common, it is crucial that the user is familiar to some degree with MongoDB. The best way to get acquainted to this solutions is to read their documentation and follow their examples - it usually doesn't take more then 5-10 minutes to go through them and if you are coming from an RDMBS-only background many times these exercises can be an eye opener.

The jumping off ground for learning about MongoDB is www.mongodb.org. Here is a list of other useful resources:

- The manual introduces MongoDB and contains links to getting started guides, reference documentation and tutorials.
- The online shell provides a convenient way to interact with a MongoDB instance in combination with the online tutorial.
- MongoDB Java Language Center
- Several books available for purchase
- Karl Seguin's online book: The Little MongoDB Book

Chapter 3. Requirements

Spring Data MongoDB 1.x binaries requires JDK level 6.0 and above, and Spring Framework 5.0.2.RELEASE and above.

In terms of document stores, MongoDB at least 2.6.

Chapter 4. Additional Help Resources

Learning a new framework is not always straight forward. In this section, we try to provide what we think is an easy to follow guide for starting with Spring Data MongoDB module. However, if you encounter issues or you are just looking for an advice, feel free to use one of the links below:

4.1. Support

There are a few support options available:

4.1.1. Community Forum

Spring Data on Stackoverflow Stackoverflow 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.

4.1.2. Professional Support

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

4.2. Following Development

For information on the Spring Data Mongo source code repository, nightly builds and snapshot artifacts please see the Spring Data Mongo homepage. You can help make Spring Data best serve the needs of the Spring community by interacting with developers through the Community on Stackoverflow. To follow developer activity look for the mailing list information on the Spring Data Mongo homepage. 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. Lastly, you can follow the Spring blog or the project team on Twitter (SpringData).

Chapter 5. New & Noteworthy

5.1. What's new in Spring Data MongoDB 2.0

- Upgrade to Java 8.
- Usage of the `Document` API instead of `DBObject`.
- Reactive MongoDB support.
- Tailable Cursor queries.
- Support for aggregation result streaming via Java 8 `Stream`.
- Fluent Collection API for CRUD and aggregation operations.
- Kotlin extensions for Template and Collection API.
- Integration of collations for collection and index creation and query operations.
- Query-by-Example support without type matching.
- Add support for isolation `Update`s.
- Tooling support for null-safety via Spring's `@NonNullApi` and `@Nullable` annotations.
- Deprecated cross-store support and removed Log4j appender.

5.2. What's new in Spring Data MongoDB 1.10

- Compatible with MongoDB Server 3.4 and the MongoDB Java Driver 3.4.
- New annotations for `@CountQuery`, `@DeleteQuery` and `@ExistsQuery`.
- Extended support for MongoDB 3.2 and MongoDB 3.4 aggregation operators (see Supported Aggregation Operations).
- Support partial filter expression when creating indexes.
- Publish lifecycle events when loading/converting `DBRef`s.
- Added any-match mode for Query By Example.
- Support for `$caseSensitive` and `$diacriticSensitive` text search.
- Support for GeoJSON Polygon with hole.
- Performance improvements by bulk fetching `DBRef`s.
- Multi-faceted aggregations using `$facet`, `$bucket` and `$bucketAuto` via `Aggregation`.

5.3. What's new in Spring Data MongoDB 1.9

- The following annotations have been enabled to build own, composed annotations: `@Document`, `@Id`, `@Field`, `@Indexed`, `@CompoundIndex`, `@GeoSpatialIndexed`, `@TextIndexed`, `@Query`, `@Meta`.
- Support for Projections in repository query methods.
- Support for Query by Example.

- Out-of-the-box support for `java.util.Currency` in object mapping.
- Add support for the bulk operations introduced in MongoDB 2.6.
- Upgrade to Querydsl 4.
- Assert compatibility with MongoDB 3.0 and MongoDB Java Driver 3.2 (see: MongoDB 3.0 Support).

5.4. What's new in Spring Data MongoDB 1.8

- `Criteria` offers support for creating `$geoIntersects`.
- Support SpEL expressions in `@Query`.
- `MongoMappingEvents` expose the collection name they are issued for.
- Improved support for `<mongo:mongo-client credentials="…" />`.
- Improved index creation failure error message.

5.5. What's new in Spring Data MongoDB 1.7

- Assert compatibility with MongoDB 3.0 and MongoDB Java Driver 3-beta3 (see: MongoDB 3.0 Support).
- Support JSR-310 and ThreeTen back-port date/time types.
- Allow `Stream` as query method return type (see: Query methods).
- Added GeoJSON support in both domain types and queries (see: GeoJSON Support).
- `QueryDslPredicateExcecutor` now supports `findAll(OrderSpecifier<?>… orders)`.
- Support calling JavaScript functions via Script Operations.
- Improve support for `CONTAINS` keyword on collection like properties.
- Support for `$bit`, `$mul` and `$position` operators to `Update`.

Chapter 6. Dependencies

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

Example 1. Using the Spring Data release train BOM

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

The current release train version is `Kay-SR2`. The train names are ascending alphabetically and currently available ones are listed here. The version name follows the following pattern: `${name}-${release}` where release can be one of the following:

- `BUILD-SNAPSHOT` - current snapshots
- `M1`, `M2` etc. - milestones
- `RC1`, `RC2` etc. - release candidates
- `RELEASE` - GA release
- `SR1`, `SR2` etc. - service releases

A working example of using the BOMs can be found in our Spring Data examples repository. If that's in place declare the Spring Data modules you'd like to use without a version in the `<dependencies />` block.

Example 2. Declaring a dependency to a Spring Data module

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

6.1. Dependency management with Spring Boot

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

6.2. Spring Framework

The current version of Spring Data modules require Spring Framework in version 5.0.2.RELEASE or better. The modules might also work with an older bugfix version of that minor version. However, using the most recent version within that generation is highly recommended.

Chapter 7. Working with Spring Data Repositories

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

IMPORTANT

Spring Data repository documentation and your module

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

7.1. Core concepts

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

Example 3. CrudRepository interface

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

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

  Optional<T> findById(ID primaryKey); ②

  Iterable<T> findAll();               ③

  long count();                        ④

  void delete(T entity);               ⑤

  boolean existsById(ID primaryKey);   ⑥

  // … more functionality omitted.
}
```

① Saves the given entity.

② Returns the entity identified by the given id.

③ Returns all entities.

④ Returns the number of entities.

⑤ Deletes the given entity.

⑥ Indicates whether an entity with the given id exists.

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

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

Example 4. PagingAndSortingRepository

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

  Iterable<T> findAll(Sort sort);

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

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

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

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

Example 5. Derived Count Query

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

  long countByLastname(String lastname);
}
```

Example 6. Derived Delete Query

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

  long deleteByLastname(String lastname);

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

7.2. Query methods

Standard CRUD functionality repositories usually have queries on the underlying datastore. With Spring Data, declaring those queries becomes a four-step process:

1. Declare an interface extending Repository or one of its subinterfaces and type it to the domain class and ID type that it will handle.

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

2. Declare query methods on the interface.

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

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

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

@EnableJpaRepositories
class Config {}
```

or via XML configuration:

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

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

</beans>
```

The JPA namespace is used in this example. If you are using the repository abstraction for any other store, you need to change this to the appropriate namespace declaration of your store module which should be exchanging `jpa` in favor of, for example, `mongodb`.

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

4. Get the repository instance injected and use it.

```
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.

7.3. Defining repository interfaces

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

7.3.1. Fine-tuning repository definition

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

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

Example 7. Selectively exposing CRUD methods

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

  Optional<T> findById(ID id);

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

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

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

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

7.3.2. Null handling of repository methods

As of Spring Data 2.0, repository CRUD methods that return an individual aggregate instance use

Java 8's `Optional` to indicate the potential absence of a value. Besides that, Spring Data supports to return other wrapper types on query methods:

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

Alternatively query methods can choose not to use a wrapper type at all. The absence of a query result will then be 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 using Spring Framework's nullability annotations. They provide a tooling-friendly approach and opt-in `null` checks during runtime:

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

Spring annotations are meta-annotated with JSR 305 annotations (a dormant but widely spread JSR). JSR 305 meta-annotations allow tooling vendors like IDEA, Eclipse, or Kotlin to 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 package level using Spring's `@NonNullApi` in `package-info.java`:

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

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

Once non-null defaulting is in place, repository query method invocations will get validated at runtime for nullability constraints. Exceptions will be thrown in case a query execution result violates the defined constraint, i.e. the method would return `null` for some reason but is declared as non-nullable (the default with the annotation defined on the package the repository resides in). If you want to opt-in to nullable results again, selectively use `@Nullable` that a method. Using the aforementioned result wrapper types will continue to work as expected, i.e. an empty result will be translated into the value representing absence.

Example 9. Using different nullability constraints

```
package com.acme;                                                 ①

import org.springframework.lang.Nullable;

interface UserRepository extends Repository<User, Long> {

  User getByEmailAddress(EmailAddress emailAddress);              ②

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

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

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

② Will throw an `EmptyResultDataAccessException` in case the query executed does not produce a result. Will throw an `IllegalArgumentException` in case the `emailAddress` handed to the method is `null`.

③ Will return `null` in case the query executed does not produce a result. Also accepts `null` as value for `emailAddress`.

④ Will return `Optional.empty()` in case the query executed does not produce a result. Will throw an `IllegalArgumentException` in case 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 using method signatures but rather 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:

Example 10. Using nullability constraints on Kotlin repositories

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

  fun findByUsername(username: String): User     ①

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

① The method defines both, the parameter as non-nullable (the Kotlin default) as well as the result. The Kotlin compiler will already reject method invocations trying to hand `null` into the method. In case the query execution yields an empty result, an `EmptyResultDataAccessException` will be thrown.

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

7.3.3. Using Repositories with multiple Spring Data modules

Using a unique Spring Data module in your application makes things simple hence, 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 case, it's required for a repository definition to distinguish between persistence technologies. Spring Data enters strict repository configuration mode because it detects multiple repository factories on the class path. Strict configuration requires details on the repository or the domain class to decide about Spring Data module binding for a repository definition:

1. If the repository definition extends the module-specific repository, then it's a valid candidate for the particular Spring Data module.
2. If the domain class is annotated with the module-specific type annotation, then it's a valid candidate for the particular Spring Data module. Spring Data modules accept either 3rd party annotations (such as JPA's `@Entity`) or provide own annotations such as `@Document` for Spring Data MongoDB/Spring Data Elasticsearch.

Example 11. Repository definitions using Module-specific Interfaces

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

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

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

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

Example 12. Repository definitions using generic Interfaces

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

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

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

`AmbiguousRepository` and `AmbiguousUserRepository` extend only `Repository` and `CrudRepository` in their type hierarchy. While this is perfectly fine using a unique Spring Data module, multiple modules cannot distinguish to which particular Spring Data these repositories should be bound.

Example 13. Repository definitions using Domain Classes with Annotations

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

@Entity
class Person {
  …
}

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

@Document
class User {
  …
}
```

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

Example 14. Repository definitions using Domain Classes with mixed Annotations

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

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

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

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

Repository type details and identifying domain class annotations are used for strict repository

configuration identify repository candidates for a particular Spring Data module. Using multiple persistence technology-specific annotations on the same domain type is possible to reuse domain types across multiple persistence technologies, but then Spring Data is no longer able to determine a unique module to bind the repository.

The last way to distinguish repositories is scoping repository base packages. Base packages define the starting points for scanning for repository interface definitions which implies to have 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.

Example 15. Annotation-driven configuration of base packages

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

7.4. Defining query methods

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

7.4.1. Query lookup strategies

The following strategies are available for the repository infrastructure to resolve the query. You can configure the strategy at the namespace through the `query-lookup-strategy` attribute in case of XML configuration or via the `queryLookupStrategy` attribute of the Enable${store}Repositories annotation in case of Java config. Some strategies may not be supported for particular datastores.

- `CREATE` attempts to construct a store-specific query from the query method name. The general approach is to remove a given set of well-known prefixes from the method name and parse the rest of the method. Read more about query construction in Query creation.
- `USE_DECLARED_QUERY` tries to find a declared query and will throw an exception in case it can't find one. The query can be defined by an annotation somewhere or declared by other means. Consult the documentation of the specific store to find available options for that store. If the repository infrastructure does not find a declared query for the method at bootstrap time, it fails.
- `CREATE_IF_NOT_FOUND` (default) combines `CREATE` and `USE_DECLARED_QUERY`. It looks up a declared query first, and if no declared query is found, it creates a custom method name-based query. This is the default lookup strategy and thus will be used if you do not configure anything explicitly. It allows quick query definition by method names but also custom-tuning of these queries by introducing declared queries as needed.

7.4.2. Query creation

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

Example 16. Query creation from method names

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

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

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

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

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

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

- The expressions are usually property traversals combined with operators that can be concatenated. You can combine property expressions with `AND` and `OR`. You also get support for operators such as `Between`, `LessThan`, `GreaterThan`, `Like` for the property expressions. The supported operators can vary by datastore, so consult the appropriate part of your reference documentation.
- The method parser supports setting an `IgnoreCase` flag for individual properties (for example, `findByLastnameIgnoreCase(…)`) or for all properties of a type that support ignoring case (usually `String` instances, for example, `findByLastnameAndFirstnameAllIgnoreCase(…)`). Whether ignoring cases is supported may vary by store, so consult the relevant sections in the reference documentation for the store-specific query method.

- You can apply static ordering by appending an `OrderBy` clause to the query method that references a property and by providing a sorting direction (`Asc` or `Desc`). To create a query method that supports dynamic sorting, see Special parameter handling.

7.4.3. Property expressions

Property expressions can refer only to a direct property of the managed entity, as shown in the preceding example. At query creation time you already make sure that the parsed property is a property of the managed domain class. However, you can also define constraints by traversing nested properties. Assume a `Person` has an `Address` with a `ZipCode`. In that case a method name of

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

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

Although this should work for most cases, it is possible for the algorithm to select the wrong property. Suppose the `Person` class has an `addressZip` property as well. The algorithm would match in the first split round already and essentially choose the wrong property and finally fail (as the type of `addressZip` probably has no `code` property).

To resolve this ambiguity you can use `_` inside your method name to manually define traversal points. So our method name would end up like so:

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

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

7.4.4. Special parameter handling

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

Example 17. Using Pageable, Slice and Sort in query methods

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

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

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

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

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

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

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

7.4.5. Limiting query results

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

Example 18. Limiting the result size of a query with `Top` and `First`

```
User findFirstByOrderByLastnameAsc();

User findTopByOrderByAgeDesc();

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

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

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

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

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

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

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

7.4.6. Streaming query results

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

Example 19. Stream the result of a query with Java 8 `Stream<T>`

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

Stream<User> readAllByFirstnameNotNull();

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

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

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

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

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

7.4.7. Async query results

Repository queries can be executed asynchronously using Spring's asynchronous method execution capability. This means the method will return immediately upon invocation and the actual query execution will occur in a task that has been submitted to a Spring TaskExecutor.

```
@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 return type.

② Use a Java 8 `java.util.concurrent.CompletableFuture` as return type.

③ Use a `org.springframework.util.concurrent.ListenableFuture` as return type.

7.5. Creating repository instances

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

7.5.1. XML configuration

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

Example 21. Enabling Spring Data repositories via XML

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

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

</beans:beans>
```

In the preceding example, Spring is instructed to scan `com.acme.repositories` and all its sub-packages for interfaces extending `Repository` or one of its sub-interfaces. For each interface found, the infrastructure registers the persistence technology-specific `FactoryBean` to create the appropriate proxies that handle invocations of the query methods. Each bean is registered under a bean name that is derived from the interface name, so an interface of `UserRepository` would be registered under `userRepository`. The `base-package` attribute allows wildcards, so that you can define a pattern of scanned packages.

Using filters

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

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

Example 22. Using exclude-filter element

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

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

7.5.2. JavaConfig

The repository infrastructure can also be triggered using a store-specific

`@Enable${store}Repositories` annotation on a JavaConfig class. For an introduction into Java-based configuration of the Spring container, see the reference documentation. [1: JavaConfig in the Spring reference documentation]

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

Example 23. Sample annotation based repository configuration

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

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

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

7.5.3. Standalone usage

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

Example 24. Standalone usage of repository factory

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

7.6. Custom implementations for Spring Data repositories

In this section you will learn about repository customization and how fragments form a composite repository.

When query method require a different behavior or can't be implemented by query derivation than it's necessary to provide a custom implementation. Spring Data repositories easily allow you to provide custom repository code and integrate it with generic CRUD abstraction and query

method functionality.

7.6.1. Customizing individual repositories

To enrich a repository with custom functionality, you first define a fragment interface and an implementation for the custom functionality. Then let your repository interface additionally extend from the fragment interface.

Example 25. Interface for custom repository functionality

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

Example 26. Implementation of custom repository functionality

```
class CustomizedUserRepositoryImpl implements CustomizedUserRepository {

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

NOTE	The most important bit for the class to be found is the `Impl` postfix of the name on it compared to the fragment interface.

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

Example 27. Changes to your repository interface

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

  // Declare query methods here
}
```

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

Spring Data repositories are implemented by using fragments that form a repository composition. Fragments are the base repository, functional aspects such as QueryDsl and custom interfaces along

with their implementation. Each time you add an interface to your repository interface, you enhance the composition by adding a fragment. The base repository and repository aspect implementations are provided by each Spring Data module.

Example 28. Fragments with their implementations

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

class HumanRepositoryImpl implements HumanRepository {

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

interface EmployeeRepository {

  void someEmployeeMethod(User user);

  User anotherEmployeeMethod(User user);
}

class ContactRepositoryImpl implements ContactRepository {

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

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

Example 29. Changes to your repository interface

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

  // Declare query methods here
}
```

Repositories may be composed of multiple custom implementations that are imported in the order of their declaration. Custom implementations have a higher priority than the base implementation and repository aspects. This ordering allows you to override base repository and aspect methods

and resolves ambiguity if two fragments contribute the same method signature. Repository fragments are not limited to be used in a single repository interface. Multiple repositories may use a fragment interface to reuse customizations across different repositories.

Example 30. Fragments overriding `save(…)`

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

class CustomizedSaveImpl<T> implements CustomizedSave<T> {

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

Example 31. Customized repository interfaces

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

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

Configuration

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

Example 32. Configuration example

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

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

The first configuration example will try to look up a class `com.acme.repository.CustomizedUserRepositoryImpl` to act as custom repository implementation,

whereas the second example will try to lookup `com.acme.repository.CustomizedUserRepositoryFooBar`.

Resolution of ambiguity

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

Given the following two custom implementations for the `CustomizedUserRepository` introduced above the first implementation will get picked. Its bean name is `customizedUserRepositoryImpl` matches that of the fragment interface (`CustomizedUserRepository`) plus the postfix `Impl`.

Example 33. Resolution of amibiguous implementations

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

class CustomizedUserRepositoryImpl implements CustomizedUserRepository {

  // Your custom implementation
}
```

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

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

  // Your custom implementation
}
```

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

Manual wiring

The approach just shown works well if your custom implementation uses annotation-based configuration and autowiring only, as it will be treated as any other Spring bean. If your implementation fragment bean needs special wiring, you simply declare the bean and name it after the conventions just described. The infrastructure will then refer to the manually defined bean definition by name instead of creating one itself.

Example 34. Manual wiring of custom implementations

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

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

7.6.2. Customize the base repository

The preceding approach requires customization of all repository interfaces when you want to customize the base repository behavior, so all repositories are affected. To change behavior for all repositories, you need to create an implementation that extends the persistence technology-specific repository base class. This class will then act as a custom base class for the repository proxies.

Example 35. Custom repository base class

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

  private final EntityManager entityManager;

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

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

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

WARNING The class needs to have a constructor of the super class which the store-specific repository factory implementation is using. In case the repository base class has multiple constructors, override the one taking an `EntityInformation` plus a store specific infrastructure object (e.g. an `EntityManager` or a template class).

The final step is to make the Spring Data infrastructure aware of the customized repository base class. In JavaConfig this is achieved by using the `repositoryBaseClass` attribute of the `@Enable…Repositories` annotation:

Example 36. Configuring a custom repository base class using JavaConfig

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

A corresponding attribute is available in the XML namespace.

Example 37. Configuring a custom repository base class using XML

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

7.7. 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 `@DomainEvents` you can use on a method of your aggregate root to make that publication as easy as possible.

Example 38. Exposing domain events from an aggregate root

```
class AnAggregateRoot {

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

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

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

② After all events have been published, a method annotated with `@AfterDomainEventPublication`. It e.g. can be used to potentially clean the list of events to be published.

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

7.8. 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.

7.8.1. Querydsl Extension

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

Several Spring Data modules offer integration with Querydsl via `QueryDslPredicateExecutor`.

Example 39. QueryDslPredicateExecutor interface

```
public interface QueryDslPredicateExecutor<T> {

  Optional<T> findById(Predicate predicate);  ①

  Iterable<T> findAll(Predicate predicate);   ②

  long count(Predicate predicate);            ③

  boolean exists(Predicate predicate);        ④

  // … more functionality omitted.
}
```

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

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

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

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

To make use of Querydsl support simply extend `QueryDslPredicateExecutor` on your repository interface.

Example 40. Querydsl integration on repositories

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

}
```

The above enables to write typesafe queries using Querydsl `Predicate` s.

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

userRepository.findAll(predicate);
```

7.8.2. Web support

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

Spring Data modules ships with a variety of web support if the module supports the repository programming model. The web related stuff requires Spring MVC JARs on the classpath, some of them even provide integration with Spring HATEOAS [2: Spring HATEOAS - https://github.com/SpringSource/spring-hateoas]. In general, the integration support is enabled by using the `@EnableSpringDataWebSupport` annotation in your JavaConfig configuration class.

Example 41. Enabling Spring Data web support

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

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

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

Example 42. Enabling Spring Data web support in XML

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

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

Basic web support

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

- A `DomainClassConverter` to enable Spring MVC to resolve instances of repository managed domain classes from request parameters or path variables.
- `HandlerMethodArgumentResolver` implementations to let Spring MVC resolve Pageable and Sort instances from request parameters.

DomainClassConverter

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

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

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

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

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

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

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

HandlerMethodArgumentResolvers for Pageable and Sort

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

Example 44. Using Pageable as controller method argument

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

  private final UserRepository repository;

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

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

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

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

Table 1. Request parameters evaluated for Pageable instances

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

To customize this behavior register a bean implementing the interface `PageableHandlerMethodArgumentResolverCustomizer` or `SortHandlerMethodArgumentResolverCustomizer` respectively. It's `customize()` method will get called allowing you to change settings. Like in the following example.

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

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

In case you need multiple `Pageable` or `Sort` instances to be resolved from the request (for multiple tables, for example) you can use Spring's `@Qualifier` annotation to distinguish one from another.

The request parameters then have to be prefixed with `${qualifier}_`. So for a method signature like this:

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

you have to populate `foo_page` and `bar_page` etc.

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

Example 45. Using a PagedResourcesAssembler as controller method argument

```
@Controller
class PersonController {

  @Autowired PersonRepository repository;

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

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

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

- The content of the `Page` will become the content of the `PagedResources` instance.
- The `PagedResources` will get a `PageMetadata` instance attached populated with information form the `Page` and the underlying `PageRequest`.
- The `PagedResources` gets `prev` and `next` links attached depending on the page's state. The links will point to the URI the method invoked is mapped to. The pagination parameters added to the method will match the setup of the `PageableHandlerMethodArgumentResolver` to make sure the links can be resolved later on.

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

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

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

Querydsl web support

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

This means that given the `User` object from previous samples a query string

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

can be resolved to

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

using the `QuerydslPredicateArgumentResolver`.

NOTE	The feature will be automatically enabled along `@EnableSpringDataWebSupport` when Querydsl is found on the classpath.

Adding a `@QuerydslPredicate` to the method signature will provide a ready to use `Predicate` which can be executed via the `QueryDslPredicateExecutor`.

TIP Type information is typically resolved from the methods return type. Since those information does not necessarily match the domain type it might be a good idea to use the `root` attribute of `QuerydslPredicate`.

```
@Controller
class UserController {

  @Autowired UserRepository repository;

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

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

    return "index";
  }
}
```

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

The default binding is as follows:

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

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

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

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

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

① `QueryDslPredicateExecutor` provides access to specific finder methods for `Predicate`.

② `QuerydslBinderCustomizer` defined on the repository interface will be 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.

7.8.3. Repository populators

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

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

Example 46. Data defined in JSON

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

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

Example 47. Declaring a Jackson repository populator

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

  <repository:jackson2-populator locations="classpath:data.json" />

</beans>
```

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

The type to which the JSON object will be unmarshalled to will be determined by inspecting the `_class` attribute of the JSON document. The infrastructure will eventually select the appropriate repository to handle the object just deserialized.

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

Example 48. Declaring an unmarshalling repository populator (using JAXB)

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

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

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

</beans>
```

7.8.4. Legacy web support

Domain class web binding for Spring MVC

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

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

  private final UserRepository userRepository;

  UserController(UserRepository userRepository) {
    Assert.notNull(repository, "Repository must not be null!");
    this.userRepository = userRepository;
  }

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

    // Do null check for id
    User user = userRepository.findById(id);
    // Do null check for user

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

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

PropertyEditors

For Spring versions before 3.0 simple Java `PropertyEditors` had to be used. To integrate with that, Spring Data offers a `DomainClassPropertyEditorRegistrar`, which looks up all Spring Data repositories registered in the `ApplicationContext` and registers a custom `PropertyEditor` for the managed domain class.

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

If you have configured Spring MVC as in the preceding example, you can configure your controller as follows, which reduces a lot of the clutter and boilerplate.

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

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

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

Reference Documentation

Chapter 8. Introduction

8.1. Document Structure

This part of the reference documentation explains the core functionality offered by Spring Data MongoDB.

MongoDB support introduces the MongoDB module feature set.

MongoDB repositories introduces the repository support for MongoDB.

Chapter 9. MongoDB support

The MongoDB support contains a wide range of features which are summarized below.

- Spring configuration support using Java based @Configuration classes or an XML namespace for a Mongo driver instance and replica sets
- MongoTemplate helper class that increases productivity performing common Mongo operations. Includes integrated object mapping between documents and POJOs.
- Exception translation into Spring's portable Data Access Exception hierarchy
- Feature Rich Object Mapping integrated with Spring's Conversion Service
- Annotation based mapping metadata but extensible to support other metadata formats
- Persistence and mapping lifecycle events
- Java based Query, Criteria, and Update DSLs
- Automatic implementation of Repository interfaces including support for custom finder methods.
- QueryDSL integration to support type-safe queries.
- Cross-store persistence - support for JPA Entities with fields transparently persisted/retrieved using MongoDB (deprecated - will be removed without replacement)
- GeoSpatial integration

For most tasks you will find yourself using `MongoTemplate` or the Repository support that both leverage the rich mapping functionality. `MongoTemplate` is the place to look for accessing functionality such as incrementing counters or ad-hoc CRUD operations. `MongoTemplate` also provides callback methods so that it is easy for you to get a hold of the low level API artifacts such as `com.mongo.DB` to communicate directly with MongoDB. The goal with naming conventions on various API artifacts is to copy those in the base MongoDB Java driver so you can easily map your existing knowledge onto the Spring APIs.

9.1. Getting Started

Spring MongoDB support requires MongoDB 2.6 or higher and Java SE 8 or higher. An easy way to bootstrap setting up a working environment is to create a Spring based project in STS.

First you need to set up a running Mongodb server. Refer to the Mongodb Quick Start guide for an explanation on how to startup a MongoDB instance. Once installed starting MongoDB is typically a matter of executing the following command: `MONGO_HOME/bin/mongod`

To create a Spring project in STS go to File → New → Spring Template Project → Simple Spring Utility Project → press Yes when prompted. Then enter a project and a package name such as org.spring.mongodb.example.

Then add the following to pom.xml dependencies section.

```
<dependencies>

  <!-- other dependency elements omitted -->

  <dependency>
    <groupId>org.springframework.data</groupId>
    <artifactId>spring-data-mongodb</artifactId>
    <version>{version}</version>
  </dependency>

</dependencies>
```

Also change the version of Spring in the pom.xml to be

```
<spring.framework.version>{springVersion}</spring.framework.version>
```

You will also need to add the location of the Spring Milestone repository for maven to your `pom.xml` which is at the same level of your `<dependencies/>` element

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

The repository is also browseable here.

You may also want to set the logging level to `DEBUG` to see some additional information, edit the `log4j.properties` file to have

```
log4j.category.org.springframework.data.mongodb=DEBUG
log4j.appender.stdout.layout.ConversionPattern=%d{ABSOLUTE} %5p %40.40c:%4L - %m%n
```

Create a simple Person class to persist:

```
package org.spring.mongodb.example;

public class Person {

  private String id;
  private String name;
  private int age;

  public Person(String name, int age) {
    this.name = name;
    this.age = age;
  }

  public String getId() {
    return id;
  }
  public String getName() {
    return name;
  }
  public int getAge() {
    return age;
  }

  @Override
  public String toString() {
    return "Person [id=" + id + ", name=" + name + ", age=" + age + "]";
  }
}
```

And a main application to run

```
package org.spring.mongodb.example;

import static org.springframework.data.mongodb.core.query.Criteria.where;

import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;
import org.springframework.data.mongodb.core.MongoOperations;
import org.springframework.data.mongodb.core.MongoTemplate;
import org.springframework.data.mongodb.core.query.Query;

import com.mongodb.MongoClient;

public class MongoApp {

  private static final Log log = LogFactory.getLog(MongoApp.class);

  public static void main(String[] args) throws Exception {

    MongoOperations mongoOps = new MongoTemplate(new MongoClient(), "database");
    mongoOps.insert(new Person("Joe", 34));

    log.info(mongoOps.findOne(new Query(where("name").is("Joe")), Person.class));

    mongoOps.dropCollection("person");
  }
}
```

This will produce the following output

```
10:01:32,062 DEBUG apping.MongoPersistentEntityIndexCreator:  80 - Analyzing class
class org.spring.example.Person for index information.
10:01:32,265 DEBUG ramework.data.mongodb.core.MongoTemplate: 631 - insert Document
containing fields: [_class, age, name] in collection: Person
10:01:32,765 DEBUG ramework.data.mongodb.core.MongoTemplate:1243 - findOne using
query: { "name" : "Joe"} in db.collection: database.Person
10:01:32,953  INFO      org.spring.mongodb.example.MongoApp:  25 - Person
[id=4ddbba3c0be56b7e1b210166, name=Joe, age=34]
10:01:32,984 DEBUG ramework.data.mongodb.core.MongoTemplate: 375 - Dropped collection
[database.person]
```

Even in this simple example, there are few things to take notice of

- You can instantiate the central helper class of Spring Mongo, `MongoTemplate`, using the standard `com.mongodb.MongoClient` object and the name of the database to use.
- The mapper works against standard POJO objects without the need for any additional metadata (though you can optionally provide that information. See here.).
- Conventions are used for handling the id field, converting it to be a `ObjectId` when stored in the

database.

- Mapping conventions can use field access. Notice the Person class has only getters.
- If the constructor argument names match the field names of the stored document, they will be used to instantiate the object

9.2. Examples Repository

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

9.3. Connecting to MongoDB with Spring

One of the first tasks when using MongoDB and Spring is to create a `com.mongodb.MongoClient` object using the IoC container. There are two main ways to do this, either using Java based bean metadata or XML based bean metadata. These are discussed in the following sections.

NOTE	For those not familiar with how to configure the Spring container using Java based bean metadata instead of XML based metadata see the high level introduction in the reference docs here as well as the detailed documentation here.

9.3.1. Registering a Mongo instance using Java based metadata

An example of using Java based bean metadata to register an instance of a `com.mongodb.MongoClient` is shown below

Example 49. Registering a com.mongodb.MongoClient object using Java based bean metadata

```
@Configuration
public class AppConfig {

  /*
   * Use the standard Mongo driver API to create a com.mongodb.MongoClient
instance.
   */
   public @Bean MongoClient mongoClient() {
       return new MongoClient("localhost");
   }
}
```

This approach allows you to use the standard `com.mongodb.MongoClient` instance with the container using Spring's `MongoClientFactoryBean`. As compared to instantiating a `com.mongodb.MongoClient` instance directly, the FactoryBean has the added advantage of also providing the container with an ExceptionTranslator implementation that translates MongoDB exceptions to exceptions in Spring's portable `DataAccessException` hierarchy for data access classes annotated with the `@Repository` annotation. This hierarchy and use of `@Repository` is described in Spring's DAO support features.

An example of a Java based bean metadata that supports exception translation on `@Repository` annotated classes is shown below:

Example 50. Registering a com.mongodb.MongoClient object using Spring's MongoClientFactoryBean and enabling Spring's exception translation support

```
@Configuration
public class AppConfig {

    /*
     * Factory bean that creates the com.mongodb.MongoClient instance
     */
     public @Bean MongoClientFactoryBean mongo() {
          MongoClientFactoryBean mongo = new MongoClientFactoryBean();
          mongo.setHost("localhost");
          return mongo;
     }
}
```

To access the `com.mongodb.MongoClient` object created by the `MongoClientFactoryBean` in other `@Configuration` or your own classes, use a "private @Autowired Mongo mongo;" field.

9.3.2. Registering a Mongo instance using XML based metadata

While you can use Spring's traditional `<beans/>` XML namespace to register an instance of `com.mongodb.MongoClient` with the container, the XML can be quite verbose as it is general purpose. XML namespaces are a better alternative to configuring commonly used objects such as the Mongo instance. The mongo namespace allows you to create a Mongo instance server location, replica-sets, and options.

To use the Mongo namespace elements you will need to reference the Mongo schema:

Example 51. XML schema to configure MongoDB

```
<?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:context="http://www.springframework.org/schema/context"
          xmlns:mongo="http://www.springframework.org/schema/data/mongo"
          xsi:schemaLocation=
          "http://www.springframework.org/schema/context
          http://www.springframework.org/schema/context/spring-context-3.0.xsd
          *http://www.springframework.org/schema/data/mongo
http://www.springframework.org/schema/data/mongo/spring-mongo-1.0.xsd*
          http://www.springframework.org/schema/beans
          http://www.springframework.org/schema/beans/spring-beans-3.0.xsd">

    <!-- Default bean name is 'mongo' -->
    *<mongo:mongo-client host="localhost" port="27017"/>*

</beans>
```

A more advanced configuration with `MongoClientOptions` is shown below (note these are not recommended values)

Example 52. XML schema to configure a com.mongodb.MongoClient object with MongoClientOptions

```
<beans>

  <mongo:mongo-client host="localhost" port="27017">
    <mongo:client-options connections-per-host="8"
                  threads-allowed-to-block-for-connection-multiplier="4"
                  connect-timeout="1000"
                  max-wait-time="1500}"
                  auto-connect-retry="true"
                  socket-keep-alive="true"
                  socket-timeout="1500"
                  slave-ok="true"
                  write-number="1"
                  write-timeout="0"
                  write-fsync="true"/>
  </mongo:mongo-client>

</beans>
```

A configuration using replica sets is shown below.

Example 53. XML schema to configure com.mongodb.MongoClient object with Replica Sets

```
<mongo:mongo-client id="replicaSetMongo" replica-set=
"127.0.0.1:27017,localhost:27018"/>
```

9.3.3. The MongoDbFactory interface

While com.mongodb.MongoClient is the entry point to the MongoDB driver API, connecting to a specific MongoDB database instance requires additional information such as the database name and an optional username and password. With that information you can obtain a com.mongodb.DB object and access all the functionality of a specific MongoDB database instance. Spring provides the org.springframework.data.mongodb.core.MongoDbFactory interface shown below to bootstrap connectivity to the database.

```
public interface MongoDbFactory {

  MongoDatabase getDb() throws DataAccessException;

  MongoDatabase getDb(String dbName) throws DataAccessException;
}
```

The following sections show how you can use the container with either Java or the XML based metadata to configure an instance of the MongoDbFactory interface. In turn, you can use the MongoDbFactory instance to configure MongoTemplate.

Instead of using the IoC container to create an instance of MongoTemplate, you can just use them in standard Java code as shown below.

```
public class MongoApp {

  private static final Log log = LogFactory.getLog(MongoApp.class);

  public static void main(String[] args) throws Exception {

    MongoOperations mongoOps = new MongoTemplate(*new SimpleMongoDbFactory(new
MongoClient(), "database")*);

    mongoOps.insert(new Person("Joe", 34));

    log.info(mongoOps.findOne(new Query(where("name").is("Joe")), Person.class));

    mongoOps.dropCollection("person");
  }
}
```

The code in bold highlights the use of SimpleMongoDbFactory and is the only difference between

the listing shown in the getting started section.

9.3.4. Registering a MongoDbFactory instance using Java based metadata

To register a MongoDbFactory instance with the container, you write code much like what was highlighted in the previous code listing. A simple example is shown below

```
@Configuration
public class MongoConfiguration {

  public @Bean MongoDbFactory mongoDbFactory() {
    return new SimpleMongoDbFactory(new MongoClient(), "database");
  }
}
```

MongoDB Server generation 3 changed the authentication model when connecting to the DB. Therefore some of the configuration options available for authentication are no longer valid. Please use the `MongoClient` specific options for setting credentials via `MongoCredential` to provide authentication data.

```
@Configuration
public class ApplicationContextEventTestsAppConfig extends AbstractMongoConfiguration
{

  @Override
  public String getDatabaseName() {
    return "database";
  }

  @Override
  @Bean
  public MongoClient mongoClient() {
    return new MongoClient(singletonList(new ServerAddress("127.0.0.1", 27017)),
      singletonList(MongoCredential.createCredential("name", "db", "pwd".toCharArray(
))));
  }
}
```

In order to use authentication with XML configuration use the `credentials` attribue on `<mongo-client>`.

NOTE Username/password credentials used in XML configuration must be URL encoded when these contain reserved characters such as `:`, `%`, `@`, `,`. Example: `m0ng0@dmin:mo_res:bw6},Qsdxx@admin@database` → `m0ng0%40dmin:mo_res%3Abw6%7D%2CQsdxx%40admin@database` See section 2.2 of RFC 3986 for further details.

9.3.5. Registering a MongoDbFactory instance using XML based metadata

The mongo namespace provides a convenient way to create a `SimpleMongoDbFactory` as compared to using the `<beans/>` namespace. Simple usage is shown below

```
<mongo:db-factory dbname="database">
```

If you need to configure additional options on the `com.mongodb.MongoClient` instance that is used to create a `SimpleMongoDbFactory` you can refer to an existing bean using the `mongo-ref` attribute as shown below. To show another common usage pattern, this listing shows the use of a property placeholder to parametrise the configuration and creating `MongoTemplate`.

```
<context:property-placeholder location=
"classpath:/com/myapp/mongodb/config/mongo.properties"/>

<mongo:mongo-client host="${mongo.host}" port="${mongo.port}">
  <mongo:client-options
     connections-per-host="${mongo.connectionsPerHost}"
     threads-allowed-to-block-for-connection-multiplier=
"${mongo.threadsAllowedToBlockForConnectionMultiplier}"
     connect-timeout="${mongo.connectTimeout}"
     max-wait-time="${mongo.maxWaitTime}"
     auto-connect-retry="${mongo.autoConnectRetry}"
     socket-keep-alive="${mongo.socketKeepAlive}"
     socket-timeout="${mongo.socketTimeout}"
     slave-ok="${mongo.slaveOk}"
     write-number="1"
     write-timeout="0"
     write-fsync="true"/>
</mongo:mongo-client>

<mongo:db-factory dbname="database" mongo-ref="mongoClient"/>

<bean id="anotherMongoTemplate" class=
"org.springframework.data.mongodb.core.MongoTemplate">
  <constructor-arg name="mongoDbFactory" ref="mongoDbFactory"/>
</bean>
```

9.4. Introduction to MongoTemplate

The class `MongoTemplate`, located in the package `org.springframework.data.mongodb.core`, is the central class of the Spring's MongoDB support providing a rich feature set to interact with the database. The template offers convenience operations to create, update, delete and query for MongoDB documents and provides a mapping between your domain objects and MongoDB documents.

NOTE	Once configured, `MongoTemplate` is thread-safe and can be reused across multiple instances.

The mapping between MongoDB documents and domain classes is done by delegating to an implementation of the interface `MongoConverter`. Spring provides the `MappingMongoConverter`, but you can also write your own converter. Please refer to the section on MongoConverters for more detailed information.

The `MongoTemplate` class implements the interface `MongoOperations`. In as much as possible, the methods on `MongoOperations` are named after methods available on the MongoDB driver `Collection` object to make the API familiar to existing MongoDB developers who are used to the driver API. For example, you will find methods such as "find", "findAndModify", "findOne", "insert", "remove", "save", "update" and "updateMulti". The design goal was to make it as easy as possible to transition between the use of the base MongoDB driver and `MongoOperations`. A major difference in between the two APIs is that MongoOperations can be passed domain objects instead of `Document` and there are fluent APIs for `Query`, `Criteria`, and `Update` operations instead of populating a `Document` to specify the parameters for those operations.

NOTE	The preferred way to reference the operations on `MongoTemplate` instance is via its interface `MongoOperations`.

The default converter implementation used by `MongoTemplate` is MappingMongoConverter. While the `MappingMongoConverter` can make use of additional metadata to specify the mapping of objects to documents it is also capable of converting objects that contain no additional metadata by using some conventions for the mapping of IDs and collection names. These conventions as well as the use of mapping annotations is explained in the Mapping chapter.

Another central feature of MongoTemplate is exception translation of exceptions thrown in the MongoDB Java driver into Spring's portable Data Access Exception hierarchy. Refer to the section on exception translation for more information.

While there are many convenience methods on `MongoTemplate` to help you easily perform common tasks if you should need to access the MongoDB driver API directly to access functionality not explicitly exposed by the MongoTemplate you can use one of several Execute callback methods to access underlying driver APIs. The execute callbacks will give you a reference to either a `com.mongodb.Collection` or a `com.mongodb.DB` object. Please see the section mongo.executioncallback[Execution Callbacks] for more information.

Now let's look at an example of how to work with the `MongoTemplate` in the context of the Spring container.

9.4.1. Instantiating MongoTemplate

You can use Java to create and register an instance of `MongoTemplate` as shown below.

Example 54. Registering a com.mongodb.MongoClient object and enabling Spring's exception translation support

```
@Configuration
public class AppConfig {

  public @Bean MongoClient mongoClient() {
      return new MongoClient("localhost");
  }

  public @Bean MongoTemplate mongoTemplate() {
      return new MongoTemplate(mongoClient(), "mydatabase");
  }
}
```

There are several overloaded constructors of MongoTemplate. These are

- `MongoTemplate(MongoClient mongo, String databaseName)` - takes the `com.mongodb.MongoClient` object and the default database name to operate against.
- `MongoTemplate(MongoDbFactory mongoDbFactory)` - takes a MongoDbFactory object that encapsulated the `com.mongodb.MongoClient` object, database name, and username and password.
- `MongoTemplate(MongoDbFactory mongoDbFactory, MongoConverter mongoConverter)` - adds a MongoConverter to use for mapping.

You can also configure a MongoTemplate using Spring's XML <beans/> schema.

```
<mongo:mongo-client host="localhost" port="27017"/>

<bean id="mongoTemplate" class="org.springframework.data.mongodb.core.MongoTemplate">
  <constructor-arg ref="mongoClient"/>
  <constructor-arg name="databaseName" value="geospatial"/>
</bean>
```

Other optional properties that you might like to set when creating a `MongoTemplate` are the default `WriteResultCheckingPolicy`, `WriteConcern`, and `ReadPreference`.

NOTE	The preferred way to reference the operations on `MongoTemplate` instance is via its interface `MongoOperations`.

9.4.2. WriteResultChecking Policy

When in development it is very handy to either log or throw an exception if the `com.mongodb.WriteResult` returned from any MongoDB operation contains an error. It is quite common to forget to do this during development and then end up with an application that looks like it runs successfully but in fact the database was not modified according to your expectations. Set MongoTemplate's property to an enum with the following values, `EXCEPTION`, or `NONE` to either

throw an Exception or do nothing. The default is to use a `WriteResultChecking` value of `NONE`.

9.4.3. WriteConcern

You can set the `com.mongodb.WriteConcern` property that the `MongoTemplate` will use for write operations if it has not yet been specified via the driver at a higher level such as `com.mongodb.MongoClient`. If MongoTemplate's `WriteConcern` property is not set it will default to the one set in the MongoDB driver's DB or Collection setting.

9.4.4. WriteConcernResolver

For more advanced cases where you want to set different `WriteConcern` values on a per-operation basis (for remove, update, insert and save operations), a strategy interface called `WriteConcernResolver` can be configured on `MongoTemplate`. Since `MongoTemplate` is used to persist POJOs, the `WriteConcernResolver` lets you create a policy that can map a specific POJO class to a `WriteConcern` value. The `WriteConcernResolver` interface is shown below.

```
public interface WriteConcernResolver {
  WriteConcern resolve(MongoAction action);
}
```

The passed in argument, MongoAction, is what you use to determine the `WriteConcern` value to be used or to use the value of the Template itself as a default. `MongoAction` contains the collection name being written to, the `java.lang.Class` of the POJO, the converted `Document`, as well as the operation as an enumeration (`MongoActionOperation`: REMOVE, UPDATE, INSERT, INSERT_LIST, SAVE) and a few other pieces of contextual information. For example,

```
private class MyAppWriteConcernResolver implements WriteConcernResolver {

  public WriteConcern resolve(MongoAction action) {
    if (action.getEntityClass().getSimpleName().contains("Audit")) {
      return WriteConcern.NONE;
    } else if (action.getEntityClass().getSimpleName().contains("Metadata")) {
      return WriteConcern.JOURNAL_SAFE;
    }
    return action.getDefaultWriteConcern();
  }
}
```

9.5. Saving, Updating, and Removing Documents

`MongoTemplate` provides a simple way for you to save, update, and delete your domain objects and map those objects to documents stored in MongoDB.

Given a simple class such as Person

```
public class Person {

  private String id;
  private String name;
  private int age;

  public Person(String name, int age) {
    this.name = name;
    this.age = age;
  }

  public String getId() {
    return id;
  }
  public String getName() {
    return name;
  }
  public int getAge() {
    return age;
  }

  @Override
  public String toString() {
    return "Person [id=" + id + ", name=" + name + ", age=" + age + "]";
  }

}
```

You can save, update and delete the object as shown below.

NOTE `MongoOperations` is the interface that `MongoTemplate` implements.

```
package org.spring.example;

import static org.springframework.data.mongodb.core.query.Criteria.where;
import static org.springframework.data.mongodb.core.query.Update.update;
import static org.springframework.data.mongodb.core.query.Query.query;

import java.util.List;

import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;
import org.springframework.data.mongodb.core.MongoOperations;
import org.springframework.data.mongodb.core.MongoTemplate;
import org.springframework.data.mongodb.core.SimpleMongoDbFactory;

import com.mongodb.MongoClient;

public class MongoApp {
```

```
  private static final Log log = LogFactory.getLog(MongoApp.class);

  public static void main(String[] args) {

    MongoOperations mongoOps = new MongoTemplate(new SimpleMongoDbFactory(new
MongoClient(), "database"));

    Person p = new Person("Joe", 34);

    // Insert is used to initially store the object into the database.
    mongoOps.insert(p);
    log.info("Insert: " + p);

    // Find
    p = mongoOps.findById(p.getId(), Person.class);
    log.info("Found: " + p);

    // Update
    mongoOps.updateFirst(query(where("name").is("Joe")), update("age", 35), Person
.class);
    p = mongoOps.findOne(query(where("name").is("Joe")), Person.class);
    log.info("Updated: " + p);

    // Delete
    mongoOps.remove(p);

    // Check that deletion worked
    List<Person> people =  mongoOps.findAll(Person.class);
    log.info("Number of people = : " + people.size());


    mongoOps.dropCollection(Person.class);
  }
}
```

This would produce the following log output (including debug messages from `MongoTemplate` itself)

```
DEBUG apping.MongoPersistentEntityIndexCreator:  80 - Analyzing class class
org.spring.example.Person for index information.
DEBUG work.data.mongodb.core.MongoTemplate: 632 - insert Document containing fields:
[_class, age, name] in collection: person
INFO               org.spring.example.MongoApp:  30 - Insert: Person
[id=4ddc6e784ce5b1eba3ceaf5c, name=Joe, age=34]
DEBUG work.data.mongodb.core.MongoTemplate:1246 - findOne using query: { "_id" : {
"$oid" : "4ddc6e784ce5b1eba3ceaf5c"}} in db.collection: database.person
INFO               org.spring.example.MongoApp:  34 - Found: Person
[id=4ddc6e784ce5b1eba3ceaf5c, name=Joe, age=34]
DEBUG work.data.mongodb.core.MongoTemplate: 778 - calling update using query: { "name"
: "Joe"} and update: { "$set" : { "age" : 35}} in collection: person
DEBUG work.data.mongodb.core.MongoTemplate:1246 - findOne using query: { "name" :
"Joe"} in db.collection: database.person
INFO               org.spring.example.MongoApp:  39 - Updated: Person
[id=4ddc6e784ce5b1eba3ceaf5c, name=Joe, age=35]
DEBUG work.data.mongodb.core.MongoTemplate: 823 - remove using query: { "id" :
"4ddc6e784ce5b1eba3ceaf5c"} in collection: person
INFO               org.spring.example.MongoApp:  46 - Number of people = : 0
DEBUG work.data.mongodb.core.MongoTemplate: 376 - Dropped collection [database.person]
```

There was implicit conversion using the `MongoConverter` between a `String` and `ObjectId` as stored in the database and recognizing a convention of the property "Id" name.

NOTE	This example is meant to show the use of save, update and remove operations on MongoTemplate and not to show complex mapping functionality

The query syntax used in the example is explained in more detail in the section Querying Documents.

9.5.1. How the `_id` field is handled in the mapping layer

MongoDB requires that you have an `_id` field for all documents. If you don't provide one the driver will assign a `ObjectId` with a generated value. When using the `MappingMongoConverter` there are certain rules that govern how properties from the Java class is mapped to this `_id` field.

The following outlines what property will be mapped to the `_id` document field:

- A property or field annotated with `@Id` (`org.springframework.data.annotation.Id`) will be mapped to the `_id` field.
- A property or field without an annotation but named `id` will be mapped to the `_id` field.

The following outlines what type conversion, if any, will be done on the property mapped to the _id document field when using the `MappingMongoConverter`, the default for `MongoTemplate`.

- An id property or field declared as a String in the Java class will be converted to and stored as an `ObjectId` if possible using a Spring `Converter<String, ObjectId>`. Valid conversion rules are delegated to the MongoDB Java driver. If it cannot be converted to an ObjectId, then the value

will be stored as a string in the database.

- An id property or field declared as `BigInteger` in the Java class will be converted to and stored as an `ObjectId` using a Spring `Converter<BigInteger, ObjectId>`.

If no field or property specified above is present in the Java class then an implicit `_id` file will be generated by the driver but not mapped to a property or field of the Java class.

When querying and updating `MongoTemplate` will use the converter to handle conversions of the `Query` and `Update` objects that correspond to the above rules for saving documents so field names and types used in your queries will be able to match what is in your domain classes.

9.5.2. Type mapping

As MongoDB collections can contain documents that represent instances of a variety of types. A great example here is if you store a hierarchy of classes or simply have a class with a property of type `Object`. In the latter case the values held inside that property have to be read in correctly when retrieving the object. Thus we need a mechanism to store type information alongside the actual document.

To achieve that the `MappingMongoConverter` uses a `MongoTypeMapper` abstraction with `DefaultMongoTypeMapper` as it's main implementation. Its default behavior is storing the fully qualified classname under `_class` inside the document. Type hints are written for top-level documents as well as for every value if it's a complex type and a subtype of the property type declared.

Example 55. Type mapping

```
public class Sample {
  Contact value;
}

public abstract class Contact { ··· }

public class Person extends Contact { ··· }

Sample sample = new Sample();
sample.value = new Person();

mongoTemplate.save(sample);

{
  "value" : { "_class" : "com.acme.Person" },
  "_class" : "com.acme.Sample"
}
```

As you can see we store the type information as last field for the actual root class as well as for the nested type as it is complex and a subtype of `Contact`. So if you're now using `mongoTemplate.findAll(Object.class, "sample")` we are able to find out that the document stored

shall be a `Sample` instance. We are also able to find out that the value property shall be a `Person` actually.

Customizing type mapping

In case you want to avoid writing the entire Java class name as type information but rather like to use some key you can use the `@TypeAlias` annotation at the entity class being persisted. If you need to customize the mapping even more have a look at the `TypeInformationMapper` interface. An instance of that interface can be configured at the `DefaultMongoTypeMapper` which can be configured in turn on `MappingMongoConverter`.

Example 56. Defining a TypeAlias for an Entity

```
@TypeAlias("pers")
class Person {

}
```

Note that the resulting document will contain `"pers"` as the value in the `_class` Field.

Configuring custom type mapping

The following example demonstrates how to configure a custom `MongoTypeMapper` in `MappingMongoConverter`.

Example 57. Configuring a custom MongoTypeMapper via Spring Java Config

```
class CustomMongoTypeMapper extends DefaultMongoTypeMapper {
  //implement custom type mapping here
}
```

```
@Configuration
class SampleMongoConfiguration extends AbstractMongoConfiguration {

  @Override
  protected String getDatabaseName() {
    return "database";
  }

  @Override
  public MongoClient mongoClient() {
    return new MongoClient();
  }

  @Bean
  @Override
  public MappingMongoConverter mappingMongoConverter() throws Exception {
    MappingMongoConverter mmc = super.mappingMongoConverter();
    mmc.setTypeMapper(customTypeMapper());
    return mmc;
  }

  @Bean
  public MongoTypeMapper customTypeMapper() {
    return new CustomMongoTypeMapper();
  }
}
```

Note that we are extending the `AbstractMongoConfiguration` class and override the bean definition of the `MappingMongoConverter` where we configure our custom `MongoTypeMapper`.

Example 58. Configuring a custom MongoTypeMapper via XML

```
<mongo:mapping-converter type-mapper-ref="customMongoTypeMapper"/>

<bean name="customMongoTypeMapper" class="com.bubu.mongo.CustomMongoTypeMapper"/>
```

9.5.3. Methods for saving and inserting documents

There are several convenient methods on `MongoTemplate` for saving and inserting your objects. To have more fine-grained control over the conversion process you can register Spring converters with the `MappingMongoConverter`, for example `Converter<Person, Document>` and `Converter<Document, Person>`.

NOTE The difference between insert and save operations is that a save operation will perform an insert if the object is not already present.

The simple case of using the save operation is to save a POJO. In this case the collection name will

be determined by name (not fully qualified) of the class. You may also call the save operation with a specific collection name. The collection to store the object can be overridden using mapping metadata.

When inserting or saving, if the Id property is not set, the assumption is that its value will be auto-generated by the database. As such, for auto-generation of an ObjectId to succeed the type of the Id property/field in your class must be either a `String`, `ObjectId`, or `BigInteger`.

Here is a basic example of using the save operation and retrieving its contents.

Example 59. Inserting and retrieving documents using the MongoTemplate

```
import static org.springframework.data.mongodb.core.query.Criteria.where;
import static org.springframework.data.mongodb.core.query.Criteria.query;
...

Person p = new Person("Bob", 33);
mongoTemplate.insert(p);

Person qp = mongoTemplate.findOne(query(where("age").is(33)), Person.class);
```

The insert/save operations available to you are listed below.

- `void` **save** `(Object objectToSave)` Save the object to the default collection.
- `void` **save** `(Object objectToSave, String collectionName)` Save the object to the specified collection.

A similar set of insert operations is listed below

- `void` **insert** `(Object objectToSave)` Insert the object to the default collection.
- `void` **insert** `(Object objectToSave, String collectionName)` Insert the object to the specified collection.

Which collection will my documents be saved into?

There are two ways to manage the collection name that is used for operating on the documents. The default collection name that is used is the class name changed to start with a lower-case letter. So a `com.test.Person` class would be stored in the "person" collection. You can customize this by providing a different collection name using the @Document annotation. You can also override the collection name by providing your own collection name as the last parameter for the selected MongoTemplate method calls.

Inserting or saving individual objects

The MongoDB driver supports inserting a collection of documents in one operation. The methods in the MongoOperations interface that support this functionality are listed below

- **insert** inserts an object. If there is an existing document with the same id then an error is

generated.

- **insertAll** takes a `Collection` of objects as the first parameter. This method inspects each object and inserts it to the appropriate collection based on the rules specified above.
- **save** saves the object overwriting any object that might exist with the same id.

Inserting several objects in a batch

The MongoDB driver supports inserting a collection of documents in one operation. The methods in the MongoOperations interface that support this functionality are listed below

- **insert** methods that take a `Collection` as the first argument. This inserts a list of objects in a single batch write to the database.

9.5.4. Updating documents in a collection

For updates we can elect to update the first document found using `MongoOperation` 's method `updateFirst` or we can update all documents that were found to match the query using the method `updateMulti`. Here is an example of an update of all SAVINGS accounts where we are adding a one-time $50.00 bonus to the balance using the `$inc` operator.

Example 60. Updating documents using the MongoTemplate

```
import static org.springframework.data.mongodb.core.query.Criteria.where;
import static org.springframework.data.mongodb.core.query.Query;
import static org.springframework.data.mongodb.core.query.Update;

...

WriteResult wr = mongoTemplate.updateMulti(new Query(where("accounts.accountType"
).is(Account.Type.SAVINGS)),
  new Update().inc("accounts.$.balance", 50.00), Account.class);
```

In addition to the `Query` discussed above we provide the update definition using an `Update` object. The `Update` class has methods that match the update modifiers available for MongoDB.

As you can see most methods return the `Update` object to provide a fluent style for the API.

Methods for executing updates for documents

- **updateFirst** Updates the first document that matches the query document criteria with the provided updated document.
- **updateMulti** Updates all objects that match the query document criteria with the provided updated document.

Methods for the Update class

The Update class can be used with a little 'syntax sugar' as its methods are meant to be chained

together and you can kick-start the creation of a new Update instance via the static method `public static Update update(String key, Object value)` and using static imports.

Here is a listing of methods on the Update class

- `Update` **addToSet** `(String key, Object value)` Update using the `$addToSet` update modifier
- `Update` **currentDate** `(String key)` Update using the `$currentDate` update modifier
- `Update` **currentTimestamp** `(String key)` Update using the `$currentDate` update modifier with `$type timestamp`
- `Update` **inc** `(String key, Number inc)` Update using the `$inc` update modifier
- `Update` **max** `(String key, Object max)` Update using the `$max` update modifier
- `Update` **min** `(String key, Object min)` Update using the `$min` update modifier
- `Update` **multiply** `(String key, Number multiplier)` Update using the `$mul` update modifier
- `Update` **pop** `(String key, Update.Position pos)` Update using the `$pop` update modifier
- `Update` **pull** `(String key, Object value)` Update using the `$pull` update modifier
- `Update` **pullAll** `(String key, Object[] values)` Update using the `$pullAll` update modifier
- `Update` **push** `(String key, Object value)` Update using the `$push` update modifier
- `Update` **pushAll** `(String key, Object[] values)` Update using the `$pushAll` update modifier
- `Update` **rename** `(String oldName, String newName)` Update using the `$rename` update modifier
- `Update` **set** `(String key, Object value)` Update using the `$set` update modifier
- `Update` **setOnInsert** `(String key, Object value)` Update using the `$setOnInsert` update modifier
- `Update` **unset** `(String key)` Update using the `$unset` update modifier

Some update modifiers like `$push` and `$addToSet` allow nesting of additional operators.

```
// { $push : { "category" : { "$each" : [ "spring" , "data" ] } } }
new Update().push("category").each("spring", "data")

// { $push : { "key" : { "$position" : 0 , "$each" : [ "Arya" , "Arry" , "Weasel" ] }
} }
new Update().push("key").atPosition(Position.FIRST).each(Arrays.asList("Arya", "Arry",
"Weasel"));

// { $push : { "key" : { "$slice" : 5 , "$each" : [ "Arya" , "Arry" , "Weasel" ] } } }
new Update().push("key").slice(5).each(Arrays.asList("Arya", "Arry", "Weasel"));
```

```
// { $addToSet : { "values" : { "$each" : [ "spring" , "data" , "mongodb" ] } } }
new Update().addToSet("values").each("spring", "data", "mongodb");
```

9.5.5. Upserting documents in a collection

Related to performing an `updateFirst` operations, you can also perform an upsert operation which will perform an insert if no document is found that matches the query. The document that is inserted is a combination of the query document and the update document. Here is an example

```
template.upsert(query(where("ssn").is(1111).and("firstName").is("Joe").and("Fraizer").
is("Update")), update("address", addr), Person.class);
```

9.5.6. Finding and Upserting documents in a collection

The `findAndModify(…)` method on DBCollection can update a document and return either the old or newly updated document in a single operation. `MongoTemplate` provides a findAndModify method that takes `Query` and `Update` classes and converts from `Document` to your POJOs. Here are the methods

```
<T> T findAndModify(Query query, Update update, Class<T> entityClass);

<T> T findAndModify(Query query, Update update, Class<T> entityClass, String
collectionName);

<T> T findAndModify(Query query, Update update, FindAndModifyOptions options, Class<T>
entityClass);

<T> T findAndModify(Query query, Update update, FindAndModifyOptions options, Class<T>
entityClass, String collectionName);
```

As an example usage, we will insert of few `Person` objects into the container and perform a simple findAndUpdate operation

```
mongoTemplate.insert(new Person("Tom", 21));
mongoTemplate.insert(new Person("Dick", 22));
mongoTemplate.insert(new Person("Harry", 23));

Query query = new Query(Criteria.where("firstName").is("Harry"));
Update update = new Update().inc("age", 1);
Person p = mongoTemplate.findAndModify(query, update, Person.class); // return's old
person object

assertThat(p.getFirstName(), is("Harry"));
assertThat(p.getAge(), is(23));
p = mongoTemplate.findOne(query, Person.class);
assertThat(p.getAge(), is(24));

// Now return the newly updated document when updating
p = template.findAndModify(query, update, new FindAndModifyOptions().returnNew(true),
Person.class);
assertThat(p.getAge(), is(25));
```

The `FindAndModifyOptions` lets you set the options of returnNew, upsert, and remove. An example extending off the previous code snippet is shown below

```
Query query2 = new Query(Criteria.where("firstName").is("Mary"));
p = mongoTemplate.findAndModify(query2, update, new FindAndModifyOptions().returnNew
(true).upsert(true), Person.class);
assertThat(p.getFirstName(), is("Mary"));
assertThat(p.getAge(), is(1));
```

9.5.7. Methods for removing documents

You can use several overloaded methods to remove an object from the database.

- **remove** Remove the given document based on one of the following: a specific object instance, a query document criteria combined with a class or a query document criteria combined with a specific collection name.

9.5.8. Optimistic locking

The `@Version` annotation provides a JPA similar semantic in the context of MongoDB and makes sure updates are only applied to documents with matching version. Therefore the actual value of the version property is added to the update query in a way that the update won't have any effect if another operation altered the document in between. In that case an `OptimisticLockingFailureException` is thrown.

```
@Document
class Person {

  @Id String id;
  String firstname;
  String lastname;
  @Version Long version;
}

Person daenerys = template.insert(new Person("Daenerys")); ①

Person tmp = teplate.findOne(query(where("id").is(daenerys.getId())), Person.
class); ②

daenerys.setLastname("Targaryen");
template.save(daenerys); ③

template.save(tmp); // throws OptimisticLockingFailureException ④
```

① Intially insert document. `version` is set to `0`.

② Load the just inserted document `version` is still `0`.

③ Update document with `version = 0`. Set the `lastname` and bump `version` to `1`.

④ Try to update previously loaded document sill having `version = 0` fails with `OptimisticLockingFailureException` as the current `version` is `1`.

IMPORTANT	Using MongoDB driver version 3 requires to set the `WriteConcern` to `ACKNOWLEDGED`. Otherwise `OptimisticLockingFailureException` can be silently swallowed.

9.6. Querying Documents

You can express your queries using the `Query` and `Criteria` classes which have method names that mirror the native MongoDB operator names such as `lt`, `lte`, `is`, and others. The `Query` and `Criteria` classes follow a fluent API style so that you can easily chain together multiple method criteria and queries while having easy to understand the code. Static imports in Java are used to help remove the need to see the 'new' keyword for creating `Query` and `Criteria` instances so as to improve readability. If you like to create `Query` instances from a plain JSON String use `BasicQuery`.

Example 61. Creating a Query instance from a plain JSON String

```
BasicQuery query = new BasicQuery("{ age : { $lt : 50 }, accounts.balance : { $gt
: 1000.00 }}");
List<Person> result = mongoTemplate.find(query, Person.class);
```

GeoSpatial queries are also supported and are described more in the section GeoSpatial Queries.

Map-Reduce operations are also supported and are described more in the section Map-Reduce.

9.6.1. Querying documents in a collection

We saw how to retrieve a single document using the findOne and findById methods on MongoTemplate in previous sections which return a single domain object. We can also query for a collection of documents to be returned as a list of domain objects. Assuming that we have a number of Person objects with name and age stored as documents in a collection and that each person has an embedded account document with a balance. We can now run a query using the following code.

Example 62. Querying for documents using the MongoTemplate

```
import static org.springframework.data.mongodb.core.query.Criteria.where;
import static org.springframework.data.mongodb.core.query.Query.query;

...

List<Person> result = mongoTemplate.find(query(where("age").lt(50)
  .and("accounts.balance").gt(1000.00d)), Person.class);
```

All find methods take a `Query` object as a parameter. This object defines the criteria and options used to perform the query. The criteria is specified using a `Criteria` object that has a static factory method named `where` used to instantiate a new `Criteria` object. We recommend using a static import for `org.springframework.data.mongodb.core.query.Criteria.where` and `Query.query` to make the query more readable.

This query should return a list of `Person` objects that meet the specified criteria. The `Criteria` class has the following methods that correspond to the operators provided in MongoDB.

As you can see most methods return the `Criteria` object to provide a fluent style for the API.

Methods for the Criteria class

- `Criteria` **all** `(Object o)` Creates a criterion using the `$all` operator
- `Criteria` **and** `(String key)` Adds a chained `Criteria` with the specified `key` to the current `Criteria` and returns the newly created one
- `Criteria` **andOperator** `(Criteria… criteria)` Creates an and query using the `$and` operator for

all of the provided criteria (requires MongoDB 2.0 or later)

- `Criteria` **elemMatch** `(Criteria c)` Creates a criterion using the `$elemMatch` operator
- `Criteria` **exists** `(boolean b)` Creates a criterion using the `$exists` operator
- `Criteria` **gt** `(Object o)` Creates a criterion using the `$gt` operator
- `Criteria` **gte** `(Object o)` Creates a criterion using the `$gte` operator
- `Criteria` **in** `(Object… o)` Creates a criterion using the `$in` operator for a varargs argument.
- `Criteria` **in** `(Collection<?> collection)` Creates a criterion using the `$in` operator using a collection
- `Criteria` **is** `(Object o)` Creates a criterion using field matching (`{ key:value }`). If the specified value is a document, the order of the fields and exact equality in the document matters.
- `Criteria` **lt** `(Object o)` Creates a criterion using the `$lt` operator
- `Criteria` **lte** `(Object o)` Creates a criterion using the `$lte` operator
- `Criteria` **mod** `(Number value, Number remainder)` Creates a criterion using the `$mod` operator
- `Criteria` **ne** `(Object o)` Creates a criterion using the `$ne` operator
- `Criteria` **nin** `(Object… o)` Creates a criterion using the `$nin` operator
- `Criteria` **norOperator** `(Criteria… criteria)` Creates an nor query using the `$nor` operator for all of the provided criteria
- `Criteria` **not** `()` Creates a criterion using the `$not` meta operator which affects the clause directly following
- `Criteria` **orOperator** `(Criteria… criteria)` Creates an or query using the `$or` operator for all of the provided criteria
- `Criteria` **regex** `(String re)` Creates a criterion using a `$regex`
- `Criteria` **size** `(int s)` Creates a criterion using the `$size` operator
- `Criteria` **type** `(int t)` Creates a criterion using the `$type` operator

There are also methods on the Criteria class for geospatial queries. Here is a listing but look at the section on GeoSpatial Queries to see them in action.

- `Criteria` **within** `(Circle circle)` Creates a geospatial criterion using `$geoWithin $center` operators.
- `Criteria` **within** `(Box box)` Creates a geospatial criterion using a `$geoWithin $box` operation.
- `Criteria` **withinSphere** `(Circle circle)` Creates a geospatial criterion using `$geoWithin $center` operators.
- `Criteria` **near** `(Point point)` Creates a geospatial criterion using a `$near` operation
- `Criteria` **nearSphere** `(Point point)` Creates a geospatial criterion using `$nearSphere$center` operations. This is only available for MongoDB 1.7 and higher.
- `Criteria` **minDistance** `(double minDistance)` Creates a geospatial criterion using the `$minDistance` operation, for use with $near.
- `Criteria` **maxDistance** `(double maxDistance)` Creates a geospatial criterion using the

`$maxDistance` operation, for use with $near.

The `Query` class has some additional methods used to provide options for the query.

Methods for the Query class

- `Query` **addCriteria** `(Criteria criteria)` used to add additional criteria to the query
- `Field` **fields** `()` used to define fields to be included in the query results
- `Query` **limit** `(int limit)` used to limit the size of the returned results to the provided limit (used for paging)
- `Query` **skip** `(int skip)` used to skip the provided number of documents in the results (used for paging)
- `Query` **with** `(Sort sort)` used to provide sort definition for the results

9.6.2. Methods for querying for documents

The query methods need to specify the target type T that will be returned and they are also overloaded with an explicit collection name for queries that should operate on a collection other than the one indicated by the return type.

- **findAll** Query for a list of objects of type T from the collection.
- **findOne** Map the results of an ad-hoc query on the collection to a single instance of an object of the specified type.
- **findById** Return an object of the given id and target class.
- **find** Map the results of an ad-hoc query on the collection to a List of the specified type.
- **findAndRemove** Map the results of an ad-hoc query on the collection to a single instance of an object of the specified type. The first document that matches the query is returned and also removed from the collection in the database.

9.6.3. GeoSpatial Queries

MongoDB supports GeoSpatial queries through the use of operators such as `$near`, `$within`, `geoWithin` and `$nearSphere`. Methods specific to geospatial queries are available on the `Criteria` class. There are also a few shape classes, `Box`, `Circle`, and `Point` that are used in conjunction with geospatial related `Criteria` methods.

To understand how to perform GeoSpatial queries we will use the following Venue class taken from the integration tests which relies on using the rich `MappingMongoConverter`.

```
@Document(collection="newyork")
public class Venue {

  @Id
  private String id;
  private String name;
  private double[] location;

  @PersistenceConstructor
  Venue(String name, double[] location) {
    super();
    this.name = name;
    this.location = location;
  }

  public Venue(String name, double x, double y) {
    super();
    this.name = name;
    this.location = new double[] { x, y };
  }

  public String getName() {
    return name;
  }

  public double[] getLocation() {
    return location;
  }

  @Override
  public String toString() {
    return "Venue [id=" + id + ", name=" + name + ", location="
        + Arrays.toString(location) + "]";
  }
}
```

To find locations within a `Circle`, the following query can be used.

```
Circle circle = new Circle(-73.99171, 40.738868, 0.01);
List<Venue> venues =
    template.find(new Query(Criteria.where("location").within(circle)), Venue.class);
```

To find venues within a `Circle` using spherical coordinates the following query can be used

```
Circle circle = new Circle(-73.99171, 40.738868, 0.003712240453784);
List<Venue> venues =
    template.find(new Query(Criteria.where("location").withinSphere(circle)), Venue
.class);
```

To find venues within a `Box` the following query can be used

```
//lower-left then upper-right
Box box = new Box(new Point(-73.99756, 40.73083), new Point(-73.988135, 40.741404));
List<Venue> venues =
    template.find(new Query(Criteria.where("location").within(box)), Venue.class);
```

To find venues near a `Point`, the following queries can be used

```
Point point = new Point(-73.99171, 40.738868);
List<Venue> venues =
    template.find(new Query(Criteria.where("location").near(point).maxDistance(0.01)),
Venue.class);
```

```
Point point = new Point(-73.99171, 40.738868);
List<Venue> venues =
    template.find(new Query(Criteria.where("location").near(point).minDistance(0.01)
.maxDistance(100)), Venue.class);
```

To find venues near a `Point` using spherical coordinates the following query can be used

```
Point point = new Point(-73.99171, 40.738868);
List<Venue> venues =
    template.find(new Query(
        Criteria.where("location").nearSphere(point).maxDistance(0.003712240453784)),
        Venue.class);
```

Geo near queries

MongoDB supports querying the database for geo locations and calculation the distance from a given origin at the very same time. With geo-near queries it's possible to express queries like: "find all restaurants in the surrounding 10 miles". To do so `MongoOperations` provides `geoNear(…)` methods taking a `NearQuery` as argument as well as the already familiar entity type and collection

```
Point location = new Point(-73.99171, 40.738868);
NearQuery query = NearQuery.near(location).maxDistance(new Distance(10, Metrics.MILES
));

GeoResults<Restaurant> = operations.geoNear(query, Restaurant.class);
```

As you can see we use the `NearQuery` builder API to set up a query to return all `Restaurant` instances surrounding the given `Point` by 10 miles maximum. The `Metrics` enum used here actually implements an interface so that other metrics could be plugged into a distance as well. A `Metric` is backed by a multiplier to transform the distance value of the given metric into native distances. The sample shown here would consider the 10 to be miles. Using one of the pre-built in metrics (miles and kilometers) will automatically trigger the spherical flag to be set on the query. If you want to avoid that, simply hand in plain `double` values into `maxDistance(…)`. For more information see the JavaDoc of `NearQuery` and `Distance`.

The geo near operations return a `GeoResults` wrapper object that encapsulates `GeoResult` instances. The wrapping `GeoResults` allows accessing the average distance of all results. A single `GeoResult` object simply carries the entity found plus its distance from the origin.

9.6.4. GeoJSON Support

MongoDB supports GeoJSON and simple (legacy) coordinate pairs for geospatial data. Those formats can both be used for storing as well as querying data.

NOTE	Please refer to the MongoDB manual on GeoJSON support to learn about requirements and restrictions.

GeoJSON types in domain classes

Usage of GeoJSON types in domain classes is straight forward. The `org.springframework.data.mongodb.core.geo` package contains types like `GeoJsonPoint`, `GeoJsonPolygon` and others. Those are extensions to the existing `org.springframework.data.geo` types.

```
public class Store {

    String id;

    /**
     * location is stored in GeoJSON format.
     * {
     *   "type" : "Point",
     *   "coordinates" : [ x, y ]
     * }
     */
    GeoJsonPoint location;
}
```

GeoJSON types in repository query methods

Using GeoJSON types as repository query parameters forces usage of the `$geometry` operator when creating the query.

```
public interface StoreRepository extends CrudRepository<Store, String> {

    List<Store> findByLocationWithin(Polygon polygon);  ①

}

/*
 * {
 *   "location": {
 *     "$geoWithin": {
 *       "$geometry": {
 *         "type": "Polygon",
 *         "coordinates": [
 *           [
 *             [-73.992514,40.758934],
 *             [-73.961138,40.760348],
 *             [-73.991658,40.730006],
 *             [-73.992514,40.758934]
 *           ]
 *         ]
 *       }
 *     }
 *   }
 * }
 */
repo.findByLocationWithin(                                  ②
  new GeoJsonPolygon(
    new Point(-73.992514, 40.758934),
    new Point(-73.961138, 40.760348),
    new Point(-73.991658, 40.730006),
    new Point(-73.992514, 40.758934)));                     ③

/*
 * {
 *   "location" : {
 *     "$geoWithin" : {
 *       "$polygon" : [ [-73.992514,40.758934] , [-73.961138,40.760348] , [-
73.991658,40.730006] ]
 *     }
 *   }
 * }
 */
repo.findByLocationWithin(                                  ④
  new Polygon(
    new Point(-73.992514, 40.758934),
    new Point(-73.961138, 40.760348),
    new Point(-73.991658, 40.730006));
```

① Repository method definition using the commons type allows calling it with both GeoJSON and legacy format.

② Use GeoJSON type the make use of `$geometry` operator.

③ Plase note that GeoJSON polygons need the define a closed ring.

④ Use legacy format `$polygon` operator.

9.6.5. Full Text Queries

Since MongoDB 2.6 full text queries can be executed using the `$text` operator. Methods and operations specific for full text queries are available in `TextQuery` and `TextCriteria`. When doing full text search please refer to the MongoDB reference for its behavior and limitations.

Full Text Search

Before we are actually able to use full text search we have to ensure to set up the search index correctly. Please refer to section Text Index for creating index structures.

```
db.foo.createIndex(
{
  title : "text",
  content : "text"
},
{
  weights : {
              title : 3
            }
}
)
```

A query searching for `coffee cake`, sorted by relevance according to the `weights` can be defined and executed as:

```
Query query = TextQuery.searching(new TextCriteria().matchingAny("coffee", "cake"))
.sortByScore();
List<Document> page = template.find(query, Document.class);
```

Exclusion of search terms can directly be done by prefixing the term with `-` or using `notMatching`

```
// search for 'coffee' and not 'cake'
TextQuery.searching(new TextCriteria().matching("coffee").matching("-cake"));
TextQuery.searching(new TextCriteria().matching("coffee").notMatching("cake"));
```

As `TextCriteria.matching` takes the provided term as is. Therefore phrases can be defined by putting them between double quotes (eg. `\"coffee cake\"`) or using `TextCriteria.phrase`.

```
// search for phrase 'coffee cake'
TextQuery.searching(new TextCriteria().matching("\"coffee cake\""));
TextQuery.searching(new TextCriteria().phrase("coffee cake"));
```

The flags for `$caseSensitive` and `$diacriticSensitive` can be set via the according methods on `TextCriteria`. Please note that these two optional flags have been introduced in MongoDB 3.2 and will not be included in the query unless explicitly set.

9.6.6. Collations

MongoDB supports since 3.4 collations for collection and index creation and various query operations. Collations define string comparison rules based on the ICU collations. A collation document consists of various properties that are encapsulated in `Collation`:

```
Collation collation = Collation.of("fr")          ①

  .strength(ComparisonLevel.secondary()           ②
    .includeCase())

  .numericOrderingEnabled()                       ③

  .alternate(Alternate.shifted().punct())         ④

  .forwardDiacriticSort()                         ⑤

  .normalizationEnabled();                        ⑥
```

① `Collation` requires a locale for creation. This can be either a string representation of the locale, a `Locale` (considering language, country and variant) or a `CollationLocale`. The locale is mandatory for creation.

② Collation strength defines comparison levels denoting differences between characters. You can configure various options (case-sensitivity, case-ordering) depending on the selected strength.

③ Specify whether to compare numeric strings as numbers or as strings.

④ Specify whether the collation should consider whitespace and punctuation as base characters for purposes of comparison.

⑤ Specify whether strings with diacritics sort from back of the string, such as with some French dictionary ordering.

⑥ Specify whether to check if text requires normalization and to perform normalization.

Collations can be used to create collections and indexes. If you create a collection specifying a collation, the collation is applied to index creation and queries unless you specify a different collation. A collation is valid for a whole operation and cannot be specified on a per-field basis.

```
Collation french = Collation.of("fr");
Collation german = Collation.of("de");

template.createCollection(Person.class, CollectionOptions.just(collation));

template.indexOps(Person.class).ensureIndex(new Index("name", Direction.ASC).
collation(german));
```

NOTE MongoDB uses simple binary comparison if no collation is specified (`Collation.simple()`).

Using collations with collection operations is a matter of specifying a `Collation` instance in your query or operation options.

Example 63. Using collation with `find`

```
Collation collation = Collation.of("de");

Query query = new Query(Criteria.where("firstName").is("Amél")).collation
(collation);

List<Person> results = template.find(query, Person.class);
```

Example 64. Using collation with `aggregate`

```
Collation collation = Collation.of("de");

AggregationOptions options = new AggregationOptions.Builder().collation(collation
).build();

Aggregation aggregation = newAggregation(
  project("tags"),
  unwind("tags"),
  group("tags")
    .count().as("count")
).withOptions(options);

AggregationResults<TagCount> results = template.aggregate(aggregation, "tags",
TagCount.class);
```

WARNING Indexes are only used if the collation used for the operation and the index collation matches.

9.6.7. Fluent Template API

The `MongoOperations` interface is one of the central components when it comes to more low level interaction with MongoDB. It offers a wide range of methods covering needs from collection / index creation and CRUD operations to more advanced functionality like map-reduce and aggregations. One can find multiple overloads for each and every method. Most of them just cover optional / nullable parts of the API.

`FluentMongoOperations` provide a more narrow interface for common methods of `MongoOperations` providing a more readable, fluent API. The entry points `insert(…)`, `find(…)`, `update(…)`, etc. follow a natural naming schema based on the operation to execute. Moving on from the entry point the API is designed to only offer context dependent methods guiding towards a terminating method that invokes the actual `MongoOperations` counterpart.

```
List<SWCharacter> all = ops.find(SWCharacter.class)
  .inCollection("star-wars")                          ①
  .all();
```

① Skip this step if `SWCharacter` defines the collection via `@Document` or if using the class name as the collection name is just fine.

Sometimes a collection in MongoDB holds entities of different types. Like a `Jedi` within a collection of `SWCharacters`. To use different types for `Query` and return value mapping one can use `as(Class<?> targetType)` map results differently.

```
List<Jedi> all = ops.find(SWCharacter.class)    ①
  .as(Jedi.class)                               ②
  .matching(query(where("jedi").is(true)))
  .all();
```

① The query fields are mapped against the `SWCharacter` type.

② Resulting documents are mapped into `Jedi`.

TIP It is possible to directly apply Projections to resulting documents by providing just the `interface` type via `as(Class<?>)`.

Switching between retrieving a single entity, multiple ones as `List` or `Stream` like is done via the terminating methods `first()`, `one()`, `all()` or `stream()`.

When writing a geo-spatial query via `near(NearQuery)` the number of terminating methods is altered to just the ones valid for executing a `geoNear` command in MongoDB fetching entities as `GeoResult` within `GeoResults`.

```
GeoResults<Jedi> results = mongoOps.query(SWCharacter.class)
  .as(Jedi.class)
  .near(alderaan) // NearQuery.near(-73.9667, 40.78).maxDis···
  .all();
```

9.7. Query by Example

9.7.1. Introduction

This chapter will give you an introduction to Query by Example and explain how to use Examples.

Query by Example (QBE) is a user-friendly querying technique with a simple interface. It allows dynamic query creation and does not require to write queries containing field names. In fact, Query by Example does not require to write queries using store-specific query languages at all.

9.7.2. Usage

The Query by Example API consists of three parts:

- Probe: That is the actual example of a domain object with populated fields.
- `ExampleMatcher`: The `ExampleMatcher` carries details on how to match particular fields. It can be reused across multiple Examples.
- `Example`: An `Example` consists of the probe and the `ExampleMatcher`. It is used to create the query.

Query by Example is suited for several use-cases but also comes with limitations:

When to use

- Querying your data store with a set of static or dynamic constraints
- Frequent refactoring of the domain objects without worrying about breaking existing queries
- Works independently from the underlying data store API

Limitations

- No support for nested/grouped property constraints like `firstname = ?0 or (firstname = ?1 and lastname = ?2)`
- Only supports starts/contains/ends/regex matching for strings and exact matching for other property types

Before getting started with Query by Example, you need to have a domain object. To get started, simply create an interface for your repository:

Example 65. Sample Person object

```
public class Person {

  @Id
  private String id;
  private String firstname;
  private String lastname;
  private Address address;

  // … getters and setters omitted
}
```

This is a simple domain object. You can use it to create an `Example`. By default, fields having `null` values are ignored, and strings are matched using the store specific defaults. Examples can be built by either using the `of` factory method or by using `ExampleMatcher`. `Example` is immutable.

Example 66. Simple Example

```
Person person = new Person();                          ①
person.setFirstname("Dave");                           ②

Example<Person> example = Example.of(person);          ③
```

① Create a new instance of the domain object

② Set the properties to query

③ Create the `Example`

Examples are ideally be executed with repositories. To do so, let your repository interface extend `QueryByExampleExecutor<T>`. Here's an excerpt from the `QueryByExampleExecutor` interface:

Example 67. The `QueryByExampleExecutor`

```
public interface QueryByExampleExecutor<T> {

  <S extends T> S findOne(Example<S> example);

  <S extends T> Iterable<S> findAll(Example<S> example);

  // … more functionality omitted.
}
```

9.7.3. Example matchers

Examples are not limited to default settings. You can specify own defaults for string matching, null handling and property-specific settings using the `ExampleMatcher`.

Example 68. Example matcher with customized matching

```
Person person = new Person();                          ①
person.setFirstname("Dave");                           ②

ExampleMatcher matcher = ExampleMatcher.matching()     ③
  .withIgnorePaths("lastname")                         ④
  .withIncludeNullValues()                             ⑤
  .withStringMatcherEnding();                          ⑥

Example<Person> example = Example.of(person, matcher); ⑦
```

① Create a new instance of the domain object.

② Set properties.

③ Create an `ExampleMatcher` to expect all values to match. It's usable at this stage even without further configuration.

④ Construct a new `ExampleMatcher` to ignore the property path `lastname`.

⑤ Construct a new `ExampleMatcher` to ignore the property path `lastname` and to include null values.

⑥ Construct a new `ExampleMatcher` to ignore the property path `lastname`, to include null values, and use perform suffix string matching.

⑦ Create a new `Example` based on the domain object and the configured `ExampleMatcher`.

By default the `ExampleMatcher` will expect all values set on the probe to match. If you want to get results matching any of the predicates defined implicitly, use `ExampleMatcher.matchingAny()`.

You can specify behavior for individual properties (e.g. "firstname" and "lastname", "address.city" for nested properties). You can tune it with matching options and case sensitivity.

Example 69. Configuring matcher options

```
ExampleMatcher matcher = ExampleMatcher.matching()
  .withMatcher("firstname", endsWith())
  .withMatcher("lastname", startsWith().ignoreCase());
}
```

Another style to configure matcher options is by using Java 8 lambdas. This approach is a callback that asks the implementor to modify the matcher. It's not required to return the matcher because configuration options are held within the matcher instance.

Example 70. Configuring matcher options with lambdas

```
ExampleMatcher matcher = ExampleMatcher.matching()
  .withMatcher("firstname", match -> match.endsWith())
  .withMatcher("firstname", match -> match.startsWith());
}
```

Queries created by `Example` use a merged view of the configuration. Default matching settings can be set at `ExampleMatcher` level while individual settings can be applied to particular property paths. Settings that are set on `ExampleMatcher` are inherited by property path settings unless they are defined explicitly. Settings on a property patch have higher precedence than default settings.

Table 2. Scope of `ExampleMatcher` settings

Setting	Scope
Null-handling	`ExampleMatcher`
String matching	`ExampleMatcher` and property path
Ignoring properties	Property path
Case sensitivity	`ExampleMatcher` and property path
Value transformation	Property path

9.7.4. Executing an example

Example 71. Query by Example using a Repository

```
public interface PersonRepository extends QueryByExampleExecutor<Person> {

}

public class PersonService {

  @Autowired PersonRepository personRepository;

  public List<Person> findPeople(Person probe) {
    return personRepository.findAll(Example.of(probe));
  }
}
```

An `Example` containing an untyped `ExampleSpec` uses the Repository type and its collection name. Typed `ExampleSpec` use their type as result type and the collection name from the Repository.

NOTE	When including `null` values in the `ExampleSpec` Spring Data Mongo uses embedded document matching instead of dot notation property matching. This forces exact document matching for all property values and the property order in the embedded document.

Spring Data MongoDB provides support for the following matching options:

Table 3. `StringMatcher` options

Matching	Logical result
`DEFAULT` (case-sensitive)	`{"firstname" : firstname}`
`DEFAULT` (case-insensitive)	`{"firstname" : { $regex: firstname, $options: 'i'}}`
`EXACT` (case-sensitive)	`{"firstname" : { $regex: /^firstname$/}}`
`EXACT` (case-insensitive)	`{"firstname" : { $regex: /^firstname$/, $options: 'i'}}`
`STARTING` (case-sensitive)	`{"firstname" : { $regex: /^firstname/}}`
`STARTING` (case-insensitive)	`{"firstname" : { $regex: /^firstname/, $options: 'i'}}`
`ENDING` (case-sensitive)	`{"firstname" : { $regex: /firstname$/}}`
`ENDING` (case-insensitive)	`{"firstname" : { $regex: /firstname$/, $options: 'i'}}`
`CONTAINING` (case-sensitive)	`{"firstname" : { $regex: /.*firstname.*/}}`
`CONTAINING` (case-insensitive)	`{"firstname" : { $regex: /.*firstname.*/, $options: 'i'}}`
`REGEX` (case-sensitive)	`{"firstname" : { $regex: /firstname/}}`
`REGEX` (case-insensitive)	`{"firstname" : { $regex: /firstname/, $options: 'i'}}`

9.7.5. Untyped Example

By default `Example` is strictly typed. This means the mapped query will have a type match included restricting it to probe assignable types. Eg. when sticking with the default type key `_class` the query has restrictions like `_class : { $in : [ com.acme.Person] }`.

By using the `UntypedExampleMatcher` it is possible bypasses the default behavior and skip the type restriction. So as long as field names match nearly any domain type can be used as the probe for creating the reference.

Example 72. Untyped Example Query

```
class JustAnArbitraryClassWithMatchingFieldName {
  @Field("lastname") String value;
}

JustAnArbitraryClassWithMatchingFieldNames probe = new
JustAnArbitraryClassWithMatchingFieldNames();
probe.value = "stark";

Example example = Example.of(probe, UntypedExampleMatcher.matching());

Query query = new Query(new Criteria().alike(example));
List<Person> result = template.find(query, Person.class);
```

9.8. Map-Reduce Operations

You can query MongoDB using Map-Reduce which is useful for batch processing, data aggregation, and for when the query language doesn't fulfill your needs.

Spring provides integration with MongoDB's map reduce by providing methods on MongoOperations to simplify the creation and execution of Map-Reduce operations. It can convert the results of a Map-Reduce operation to a POJO also integrates with Spring's Resource abstraction abstraction. This will let you place your JavaScript files on the file system, classpath, http server or any other Spring Resource implementation and then reference the JavaScript resources via an easy URI style syntax, e.g. 'classpath:reduce.js;. Externalizing JavaScript code in files is often preferable to embedding them as Java strings in your code. Note that you can still pass JavaScript code as Java strings if you prefer.

9.8.1. Example Usage

To understand how to perform Map-Reduce operations an example from the book 'MongoDB - The definitive guide' is used. In this example we will create three documents that have the values [a,b], [b,c], and [c,d] respectfully. The values in each document are associated with the key 'x' as shown below. For this example assume these documents are in the collection named "jmr1".

```
{ "_id" : ObjectId("4e5ff893c0277826074ec533"), "x" : [ "a", "b" ] }
{ "_id" : ObjectId("4e5ff893c0277826074ec534"), "x" : [ "b", "c" ] }
{ "_id" : ObjectId("4e5ff893c0277826074ec535"), "x" : [ "c", "d" ] }
```

A map function that will count the occurrence of each letter in the array for each document is shown below

```
function () {
    for (var i = 0; i < this.x.length; i++) {
        emit(this.x[i], 1);
    }
}
```

The reduce function that will sum up the occurrence of each letter across all the documents is shown below

```
function (key, values) {
    var sum = 0;
    for (var i = 0; i < values.length; i++)
        sum += values[i];
    return sum;
}
```

Executing this will result in a collection as shown below.

```
{ "_id" : "a", "value" : 1 }
{ "_id" : "b", "value" : 2 }
{ "_id" : "c", "value" : 2 }
{ "_id" : "d", "value" : 1 }
```

Assuming that the map and reduce functions are located in `map.js` and `reduce.js` and bundled in your jar so they are available on the classpath, you can execute a map-reduce operation and obtain the results as shown below

```
MapReduceResults<ValueObject> results = mongoOperations.mapReduce("jmr1",
"classpath:map.js", "classpath:reduce.js", ValueObject.class);
for (ValueObject valueObject : results) {
  System.out.println(valueObject);
}
```

The output of the above code is

```
ValueObject [id=a, value=1.0]
ValueObject [id=b, value=2.0]
ValueObject [id=c, value=2.0]
ValueObject [id=d, value=1.0]
```

The MapReduceResults class implements `Iterable` and provides access to the raw output, as well as timing and count statistics. The `ValueObject` class is simply

```
public class ValueObject {

  private String id;
  private float value;

  public String getId() {
    return id;
  }

  public float getValue() {
    return value;
  }

  public void setValue(float value) {
    this.value = value;
  }

  @Override
  public String toString() {
    return "ValueObject [id=" + id + ", value=" + value + "]";
  }
}
```

By default the output type of INLINE is used so you don't have to specify an output collection. To specify additional map-reduce options use an overloaded method that takes an additional `MapReduceOptions` argument. The class `MapReduceOptions` has a fluent API so adding additional options can be done in a very compact syntax. Here an example that sets the output collection to "jmr1_out". Note that setting only the output collection assumes a default output type of REPLACE.

```
MapReduceResults<ValueObject> results = mongoOperations.mapReduce("jmr1",
"classpath:map.js", "classpath:reduce.js",
                                                                  new
MapReduceOptions().outputCollection("jmr1_out"), ValueObject.class);
```

There is also a static import `import static org.springframework.data.mongodb.core.mapreduce.MapReduceOptions.options;` that can be used to make the syntax slightly more compact

```
MapReduceResults<ValueObject> results = mongoOperations.mapReduce("jmr1",
"classpath:map.js", "classpath:reduce.js",
                                                              options()
.outputCollection("jmr1_out"), ValueObject.class);
```

You can also specify a query to reduce the set of data that will be used to feed into the map-reduce operation. This will remove the document that contains [a,b] from consideration for map-reduce operations.

```
Query query = new Query(where("x").ne(new String[] { "a", "b" }));
MapReduceResults<ValueObject> results = mongoOperations.mapReduce(query, "jmr1",
"classpath:map.js", "classpath:reduce.js",
                                                                    options()
.outputCollection("jmr1_out"), ValueObject.class);
```

Note that you can specify additional limit and sort values as well on the query but not skip values.

9.9. Script Operations

MongoDB allows executing JavaScript functions on the server by either directly sending the script or calling a stored one. `ScriptOperations` can be accessed via `MongoTemplate` and provides basic abstraction for `JavaScript` usage.

9.9.1. Example Usage

```
ScriptOperations scriptOps = template.scriptOps();

ExecutableMongoScript echoScript = new ExecutableMongoScript("function(x) { return
x; }");
scriptOps.execute(echoScript, "directly execute script");     ①

scriptOps.register(new NamedMongoScript("echo", echoScript)); ②
scriptOps.call("echo", "execute script via name");            ③
```

① Execute the script directly without storing the function on server side.

② Store the script using 'echo' as its name. The given name identifies the script and allows calling it later.

③ Execute the script with name 'echo' using the provided parameters.

9.10. Group Operations

As an alternative to using Map-Reduce to perform data aggregation, you can use the `group` operation which feels similar to using SQL's group by query style, so it may feel more approachable vs. using Map-Reduce. Using the group operations does have some limitations, for example it is not supported in a shared environment and it returns the full result set in a single BSON object, so the result should be small, less than 10,000 keys.

Spring provides integration with MongoDB's group operation by providing methods on MongoOperations to simplify the creation and execution of group operations. It can convert the results of the group operation to a POJO and also integrates with Spring's Resource abstraction abstraction. This will let you place your JavaScript files on the file system, classpath, http server or any other Spring Resource implementation and then reference the JavaScript resources via an easy URI style syntax, e.g. 'classpath:reduce.js;. Externalizing JavaScript code in files if often preferable

to embedding them as Java strings in your code. Note that you can still pass JavaScript code as Java strings if you prefer.

9.10.1. Example Usage

In order to understand how group operations work the following example is used, which is somewhat artificial. For a more realistic example consult the book 'MongoDB - The definitive guide'. A collection named `group_test_collection` created with the following rows.

```
{ "_id" : ObjectId("4ec1d25d41421e2015da64f1"), "x" : 1 }
{ "_id" : ObjectId("4ec1d25d41421e2015da64f2"), "x" : 1 }
{ "_id" : ObjectId("4ec1d25d41421e2015da64f3"), "x" : 2 }
{ "_id" : ObjectId("4ec1d25d41421e2015da64f4"), "x" : 3 }
{ "_id" : ObjectId("4ec1d25d41421e2015da64f5"), "x" : 3 }
{ "_id" : ObjectId("4ec1d25d41421e2015da64f6"), "x" : 3 }
```

We would like to group by the only field in each row, the `x` field and aggregate the number of times each specific value of `x` occurs. To do this we need to create an initial document that contains our count variable and also a reduce function which will increment it each time it is encountered. The Java code to execute the group operation is shown below

```
GroupByResults<XObject> results = mongoTemplate.group("group_test_collection",
                                                       GroupBy.key("x").
initialDocument("{ count: 0 }").reduceFunction("function(doc, prev) { prev.count += 1
}"),
                                                       XObject.class);
```

The first argument is the name of the collection to run the group operation over, the second is a fluent API that specifies properties of the group operation via a `GroupBy` class. In this example we are using just the `intialDocument` and `reduceFunction` methods. You can also specify a key-function, as well as a finalizer as part of the fluent API. If you have multiple keys to group by, you can pass in a comma separated list of keys.

The raw results of the group operation is a JSON document that looks like this

```
{
  "retval" : [ { "x" : 1.0 , "count" : 2.0} ,
               { "x" : 2.0 , "count" : 1.0} ,
               { "x" : 3.0 , "count" : 3.0} ] ,
  "count" : 6.0 ,
  "keys" : 3 ,
  "ok" : 1.0
}
```

The document under the "retval" field is mapped onto the third argument in the group method, in this case XObject which is shown below.

```
public class XObject {

  private float x;

  private float count;


  public float getX() {
    return x;
  }

  public void setX(float x) {
    this.x = x;
  }

  public float getCount() {
    return count;
  }

  public void setCount(float count) {
    this.count = count;
  }

  @Override
  public String toString() {
    return "XObject [x=" + x + " count = " + count + "]";
  }
}
```

You can also obtain the raw result as a `Document` by calling the method `getRawResults` on the `GroupByResults` class.

There is an additional method overload of the group method on `MongoOperations` which lets you specify a `Criteria` object for selecting a subset of the rows. An example which uses a `Criteria` object, with some syntax sugar using static imports, as well as referencing a key-function and reduce function javascript files via a Spring Resource string is shown below.

```
import static org.springframework.data.mongodb.core.mapreduce.GroupBy.keyFunction;
import static org.springframework.data.mongodb.core.query.Criteria.where;

GroupByResults<XObject> results = mongoTemplate.group(where("x").gt(0),
                                        "group_test_collection",

keyFunction("classpath:keyFunction.js").initialDocument("{ count: 0
}").reduceFunction("classpath:groupReduce.js"), XObject.class);
```

9.11. Aggregation Framework Support

Spring Data MongoDB provides support for the Aggregation Framework introduced to MongoDB in version 2.2.

The MongoDB Documentation describes the Aggregation Framework as follows:

For further information see the full reference documentation of the aggregation framework and other data aggregation tools for MongoDB.

9.11.1. Basic Concepts

The Aggregation Framework support in Spring Data MongoDB is based on the following key abstractions `Aggregation`, `AggregationOperation` and `AggregationResults`.

- `Aggregation`

  An Aggregation represents a MongoDB `aggregate` operation and holds the description of the aggregation pipeline instructions. Aggregations are created by invoking the appropriate `newAggregation(…)` static factory Method of the `Aggregation` class which takes the list of `AggregateOperation` as a parameter next to the optional input class.

  The actual aggregate operation is executed by the `aggregate` method of the `MongoTemplate` which also takes the desired output class as parameter.

- `AggregationOperation`

  An `AggregationOperation` represents a MongoDB aggregation pipeline operation and describes the processing that should be performed in this aggregation step. Although one could manually create an `AggregationOperation` the recommended way to construct an `AggregateOperation` is to use the static factory methods provided by the `Aggregate` class.

- `AggregationResults`

  `AggregationResults` is the container for the result of an aggregate operation. It provides access to the raw aggregation result in the form of an `Document`, to the mapped objects and information which performed the aggregation.

  The canonical example for using the Spring Data MongoDB support for the MongoDB Aggregation Framework looks as follows:

```
import static org.springframework.data.mongodb.core.aggregation.Aggregation.*;

Aggregation agg = newAggregation(
    pipelineOP1(),
    pipelineOP2(),
    pipelineOPn()
);

AggregationResults<OutputType> results = mongoTemplate.aggregate(agg,
"INPUT_COLLECTION_NAME", OutputType.class);
List<OutputType> mappedResult = results.getMappedResults();
```

Note that if you provide an input class as the first parameter to the `newAggregation` method the `MongoTemplate` will derive the name of the input collection from this class. Otherwise if you don't not specify an input class you must provide the name of the input collection explicitly. If an input-class and an input-collection is provided the latter takes precedence.

9.11.2. Supported Aggregation Operations

The MongoDB Aggregation Framework provides the following types of Aggregation Operations:

- Pipeline Aggregation Operators
- Group Aggregation Operators
- Boolean Aggregation Operators
- Comparison Aggregation Operators
- Arithmetic Aggregation Operators
- String Aggregation Operators
- Date Aggregation Operators
- Array Aggregation Operators
- Conditional Aggregation Operators
- Lookup Aggregation Operators

At the time of this writing we provide support for the following Aggregation Operations in Spring Data MongoDB.

Table 4. Aggregation Operations currently supported by Spring Data MongoDB

Pipeline Aggregation Operators	bucket, bucketAuto, count, facet, geoNear, graphLookup, group, limit, lookup, match, project, replaceRoot, skip, sort, unwind
Set Aggregation Operators	setEquals, setIntersection, setUnion, setDifference, setIsSubset, anyElementTrue, allElementsTrue
Group Aggregation Operators	addToSet, first, last, max, min, avg, push, sum, (*count), stdDevPop, stdDevSamp

Arithmetic Aggregation Operators	abs, add (*via plus), ceil, divide, exp, floor, ln, log, log10, mod, multiply, pow, sqrt, subtract (*via minus), trunc
String Aggregation Operators	concat, substr, toLower, toUpper, stcasecmp, indexOfBytes, indexOfCP, split, strLenBytes, strLenCP, substrCP,
Comparison Aggregation Operators	eq (*via: is), gt, gte, lt, lte, ne
Array Aggregation Operators	arrayElementAt, concatArrays, filter, in, indexOfArray, isArray, range, reverseArray, reduce, size, slice, zip
Literal Operators	literal
Date Aggregation Operators	dayOfYear, dayOfMonth, dayOfWeek, year, month, week, hour, minute, second, millisecond, dateToString, isoDayOfWeek, isoWeek, isoWeekYear
Variable Operators	map
Conditional Aggregation Operators	cond, ifNull, switch
Type Aggregation Operators	type

Note that the aggregation operations not listed here are currently not supported by Spring Data MongoDB. Comparison aggregation operators are expressed as `Criteria` expressions.

*) The operation is mapped or added by Spring Data MongoDB.

9.11.3. Projection Expressions

Projection expressions are used to define the fields that are the outcome of a particular aggregation step. Projection expressions can be defined via the `project` method of the `Aggregation` class either by passing a list of `String`'s or an aggregation framework `Fields` object. The projection can be extended with additional fields through a fluent API via the `and(String)` method and aliased via the `as(String)` method. Note that one can also define fields with aliases via the static factory method `Fields.field` of the aggregation framework that can then be used to construct a new `Fields` instance. References to projected fields in later aggregation stages are only valid by using the field name of included fields or their alias of aliased or newly defined fields. Fields not included in the projection cannot be referenced in later aggregation stages.

Example 73. Projection expression examples

```
// will generate {$project: {name: 1, netPrice: 1}}
project("name", "netPrice")

// will generate {$project: {bar: $foo}}
project().and("foo").as("bar")

// will generate {$project: {a: 1, b: 1, bar: $foo}}
project("a","b").and("foo").as("bar")
```

Example 74. Multi-Stage Aggregation using Projection and Sorting

```
// will generate {$project: {name: 1, netPrice: 1}}, {$sort: {name: 1}}
project("name", "netPrice"), sort(ASC, "name")

// will generate {$project: {bar: $foo}}, {$sort: {bar: 1}}
project().and("foo").as("bar"), sort(ASC, "bar")

// this will not work
project().and("foo").as("bar"), sort(ASC, "foo")
```

More examples for project operations can be found in the `AggregationTests` class. Note that further details regarding the projection expressions can be found in the corresponding section of the MongoDB Aggregation Framework reference documentation.

9.11.4. Faceted classification

MongoDB supports as of Version 3.4 faceted classification using the Aggregation Framework. A faceted classification uses semantic categories, either general or subject-specific, that are combined to create the full classification entry. Documents flowing through the aggregation pipeline are classificated into buckets. A multi-faceted classification enables various aggregations on the same set of input documents, without needing to retrieve the input documents multiple times.

Buckets

Bucket operations categorize incoming documents into groups, called buckets, based on a specified expression and bucket boundaries. Bucket operations require a grouping field or grouping expression. They can be defined via the `bucket()`/`bucketAuto()` methods of the `Aggregate` class. `BucketOperation` and `BucketAutoOperation` can expose accumulations based on aggregation expressions for input documents. The bucket operation can be extended with additional parameters through a fluent API via the `with…()` methods, the `andOutput(String)` method and aliased via the `as(String)` method. Each bucket is represented as a document in the output.

`BucketOperation` takes a defined set of boundaries to group incoming documents into these categories. Boundaries are required to be sorted.

Example 75. Bucket operation examples

```
// will generate {$bucket: {groupBy: $price, boundaries: [0, 100, 400]}}
bucket("price").withBoundaries(0, 100, 400);

// will generate {$bucket: {groupBy: $price, default: "Other" boundaries: [0,
100]}}
bucket("price").withBoundaries(0, 100).withDefault("Other");

// will generate {$bucket: {groupBy: $price, boundaries: [0, 100], output: {
count: { $sum: 1}}}}
bucket("price").withBoundaries(0, 100).andOutputCount().as("count");

// will generate {$bucket: {groupBy: $price, boundaries: [0, 100], 5, output: {
titles: { $push: "$title"}}}
bucket("price").withBoundaries(0, 100).andOutput("title").push().as("titles");
```

`BucketAutoOperation` determines boundaries itself in an attempt to evenly distribute documents into a specified number of buckets. `BucketAutoOperation` optionally takes a granularity specifies the preferred number series to use to ensure that the calculated boundary edges end on preferred round numbers or their powers of 10.

Example 76. Bucket operation examples

```
// will generate {$bucketAuto: {groupBy: $price, buckets: 5}}
bucketAuto("price", 5)

// will generate {$bucketAuto: {groupBy: $price, buckets: 5, granularity: "E24"}}
bucketAuto("price", 5).withGranularity(Granularities.E24).withDefault("Other");

// will generate {$bucketAuto: {groupBy: $price, buckets: 5, output: { titles: {
$push: "$title"}}}
bucketAuto("price", 5).andOutput("title").push().as("titles");
```

Bucket operations can use `AggregationExpression` via `andOutput()` and SpEL expressions via `andOutputExpression()` to create output fields in buckets.

Note that further details regarding bucket expressions can be found in the `$bucket` section and `$bucketAuto` section of the MongoDB Aggregation Framework reference documentation.

Multi-faceted aggregation

Multiple aggregation pipelines can be used to create multi-faceted aggregations which characterize data across multiple dimensions, or facets, within a single aggregation stage. Multi-faceted aggregations provide multiple filters and categorizations to guide data browsing and analysis. A common implementation of faceting is how many online retailers provide ways to narrow down

search results by applying filters on product price, manufacturer, size, etc.

A `FacetOperation` can be defined via the `facet()` method of the `Aggregation` class. It can be customized with multiple aggregation pipelines via the `and()` method. Each sub-pipeline has its own field in the output document where its results are stored as an array of documents.

Sub-pipelines can project and filter input documents prior grouping. Common cases are extraction of date parts or calculations before categorization.

Example 77. Facet operation examples

```
// will generate {$facet: {categorizedByPrice: [ { $match: { price: {$exists :
true}}}, { $bucketAuto: {groupBy: $price, buckets: 5}}]}}
facet(match(Criteria.where("price").exists(true)), bucketAuto("price", 5)).as(
"categorizedByPrice"))

// will generate {$facet: {categorizedByYear: [
//                           { $project: { title: 1, publicationYear: { $year:
"publicationDate"}}},
//                           { $bucketAuto: {groupBy: $price, buckets: 5, output:
{ titles: {$push:"$title"}}}
//                         ]}}
facet(project("title").and("publicationDate").extractYear().as("publicationYear"),
      bucketAuto("publicationYear", 5).andOutput("title").push().as("titles"))
  .as("categorizedByYear"))
```

Note that further details regarding facet operation can be found in the `$facet` section of the MongoDB Aggregation Framework reference documentation.

Spring Expression Support in Projection Expressions

We support the use of SpEL expression in projection expressions via the `andExpression` method of the `ProjectionOperation` and `BucketOperation` classes. This allows you to define the desired expression as a SpEL expression which is translated into a corresponding MongoDB projection expression part on query execution. This makes it much easier to express complex calculations.

Complex calculations with SpEL expressions

The following SpEL expression:

```
1 + (q + 1) / (q - 1)
```

will be translated into the following projection expression part:

```
{ "$add" : [ 1, {
    "$divide" : [ {
        "$add":["$q", 1]}, {
        "$subtract":[ "$q", 1]}
    ]
}]}
```

Have a look at an example in more context in Aggregation Framework Example 5 and Aggregation Framework Example 6. You can find more usage examples for supported SpEL expression constructs in `SpelExpressionTransformerUnitTests`.

Table 5. Supported SpEL transformations

a == b	{ $eq : [$a, $b] }
a != b	{ $ne : [$a , $b] }
a > b	{ $gt : [$a, $b] }
a >= b	{ $gte : [$a, $b] }
a < b	{ $lt : [$a, $b] }
a <= b	{ $lte : [$a, $b] }
a + b	{ $add : [$a, $b] }
a - b	{ $subtract : [$a, $b] }
a * b	{ $multiply : [$a, $b] }
a / b	{ $divide : [$a, $b] }
a^b	{ $pow : [$a, $b] }
a % b	{ $mod : [$a, $b] }
a && b	{ $and : [$a, $b] }
a || b	{ $or : [$a, $b] }
!a	{ $not : [$a] }

Next to the transformations shown in Supported SpEL transformations it is possible to use standard SpEL operations like `new` to eg. create arrays and reference expressions via their name followed by the arguments to use in brackets.

```
// { $setEquals : [$a, [5, 8, 13] ] }
.andExpression("setEquals(a, new int[]{5, 8, 13})");
```

Aggregation Framework Examples

The following examples demonstrate the usage patterns for the MongoDB Aggregation Framework with Spring Data MongoDB.

Aggregation Framework Example 1

In this introductory example we want to aggregate a list of tags to get the occurrence count of a

particular tag from a MongoDB collection called `"tags"` sorted by the occurrence count in descending order. This example demonstrates the usage of grouping, sorting, projections (selection) and unwinding (result splitting).

```
class TagCount {
 String tag;
 int n;
}
```

```
import static org.springframework.data.mongodb.core.aggregation.Aggregation.*;

Aggregation agg = newAggregation(
    project("tags"),
    unwind("tags"),
    group("tags").count().as("n"),
    project("n").and("tag").previousOperation(),
    sort(DESC, "n")
);

AggregationResults<TagCount> results = mongoTemplate.aggregate(agg, "tags", TagCount
.class);
List<TagCount> tagCount = results.getMappedResults();
```

- In order to do this we first create a new aggregation via the `newAggregation` static factory method to which we pass a list of aggregation operations. These aggregate operations define the aggregation pipeline of our `Aggregation`.
- As a second step we select the `"tags"` field (which is an array of strings) from the input collection with the `project` operation.
- In a third step we use the `unwind` operation to generate a new document for each tag within the `"tags"` array.
- In the forth step we use the `group` operation to define a group for each `"tags"`-value for which we aggregate the occurrence count via the `count` aggregation operator and collect the result in a new field called `"n"`.
- As a fifth step we select the field `"n"` and create an alias for the id-field generated from the previous group operation (hence the call to `previousOperation()`) with the name `"tag"`.
- As the sixth step we sort the resulting list of tags by their occurrence count in descending order via the `sort` operation.
- Finally we call the `aggregate` Method on the MongoTemplate in order to let MongoDB perform the actual aggregation operation with the created `Aggregation` as an argument.

Note that the input collection is explicitly specified as the `"tags"` parameter to the `aggregate` Method. If the name of the input collection is not specified explicitly, it is derived from the input-class passed as first parameter to the `newAggreation` Method.

Aggregation Framework Example 2

This example is based on the Largest and Smallest Cities by State example from the MongoDB Aggregation Framework documentation. We added additional sorting to produce stable results with different MongoDB versions. Here we want to return the smallest and largest cities by population for each state, using the aggregation framework. This example demonstrates the usage of grouping, sorting and projections (selection).

```
class ZipInfo {
   String id;
   String city;
   String state;
   @Field("pop") int population;
   @Field("loc") double[] location;
}

class City {
   String name;
   int population;
}

class ZipInfoStats {
   String id;
   String state;
   City biggestCity;
   City smallestCity;
}
```

```
import static org.springframework.data.mongodb.core.aggregation.Aggregation.*;

TypedAggregation<ZipInfo> aggregation = newAggregation(ZipInfo.class,
    group("state", "city")
       .sum("population").as("pop"),
    sort(ASC, "pop", "state", "city"),
    group("state")
       .last("city").as("biggestCity")
       .last("pop").as("biggestPop")
       .first("city").as("smallestCity")
       .first("pop").as("smallestPop"),
    project()
       .and("state").previousOperation()
       .and("biggestCity")
          .nested(bind("name", "biggestCity").and("population", "biggestPop"))
       .and("smallestCity")
          .nested(bind("name", "smallestCity").and("population", "smallestPop")),
    sort(ASC, "state")
);

AggregationResults<ZipInfoStats> result = mongoTemplate.aggregate(aggregation,
ZipInfoStats.class);
ZipInfoStats firstZipInfoStats = result.getMappedResults().get(0);
```

- The class `ZipInfo` maps the structure of the given input-collection. The class `ZipInfoStats` defines the structure in the desired output format.
- As a first step we use the `group` operation to define a group from the input-collection. The grouping criteria is the combination of the fields `"state"` and `"city"` which forms the id structure of the group. We aggregate the value of the `"population"` property from the grouped elements with by using the `sum` operator saving the result in the field `"pop"`.
- In a second step we use the `sort` operation to sort the intermediate-result by the fields `"pop"`, `"state"` and `"city"` in ascending order, such that the smallest city is at the top and the biggest city is at the bottom of the result. Note that the sorting on `"state"` and `"city"` is implicitly performed against the group id fields which Spring Data MongoDB took care of.
- In the third step we use a `group` operation again to group the intermediate result by `"state"`. Note that `"state"` again implicitly references an group-id field. We select the name and the population count of the biggest and smallest city with calls to the `last(…)` and `first(…)` operator respectively via the `project` operation.
- As the forth step we select the `"state"` field from the previous `group` operation. Note that `"state"` again implicitly references an group-id field. As we do not want an implicitly generated id to appear, we exclude the id from the previous operation via `and(previousOperation()).exclude()`. As we want to populate the nested `City` structures in our output-class accordingly we have to emit appropriate sub-documents with the nested method.
- Finally as the fifth step we sort the resulting list of `StateStats` by their state name in ascending order via the `sort` operation.

Note that we derive the name of the input-collection from the `ZipInfo`-class passed as first parameter to the `newAggregation`-Method.

Aggregation Framework Example 3

This example is based on the States with Populations Over 10 Million example from the MongoDB Aggregation Framework documentation. We added additional sorting to produce stable results with different MongoDB versions. Here we want to return all states with a population greater than 10 million, using the aggregation framework. This example demonstrates the usage of grouping, sorting and matching (filtering).

```
class StateStats {
   @Id String id;
   String state;
   @Field("totalPop") int totalPopulation;
}
```

```
import static org.springframework.data.mongodb.core.aggregation.Aggregation.*;

TypedAggregation<ZipInfo> agg = newAggregation(ZipInfo.class,
    group("state").sum("population").as("totalPop"),
    sort(ASC, previousOperation(), "totalPop"),
    match(where("totalPop").gte(10 * 1000 * 1000))
);

AggregationResults<StateStats> result = mongoTemplate.aggregate(agg, StateStats.class
);
List<StateStats> stateStatsList = result.getMappedResults();
```

- As a first step we group the input collection by the `"state"` field and calculate the sum of the `"population"` field and store the result in the new field `"totalPop"`.
- In the second step we sort the intermediate result by the id-reference of the previous group operation in addition to the `"totalPop"` field in ascending order.
- Finally in the third step we filter the intermediate result by using a `match` operation which accepts a `Criteria` query as an argument.

Note that we derive the name of the input-collection from the `ZipInfo`-class passed as first parameter to the `newAggregation`-Method.

Aggregation Framework Example 4

This example demonstrates the use of simple arithmetic operations in the projection operation.

```
class Product {
    String id;
    String name;
    double netPrice;
    int spaceUnits;
}
```

```
import static org.springframework.data.mongodb.core.aggregation.Aggregation.*;

TypedAggregation<Product> agg = newAggregation(Product.class,
    project("name", "netPrice")
        .and("netPrice").plus(1).as("netPricePlus1")
        .and("netPrice").minus(1).as("netPriceMinus1")
        .and("netPrice").multiply(1.19).as("grossPrice")
        .and("netPrice").divide(2).as("netPriceDiv2")
        .and("spaceUnits").mod(2).as("spaceUnitsMod2")
);

AggregationResults<Document> result = mongoTemplate.aggregate(agg, Document.class);
List<Document> resultList = result.getMappedResults();
```

Note that we derive the name of the input-collection from the `Product`-class passed as first parameter to the `newAggregation`-Method.

Aggregation Framework Example 5

This example demonstrates the use of simple arithmetic operations derived from SpEL Expressions in the projection operation.

```
class Product {
    String id;
    String name;
    double netPrice;
    int spaceUnits;
}
```

```
import static org.springframework.data.mongodb.core.aggregation.Aggregation.*;

TypedAggregation<Product> agg = newAggregation(Product.class,
    project("name", "netPrice")
        .andExpression("netPrice + 1").as("netPricePlus1")
        .andExpression("netPrice - 1").as("netPriceMinus1")
        .andExpression("netPrice / 2").as("netPriceDiv2")
        .andExpression("netPrice * 1.19").as("grossPrice")
        .andExpression("spaceUnits % 2").as("spaceUnitsMod2")
        .andExpression("(netPrice * 0.8  + 1.2) * 1.19").as(
"grossPriceIncludingDiscountAndCharge")

);

AggregationResults<Document> result = mongoTemplate.aggregate(agg, Document.class);
List<Document> resultList = result.getMappedResults();
```

Aggregation Framework Example 6

This example demonstrates the use of complex arithmetic operations derived from SpEL Expressions in the projection operation.

Note: The additional parameters passed to the `addExpression` Method can be referenced via indexer expressions according to their position. In this example we reference the parameter which is the first parameter of the parameters array via `[0]`. External parameter expressions are replaced with their respective values when the SpEL expression is transformed into a MongoDB aggregation framework expression.

```
class Product {
    String id;
    String name;
    double netPrice;
    int spaceUnits;
}
```

```
import static org.springframework.data.mongodb.core.aggregation.Aggregation.*;

double shippingCosts = 1.2;

TypedAggregation<Product> agg = newAggregation(Product.class,
    project("name", "netPrice")
        .andExpression("(netPrice * (1-discountRate)  + [0]) * (1+taxRate)",
shippingCosts).as("salesPrice")
);

AggregationResults<Document> result = mongoTemplate.aggregate(agg, Document.class);
List<Document> resultList = result.getMappedResults();
```

Note that we can also refer to other fields of the document within the SpEL expression.

Aggregation Framework Example 7

This example uses conditional projection. It's derived from the $cond reference documentation.

```
public class InventoryItem {

  @Id int id;
  String item;
  String description;
  int qty;
}

public class InventoryItemProjection {

  @Id int id;
  String item;
  String description;
  int qty;
  int discount
}
```

```
import static org.springframework.data.mongodb.core.aggregation.Aggregation.*;

TypedAggregation<InventoryItem> agg = newAggregation(InventoryItem.class,
  project("item").and("discount")
    .applyCondition(ConditionalOperator.newBuilder().when(Criteria.where("qty").gte
(250))
      .then(30)
      .otherwise(20))
    .and(ifNull("description", "Unspecified")).as("description")
);

AggregationResults<InventoryItemProjection> result = mongoTemplate.aggregate(agg,
"inventory", InventoryItemProjection.class);
List<InventoryItemProjection> stateStatsList = result.getMappedResults();
```

- This one-step aggregation uses a projection operation with the `inventory` collection. We project the `discount` field using a conditional operation for all inventory items that have a `qty` greater or equal to `250`. A second conditional projection is performed for the `description` field. We apply the description `Unspecified` to all items that either do not have a `description` field of items that have a `null` description.

9.12. Overriding default mapping with custom converters

In order to have more fine-grained control over the mapping process you can register Spring

converters with the `MongoConverter` implementations such as the `MappingMongoConverter`.

The `MappingMongoConverter` checks to see if there are any Spring converters that can handle a specific class before attempting to map the object itself. To 'hijack' the normal mapping strategies of the `MappingMongoConverter`, perhaps for increased performance or other custom mapping needs, you first need to create an implementation of the Spring `Converter` interface and then register it with the MappingConverter.

NOTE	For more information on the Spring type conversion service see the reference docs here.

9.12.1. Saving using a registered Spring Converter

An example implementation of the `Converter` that converts from a Person object to a `org.bson.Document` is shown below

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

import org.bson.Document;

public class PersonWriteConverter implements Converter<Person, Document> {

  public Document convert(Person source) {
    Document document = new Document();
    document.put("_id", source.getId());
    document.put("name", source.getFirstName());
    document.put("age", source.getAge());
    return document;
  }
}
```

9.12.2. Reading using a Spring Converter

An example implementation of a Converter that converts from a Document to a Person object is shown below.

```
public class PersonReadConverter implements Converter<Document, Person> {

  public Person convert(Document source) {
    Person p = new Person((ObjectId) source.get("_id"), (String) source.get("name"));
    p.setAge((Integer) source.get("age"));
    return p;
  }
}
```

9.12.3. Registering Spring Converters with the MongoConverter

The Mongo Spring namespace provides a convenience way to register Spring `Converter` s with the `MappingMongoConverter`. The configuration snippet below shows how to manually register converter beans as well as configuring the wrapping `MappingMongoConverter` into a `MongoTemplate`.

```
<mongo:db-factory dbname="database"/>

<mongo:mapping-converter>
  <mongo:custom-converters>
    <mongo:converter ref="readConverter"/>
    <mongo:converter>
      <bean class="org.springframework.data.mongodb.test.PersonWriteConverter"/>
    </mongo:converter>
  </mongo:custom-converters>
</mongo:mapping-converter>

<bean id="readConverter" class=
"org.springframework.data.mongodb.test.PersonReadConverter"/>

<bean id="mongoTemplate" class="org.springframework.data.mongodb.core.MongoTemplate">
  <constructor-arg name="mongoDbFactory" ref="mongoDbFactory"/>
  <constructor-arg name="mongoConverter" ref="mappingConverter"/>
</bean>
```

You can also use the base-package attribute of the custom-converters element to enable classpath scanning for all `Converter` and `GenericConverter` implementations below the given package.

```
<mongo:mapping-converter>
  <mongo:custom-converters base-package="com.acme.**.converters" />
</mongo:mapping-converter>
```

9.12.4. 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 MongoDB can handle natively we will register the converter instance as reading or writing one. Have a look at the following samples:

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

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

In case you write a `Converter` whose source and target type are native Mongo types there's no way for us to determine whether we should consider it as reading or writing converter. Registering the converter instance as both might lead to unwanted results then. E.g. 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 be generally able to force the infrastructure to register a converter for one way only we provide `@ReadingConverter` as well as `@WritingConverter` to be used in the converter implementation.

9.13. Index and Collection management

`MongoTemplate` provides a few methods for managing indexes and collections. These are collected into a helper interface called `IndexOperations`. You access these operations by calling the method `indexOps` and pass in either the collection name or the `java.lang.Class` of your entity (the collection name will be derived from the .class either by name or via annotation metadata).

The `IndexOperations` interface is shown below

```
public interface IndexOperations {

  void ensureIndex(IndexDefinition indexDefinition);

  void dropIndex(String name);

  void dropAllIndexes();

  void resetIndexCache();

  List<IndexInfo> getIndexInfo();
}
```

9.13.1. Methods for creating an Index

We can create an index on a collection to improve query performance.

Creating an index using the MongoTemplate

```
mongoTemplate.indexOps(Person.class).ensureIndex(new Index().on("name",Order.
ASCENDING));
```

- **ensureIndex** Ensure that an index for the provided IndexDefinition exists for the collection.

You can create standard, geospatial and text indexes using the classes `IndexDefinition`, `GeoSpatialIndex` and `TextIndexDefinition`. For example, given the Venue class defined in a previous section, you would declare a geospatial query as shown below.

```
mongoTemplate.indexOps(Venue.class).ensureIndex(new GeospatialIndex("location"));
```

NOTE `Index` and `GeospatialIndex` support configuration of collations.

9.13.2. Accessing index information

The IndexOperations interface has the method getIndexInfo that returns a list of IndexInfo objects. This contains all the indexes defined on the collection. Here is an example that defines an index on the Person class that has age property.

```
template.indexOps(Person.class).ensureIndex(new Index().on("age", Order.DESCENDING)
.unique(Duplicates.DROP));

List<IndexInfo> indexInfoList = template.indexOps(Person.class).getIndexInfo();

// Contains
// [IndexInfo [fieldSpec={_id=ASCENDING}, name=_id_, unique=false,
dropDuplicates=false, sparse=false],
//  IndexInfo [fieldSpec={age=DESCENDING}, name=age_-1, unique=true,
dropDuplicates=true, sparse=false]]
```

9.13.3. Methods for working with a Collection

It's time to look at some code examples showing how to use the `MongoTemplate`. First we look at creating our first collection.

Example 78. Working with collections using the MongoTemplate

```
DBCollection collection = null;
if (!mongoTemplate.getCollectionNames().contains("MyNewCollection")) {
    collection = mongoTemplate.createCollection("MyNewCollection");
}

mongoTemplate.dropCollection("MyNewCollection");
```

- **getCollectionNames** Returns a set of collection names.
- **collectionExists** Check to see if a collection with a given name exists.
- **createCollection** Create an uncapped collection
- **dropCollection** Drop the collection
- **getCollection** Get a collection by name, creating it if it doesn't exist.

NOTE	Collection creation allows customization via `CollectionOptions` and supports collations.

9.14. Executing Commands

You can also get at the MongoDB driver's `MongoDatabase.runCommand( )` method using the `executeCommand(…)` methods on `MongoTemplate`. These will also perform exception translation into

Spring's `DataAccessException` hierarchy.

9.14.1. Methods for executing commands

- `Document` **executeCommand** `(Document command)` Execute a MongoDB command.
- `Document` **executeCommand** `(Document command, ReadPreference readPreference)` Execute a MongoDB command using the given nullable MongoDB `ReadPreference`.
- `Document` **executeCommand** `(String jsonCommand)` Execute the a MongoDB command expressed as a JSON string.

9.15. Lifecycle Events

Built into the MongoDB mapping framework are several `org.springframework.context.ApplicationEvent` events that your application can respond to by registering special beans in the `ApplicationContext`. By being based off Spring's ApplicationContext event infrastructure this enables other products, such as Spring Integration, to easily receive these events as they are a well known eventing mechanism in Spring based applications.

To intercept an object before it goes through the conversion process (which turns your domain object into a `org.bson.Document`), you'd register a subclass of `AbstractMongoEventListener` that overrides the `onBeforeConvert` method. When the event is dispatched, your listener will be called and passed the domain object before it goes into the converter.

```
public class BeforeConvertListener extends AbstractMongoEventListener<Person> {
  @Override
  public void onBeforeConvert(BeforeConvertEvent<Person> event) {
    ... does some auditing manipulation, set timestamps, whatever ...
  }
}
```

To intercept an object before it goes into the database, you'd register a subclass of `org.springframework.data.mongodb.core.mapping.event.AbstractMongoEventListener` that overrides the `onBeforeSave` method. When the event is dispatched, your listener will be called and passed the domain object and the converted `com.mongodb.Document`.

```
public class BeforeSaveListener extends AbstractMongoEventListener<Person> {
  @Override
  public void onBeforeSave(BeforeSaveEvent<Person> event) {
    … change values, delete them, whatever …
  }
}
```

Simply declaring these beans in your Spring ApplicationContext will cause them to be invoked

whenever the event is dispatched.

The list of callback methods that are present in AbstractMappingEventListener are

- `onBeforeConvert` - called in MongoTemplate insert, insertList and save operations before the object is converted to a Document using a MongoConveter.
- `onBeforeSave` - called in MongoTemplate insert, insertList and save operations **before** inserting/saving the Document in the database.
- `onAfterSave` - called in MongoTemplate insert, insertList and save operations **after** inserting/saving the Document in the database.
- `onAfterLoad` - called in MongoTemplate find, findAndRemove, findOne and getCollection methods after the Document is retrieved from the database.
- `onAfterConvert` - called in MongoTemplate find, findAndRemove, findOne and getCollection methods after the Document retrieved from the database was converted to a POJO.

NOTE	Lifecycle events are only emitted for root level types. Complex types used as properties within a document root are not subject of event publication unless they are document references annotated with `@DBRef`.

9.16. Exception Translation

The Spring framework provides exception translation for a wide variety of database and mapping technologies. This has traditionally been for JDBC and JPA. The Spring support for MongoDB extends this feature to the MongoDB Database by providing an implementation of the `org.springframework.dao.support.PersistenceExceptionTranslator` interface.

The motivation behind mapping to Spring's consistent data access exception hierarchy is that you are then able to write portable and descriptive exception handling code without resorting to coding against MongoDB error codes. All of Spring's data access exceptions are inherited from the root `DataAccessException` class so you can be sure that you will be able to catch all database related exception within a single try-catch block. Note, that not all exceptions thrown by the MongoDB driver inherit from the MongoException class. The inner exception and message are preserved so no information is lost.

Some of the mappings performed by the `MongoExceptionTranslator` are: com.mongodb.Network to DataAccessResourceFailureException and `MongoException` error codes 1003, 12001, 12010, 12011, 12012 to `InvalidDataAccessApiUsageException`. Look into the implementation for more details on the mapping.

9.17. Execution callbacks

One common design feature of all Spring template classes is that all functionality is routed into one of the templates execute callback methods. This helps ensure that exceptions and any resource management that maybe required are performed consistency. While this was of much greater need in the case of JDBC and JMS than with MongoDB, it still offers a single spot for exception translation and logging to occur. As such, using these execute callback is the preferred way to access the

MongoDB driver's `DB` and `DBCollection` objects to perform uncommon operations that were not exposed as methods on `MongoTemplate`.

Here is a list of execute callback methods.

- `<T> T` **execute** `(Class<?> entityClass, CollectionCallback<T> action)` Executes the given CollectionCallback for the entity collection of the specified class.
- `<T> T` **execute** `(String collectionName, CollectionCallback<T> action)` Executes the given CollectionCallback on the collection of the given name.
- `<T> T` **execute** `(DbCallback<T> action) Spring Data MongoDB provides support for the Aggregation Framework introduced to MongoDB in version 2.2.` Executes a DbCallback translating any exceptions as necessary.
- `<T> T` **execute** `(String collectionName, DbCallback<T> action)` Executes a DbCallback on the collection of the given name translating any exceptions as necessary.
- `<T> T` **executeInSession** `(DbCallback<T> action)` Executes the given DbCallback within the same connection to the database so as to ensure consistency in a write heavy environment where you may read the data that you wrote.

Here is an example that uses the `CollectionCallback` to return information about an index

```
boolean hasIndex = template.execute("geolocation", new CollectionCallbackBoolean>() {
  public Boolean doInCollection(Venue.class, DBCollection collection) throws
MongoException, DataAccessException {
    List<Document> indexes = collection.getIndexInfo();
    for (Document document : indexes) {
      if ("location_2d".equals(document.get("name"))) {
        return true;
      }
    }
    return false;
  }
});
```

9.18. GridFS support

MongoDB supports storing binary files inside it's filesystem GridFS. Spring Data MongoDB provides a `GridFsOperations` interface as well as the according implementation `GridFsTemplate` to easily interact with the filesystem. You can setup a `GridFsTemplate` instance by handing it a `MongoDbFactory` as well as a `MongoConverter`:

Example 79. JavaConfig setup for a GridFsTemplate

```
class GridFsConfiguration extends AbstractMongoConfiguration {

  // … further configuration omitted

  @Bean
  public GridFsTemplate gridFsTemplate() {
    return new GridFsTemplate(mongoDbFactory(), mappingMongoConverter());
  }
}
```

An according XML configuration looks like this:

Example 80. XML configuration for a GridFsTemplate

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

  <mongo:db-factory id="mongoDbFactory" dbname="database" />
  <mongo:mapping-converter id="converter" />

  <bean class="org.springframework.data.mongodb.gridfs.GridFsTemplate">
    <constructor-arg ref="mongoDbFactory" />
    <constructor-arg ref="converter" />
  </bean>

</beans>
```

The template can now be injected and used to perform storage and retrieval operations.

Example 81. Using GridFsTemplate to store files

```
class GridFsClient {

  @Autowired
  GridFsOperations operations;

  @Test
  public void storeFileToGridFs() {

    FileMetadata metadata = new FileMetadata();
    // populate metadata
    Resource file = ⋯ // lookup File or Resource

    operations.store(file.getInputStream(), "filename.txt", metadata);
  }
}
```

The `store(⋯)` operations take an `InputStream`, a filename and optionally metadata information about the file to store. The metadata can be an arbitrary object which will be marshaled by the `MongoConverter` configured with the `GridFsTemplate`. Alternatively you can also provide a `Document` as well.

Reading files from the filesystem can either be achieved through the `find(⋯)` or `getResources(⋯)` methods. Let's have a look at the `find(⋯)` methods first. You can either find a single file matching a `Query` or multiple ones. To easily define file queries we provide the `GridFsCriteria` helper class. It provides static factory methods to encapsulate default metadata fields (e.g. `whereFilename()`, `whereContentType()`) or the custom one through `whereMetaData()`.

Example 82. Using GridFsTemplate to query for files

```
class GridFsClient {

  @Autowired
  GridFsOperations operations;

  @Test
  public void findFilesInGridFs() {
    GridFSFindIterable result = operations.find(query(whereFilename().is(
"filename.txt")))
  }
}
```

NOTE	Currently MongoDB does not support defining sort criteria when retrieving files from GridFS. Thus any sort criteria defined on the `Query` instance handed into the `find(…)` method will be disregarded.

The other option to read files from the GridFs is using the methods introduced by the `ResourcePatternResolver` interface. They allow handing an Ant path into the method ar thus retrieve files matching the given pattern.

Example 83. Using GridFsTemplate to read files

```
class GridFsClient {

  @Autowired
  GridFsOperations operations;

  @Test
  public void readFilesFromGridFs() {
    GridFsResources[] txtFiles = operations.getResources("*.txt");
  }
}
```

`GridFsOperations` extending `ResourcePatternResolver` allows the `GridFsTemplate` e.g. to be plugged into an `ApplicationContext` to read Spring Config files from a MongoDB.

Chapter 10. Reactive MongoDB support

The reactive MongoDB support contains a basic set of features which are summarized below.

- Spring configuration support using Java based `@Configuration` classes a `MongoClient` instance and replica sets.
- `ReactiveMongoTemplate` helper class that increases productivity using `MongoOperations in a reactive manner. Includes integrated object mapping between `Documents and POJOs.
- Exception translation into Spring's portable Data Access Exception hierarchy.
- Feature Rich Object Mapping integrated with Spring's `ConversionService`.
- Annotation based mapping metadata but extensible to support other metadata formats.
- Persistence and mapping lifecycle events.
- Java based `Query`, `Criteria`, and `Update` DSLs.
- Automatic implementation of reactive Repository interfaces including support for custom finder methods.

For most tasks you will find yourself using `ReactiveMongoTemplate` or the Repository support that both leverage the rich mapping functionality. `ReactiveMongoTemplate` is the place to look for accessing functionality such as incrementing counters or ad-hoc CRUD operations. `ReactiveMongoTemplate` also provides callback methods so that it is easy for you to get a hold of the low level API artifacts such as `MongoDatabase` to communicate directly with MongoDB. The goal with naming conventions on various API artifacts is to copy those in the base MongoDB Java driver so you can easily map your existing knowledge onto the Spring APIs.

10.1. Getting Started

Spring MongoDB support requires MongoDB 2.6 or higher and Java SE 8 or higher.

First you need to set up a running Mongodb server. Refer to the Mongodb Quick Start guide for an explanation on how to startup a MongoDB instance. Once installed starting MongoDB is typically a matter of executing the following command: `MONGO_HOME/bin/mongod`

To create a Spring project in STS go to File → New → Spring Template Project → Simple Spring Utility Project → press Yes when prompted. Then enter a project and a package name such as org.spring.mongodb.example.

Then add the following to pom.xml dependencies section.

```
<dependencies>

  <!-- other dependency elements omitted -->

  <dependency>
    <groupId>org.springframework.data</groupId>
    <artifactId>spring-data-mongodb</artifactId>
    <version>{version}</version>
  </dependency>

  <dependency>
    <groupId>org.mongodb</groupId>
    <artifactId>mongodb-driver-reactivestreams</artifactId>
    <version>{mongo.reactivestreams}</version>
  </dependency>

  <dependency>
    <groupId>io.projectreactor</groupId>
    <artifactId>reactor-core</artifactId>
    <version>{reactor}</version>
  </dependency>

</dependencies>
```

NOTE	MongoDB uses two different drivers for blocking and reactive (non-blocking) data access. While blocking operations are provided by default, you're have to opt-in for reactive usage.

Create a simple `Person` class to persist:

```
@Document
public class Person {

  private String id;
  private String name;
  private int age;

  public Person(String name, int age) {
    this.name = name;
    this.age = age;
  }

  public String getId() {
    return id;
  }
  public String getName() {
    return name;
  }
  public int getAge() {
    return age;
  }

  @Override
  public String toString() {
    return "Person [id=" + id + ", name=" + name + ", age=" + age + "]";
  }
}
```

And a main application to run

```
public class ReactiveMongoApp {

  private static final Logger log = LoggerFactory.getLogger(ReactiveMongoApp.class);

  public static void main(String[] args) throws Exception {

    CountDownLatch latch = new CountDownLatch(1);

    ReactiveMongoTemplate mongoOps = new ReactiveMongoTemplate(MongoClients.create(),
"database");

    mongoOps.insert(new Person("Joe", 34))
          .flatMap(p -> mongoOps.findOne(new Query(where("name").is("Joe")), Person
.class))
          .doOnNext(person -> log.info(person.toString()))
          .flatMap(person -> mongoOps.dropCollection("person"))
          .doOnComplete(latch::countDown)
          .subscribe();

    latch.await();
  }
}
```

This will produce the following output

```
2016-09-20 14:56:57,373 DEBUG .index.MongoPersistentEntityIndexCreator: 124 -
Analyzing class class example.ReactiveMongoApp$Person for index information.
2016-09-20 14:56:57,452 DEBUG .data.mongodb.core.ReactiveMongoTemplate: 975 -
Inserting Document containing fields: [_class, name, age] in collection: person
2016-09-20 14:56:57,541 DEBUG .data.mongodb.core.ReactiveMongoTemplate:1503 - findOne
using query: { "name" : "Joe"} fields: null for class: class
example.ReactiveMongoApp$Person in collection: person
2016-09-20 14:56:57,545 DEBUG .data.mongodb.core.ReactiveMongoTemplate:1979 - findOne
using query: { "name" : "Joe"} in db.collection: database.person
2016-09-20 14:56:57,567  INFO                example.ReactiveMongoApp:  43 - Person
[id=57e1321977ac501c68d73104, name=Joe, age=34]
2016-09-20 14:56:57,573 DEBUG .data.mongodb.core.ReactiveMongoTemplate: 528 - Dropped
collection [person]
```

Even in this simple example, there are few things to take notice of

- You can instantiate the central helper class of Spring Mongo, `MongoTemplate`, using the standard `com.mongodb.reactivestreams.client.MongoClient` object and the name of the database to use.
- The mapper works against standard POJO objects without the need for any additional metadata (though you can optionally provide that information. See here.).
- Conventions are used for handling the id field, converting it to be a ObjectId when stored in the database.

- Mapping conventions can use field access. Notice the Person class has only getters.
- If the constructor argument names match the field names of the stored document, they will be used to instantiate the object

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

10.2. Connecting to MongoDB with Spring and the Reactive Streams Driver

One of the first tasks when using MongoDB and Spring is to create a `com.mongodb.reactivestreams.client.MongoClient` object using the IoC container.

10.2.1. Registering a MongoClient instance using Java based metadata

An example of using Java based bean metadata to register an instance of a `com.mongodb.reactivestreams.client.MongoClient` is shown below

Example 84. Registering a com.mongodb.MongoClient object using Java based bean metadata

```
@Configuration
public class AppConfig {

  /*
   * Use the Reactive Streams Mongo Client API to create a
com.mongodb.reactivestreams.client.MongoClient instance.
   */
   public @Bean MongoClient reactiveMongoClient()  {
       return MongoClients.create("mongodb://localhost");
   }
}
```

This approach allows you to use the standard `com.mongodb.reactivestreams.client.MongoClient` API that you may already be used to using.

An alternative is to register an instance of `com.mongodb.reactivestreams.client.MongoClient` instance with the container using Spring's `ReactiveMongoClientFactoryBean`. As compared to instantiating a `com.mongodb.reactivestreams.client.MongoClient` instance directly, the FactoryBean approach has the added advantage of also providing the container with an `ExceptionTranslator` implementation that translates MongoDB exceptions to exceptions in Spring's portable `DataAccessException` hierarchy for data access classes annotated with the `@Repository` annotation. This hierarchy and use of `@Repository` is described in Spring's DAO support features.

An example of a Java based bean metadata that supports exception translation on `@Repository` annotated classes is shown below:

Example 85. Registering a com.mongodb.MongoClient object using Spring's MongoClientFactoryBean and enabling Spring's exception translation support

```
@Configuration
public class AppConfig {

    /*
     * Factory bean that creates the
com.mongodb.reactivestreams.client.MongoClient instance
     */
     public @Bean ReactiveMongoClientFactoryBean mongoClient() {

          ReactiveMongoClientFactoryBean clientFactory = new
ReactiveMongoClientFactoryBean();
          clientFactory.setHost("localhost");

          return clientFactory;
     }
}
```

To access the `com.mongodb.reactivestreams.client.MongoClient` object created by the `ReactiveMongoClientFactoryBean` in other `@Configuration` or your own classes, just obtain the `MongoClient` from the context.

10.2.2. The ReactiveMongoDatabaseFactory interface

While `com.mongodb.reactivestreams.client.MongoClient` is the entry point to the reactive MongoDB driver API, connecting to a specific MongoDB database instance requires additional information such as the database name. With that information you can obtain a `com.mongodb.reactivestreams.client.MongoDatabase` object and access all the functionality of a specific MongoDB database instance. Spring provides the `org.springframework.data.mongodb.core.ReactiveMongoDatabaseFactory` interface shown below to bootstrap connectivity to the database.

```
public interface ReactiveMongoDatabaseFactory {

  /**
   * Creates a default {@link MongoDatabase} instance.
   *
   * @return
   * @throws DataAccessException
   */
  MongoDatabase getMongoDatabase() throws DataAccessException;

  /**
   * Creates a {@link MongoDatabase} instance to access the database with the given
name.
   *
   * @param dbName must not be {@literal null} or empty.
   * @return
   * @throws DataAccessException
   */
  MongoDatabase getMongoDatabase(String dbName) throws DataAccessException;

  /**
   * Exposes a shared {@link MongoExceptionTranslator}.
   *
   * @return will never be {@literal null}.
   */
  PersistenceExceptionTranslator getExceptionTranslator();
}
```

The class `org.springframework.data.mongodb.core.SimpleReactiveMongoDatabaseFactory` provides implements the ReactiveMongoDatabaseFactory interface and is created with a standard `com.mongodb.reactivestreams.client.MongoClient` instance and the database name.

Instead of using the IoC container to create an instance of `ReactiveMongoTemplate`, you can just use them in standard Java code as shown below.

```
public class MongoApp {

  private static final Log log = LogFactory.getLog(MongoApp.class);

  public static void main(String[] args) throws Exception {

    ReactiveMongoOperations mongoOps = new ReactiveMongoOperations(new
SimpleReactiveMongoDatabaseFactory(MongoClient.create(), "database"));

    mongoOps.insert(new Person("Joe", 34))
        .flatMap(p -> mongoOps.findOne(new Query(where("name").is("Joe")), Person
.class))
        .doOnNext(person -> log.info(person.toString()))
        .flatMap(person -> mongoOps.dropCollection("person"))
        .subscribe();
  }
}
```

The use of `SimpleMongoDbFactory` is the only difference between the listing shown in the getting started section.

10.2.3. Registering a ReactiveMongoDatabaseFactory instance using Java based metadata

To register a `ReactiveMongoDatabaseFactory` instance with the container, you write code much like what was highlighted in the previous code listing. A simple example is shown below

```
@Configuration
public class MongoConfiguration {

  public @Bean ReactiveMongoDatabaseFactory reactiveMongoDatabaseFactory() {
    return new SimpleReactiveMongoDatabaseFactory(MongoClients.create(), "database");
  }
}
```

To define the username and password create MongoDB connection string and pass it into the factory method as shown below. This listing also shows using `ReactiveMongoDatabaseFactory` register an instance of `ReactiveMongoTemplate` with the container.

```
@Configuration
public class MongoConfiguration {

  public @Bean ReactiveMongoDatabaseFactory reactiveMongoDatabaseFactory() {
    return new SimpleReactiveMongoDatabaseFactory(MongoClients.create(
"mongodb://joe:secret@localhost"), "database");
  }

  public @Bean ReactiveMongoTemplate reactiveMongoTemplate() {
    return new ReactiveMongoTemplate(reactiveMongoDatabaseFactory());
  }
}
```

10.3. Introduction to ReactiveMongoTemplate

The class `ReactiveMongoTemplate`, located in the package `org.springframework.data.mongodb`, is the central class of the Spring's Reactive MongoDB support providing a rich feature set to interact with the database. The template offers convenience operations to create, update, delete and query for MongoDB documents and provides a mapping between your domain objects and MongoDB documents.

NOTE Once configured, `ReactiveMongoTemplate` is thread-safe and can be reused across multiple instances.

The mapping between MongoDB documents and domain classes is done by delegating to an implementation of the interface `MongoConverter`. Spring provides a default implementation with `MongoMappingConverter`, but you can also write your own converter. Please refer to the section on MongoConverters for more detailed information.

The `ReactiveMongoTemplate` class implements the interface `ReactiveMongoOperations`. In as much as possible, the methods on `ReactiveMongoOperations` are named after methods available on the MongoDB driver `Collection` object as as to make the API familiar to existing MongoDB developers who are used to the driver API. For example, you will find methods such as "find", "findAndModify", "findOne", "insert", "remove", "save", "update" and "updateMulti". The design goal was to make it as easy as possible to transition between the use of the base MongoDB driver and `ReactiveMongoOperations`. A major difference in between the two APIs is that `ReactiveMongoOperations` can be passed domain objects instead of `Document` and there are fluent APIs for `Query`, `Criteria`, and `Update` operations instead of populating a `Document` to specify the parameters for those operations.

NOTE The preferred way to reference the operations on `ReactiveMongoTemplate` instance is via its interface `ReactiveMongoOperations`.

The default converter implementation used by `ReactiveMongoTemplate` is `MappingMongoConverter`. While the `MappingMongoConverter` can make use of additional metadata to specify the mapping of objects to documents it is also capable of converting objects that contain no additional metadata by using some conventions for the mapping of IDs and collection names. These conventions as well as

the use of mapping annotations is explained in the Mapping chapter.

Another central feature of `ReactiveMongoTemplate` is exception translation of exceptions thrown in the MongoDB Java driver into Spring's portable Data Access Exception hierarchy. Refer to the section on exception translation for more information.

While there are many convenience methods on `ReactiveMongoTemplate` to help you easily perform common tasks if you should need to access the MongoDB driver API directly to access functionality not explicitly exposed by the MongoTemplate you can use one of several Execute callback methods to access underlying driver APIs. The execute callbacks will give you a reference to either a `com.mongodb.reactivestreams.client.MongoCollection` or a `com.mongodb.reactivestreams.client.MongoDatabase` object. Please see the section Execution Callbacks for more information.

Now let's look at a examples of how to work with the `ReactiveMongoTemplate` in the context of the Spring container.

10.3.1. Instantiating ReactiveMongoTemplate

You can use Java to create and register an instance of `ReactiveMongoTemplate` as shown below.

Example 86. Registering a `com.mongodb.reactivestreams.client.MongoClient` object and enabling Spring's exception translation support

```
@Configuration
public class AppConfig {

  public @Bean MongoClient reactiveMongoClient() {
      return MongoClients.create("mongodb://localhost");
  }

  public @Bean ReactiveMongoTemplate reactiveMongoTemplate() {
      return new ReactiveMongoTemplate(reactiveMongoClient(), "mydatabase");
  }
}
```

There are several overloaded constructors of `ReactiveMongoTemplate`. These are

- `ReactiveMongoTemplate(MongoClient mongo, String databaseName)` - takes the `com.mongodb.MongoClient` object and the default database name to operate against.
- `ReactiveMongoTemplate(ReactiveMongoDatabaseFactory mongoDatabaseFactory)` - takes a ReactiveMongoDatabaseFactory object that encapsulated the `com.mongodb.reactivestreams.client.MongoClient` object and database name.
- `ReactiveMongoTemplate(ReactiveMongoDatabaseFactory mongoDatabaseFactory, MongoConverter mongoConverter)` - adds a `MongoConverter` to use for mapping.

Other optional properties that you might like to set when creating a `ReactiveMongoTemplate` are the

default `WriteResultCheckingPolicy`, `WriteConcern`, and `ReadPreference`.

NOTE	The preferred way to reference the operations on `ReactiveMongoTemplate` instance is via its interface `ReactiveMongoOperations`.

10.3.2. WriteResultChecking Policy

When in development it is very handy to either log or throw an `Exception` if the `com.mongodb.WriteResult` returned from any MongoDB operation contains an error. It is quite common to forget to do this during development and then end up with an application that looks like it runs successfully but in fact the database was not modified according to your expectations. Set MongoTemplate's property to an enum with the following values, `LOG`, `EXCEPTION`, or `NONE` to either log the error, throw and exception or do nothing. The default is to use a `WriteResultChecking` value of `NONE`.

10.3.3. WriteConcern

You can set the `com.mongodb.WriteConcern` property that the `ReactiveMongoTemplate` will use for write operations if it has not yet been specified via the driver at a higher level such as `MongoDatabase`. If ReactiveMongoTemplate's `WriteConcern` property is not set it will default to the one set in the MongoDB driver's `MongoDatabase` or `MongoCollection` setting.

10.3.4. WriteConcernResolver

For more advanced cases where you want to set different `WriteConcern` values on a per-operation basis (for remove, update, insert and save operations), a strategy interface called `WriteConcernResolver` can be configured on `ReactiveMongoTemplate`. Since `ReactiveMongoTemplate` is used to persist POJOs, the `WriteConcernResolver` lets you create a policy that can map a specific POJO class to a `WriteConcern` value. The `WriteConcernResolver` interface is shown below.

```
public interface WriteConcernResolver {
  WriteConcern resolve(MongoAction action);
}
```

The passed in argument, `MongoAction`, is what you use to determine the `WriteConcern` value to be used or to use the value of the Template itself as a default. `MongoAction` contains the collection name being written to, the `java.lang.Class` of the POJO, the converted `DBObject`, as well as the operation as an enumeration (`MongoActionOperation`: REMOVE, UPDATE, INSERT, INSERT_LIST, SAVE) and a few other pieces of contextual information. For example,

```
private class MyAppWriteConcernResolver implements WriteConcernResolver {

  public WriteConcern resolve(MongoAction action) {
    if (action.getEntityClass().getSimpleName().contains("Audit")) {
      return WriteConcern.NONE;
    } else if (action.getEntityClass().getSimpleName().contains("Metadata")) {
      return WriteConcern.JOURNAL_SAFE;
    }
    return action.getDefaultWriteConcern();
  }
}
```

10.4. Saving, Updating, and Removing Documents

`ReactiveMongoTemplate` provides a simple way for you to save, update, and delete your domain objects and map those objects to documents stored in MongoDB.

Given a simple class such as Person

```
public class Person {

  private String id;
  private String name;
  private int age;

  public Person(String name, int age) {
    this.name = name;
    this.age = age;
  }

  public String getId() {
    return id;
  }
  public String getName() {
    return name;
  }
  public int getAge() {
    return age;
  }

  @Override
  public String toString() {
    return "Person [id=" + id + ", name=" + name + ", age=" + age + "]";
  }

}
```

You can save, update and delete the object as shown below.

```
public class ReactiveMongoApp {

  private static final Logger log = LoggerFactory.getLogger(ReactiveMongoApp.class);

  public static void main(String[] args) throws Exception {

    CountDownLatch latch = new CountDownLatch(1);

    ReactiveMongoTemplate mongoOps = new ReactiveMongoTemplate(MongoClients.create(),
"database");

    mongoOps.insert(new Person("Joe", 34)).doOnNext(person -> log.info("Insert: " +
person))
      .flatMap(person -> mongoOps.findById(person.getId(), Person.class))
      .doOnNext(person -> log.info("Found: " + person))
      .zipWith(person -> mongoOps.updateFirst(query(where("name").is("Joe")), update(
"age", 35), Person.class))
      .flatMap(tuple -> mongoOps.remove(tuple.getT1())).flatMap(deleteResult ->
mongoOps.findAll(Person.class))
      .count().doOnSuccess(count -> {
        log.info("Number of people: " + count);
        latch.countDown();
      })

      .subscribe();

    latch.await();
  }
}
```

There was implicit conversion using the `MongoConverter` between a `String` and `ObjectId` as stored in the database and recognizing a convention of the property "Id" name.

NOTE	This example is meant to show the use of save, update and remove operations on `ReactiveMongoTemplate` and not to show complex mapping or functional chaining functionality

The query syntax used in the example is explained in more detail in the section Querying Documents. Additional documentation can be found in the blocking MongoTemplate section.

10.5. Execution callbacks

One common design feature of all Spring template classes is that all functionality is routed into one of the templates execute callback methods. This helps ensure that exceptions and any resource management that maybe required are performed consistency. While this was of much greater need in the case of JDBC and JMS than with MongoDB, it still offers a single spot for exception translation and logging to occur. As such, using the execute callback is the preferred way to access the MongoDB driver's `MongoDatabase` and `MongoCollection` objects to perform uncommon operations that

were not exposed as methods on `ReactiveMongoTemplate`.

Here is a list of execute callback methods.

- `<T> Flux<T>` **execute** `(Class<?> entityClass, ReactiveCollectionCallback<T> action)` Executes the given ReactiveCollectionCallback for the entity collection of the specified class.
- `<T> Flux<T>` **execute** `(String collectionName, ReactiveCollectionCallback<T> action)` Executes the given ReactiveCollectionCallback on the collection of the given name.
- `<T> Flux<T>` **execute** `(ReactiveDatabaseCallback<T> action)` Executes a ReactiveDatabaseCallback translating any exceptions as necessary.

Here is an example that uses the `ReactiveCollectionCallback` to return information about an index

```
Flux<Boolean> hasIndex = operations.execute("geolocation",
    collection -> Flux.from(collection.listIndexes(Document.class))
      .filter(document -> document.get("name").equals("fancy-index-name"))
      .flatMap(document -> Mono.just(true))
      .defaultIfEmpty(false));
```

Chapter 11. MongoDB repositories

11.1. Introduction

This chapter will point out the specialties for repository support for MongoDB. This builds on the core repository support explained in Working with Spring Data Repositories. So make sure you've got a sound understanding of the basic concepts explained there.

11.2. Usage

To access domain entities stored in a MongoDB you can leverage our sophisticated repository support that eases implementing those quite significantly. To do so, simply create an interface for your repository:

Example 87. Sample Person entity

```
public class Person {

  @Id
  private String id;
  private String firstname;
  private String lastname;
  private Address address;

  // … getters and setters omitted
}
```

We have a quite simple domain object here. Note that it has a property named `id` of type `ObjectId`. The default serialization mechanism used in `MongoTemplate` (which is backing the repository support) regards properties named id as document id. Currently we support `String`, `ObjectId` and `BigInteger` as id-types.

Example 88. Basic repository interface to persist Person entities

```
public interface PersonRepository extends PagingAndSortingRepository<Person, Long>
{

  // additional custom finder methods go here
}
```

Right now this interface simply serves typing purposes but we will add additional methods to it later. In your Spring configuration simply add

Example 89. General MongoDB repository Spring configuration

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:mongo="http://www.springframework.org/schema/data/mongo"
  xsi:schemaLocation="http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
    http://www.springframework.org/schema/data/mongo
    http://www.springframework.org/schema/data/mongo/spring-mongo-1.0.xsd">

  <mongo:mongo-client id="mongoClient" />

  <bean id="mongoTemplate" class=
"org.springframework.data.mongodb.core.MongoTemplate">
    <constructor-arg ref="mongoClient" />
    <constructor-arg value="databaseName" />
  </bean>

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

</beans>
```

This namespace element will cause the base packages to be scanned for interfaces extending `MongoRepository` and create Spring beans for each of them found. By default the repositories will get a `MongoTemplate` Spring bean wired that is called `mongoTemplate`, so you only need to configure `mongo-template-ref` explicitly if you deviate from this convention.

If you'd rather like to go with JavaConfig use the `@EnableMongoRepositories` annotation. The annotation carries the very same attributes like the namespace element. If no base package is configured the infrastructure will scan the package of the annotated configuration class.

Example 90. JavaConfig for repositories

```
@Configuration
@EnableMongoRepositories
class ApplicationConfig extends AbstractMongoConfiguration {

  @Override
  protected String getDatabaseName() {
    return "e-store";
  }

  @Override
  public MongoClient mongoClient() {
    return new MongoClient();
  }

  @Override
  protected String getMappingBasePackage() {
    return "com.oreilly.springdata.mongodb"
  }
}
```

As our domain repository extends `PagingAndSortingRepository` it provides you with CRUD operations as well as methods for paginated and sorted access to the entities. Working with the repository instance is just a matter of dependency injecting it into a client. So accessing the second page of `Person` s at a page size of 10 would simply look something like this:

Example 91. Paging access to Person entities

```
@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration
public class PersonRepositoryTests {

    @Autowired PersonRepository repository;

    @Test
    public void readsFirstPageCorrectly() {

      Page<Person> persons = repository.findAll(PageRequest.of(0, 10));
      assertThat(persons.isFirstPage(), is(true));
    }
}
```

The sample creates an application context with Spring's unit test support which will perform annotation based dependency injection into test cases. Inside the test method we simply use the repository to query the datastore. We hand the repository a `PageRequest` instance that requests the

first page of persons at a page size of 10.

11.3. Query methods

Most of the data access operations you usually trigger on a repository result a query being executed against the MongoDB databases. Defining such a query is just a matter of declaring a method on the repository interface

Example 92. PersonRepository with query methods

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

    List<Person> findByLastname(String lastname);                    ①

    Page<Person> findByFirstname(String firstname, Pageable pageable); ②

    Person findByShippingAddresses(Address address);                 ③

    Stream<Person> findAllBy();                                      ④
}
```

① The method shows a query for all people with the given lastname. The query will be derived parsing the method name for constraints which can be concatenated with `And` and `Or`. Thus the method name will result in a query expression of `{"lastname" : lastname}`.

② Applies pagination to a query. Just equip your method signature with a `Pageable` parameter and let the method return a `Page` instance and we will automatically page the query accordingly.

③ Shows that you can query based on properties which are not a primitive type.

④ Uses a Java 8 `Stream` which reads and converts individual elements while iterating the stream.

NOTE Note that for version 1.0 we currently don't support referring to parameters that are mapped as `DBRef` in the domain class.

Table 6. Supported keywords for query methods

Keyword	Sample	Logical result
`After`	`findByBirthdateAfter(Date date)`	`{"birthdate" : {"$gt" : date}}`
`GreaterThan`	`findByAgeGreaterThan(int age)`	`{"age" : {"$gt" : age}}`
`GreaterThanEqual`	`findByAgeGreaterThanEqual(int age)`	`{"age" : {"$gte" : age}}`
`Before`	`findByBirthdateBefore(Date date)`	`{"birthdate" : {"$lt" : date}}`
`LessThan`	`findByAgeLessThan(int age)`	`{"age" : {"$lt" : age}}`

Keyword	Sample	Logical result
`LessThanEqual`	`findByAgeLessThanEqual(int age)`	`{"age" : {"$lte" : age}}`
`Between`	`findByAgeBetween(int from, int to)`	`{"age" : {"$gt" : from, "$lt" : to}}`
`In`	`findByAgeIn(Collection ages)`	`{"age" : {"$in" : [ages…]}}`
`NotIn`	`findByAgeNotIn(Collection ages)`	`{"age" : {"$nin" : [ages…]}}`
`IsNotNull`, `NotNull`	`findByFirstnameNotNull()`	`{"firstname" : {"$ne" : null}}`
`IsNull`, `Null`	`findByFirstnameNull()`	`{"firstname" : null}`
`Like`, `StartingWith`, `EndingWith`	`findByFirstnameLike(String name)`	`{"firstname" : name} (name as regex)`
`NotLike`, `IsNotLike`	`findByFirstnameNotLike(String name)`	`{"firstname" : { "$not" : name }} (name as regex)`
`Containing` on String	`findByFirstnameContaining(String name)`	`{"firstname" : name} (name as regex)`
`NotContaining` on String	`findByFirstnameNotContaining(String name)`	`{"firstname" : { "$not" : name}} (name as regex)`
`Containing` on Collection	`findByAddressesContaining(Address address)`	`{"addresses" : { "$in" : address}}`
`NotContaining` on Collection	`findByAddressesNotContaining(Address address)`	`{"addresses" : { "$not" : { "$in" : address}}}`
`Regex`	`findByFirstnameRegex(String firstname)`	`{"firstname" : {"$regex" : firstname }}`
`(No keyword)`	`findByFirstname(String name)`	`{"firstname" : name}`
`Not`	`findByFirstnameNot(String name)`	`{"firstname" : {"$ne" : name}}`
`Near`	`findByLocationNear(Point point)`	`{"location" : {"$near" : [x,y]}}`
`Near`	`findByLocationNear(Point point, Distance max)`	`{"location" : {"$near" : [x,y], "$maxDistance" : max}}`
`Near`	`findByLocationNear(Point point, Distance min, Distance max)`	`{"location" : {"$near" : [x,y], "$minDistance" : min, "$maxDistance" : max}}`
`Within`	`findByLocationWithin(Circle circle)`	`{"location" : {"$geoWithin" : {"$center" : [ [x, y], distance]}}}`
`Within`	`findByLocationWithin(Box box)`	`{"location" : {"$geoWithin" : {"$box" : [ [x1, y1], x2, y2]}}}`
`IsTrue`, `True`	`findByActiveIsTrue()`	`{"active" : true}`
`IsFalse`, `False`	`findByActiveIsFalse()`	`{"active" : false}`
`Exists`	`findByLocationExists(boolean exists)`	`{"location" : {"$exists" : exists }}`

NOTE If the property criterion compares a document, the order of the fields and exact equality in the document matters.

11.3.1. Repository delete queries

The above keywords can be used in conjunction with `delete…By` or `remove…By` to create queries deleting matching documents.

Example 93. `Delete…By` Query

```
public interface PersonRepository extends MongoRepository<Person, String> {

  List <Person> deleteByLastname(String lastname);

  Long deletePersonByLastname(String lastname);
}
```

Using return type `List` will retrieve and return all matching documents before actually deleting them. A numeric return type directly removes the matching documents returning the total number of documents removed.

11.3.2. Geo-spatial repository queries

As you've just seen there are a few keywords triggering geo-spatial operations within a MongoDB query. The `Near` keyword allows some further modification. Let's have a look at some examples:

Example 94. Advanced `Near` queries

```
public interface PersonRepository extends MongoRepository<Person, String>

  // { 'location' : { '$near' : [point.x, point.y], '$maxDistance' : distance}}
  List<Person> findByLocationNear(Point location, Distance distance);
}
```

Adding a `Distance` parameter to the query method allows restricting results to those within the given distance. If the `Distance` was set up containing a `Metric` we will transparently use `$nearSphere` instead of $code.

Example 95. Using `Distance` with `Metrics`

```
Point point = new Point(43.7, 48.8);
Distance distance = new Distance(200, Metrics.KILOMETERS);
… = repository.findByLocationNear(point, distance);
// {'location' : {'$nearSphere' : [43.7, 48.8], '$maxDistance' :
0.0313571188577479 6}}
```

As you can see using a `Distance` equipped with a `Metric` causes `$nearSphere` clause to be added

instead of a plain `$near`. Beyond that the actual distance gets calculated according to the `Metrics` used.

NOTE	Using `@GeoSpatialIndexed(type = GeoSpatialIndexType.GEO_2DSPHERE)` on the target property forces usage of `$nearSphere` operator.

Geo-near queries

```
public interface PersonRepository extends MongoRepository<Person, String>

  // {'geoNear' : 'location', 'near' : [x, y] }
  GeoResults<Person> findByLocationNear(Point location);

  // No metric: {'geoNear' : 'person', 'near' : [x, y], maxDistance : distance }
  // Metric: {'geoNear' : 'person', 'near' : [x, y], 'maxDistance' : distance,
  //          'distanceMultiplier' : metric.multiplier, 'spherical' : true }
  GeoResults<Person> findByLocationNear(Point location, Distance distance);

  // Metric: {'geoNear' : 'person', 'near' : [x, y], 'minDistance' : min,
  //          'maxDistance' : max, 'distanceMultiplier' : metric.multiplier,
  //          'spherical' : true }
  GeoResults<Person> findByLocationNear(Point location, Distance min, Distance max);

  // {'geoNear' : 'location', 'near' : [x, y] }
  GeoResults<Person> findByLocationNear(Point location);
}
```

11.3.3. MongoDB JSON based query methods and field restriction

By adding the annotation `org.springframework.data.mongodb.repository.Query` repository finder methods you can specify a MongoDB JSON query string to use instead of having the query derived from the method name. For example

```
public interface PersonRepository extends MongoRepository<Person, String>

  @Query("{ 'firstname' : ?0 }")
  List<Person> findByThePersonsFirstname(String firstname);

}
```

The placeholder `?0` lets you substitute the value from the method arguments into the JSON query string.

NOTE	`String` parameter values are escaped during the binding process, which means that it is not possible to add MongoDB specific operators via the argument.

You can also use the filter property to restrict the set of properties that will be mapped into the Java

object. For example,

```
public interface PersonRepository extends MongoRepository<Person, String>

  @Query(value="{ 'firstname' : ?0 }", fields="{ 'firstname' : 1, 'lastname' : 1}")
  List<Person> findByThePersonsFirstname(String firstname);

}
```

This will return only the firstname, lastname and Id properties of the Person objects. The age property, a java.lang.Integer, will not be set and its value will therefore be null.

11.3.4. JSON based queries with SpEL expressions

Query strings and field definitions can be used together with SpEL expressions to create dynamic queries at runtime. SpEL expressions can provide predicate values and can be used to extend predicates with subdocuments.

Expressions expose method arguments through an array that contains all arguments. The the following query uses `[0]` to declare the predicate value for `lastname` that is equivalent to the `?0` parameter binding.

```
public interface PersonRepository extends MongoRepository<Person, String>

  @Query("{'lastname': ?#{[0]} }")
  List<Person> findByQueryWithExpression(String param0);
}
```

Expressions can be used to invoke functions, evaluate conditionals and construct values. SpEL expressions reveal in conjunction with JSON a side-effect as Map-like declarations inside of SpEL read like JSON.

```
public interface PersonRepository extends MongoRepository<Person, String>

  @Query("{'id': ?#{ [0] ? {$exists :true} : [1] }}")
  List<Person> findByQueryWithExpressionAndNestedObject(boolean param0, String param1
);
}
```

SpEL in query strings can be a powerful way to enhance queries and can accept a broad range of unwanted arguments. You should make sure to sanitize strings before passing these to the query to avoid unwanted changes to your query.

Expression support is extensible through the Query SPI `org.springframework.data.repository.query.spi.EvaluationContextExtension` than can contribute properties, functions and customize the root object. Extensions are retrieved from the application context at the time of SpEL evaluation when the query is build.

```
public class SampleEvaluationContextExtension extends
EvaluationContextExtensionSupport {

  @Override
  public String getExtensionId() {
    return "security";
  }

  @Override
  public Map<String, Object> getProperties() {
    return Collections.singletonMap("principal", SecurityContextHolder.getCurrent()
.getPrincipal());
  }
}
```

NOTE	Bootstrapping `MongoRepositoryFactory` yourself is not application context-aware and requires further configuration to pick up Query SPI extensions.

11.3.5. Type-safe Query methods

MongoDB repository support integrates with the QueryDSL project which provides a means to perform type-safe queries in Java. To quote from the project description, "Instead of writing queries as inline strings or externalizing them into XML files they are constructed via a fluent API." It provides the following features

- Code completion in IDE (all properties, methods and operations can be expanded in your favorite Java IDE)
- Almost no syntactically invalid queries allowed (type-safe on all levels)
- Domain types and properties can be referenced safely (no Strings involved!)
- Adopts better to refactoring changes in domain types
- Incremental query definition is easier

Please refer to the QueryDSL documentation which describes how to bootstrap your environment for APT based code generation using Maven or Ant.

Using QueryDSL you will be able to write queries as shown below

```
QPerson person = new QPerson("person");
List<Person> result = repository.findAll(person.address.zipCode.eq("C0123"));

Page<Person> page = repository.findAll(person.lastname.contains("a"),
                                       PageRequest.of(0, 2, Direction.ASC, "lastname"
));
```

`QPerson` is a class that is generated (via the Java annotation post processing tool) which is a `Predicate` that allows you to write type safe queries. Notice that there are no strings in the query

other than the value "C0123".

You can use the generated `Predicate` class via the interface `QueryDslPredicateExecutor` which is shown below

```
public interface QueryDslPredicateExecutor<T> {

  T findOne(Predicate predicate);

  List<T> findAll(Predicate predicate);

  List<T> findAll(Predicate predicate, OrderSpecifier<?>... orders);

  Page<T> findAll(Predicate predicate, Pageable pageable);

  Long count(Predicate predicate);
}
```

To use this in your repository implementation, simply inherit from it in addition to other repository interfaces. This is shown below

```
public interface PersonRepository extends MongoRepository<Person, String>,
QueryDslPredicateExecutor<Person> {

   // additional finder methods go here
}
```

We think you will find this an extremely powerful tool for writing MongoDB queries.

11.3.6. Full-text search queries

MongoDBs full text search feature is very store specific and therefore can rather be found on `MongoRepository` than on the more general `CrudRepository`. What we need is a document with a full-text index defined for (Please see section Text Indexes for creating).

Additional methods on `MongoRepository` take `TextCriteria` as input parameter. In addition to those explicit methods, it is also possible to add a `TextCriteria` derived repository method. The criteria will be added as an additional `AND` criteria. Once the entity contains a `@TextScore` annotated property the documents full-text score will be retrieved. Furthermore the `@TextScore` annotated property will also make it possible to sort by the documents score.

```
@Document
class FullTextDocument {

  @Id String id;
  @TextIndexed String title;
  @TextIndexed String content;
  @TextScore Float score;
}

interface FullTextRepository extends Repository<FullTextDocument, String> {

  // Execute a full-text search and define sorting dynamically
  List<FullTextDocument> findAllBy(TextCriteria criteria, Sort sort);

  // Paginate over a full-text search result
  Page<FullTextDocument> findAllBy(TextCriteria criteria, Pageable pageable);

  // Combine a derived query with a full-text search
  List<FullTextDocument> findByTitleOrderByScoreDesc(String title, TextCriteria
criteria);
}


Sort sort = Sort.by("score");
TextCriteria criteria = TextCriteria.forDefaultLanguage().matchingAny("spring", "data
");
List<FullTextDocument> result = repository.findAllBy(criteria, sort);

criteria = TextCriteria.forDefaultLanguage().matching("film");
Page<FullTextDocument> page = repository.findAllBy(criteria, PageRequest.of(1, 1,
sort));
List<FullTextDocument> result = repository.findByTitleOrderByScoreDesc("mongodb",
criteria);
```

11.3.7. Projections

Spring Data query methods usually return one or multiple instances of the aggregate root managed by the repository. However, it might sometimes be desirable to rather project on certain attributes of those types. Spring Data allows to model dedicated return types to more selectively retrieve partial views onto the managed aggregates.

Imagine a sample repository and aggregate root type like this:

Example 96. A sample aggregate and repository

```
class Person {

  @Id UUID id;
  String firstname, lastname;
  Address address;

  static class Address {
    String zipCode, city, street;
  }
}

interface PersonRepository extends Repository<Person, UUID> {

  Collection<Person> findByLastname(String lastname);
}
```

Now imagine we'd want to retrieve the person's name attributes only. What means does Spring Data offer to achieve this?

Interface-based projections

The easiest way to limit the result of the queries to expose the name attributes only is by declaring an interface that will expose accessor methods for the properties to be read:

Example 97. A projection interface to retrieve a subset of attributes

```
interface NamesOnly {

  String getFirstname();
  String getLastname();
}
```

The important bit here is that the properties defined here exactly match properties in the aggregate root. This allows a query method to be added like this:

Example 98. A repository using an interface based projection with a query method

```
interface PersonRepository extends Repository<Person, UUID> {

  Collection<NamesOnly> findByLastname(String lastname);
}
```

The query execution engine will create proxy instances of that interface at runtime for each element returned and forward calls to the exposed methods to the target object.

Projections can be used recursively. If you wanted to include some of the `Address` information as well, create a projection interface for that and return that interface from the declaration of `getAddress()`.

Example 99. A projection interface to retrieve a subset of attributes

```
interface PersonSummary {

  String getFirstname();
  String getLastname();
  AddressSummary getAddress();

  interface AddressSummary {
    String getCity();
  }
}
```

On method invocation, the `address` property of the target instance will be obtained and wrapped into a projecting proxy in turn.

Closed projections

A projection interface whose accessor methods all match properties of the target aggregate are considered closed projections.

Example 100. A closed projection

```
interface NamesOnly {

  String getFirstname();
  String getLastname();
}
```

If a closed projection is used, Spring Data modules can even optimize the query execution as we exactly know about all attributes that are needed to back the projection proxy. For more details on that, please refer to the module specific part of the reference documentation.

Open projections

Accessor methods in projection interfaces can also be used to compute new values by using the `@Value` annotation on it:

Example 101. An Open Projection

```
interface NamesOnly {

  @Value("#{target.firstname + ' ' + target.lastname}")
  String getFullName();
  …
}
```

The aggregate root backing the projection is available via the `target` variable. A projection interface using `@Value` an open projection. Spring Data won't be able to apply query execution optimizations in this case as the SpEL expression could use any attributes of the aggregate root.

The expressions used in `@Value` shouldn't become too complex as you'd want to avoid programming in `String`s. For very simple expressions, one option might be to resort to default methods:

Example 102. A projection interface using a default method for custom logic

```
interface NamesOnly {

  String getFirstname();
  String getLastname();

  default String getFullName() {
    return getFirstname.concat(" ").concat(getLastname());
  }
}
```

This approach requires you to be able to implement logic purely based on the other accessor methods exposed on the projection interface. A second, more flexible option is to implement the custom logic in a Spring bean and then simply invoke that from the SpEL expression:

Example 103. Sample Person object

```
@Component
class MyBean {

  String getFullName(Person person) {
    …
  }
}

interface NamesOnly {

  @Value("#{@myBean.getFullName(target)}")
  String getFullName();
  …
}
```

Note, how the SpEL expression refers to `myBean` and invokes the `getFullName(…)` method forwarding the projection target as method parameter. Methods backed by SpEL expression evaluation can also use method parameters which can then be referred to from the expression. The method parameters are available via an `Object` array named `args`.

Example 104. Sample Person object

```
interface NamesOnly {

  @Value("#{args[0] + ' ' + target.firstname + '!'}")
  String getSalutation(String prefix);
}
```

Again, for more complex expressions rather use a Spring bean and let the expression just invoke a method as described above.

Class-based projections (DTOs)

Another way of defining projections is using value type DTOs that hold properties for the fields that are supposed to be retrieved. These DTO types can be used exactly the same way projection interfaces are used, except that no proxying is going on here and no nested projections can be applied.

In case the store optimizes the query execution by limiting the fields to be loaded, the ones to be loaded are determined from the parameter names of the constructor that is exposed.

Example 105. A projecting DTO

```
class NamesOnly {

  private final String firstname, lastname;

  NamesOnly(String firstname, String lastname) {

    this.firstname = firstname;
    this.lastname = lastname;
  }

  String getFirstname() {
    return this.firstname;
  }

  String getLastname() {
    return this.lastname;
  }

  // equals(…) and hashCode() implementations
}
```

TIP

Avoiding boilerplate code for projection DTOs

The code that needs to be written for a DTO can be dramatically simplified using Project Lombok, which provides an `@Value` annotation (not to mix up with Spring's `@Value` annotation shown in the interface examples above). The sample DTO above would become this:

```
@Value
class NamesOnly {
    String firstname, lastname;
}
```

Fields are private final by default, the class exposes a constructor taking all fields and automatically gets `equals(…)` and `hashCode()` methods implemented.

Dynamic projections

So far we have used the projection type as the return type or element type of a collection. However, it might be desirable to rather select the type to be used at invocation time. To apply dynamic projections, use a query method like this:

Example 106. A repository using a dynamic projection parameter

```
interface PersonRepository extends Repository<Person, UUID> {

  Collection<T> findByLastname(String lastname, Class<T> type);
}
```

This way the method can be used to obtain the aggregates as is, or with a projection applied:

Example 107. Using a repository with dynamic projections

```
void someMethod(PersonRepository people) {

  Collection<Person> aggregates =
    people.findByLastname("Matthews", Person.class);

  Collection<NamesOnly> aggregates =
    people.findByLastname("Matthews", NamesOnly.class);
}
```

11.4. Miscellaneous

11.4.1. CDI Integration

Instances of the repository interfaces are usually created by a container, which Spring is the most natural choice when working with Spring Data. As of version 1.3.0 Spring Data MongoDB ships with a custom CDI extension that allows using the repository abstraction in CDI environments. The extension is part of the JAR so all you need to do to activate it is dropping the Spring Data MongoDB JAR into your classpath. You can now set up the infrastructure by implementing a CDI Producer for the `MongoTemplate`:

```
class MongoTemplateProducer {

    @Produces
    @ApplicationScoped
    public MongoOperations createMongoTemplate() {

        MongoDbFactory factory = new SimpleMongoDbFactory(new MongoClient(), "
database");
        return new MongoTemplate(factory);
    }
}
```

The Spring Data MongoDB CDI extension will pick up the `MongoTemplate` available as CDI bean and

create a proxy for a Spring Data repository whenever a bean of a repository type is requested by the container. Thus obtaining an instance of a Spring Data repository is a matter of declaring an `@Inject`-ed property:

```
class RepositoryClient {

  @Inject
  PersonRepository repository;

  public void businessMethod() {
    List<Person> people = repository.findAll();
  }
}
```

Chapter 12. Reactive MongoDB repositories

12.1. Introduction

This chapter will point out the specialties for reactive repository support for MongoDB. This builds on the core repository support explained in Working with Spring Data Repositories. So make sure you've got a sound understanding of the basic concepts explained there.

12.2. Reactive Composition Libraries

The reactive space offers various reactive composition libraries. The most common libraries are RxJava and Project Reactor.

Spring Data MongoDB is built on top of the MongoDB Reactive Streams driver to provide maximal interoperability relying on the Reactive Streams initiative. Static APIs such as `ReactiveMongoOperations` are provided by using Project Reactor's `Flux` and `Mono` types. Project Reactor offers various adapters to convert reactive wrapper types (`Flux` to `Observable` and vice versa) but conversion can easily clutter your code.

Spring Data's Repository abstraction is a dynamic API, mostly defined by you and your requirements, as you're declaring query methods. Reactive MongoDB repositories can be either implemented using RxJava or Project Reactor wrapper types by simply extending from one of the library-specific repository interfaces:

- `ReactiveCrudRepository`
- `ReactiveSortingRepository`
- `RxJava2CrudRepository`
- `RxJava2SortingRepository`

Spring Data converts reactive wrapper types behind the scenes so that you can stick to your favorite composition library.

12.3. Usage

To access domain entities stored in a MongoDB you can leverage our sophisticated repository support that eases implementing those quite significantly. To do so, simply create an interface for your repository:

Example 108. Sample Person entity

```
public class Person {

  @Id
  private String id;
  private String firstname;
  private String lastname;
  private Address address;

  // … getters and setters omitted
}
```

We have a quite simple domain object here. Note that it has a property named `id` of type `ObjectId`. The default serialization mechanism used in `MongoTemplate` (which is backing the repository support) regards properties named id as document id. Currently we support `String`, `ObjectId` and `BigInteger` as id-types.

Example 109. Basic repository interface to persist Person entities

```
public interface ReactivePersonRepository extends
ReactiveSortingRepository<Person, Long> {

  Flux<Person> findByFirstname(String firstname);

  Flux<Person> findByFirstname(Publisher<String> firstname);

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

  Mono<Person> findByFirstnameAndLastname(String firstname, String lastname);
}
```

For JavaConfig use the `@EnableReactiveMongoRepositories` annotation. The annotation carries the very same attributes like the namespace element. If no base package is configured the infrastructure will scan the package of the annotated configuration class.

NOTE

MongoDB uses two different drivers for blocking and reactive (non-blocking) data access. It's required to create a connection using the Reactive Streams driver to provide the required infrastructure for Spring Data's Reactive MongoDB support hence you're required to provide a separate Configuration for MongoDB's Reactive Streams driver. Please also note that your application will operate on two different connections if using Reactive and Blocking Spring Data MongoDB Templates and Repositories.

Example 110. JavaConfig for repositories

```
@Configuration
@EnableReactiveMongoRepositories
class ApplicationConfig extends AbstractReactiveMongoConfiguration {

  @Override
  protected String getDatabaseName() {
    return "e-store";
  }

  @Override
  public MongoClient reactiveMongoClient() {
    return MongoClients.create();
  }

  @Override
  protected String getMappingBasePackage() {
    return "com.oreilly.springdata.mongodb"
  }
}
```

As our domain repository extends `ReactiveSortingRepository` it provides you with CRUD operations as well as methods for sorted access to the entities. Working with the repository instance is just a matter of dependency injecting it into a client.

Example 111. Sorted access to Person entities

```
public class PersonRepositoryTests {

    @Autowired ReactivePersonRepository repository;

    @Test
    public void sortsElementsCorrectly() {
      Flux<Person> persons = repository.findAll(Sort.by(new Order(ASC, "lastname"
)));
    }
}
```

12.4. Features

Spring Data's Reactive MongoDB support comes with a reduced feature set compared to the blocking MongoDB Repositories.

Following features are supported:

- Query Methods using String queries and Query Derivation
- Geo-spatial repository queries
- Repository delete queries
- MongoDB JSON based query methods and field restriction
- Full-text search queries
- Projections

WARNING	Reactive Repositories do not support Type-safe Query methods using Querydsl.

12.4.1. Geo-spatial repository queries

As you've just seen there are a few keywords triggering geo-spatial operations within a MongoDB query. The `Near` keyword allows some further modification. Let's have look at some examples:

Example 112. Advanced `Near` queries

```
public interface PersonRepository extends ReactiveMongoRepository<Person, String>

  // { 'location' : { '$near' : [point.x, point.y], '$maxDistance' : distance}}
  Flux<Person> findByLocationNear(Point location, Distance distance);
}
```

Adding a `Distance` parameter to the query method allows restricting results to those within the given distance. If the `Distance` was set up containing a `Metric` we will transparently use `$nearSphere` instead of $code.

NOTE	Reactive Geo-spatial repository queries support the domain type and `GeoResult<T>` results within a reactive wrapper type. `GeoPage` and `GeoResults` are not supported as they contradict the deferred result approach with pre-calculating the average distance. Howevery, you can still pass in a `Pageable` argument to page results yourself.

Example 113. Using `Distance` with `Metrics`

```
Point point = new Point(43.7, 48.8);
Distance distance = new Distance(200, Metrics.KILOMETERS);
… = repository.findByLocationNear(point, distance);
// {'location' : {'$nearSphere' : [43.7, 48.8], '$maxDistance' :
0.03135711885774796}}
```

As you can see using a `Distance` equipped with a `Metric` causes `$nearSphere` clause to be added instead of a plain `$near`. Beyond that the actual distance gets calculated according to the `Metrics` used.

NOTE	Using `@GeoSpatialIndexed(type = GeoSpatialIndexType.GEO_2DSPHERE)` on the target property forces usage of `$nearSphere` operator.

Geo-near queries

```
public interface PersonRepository extends ReactiveMongoRepository<Person, String>

  // {'geoNear' : 'location', 'near' : [x, y] }
  Flux<GeoResult<Person>> findByLocationNear(Point location);

  // No metric: {'geoNear' : 'person', 'near' : [x, y], maxDistance : distance }
  // Metric: {'geoNear' : 'person', 'near' : [x, y], 'maxDistance' : distance,
  //          'distanceMultiplier' : metric.multiplier, 'spherical' : true }
  Flux<GeoResult<Person>> findByLocationNear(Point location, Distance distance);

  // Metric: {'geoNear' : 'person', 'near' : [x, y], 'minDistance' : min,
  //          'maxDistance' : max, 'distanceMultiplier' : metric.multiplier,
  //          'spherical' : true }
  Flux<GeoResult<Person>> findByLocationNear(Point location, Distance min, Distance
max);

  // {'geoNear' : 'location', 'near' : [x, y] }
  Flux<GeoResult<Person>> findByLocationNear(Point location);
}
```

12.5. Infinite Streams with Tailable Cursors

By default, MongoDB will automatically close a cursor when the client exhausts all results supplied by the cursor. Closing a cursor on exhaustion turns a stream into a finite stream. For capped collections you may use a Tailable Cursor that remains open after the client consumes all initially returned data. Using tailable cursors with a reactive data types allows construction of infinite streams. A tailable cursor remains open until it is closed externally. It emits data as new documents arrive in a capped collection.

Tailable cursors may become dead, or invalid, if either the query returns no match or the cursor returns the document at the "end" of the collection and then the application deletes that document.

Example 114. Infinite Stream queries with ReactiveMongoOperations

```
Flux<Person> stream = template.tail(query(where("name").is("Joe")), Person.class);

Disposable subscription = stream.doOnNext(person -> System.out.println(person))
.subscribe();

// ···

// Later: Dispose the subscription to close the stream
subscription.dispose();
```

Spring Data MongoDB Reactive repositories support infinite streams by annotating a query method with `@Tailable`. This works for methods returning `Flux` and other reactive types capable of emitting multiple elements.

Example 115. Infinite Stream queries with ReactiveMongoRepository

```
public interface PersonRepository extends ReactiveMongoRepository<Person, String>
{

  @Tailable
  Flux<Person> findByFirstname(String firstname);

}

Flux<Person> stream = repository.findByFirstname("Joe");

Disposable subscription = stream.doOnNext(System.out::println).subscribe();

// ···

// Later: Dispose the subscription to close the stream
subscription.dispose();
```

TIP Capped collections can be created via `MongoOperations.createCollection`. Just provide the required `CollectionOptions.empty().capped()···`

Chapter 13. Auditing

13.1. Basics

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

13.1.1. Annotation based auditing metadata

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

Example 116. An audited entity

```
class Customer {

  @CreatedBy
  private User user;

  @CreatedDate
  private DateTime createdDate;

  // … further properties omitted
}
```

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

13.1.2. Interface-based auditing metadata

In case you don't want to use annotations to define auditing metadata you can let your domain class implement the `Auditable` interface. It exposes setter methods for all of the auditing properties.

There's also a convenience base class `AbstractAuditable` which you can extend to avoid the need to manually implement the interface methods. Be aware that this increases the coupling of your domain classes to Spring Data which might be something you want to avoid. Usually the annotation based way of defining auditing metadata is preferred as it is less invasive and more flexible.

13.1.3. AuditorAware

In case you use either `@CreatedBy` or `@LastModifiedBy`, the auditing infrastructure somehow needs to become aware of the current principal. To do so, we provide an `AuditorAware<T>` SPI interface that you have to implement to tell the infrastructure who the current user or system interacting with

the application is. The generic type `T` defines of what type the properties annotated with `@CreatedBy` or `@LastModifiedBy` have to be.

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

Example 117. Implementation of AuditorAware based on Spring Security

```
class SpringSecurityAuditorAware implements AuditorAware<User> {

  public User getCurrentAuditor() {

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

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

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

The implementation is accessing the `Authentication` object provided by Spring Security and looks up the custom `UserDetails` instance from it that you have created in your `UserDetailsService` implementation. We're assuming here that you are exposing the domain user through that `UserDetails` implementation but you could also look it up from anywhere based on the `Authentication` found.

13.2. General auditing configuration

Activating auditing functionality is just a matter of adding the Spring Data Mongo `auditing` namespace element to your configuration:

Example 118. Activating auditing using XML configuration

```
<mongo:auditing mapping-context-ref="customMappingContext" auditor-aware-ref=
"yourAuditorAwareImpl"/>
```

Since Spring Data MongoDB 1.4 auditing can be enabled by annotating a configuration class with the `@EnableMongoAuditing` annotation.

Example 119. Activating auditing using JavaConfig

```
@Configuration
@EnableMongoAuditing
class Config {

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

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

Chapter 14. Mapping

Rich mapping support is provided by the `MappingMongoConverter`. `MappingMongoConverter` has a rich metadata model that provides a full feature set of functionality to map domain objects to MongoDB documents.The mapping metadata model is populated using annotations on your domain objects. However, the infrastructure is not limited to using annotations as the only source of metadata information. The `MappingMongoConverter` also allows you to map objects to documents without providing any additional metadata, by following a set of conventions.

In this section we will describe the features of the `MappingMongoConverter`. How to use conventions for mapping objects to documents and how to override those conventions with annotation based mapping metadata.

14.1. Convention based Mapping

`MappingMongoConverter` has a few conventions for mapping objects to documents when no additional mapping metadata is provided. The conventions are:

- The short Java class name is mapped to the collection name in the following manner. The class `com.bigbank.SavingsAccount` maps to `savingsAccount` collection name.
- All nested objects are stored as nested objects in the document and **not** as DBRefs
- The converter will use any Spring Converters registered with it to override the default mapping of object properties to document field/values.
- The fields of an object are used to convert to and from fields in the document. Public JavaBean properties are not used.
- You can have a single non-zero argument constructor whose constructor argument names match top level field names of document, that constructor will be used. Otherwise the zero arg constructor will be used. if there is more than one non-zero argument constructor an exception will be thrown.

14.1.1. How the `_id` field is handled in the mapping layer

MongoDB requires that you have an `_id` field for all documents. If you don't provide one the driver will assign a ObjectId with a generated value. The "_id" field can be of any type the, other than arrays, so long as it is unique. The driver naturally supports all primitive types and Dates. When using the `MappingMongoConverter` there are certain rules that govern how properties from the Java class is mapped to this `_id` field.

The following outlines what field will be mapped to the `_id` document field:

- A field annotated with `@Id` (`org.springframework.data.annotation.Id`) will be mapped to the `_id` field.
- A field without an annotation but named `id` will be mapped to the `_id` field.
- The default field name for identifiers is `_id` and can be customized via the `@Field` annotation.

Table 7. Examples for the translation of `_id` field definitions

Field definition	Resulting Id-Fieldname in MongoDB
`String` id	`_id`
`@Field String` id	`_id`
`@Field("x") String` id	`x`
`@Id String` x	`_id`
`@Field("x") @Id String` x	`_id`

The following outlines what type conversion, if any, will be done on the property mapped to the _id document field.

- If a field named `id` is declared as a String or BigInteger in the Java class it will be converted to and stored as an ObjectId if possible. ObjectId as a field type is also valid. If you specify a value for `id` in your application, the conversion to an ObjectId is detected to the MongoDBdriver. If the specified `id` value cannot be converted to an ObjectId, then the value will be stored as is in the document's _id field.
- If a field named `id` id field is not declared as a String, BigInteger, or ObjectID in the Java class then you should assign it a value in your application so it can be stored 'as-is' in the document's _id field.
- If no field named `id` is present in the Java class then an implicit `_id` file will be generated by the driver but not mapped to a property or field of the Java class.

When querying and updating `MongoTemplate` will use the converter to handle conversions of the `Query` and `Update` objects that correspond to the above rules for saving documents so field names and types used in your queries will be able to match what is in your domain classes.

14.2. Data mapping and type conversion

This section explain how types are mapped to a MongoDB representation and vice versa. Spring Data MongoDB supports all types that can be represented as BSON, MongoDB's internal document format. In addition to these types, Spring Data MongoDB provides a set of built-in converters to map additional types. You can provide your own converters to adjust type conversion, see Overriding Mapping with explicit Converters for further details.

Table 8. Type

Type	Type conversion	Sample
`String`	native	`{"firstname" : "Dave"}`
`double`, `Double`, `float`, `Float`	native	`{"weight" : 42.5}`
`int`, `Integer`, `short`, `Short`	native 32-bit integer	`{"height" : 42}`
`long`, `Long`	native 64-bit integer	`{"height" : 42}`

Type	Type conversion	Sample
`Date`, `Timestamp`	native	`{"date" : ISODate("2019-11-12T23:00:00.809Z")}`
`byte[]`	native	`{"bin" : { "$binary" : "AQIDBA==", "$type" : "00" }}`
`java.util.UUID` (Legacy UUID)	native	`{"uuid" : { "$binary" : "MEaf1CFQ6lSphaa3b9AtlA==", "$type" : "03" }}`
`Date`	native	`{"date" : ISODate("2019-11-12T23:00:00.809Z")}`
`ObjectId`	native	`{"_id" : ObjectId("5707a2690364aba3136ab870")}`
Array, `List`, `BasicDBList`	native	`{"cookies" : [ … ]}`
`boolean`, `Boolean`	native	`{"active" : true}`
`null`	native	`{"value" : null}`
`Document`	native	`{"value" : { … }}`
`Decimal128`	native	`{"value" : NumberDecimal(…)}`
`AtomicInteger` calling `get()` before the actual conversion	converter 32-bit integer	`{"value" : "741" }`
`AtomicLong` calling `get()` before the actual conversion	converter 64-bit integer	`{"value" : "741" }`
`BigInteger`	converter `String`	`{"value" : "741" }`
`BigDecimal`	converter `String`	`{"value" : "741.99" }`
`URL`	converter	`{"website" : "http://projects.spring.io/spring-data-mongodb/" }`
`Locale`	converter	`{"locale : "en_US" }`
`char`, `Character`	converter	`{"char" : "a" }`
`NamedMongoScript`	converter `Code`	`{"_id" : "script name", value: (some javascript code)}`
`java.util.Currency`	converter	`{"currencyCode" : "EUR"}`
`LocalDate` (Joda, Java 8, JSR310-BackPort)	converter	`{"date" : ISODate("2019-11-12T00:00:00.000Z")}`

Type	Type conversion	Sample
`LocalDateTime`, `LocalTime`, `Instant` (Joda, Java 8, JSR310-BackPort)	converter	`{"date" : ISODate("2019-11-12T23:00:00.809Z")}`
`DateTime` (Joda)	converter	`{"date" : ISODate("2019-11-12T23:00:00.809Z")}`
`ZoneId` (Java 8, JSR310-BackPort)	converter	`{"zoneId" : "ECT - Europe/Paris"}`
`Box`	converter	`{"box" : { "first" : { "x" : 1.0 , "y" : 2.0} , "second" : { "x" : 3.0 , "y" : 4.0}}`
`Polygon`	converter	`{"polygon" : { "points" : [ { "x" : 1.0 , "y" : 2.0} , { "x" : 3.0 , "y" : 4.0} , { "x" : 4.0 , "y" : 5.0}]}}`
`Circle`	converter	`{"circle" : { "center" : { "x" : 1.0 , "y" : 2.0} , "radius" : 3.0 , "metric" : "NEUTRAL"}}`
`Point`	converter	`{"point" : { "x" : 1.0 , "y" : 2.0}}`
`GeoJsonPoint`	converter	`{"point" : { "type" : "Point" , "coordinates" : [3.0 , 4.0] }}`
`GeoJsonMultiPoint`	converter	`{"geoJsonLineString" : {"type":"MultiPoint", "coordinates": [ [ 0 , 0 ], [ 0 , 1 ], [ 1 , 1 ] ] }}`
`Sphere`	converter	`{"sphere" : { "center" : { "x" : 1.0 , "y" : 2.0} , "radius" : 3.0 , "metric" : "NEUTRAL"}}`
`GeoJsonPolygon`	converter	`{"polygon" : { "type" : "Polygon", "coordinates" : [[ [ 0 , 0 ], [ 3 , 6 ], [ 6 , 1 ], [ 0 , 0 ] ]] }}`
`GeoJsonMultiPolygon`	converter	`{"geoJsonMultiPolygon" : { "type" : "MultiPolygon", "coordinates" : [ [ [ [ -73.958 , 40.8003 ] , [ -73.9498 , 40.7968 ] ] ], [ [ [ -73.973 , 40.7648 ] , [ -73.9588 , 40.8003 ] ] ] ] }}`
`GeoJsonLineString`	converter	`{ "geoJsonLineString" : { "type" : "LineString", "coordinates" : [ [ 40 , 5 ], [ 41 , 6 ] ] }}`
`GeoJsonMultiLineString`	converter	`{"geoJsonLineString" : { "type" : "MultiLineString", coordinates: [ [ [ -73.97162 , 40.78205 ], [ -73.96374 , 40.77715 ] ], [ [ -73.97880 , 40.77247 ], [ -73.97036 , 40.76811 ] ] ] }}`

14.3. Mapping Configuration

Unless explicitly configured, an instance of `MappingMongoConverter` is created by default when creating a `MongoTemplate`. You can create your own instance of the `MappingMongoConverter` so as to tell it where to scan the classpath at startup your domain classes in order to extract metadata and construct indexes. Also, by creating your own instance you can register Spring converters to use for mapping specific classes to and from the database.

You can configure the `MappingMongoConverter` as well as `com.mongodb.MongoClient` and

MongoTemplate either using Java or XML based metadata. Here is an example using Spring's Java based configuration

Example 120. @Configuration class to configure MongoDB mapping support

```
@Configuration
public class GeoSpatialAppConfig extends AbstractMongoConfiguration {

  @Bean
  public MongoClient mongoClient() {
    return new MongoClient("localhost");
  }

  @Override
  public String getDatabaseName() {
    return "database";
  }

  @Override
  public String getMappingBasePackage() {
    return "com.bigbank.domain";
  }

  // the following are optional


  @Bean
  @Override
  public CustomConversions customConversions() throws Exception {
    List<Converter<?, ?>> converterList = new ArrayList<Converter<?, ?>>();
    converterList.add(new org.springframework.data.mongodb.test
.PersonReadConverter());
    converterList.add(new org.springframework.data.mongodb.test
.PersonWriteConverter());
    return new CustomConversions(converterList);
  }

  @Bean
  public LoggingEventListener<MongoMappingEvent> mappingEventsListener() {
    return new LoggingEventListener<MongoMappingEvent>();
  }
}
```

`AbstractMongoConfiguration` requires you to implement methods that define a `com.mongodb.MongoClient` as well as provide a database name. `AbstractMongoConfiguration` also has a method you can override named `getMappingBasePackage(…)` which tells the converter where to scan for classes annotated with the `@Document` annotation.

You can add additional converters to the converter by overriding the method

afterMappingMongoConverterCreation. Also shown in the above example is a `LoggingEventListener` which logs `MongoMappingEvent` s that are posted onto Spring's `ApplicationContextEvent` infrastructure.

NOTE	AbstractMongoConfiguration will create a MongoTemplate instance and registered with the container under the name `mongoTemplate`.

You can also override the method `UserCredentials getUserCredentials()` to provide the username and password information to connect to the database.

Spring's MongoDB namespace enables you to easily enable mapping functionality in XML

Example 121. XML schema to configure MongoDB mapping support

```
<?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:context="http://www.springframework.org/schema/context"
  xmlns:mongo="http://www.springframework.org/schema/data/mongo"
  xsi:schemaLocation="http://www.springframework.org/schema/context
http://www.springframework.org/schema/context/spring-context-3.0.xsd
    http://www.springframework.org/schema/data/mongo
http://www.springframework.org/schema/data/mongo/spring-mongo-1.0.xsd
    http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans-3.0.xsd">

  <!-- Default bean name is 'mongo' -->
  <mongo:mongo-client host="localhost" port="27017"/>

  <mongo:db-factory dbname="database" mongo-ref="mongoClient"/>

  <!-- by default look for a Mongo object named 'mongo' - default name used for
the converter is 'mappingConverter' -->
  <mongo:mapping-converter base-package="com.bigbank.domain">
    <mongo:custom-converters>
      <mongo:converter ref="readConverter"/>
      <mongo:converter>
        <bean class="org.springframework.data.mongodb.test.PersonWriteConverter"/>
      </mongo:converter>
    </mongo:custom-converters>
  </mongo:mapping-converter>

  <bean id="readConverter" class=
"org.springframework.data.mongodb.test.PersonReadConverter"/>

  <!-- set the mapping converter to be used by the MongoTemplate -->
  <bean id="mongoTemplate" class=
"org.springframework.data.mongodb.core.MongoTemplate">
    <constructor-arg name="mongoDbFactory" ref="mongoDbFactory"/>
    <constructor-arg name="mongoConverter" ref="mappingConverter"/>
  </bean>

  <bean class=
"org.springframework.data.mongodb.core.mapping.event.LoggingEventListener"/>

</beans>
```

The `base-package` property tells it where to scan for classes annotated with the `@org.springframework.data.mongodb.core.mapping.Document` annotation.

14.4. Metadata based Mapping

To take full advantage of the object mapping functionality inside the Spring Data/MongoDB support, you should annotate your mapped objects with the `@Document` annotation. Although it is not necessary for the mapping framework to have this annotation (your POJOs will be mapped correctly, even without any annotations), it allows the classpath scanner to find and pre-process your domain objects to extract the necessary metadata. If you don't use this annotation, your application will take a slight performance hit the first time you store a domain object because the mapping framework needs to build up its internal metadata model so it knows about the properties of your domain object and how to persist them.

Example 122. Example domain object

```
package com.mycompany.domain;

@Document
public class Person {

  @Id
  private ObjectId id;

  @Indexed
  private Integer ssn;

  private String firstName;

  @Indexed
  private String lastName;
}
```

IMPORTANT	The `@Id` annotation tells the mapper which property you want to use for the MongoDB `_id` property and the `@Indexed` annotation tells the mapping framework to call `createIndex(…)` on that property of your document, making searches faster.
IMPORTANT	Automatic index creation is only done for types annotated with `@Document`.

14.4.1. Mapping annotation overview

The MappingMongoConverter can use metadata to drive the mapping of objects to documents. An overview of the annotations is provided below

- `@Id` - applied at the field level to mark the field used for identity purpose.
- `@Document` - applied at the class level to indicate this class is a candidate for mapping to the database. You can specify the name of the collection where the database will be stored.
- `@DBRef` - applied at the field to indicate it is to be stored using a com.mongodb.DBRef.

- `@Indexed` - applied at the field level to describe how to index the field.
- `@CompoundIndex` - applied at the type level to declare Compound Indexes
- `@GeoSpatialIndexed` - applied at the field level to describe how to geoindex the field.
- `@TextIndexed` - applied at the field level to mark the field to be included in the text index.
- `@Language` - applied at the field level to set the language override property for text index.
- `@Transient` - by default all private fields are mapped to the document, this annotation excludes the field where it is applied from being stored in the database
- `@PersistenceConstructor` - marks a given constructor - even a package protected one - to use when instantiating the object from the database. Constructor arguments are mapped by name to the key values in the retrieved Document.
- `@Value` - this annotation is part of the Spring Framework . Within the mapping framework it can be applied to constructor arguments. This lets you use a Spring Expression Language statement to transform a key's value retrieved in the database before it is used to construct a domain object. In order to reference a property of a given document one has to use expressions like: `@Value("#root.myProperty")` where `root` refers to the root of the given document.
- `@Field` - applied at the field level and described the name of the field as it will be represented in the MongoDB BSON document thus allowing the name to be different than the fieldname of the class.
- `@Version` - applied at field level is used for optimistic locking and checked for modification on save operations. The initial value is `zero` which is bumped automatically on every update.

The mapping metadata infrastructure is defined in a separate spring-data-commons project that is technology agnostic. Specific subclasses are using in the MongoDB support to support annotation based metadata. Other strategies are also possible to put in place if there is demand.

Here is an example of a more complex mapping.

```
@Document
@CompoundIndexes({
    @CompoundIndex(name = "age_idx", def = "{'lastName': 1, 'age': -1}")
})
public class Person<T extends Address> {

  @Id
  private String id;

  @Indexed(unique = true)
  private Integer ssn;

  @Field("fName")
  private String firstName;

  @Indexed
  private String lastName;
```

```
  private Integer age;

  @Transient
  private Integer accountTotal;

  @DBRef
  private List<Account> accounts;

  private T address;


  public Person(Integer ssn) {
    this.ssn = ssn;
  }

  @PersistenceConstructor
  public Person(Integer ssn, String firstName, String lastName, Integer age, T
address) {
    this.ssn = ssn;
    this.firstName = firstName;
    this.lastName = lastName;
    this.age = age;
    this.address = address;
  }

  public String getId() {
    return id;
  }

  // no setter for Id.  (getter is only exposed for some unit testing)

  public Integer getSsn() {
    return ssn;
  }

// other getters/setters omitted
```

14.4.2. Customized Object Construction

The mapping subsystem allows the customization of the object construction by annotating a constructor with the `@PersistenceConstructor` annotation. The values to be used for the constructor parameters are resolved in the following way:

- If a parameter is annotated with the `@Value` annotation, the given expression is evaluated and the result is used as the parameter value.
- If the Java type has a property whose name matches the given field of the input document, then it's property information is used to select the appropriate constructor parameter to pass the input field value to. This works only if the parameter name information is present in the java `.class` files which can be achieved by compiling the source with debug information or using the

new `-parameters` command-line switch for javac in Java 8.

- Otherwise a `MappingException` will be thrown indicating that the given constructor parameter could not be bound.

```
class OrderItem {

  private @Id String id;
  private int quantity;
  private double unitPrice;

  OrderItem(String id, @Value("#root.qty ?: 0") int quantity, double unitPrice) {
    this.id = id;
    this.quantity = quantity;
    this.unitPrice = unitPrice;
  }

  // getters/setters ommitted
}

Document input = new Document("id", "4711");
input.put("unitPrice", 2.5);
input.put("qty",5);
OrderItem item = converter.read(OrderItem.class, input);
```

NOTE	The SpEL expression in the `@Value` annotation of the `quantity` parameter falls back to the value `0` if the given property path cannot be resolved.

Additional examples for using the `@PersistenceConstructor` annotation can be found in the MappingMongoConverterUnitTests test suite.

14.4.3. Compound Indexes

Compound indexes are also supported. They are defined at the class level, rather than on individual properties.

NOTE	Compound indexes are very important to improve the performance of queries that involve criteria on multiple fields

Here's an example that creates a compound index of `lastName` in ascending order and `age` in descending order:

Example 123. Example Compound Index Usage

```
package com.mycompany.domain;

@Document
@CompoundIndexes({
    @CompoundIndex(name = "age_idx", def = "{'lastName': 1, 'age': -1}")
})
public class Person {

  @Id
  private ObjectId id;
  private Integer age;
  private String firstName;
  private String lastName;

}
```

14.4.4. Text Indexes

NOTE	The text index feature is disabled by default for mongodb v.2.4.

Creating a text index allows accumulating several fields into a searchable full text index. It is only possible to have one text index per collection so all fields marked with `@TextIndexed` are combined into this index. Properties can be weighted to influence document score for ranking results. The default language for the text index is english, to change the default language set `@Document(language="spanish")` to any language you want. Using a property called `language` or `@Language` allows to define a language override on a per document base.

Example 124. Example Text Index Usage

```
@Document(language = "spanish")
class SomeEntity {

    @TextIndexed String foo;

    @Language String lang;

    Nested nested;
}

class Nested {

    @TextIndexed(weight=5) String bar;
    String roo;
}
```

14.4.5. Using DBRefs

The mapping framework doesn't have to store child objects embedded within the document. You can also store them separately and use a DBRef to refer to that document. When the object is loaded from MongoDB, those references will be eagerly resolved and you will get back a mapped object that looks the same as if it had been stored embedded within your master document.

Here's an example of using a DBRef to refer to a specific document that exists independently of the object in which it is referenced (both classes are shown in-line for brevity's sake):

```
@Document
public class Account {

  @Id
  private ObjectId id;
  private Float total;
}

@Document
public class Person {

  @Id
  private ObjectId id;
  @Indexed
  private Integer ssn;
  @DBRef
  private List<Account> accounts;
}
```

There's no need to use something like `@OneToMany` because the mapping framework sees that you want a one-to-many relationship because there is a List of objects. When the object is stored in MongoDB, there will be a list of DBRefs rather than the `Account` objects themselves.

IMPORTANT	The mapping framework does not handle cascading saves. If you change an `Account` object that is referenced by a `Person` object, you must save the Account object separately. Calling `save` on the `Person` object will not automatically save the `Account` objects in the property `accounts`.

14.4.6. Mapping Framework Events

Events are fired throughout the lifecycle of the mapping process. This is described in the Lifecycle Events section.

Simply declaring these beans in your Spring ApplicationContext will cause them to be invoked whenever the event is dispatched.

14.4.7. Overriding Mapping with explicit Converters

When storing and querying your objects it is convenient to have a `MongoConverter` instance handle the mapping of all Java types to Documents. However, sometimes you may want the `MongoConverter`s do most of the work but allow you to selectively handle the conversion for a particular type or to optimize performance.

To selectively handle the conversion yourself, register one or more one or more `org.springframework.core.convert.converter.Converter` instances with the MongoConverter.

NOTE	Spring 3.0 introduced a core.convert package that provides a general type conversion system. This is described in detail in the Spring reference documentation section entitled Spring Type Conversion.

The method `customConversions` in `AbstractMongoConfiguration` can be used to configure Converters. The examples here at the beginning of this chapter show how to perform the configuration using Java and XML.

Below is an example of a Spring Converter implementation that converts from a Document to a Person POJO.

```
@ReadingConverter
 public class PersonReadConverter implements Converter<Document, Person> {

  public Person convert(Document source) {
    Person p = new Person((ObjectId) source.get("_id"), (String) source.get("name"));
    p.setAge((Integer) source.get("age"));
    return p;
  }
}
```

Here is an example that converts from a Person to a Document.

```
@WritingConverter
public class PersonWriteConverter implements Converter<Person, Document> {

  public Document convert(Person source) {
    Document document = new Document();
    document.put("_id", source.getId());
    document.put("name", source.getFirstName());
    document.put("age", source.getAge());
    return document;
  }
}
```

Chapter 15. Cross Store support

WARNING	Deprecated - will be removed without replacement.

Sometimes you need to store data in multiple data stores and these data stores can be of different types. One might be relational while the other a document store. For this use case we have created a separate module in the MongoDB support that handles what we call cross-store support. The current implementation is based on JPA as the driver for the relational database and we allow select fields in the Entities to be stored in a Mongo database. In addition to allowing you to store your data in two stores we also coordinate persistence operations for the non-transactional MongoDB store with the transaction life-cycle for the relational database.

15.1. Cross Store Configuration

Assuming that you have a working JPA application and would like to add some cross-store persistence for MongoDB. What do you have to add to your configuration?

First of all you need to add a dependency on the module. Using Maven this is done by adding a dependency to your pom:

Example 125. Example Maven pom.xml with spring-data-mongodb-cross-store dependency

```
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi=
"http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>

  ...

    <!-- Spring Data -->
    <dependency>
      <groupId>org.springframework.data</groupId>
      <artifactId>spring-data-mongodb-cross-store</artifactId>
      <version>${spring.data.mongo.version}</version>
    </dependency>

  ...

</project>
```

Once this is done we need to enable AspectJ for the project. The cross-store support is implemented using AspectJ aspects so by enabling compile time AspectJ support the cross-store features will become available to your project. In Maven you would add an additional plugin to the <build> section of the pom:

Example 126. Example Maven pom.xml with AspectJ plugin enabled

```
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi=
"http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>

  ...

  <build>
    <plugins>

      ...

      <plugin>
        <groupId>org.codehaus.mojo</groupId>
        <artifactId>aspectj-maven-plugin</artifactId>
        <version>1.0</version>
        <dependencies>
          <!-- NB: You must use Maven 2.0.9 or above or these are ignored (see
MNG-2972) -->
          <dependency>
            <groupId>org.aspectj</groupId>
            <artifactId>aspectjrt</artifactId>
            <version>${aspectj.version}</version>
          </dependency>
          <dependency>
            <groupId>org.aspectj</groupId>
            <artifactId>aspectjtools</artifactId>
            <version>${aspectj.version}</version>
          </dependency>
        </dependencies>
        <executions>
          <execution>
            <goals>
              <goal>compile</goal>
              <goal>test-compile</goal>
            </goals>
          </execution>
        </executions>
        <configuration>
          <outxml>true</outxml>
          <aspectLibraries>
            <aspectLibrary>
              <groupId>org.springframework</groupId>
              <artifactId>spring-aspects</artifactId>
            </aspectLibrary>
            <aspectLibrary>
              <groupId>org.springframework.data</groupId>
```

```
            <artifactId>spring-data-mongodb-cross-store</artifactId>
          </aspectLibrary>
        </aspectLibraries>
        <source>1.6</source>
        <target>1.6</target>
      </configuration>
    </plugin>

    ...

  </plugins>
 </build>

...

</project>
```

Finally, you need to configure your project to use MongoDB and also configure the aspects that are used. The following XML snippet should be added to your application context:

Example 127. Example application context with MongoDB and cross-store aspect support

```
<?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:jdbc="http://www.springframework.org/schema/jdbc"
  xmlns:jpa="http://www.springframework.org/schema/data/jpa"
  xmlns:mongo="http://www.springframework.org/schema/data/mongo"
  xsi:schemaLocation="http://www.springframework.org/schema/data/mongo
    http://www.springframework.org/schema/data/mongo/spring-mongo.xsd
    http://www.springframework.org/schema/jdbc
    http://www.springframework.org/schema/jdbc/spring-jdbc-3.0.xsd
    http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
    http://www.springframework.org/schema/data/jpa
    http://www.springframework.org/schema/data/jpa/spring-jpa-1.0.xsd">

  ...

  <!--  Mongo config -->
  <mongo:mongo-client host="localhost" port="27017"/>

  <bean id="mongoTemplate" class=
"org.springframework.data.mongodb.core.MongoTemplate">
    <constructor-arg name="mongoClient" ref="mongoClient"/>
    <constructor-arg name="databaseName" value="test"/>
    <constructor-arg name="defaultCollectionName" value="cross-store"/>
  </bean>

  <bean class="org.springframework.data.mongodb.core.MongoExceptionTranslator"/>

  <!--  Mongo cross-store aspect config -->
  <bean class=
"org.springframework.data.persistence.document.mongo.MongoDocumentBacking"
        factory-method="aspectOf">
    <property name="changeSetPersister" ref="mongoChangeSetPersister"/>
  </bean>
  <bean id="mongoChangeSetPersister"
      class=
"org.springframework.data.persistence.document.mongo.MongoChangeSetPersister">
    <property name="mongoTemplate" ref="mongoTemplate"/>
    <property name="entityManagerFactory" ref="entityManagerFactory"/>
  </bean>

  ...

</beans>
```

15.2. Writing the Cross Store Application

We are assuming that you have a working JPA application so we will only cover the additional steps needed to persist part of your Entity in your Mongo database. First you need to identify the field you want persisted. It should be a domain class and follow the general rules for the Mongo mapping support covered in previous chapters. The field you want persisted in MongoDB should be annotated using the `@RelatedDocument` annotation. That is really all you need to do!. The cross-store aspects take care of the rest. This includes marking the field with `@Transient` so it won't be persisted using JPA, keeping track of any changes made to the field value and writing them to the database on successful transaction completion, loading the document from MongoDB the first time the value is used in your application. Here is an example of a simple Entity that has a field annotated with `@RelatedDocument`.

Example 128. Example of Entity with @RelatedDocument

```
@Entity
public class Customer {

  @Id
  @GeneratedValue(strategy = GenerationType.IDENTITY)
  private Long id;

  private String firstName;

  private String lastName;

  @RelatedDocument
  private SurveyInfo surveyInfo;

  // getters and setters omitted
}
```

Example 129. Example of domain class to be stored as document

```
public class SurveyInfo {

  private Map<String, String> questionsAndAnswers;

  public SurveyInfo() {
    this.questionsAndAnswers = new HashMap<String, String>();
  }

  public SurveyInfo(Map<String, String> questionsAndAnswers) {
    this.questionsAndAnswers = questionsAndAnswers;
  }

  public Map<String, String> getQuestionsAndAnswers() {
    return questionsAndAnswers;
  }

  public void setQuestionsAndAnswers(Map<String, String> questionsAndAnswers) {
    this.questionsAndAnswers = questionsAndAnswers;
  }

  public SurveyInfo addQuestionAndAnswer(String question, String answer) {
    this.questionsAndAnswers.put(question, answer);
    return this;
  }
}
```

Once the SurveyInfo has been set on the Customer object above the MongoTemplate that was configured above is used to save the SurveyInfo along with some metadata about the JPA Entity is stored in a MongoDB collection named after the fully qualified name of the JPA Entity class. The following code:

Example 130. Example of code using the JPA Entity configured for cross-store persistence

```
Customer customer = new Customer();
customer.setFirstName("Sven");
customer.setLastName("Olafsen");
SurveyInfo surveyInfo = new SurveyInfo()
  .addQuestionAndAnswer("age", "22")
  .addQuestionAndAnswer("married", "Yes")
  .addQuestionAndAnswer("citizenship", "Norwegian");
customer.setSurveyInfo(surveyInfo);
customerRepository.save(customer);
```

Executing the code above results in the following JSON document stored in MongoDB.

Example 131. Example of JSON document stored in MongoDB

```
{ "_id" : ObjectId( "4d9e8b6e3c55287f87d4b79e" ),
  "_entity_id" : 1,
  "_entity_class" :
"org.springframework.data.mongodb.examples.custsvc.domain.Customer",
  "_entity_field_name" : "surveyInfo",
  "questionsAndAnswers" : { "married" : "Yes",
    "age" : "22",
    "citizenship" : "Norwegian" },
  "_entity_field_class" :
"org.springframework.data.mongodb.examples.custsvc.domain.SurveyInfo" }
```

Chapter 16. JMX support

The JMX support for MongoDB exposes the results of executing the 'serverStatus' command on the admin database for a single MongoDB server instance. It also exposes an administrative MBean, MongoAdmin which will let you perform administrative operations such as drop or create a database. The JMX features build upon the JMX feature set available in the Spring Framework. See here for more details.

16.1. MongoDB JMX Configuration

Spring's Mongo namespace enables you to easily enable JMX functionality

Example 132. XML schema to configure MongoDB

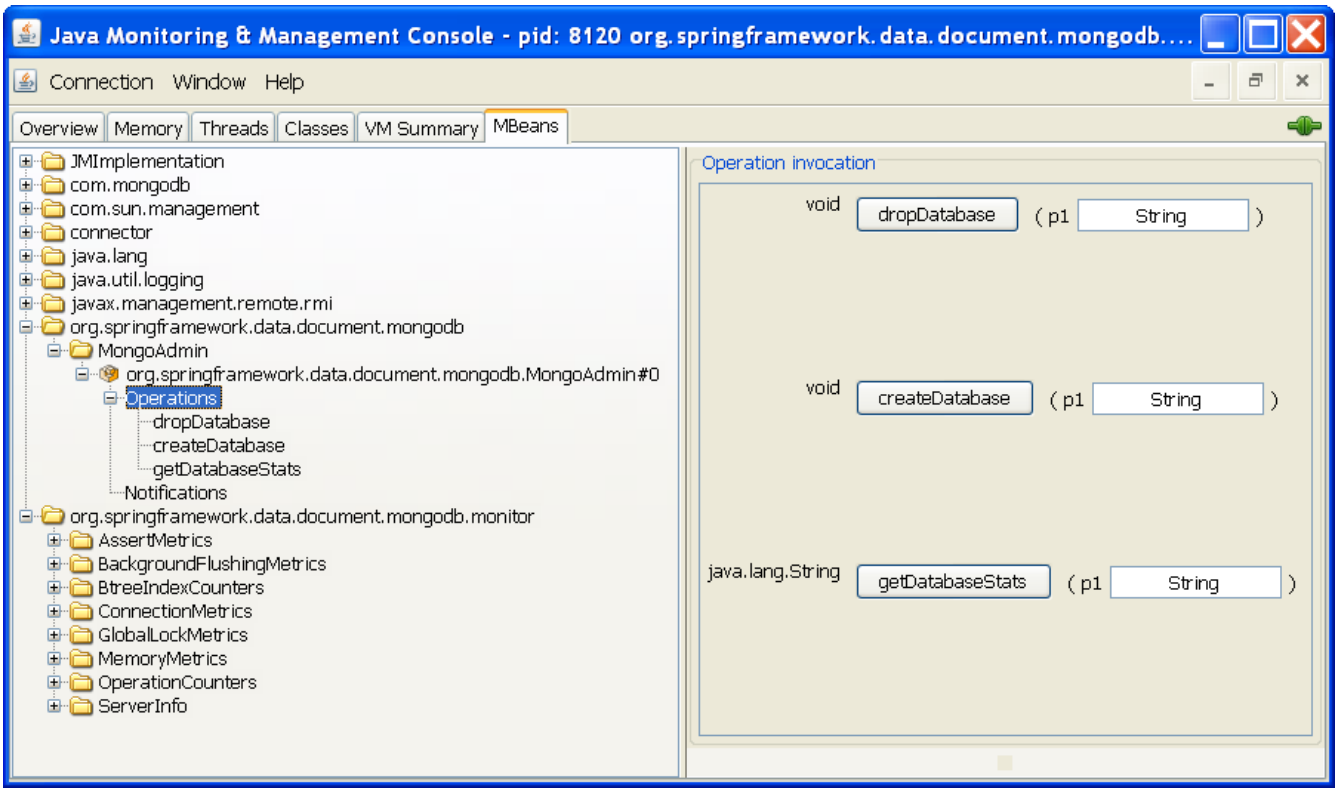
```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:context="http://www.springframework.org/schema/context"
  xmlns:mongo="http://www.springframework.org/schema/data/mongo"
  xsi:schemaLocation="
    http://www.springframework.org/schema/context
    http://www.springframework.org/schema/context/spring-context-3.0.xsd
    http://www.springframework.org/schema/data/mongo
    http://www.springframework.org/schema/data/mongo/spring-mongo-1.0.xsd
    http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans-3.0.xsd">

    <!-- Default bean name is 'mongo' -->
    <mongo:mongo-client host="localhost" port="27017"/>

    <!-- by default look for a Mongo object named 'mongo' -->
    <mongo:jmx/>

    <context:mbean-export/>

    <!-- To translate any MongoExceptions thrown in @Repository annotated classes
-->
    <context:annotation-config/>

    <bean id="registry" class=
"org.springframework.remoting.rmi.RmiRegistryFactoryBean" p:port="1099" />

    <!-- Expose JMX over RMI -->
    <bean id="serverConnector" class=
"org.springframework.jmx.support.ConnectorServerFactoryBean"
        depends-on="registry"
        p:objectName="connector:name=rmi"
        p:serviceUrl=
"service:jmx:rmi://localhost/jndi/rmi://localhost:1099/myconnector" />

</beans>
```

This will expose several MBeans

- AssertMetrics
- BackgroundFlushingMetrics
- BtreeIndexCounters
- ConnectionMetrics
- GlobalLoclMetrics

- MemoryMetrics
- OperationCounters
- ServerInfo
- MongoAdmin

This is shown below in a screenshot from JConsole



Chapter 17. MongoDB 3.0 Support

Spring Data MongoDB allows usage of both MongoDB Java driver generations 2 and 3 when connecting to a MongoDB 2.6/3.0 server running *MMap.v1* or a MongoDB server 3.0 using *MMap.v1* or the *WiredTiger* storage engine.

NOTE	Please refer to the driver and database specific documentation for major differences between those.
NOTE	Operations that are no longer valid using a 3.x MongoDB Java driver have been deprecated within Spring Data and will be removed in a subsequent release.

17.1. Using Spring Data MongoDB with MongoDB 3.0

17.1.1. Configuration Options

Some of the configuration options have been changed / removed for the *mongo-java-driver*. The following options will be ignored using the generation 3 driver:

- autoConnectRetry
- maxAutoConnectRetryTime
- slaveOk

Generally it is recommended to use the `<mongo:mongo-client ⋯ />` and `<mongo:client-options ⋯ />` elements instead of `<mongo:mongo ⋯ />` when doing XML based configuration, since those elements will only provide you with attributes valid for the 3 generation java driver.

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

  <mongo:mongo-client host="127.0.0.1" port="27017">
    <mongo:client-options write-concern="NORMAL" />
  </mongo:mongo-client>

</beans>
```

17.1.2. WriteConcern and WriteConcernChecking

The `WriteConcern.NONE`, which had been used as default by Spring Data MongoDB, was removed in 3.0. Therefore in a MongoDB 3 environment the `WriteConcern` will be defaulted to

`WriteConcern.UNACKNOWLEGED`. In case `WriteResultChecking.EXCEPTION` is enabled the `WriteConcern` will be altered to `WriteConcern.ACKNOWLEDGED` for write operations, as otherwise errors during execution would not be throw correctly, since simply not raised by the driver.

17.1.3. Authentication

MongoDB Server generation 3 changed the authentication model when connecting to the DB. Therefore some of the configuration options available for authentication are no longer valid. Please use the `MongoClient` specific options for setting credentials via `MongoCredential` to provide authentication data.

```
@Configuration
public class ApplicationContextEventTestsAppConfig extends AbstractMongoConfiguration
{

  @Override
  public String getDatabaseName() {
    return "database";
  }

  @Override
  @Bean
  public MongoClient mongoClient() {
    return new MongoClient(singletonList(new ServerAddress("127.0.0.1", 27017)),
      singletonList(MongoCredential.createCredential("name", "db", "pwd".toCharArray(
))));
  }
}
```

In order to use authentication with XML configuration use the `credentials` attribue on `<mongo-client>`.

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

  <mongo:mongo-client credentials="user:password@database" />

</beans>
```

17.1.4. Other things to be aware of

This section covers additional things to keep in mind when using the 3.0 driver.

- `IndexOperations.resetIndexCache()` is no longer supported.
- Any `MapReduceOptions.extraOption` is silently ignored.
- `WriteResult` does not longer hold error information but throws an Exception.
- `MongoOperations.executeInSession(…)` no longer calls `requestStart` / `requestDone`.
- Index name generation has become a driver internal operations, still we use the 2.x schema to generate names.
- Some Exception messages differ between the generation 2 and 3 servers as well as between *MMap.v1* and *WiredTiger* storage engine.

Appendix

Appendix A: Namespace reference

The <repositories /> element

The `<repositories />` element triggers the setup of the Spring Data repository infrastructure. The most important attribute is `base-package` which defines the package to scan for Spring Data repository interfaces. [3: see XML configuration]

Table 9. Attributes

Name	Description
`base-package`	Defines the package to be used to be scanned for repository interfaces extending *Repository (actual interface is determined by specific Spring Data module) in auto detection mode. All packages below the configured package will be scanned, too. Wildcards are allowed.
`repository-impl-postfix`	Defines the postfix to autodetect custom repository implementations. Classes whose names end with the configured postfix will be considered as candidates. Defaults to `Impl`.
`query-lookup-strategy`	Determines the strategy to be used to create finder queries. See Query lookup strategies for details. Defaults to `create-if-not-found`.
`named-queries-location`	Defines the location to look for a Properties file containing externally defined queries.
`consider-nested-repositories`	Controls whether nested repository interface definitions should be considered. Defaults to `false`.

Appendix B: Populators namespace reference

The <populator /> element

The `<populator />` element allows to populate the a data store via the Spring Data repository infrastructure. [4: see XML configuration]

Table 10. Attributes

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

Appendix C: Repository query keywords

Supported query keywords

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

Table 11. Query keywords

Logical keyword	Keyword expressions
`AND`	`And`
`OR`	`Or`
`AFTER`	`After`, `IsAfter`
`BEFORE`	`Before`, `IsBefore`
`CONTAINING`	`Containing`, `IsContaining`, `Contains`
`BETWEEN`	`Between`, `IsBetween`
`ENDING_WITH`	`EndingWith`, `IsEndingWith`, `EndsWith`
`EXISTS`	`Exists`
`FALSE`	`False`, `IsFalse`
`GREATER_THAN`	`GreaterThan`, `IsGreaterThan`
`GREATER_THAN_EQUALS`	`GreaterThanEqual`, `IsGreaterThanEqual`
`IN`	`In`, `IsIn`
`IS`	`Is`, `Equals`, (or no keyword)
`IS_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`

Appendix D: Repository query return types

Supported query return types

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

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

Table 12. Query return types

Return type	Description
`void`	Denotes no return value.
Primitives	Java primitives.
Wrapper types	Java wrapper types.
`T`	An unique entity. Expects the query method to return one result at most. In case no result is found `null` is returned. More than one result will trigger an `IncorrectResultSizeDataAccessException`.
`Iterator<T>`	An `Iterator`.
`Collection<T>`	A `Collection`.
`List<T>`	A `List`.
`Optional<T>`	A Java 8 or Guava `Optional`. Expects the query method to return one result at most. In case no result is found `Optional.empty()` /`Optional.absent()` is returned. More than one result will trigger an `IncorrectResultSizeDataAccessException`.
`Option<T>`	An either Scala or JavaSlang `Option` type. Semantically same behavior as Java 8's `Optional` described above.
`Stream<T>`	A Java 8 `Stream`.
`Future<T>`	A `Future`. Expects method to be annotated with `@Async` and requires Spring's asynchronous method execution capability enabled.
`CompletableFuture<T>`	A Java 8 `CompletableFuture`. Expects method to be annotated with `@Async` and requires Spring's asynchronous method execution capability enabled.
`ListenableFuture`	A `org.springframework.util.concurrent.ListenableFuture`. Expects method to be annotated with `@Async` and requires Spring's asynchronous method execution capability enabled.
`Slice`	A sized chunk of data with information whether there is more data available. Requires a `Pageable` method parameter.
`Page<T>`	A `Slice` with additional information, e.g. the total number of results. Requires a `Pageable` method parameter.
`GeoResult<T>`	A result entry with additional information, e.g. distance to a reference location.

Return type	Description
`GeoResults<T>`	A list of `GeoResult<T>` with additional information, e.g. average distance to a reference location.
`GeoPage<T>`	A `Page` with `GeoResult<T>`, e.g. average distance to a reference location.