Spring Datastore Document - Reference Documentation

1.0.0.M2

Mark Pollack, Thomas Risberg, Oliver Gierke, Costin Leau, Jon Brisbin

Preface

The Spring Datastore Document project applies core Spring concepts to the development of solutions using a 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.

Part I. Introduction

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

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

1. Knowing Spring

Spring Data uses heavily Spring framework's core functionality, such as the IoC container, resource abstract or AOP infrastructure. 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. These being said, the more knowledge one has about the Spring, the faster she will pick up Spring Data Document. Besides the very 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.

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 worth 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 the stores supported by DATADOC. 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 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 1. Why Spring Data - Document?

The Spring Framework is the leading full-stack Java/JEE application framework. It provides a lightweight container and a non-invasive programming model enabled by the use of dependency injection, AOP, and portable service abstractions.

NoSQL storages provide an alternative to classical RDBMS for horizontal scalability and speed. In terms of implementation, Document stores represent one of the most popular types of stores in the NoSQL space. The document database supported by Spring Data are MongoDB and CouchDB, though just MongoDB integration has been released to date.

The goal of the Spring Data Document (or DATADOC) framework is to provide an extension to the Spring programming model that supports writing applications that use Document databases. The Spring framework has always promoted a POJO programming model with a strong emphasis on portability and productivity. These values are caried over into Spring Data Document.

Notable features that are used in Spring Data Document from the Spring framework are the Features that particular, features from the Spring framework that are used are the Conversion Service, JMX Exporters, portable Data Access Exception hierarchy, Spring Expression Language, and Java based IoC container configuration. The programming model follows the familiar Spring 'template' style, so if you are familar with Spring template classes such as JdbcTemplate, JmsTemplate, RestTemplate, you will feel right at home. For example, MongoTemplate removes much of the boilerplate code you would have to write when using the MongoDB driver to save POJOs as well as a rich java based query interface to retrieve POJOs. The programming model also offers a new Repository approach in which the Spring container will provide an implementation of a Repository based soley off an interface definition which can also include custom finder methods.

Chapter 2. Requirements

Spring Data Document 1.x binaries requires JDK level 6.0 and above, and Spring Framework 3.0.x and above.

In terms of document stores, MongoDB preferably version 1.6.5 or later or CouchDB 1.0.1 or later are required.

Chapter 3. Additional Help Resources

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

3.1. Support

There are a few support options available:

3.1.1. Community Forum

The Spring Data forum is a message board for all Spring Data (not just Document) users to share information and help each other. Note that registration is needed *only* for posting.

3.1.2. Professional Support

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

3.2. 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 Spring Community forums. 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 SpringSource Data blog or the project team on Twitter (SpringData)

Chapter 4. Repositories

4.1. Introduction

Implementing a data access layer of an application has been cumbersome for quite a while. Too much boilerplate code had to be written. Domain classes were anemic and haven't been designed in a real object oriented or domain driven manner.

Using both of these technologies makes developers life a lot easier regarding rich domain model's persistence. Nevertheless the amount of boilerplate code to implement repositories especially is still quite high. So the goal of the repository abstraction of Spring Data is to reduce the effort to implement data access layers for various persistence stores significantly

The following chapters will introduce the core concepts and interfaces of Spring Data repositories.

4.2. Core concepts

The central interface in Spring Data repository abstraction is `Repository` (probably not that much of a surprise). It is typeable to the domain class to manage as well as the id type of the domain class and provides some sophisticated functionality around CRUD for the entity managed.

Example 4.1. Repository interface

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

    T save(T entity);                                                    ❶

    T findById(ID primaryKey);                                           ❷

    List<T> findAll();                                                   ❸

    Page<T> findAll(Pageable pageable);                                  ❹

    Long count();                                                        ❺

    void delete(T entity);                                               ❻

    boolean exists(ID primaryKey);                                       ❼

    // … more functionality omitted.
}
```

❶ Saves the given entity.
❷ Returns the entity identified by the given id.
❸ Returns all entities.
❹ Returns a page of entities.
❺ Returns the number of entities.
❻ Deletes the given entity.
❼ Returns whether an entity with the given id exists.

Usually we will have persistence technology specific sub-interfaces to include additional technology specific methods. We will now ship implementations for a variety of Spring Data modules that implement that interface.

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

Example 4.2. PagingAndSortingRepository

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

    List<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);
```

4.3. Query methods

Next to standard CRUD functionality repositories are usually query the underlying datastore. With Spring Data declaring those queries becomes a four-step process (we use the JPA based module as example but that works the same way for other stores):

1. Declare an interface extending the technology specific Repository sub-interface and type it to the domain class it shall handle.

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

2. Declare query methods on the interface.

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

3. Setup Spring to create proxy instances for those interfaces.

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

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

</beans>
```

4. Get the repository instance injected and use it.

```
public class SomeClient {

  @Autowired
  private PersonRepoyitory repository;

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

```
}
```

At this stage we barely scratched the surface of what's possible with the repositories but the general approach should be clear. Let's go through each of these steps and and figure out details and various options that you have at each stage.

4.3.1. Defining repository interfaces

As a very first step you define a domain class specific repository interface to start with. It's got to be typed to the domain class and an ID type so that you get CRUD methods of the `Repository` interface tailored to it.

4.3.2. Defining query methods

4.3.2.1. Query lookup strategies

The next thing we have to discuss is the definition of query methods. There's roughly two main ways how the repository proxy is generally able to come up with the store specific query from the method name. The first option is to derive the quer from the method name directly, the second is using some kind of additionally created query. What detailed options are available pretty much depends on the actual store. However there's got to be some algorithm the decision which actual query to is made.

There's three strategies for the repository infrastructure to resolve the query. The strategy to be used can be configured at the namespace through the `query-lookup-strategy` attribute. However might be the case that some of the strategies are not supported for the specific datastore. Here are your options:

CREATE

This strategy will try to construct a store specific query from the query method's 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 ???.

USE_DECLARED_QUERY

This strategy tries to find a declared query which will be used for execution first. The query could be defined by an annotation somwhere or declared by other means. Please consult the documentation of the specific store to find out what options are available for that store. If the repository infrastructure does not find a declared query for the method at bootstrap time it will fail.

CREATE_IF_NOT_FOUND (default)

This strategy is actually a combination of the both mentioned above. It will try to lookup a declared query first but create a custom method name based query if no declared query was found. This is default lookup strategy and thus will be used if you don't configure anything explicitly. It allows quick query definition by method names but also custom tuning of these queries by introducing declared queries for those who need explicit tuning.

4.3.2.2. Query creation

The query builder mechanism built into Spring Data repository infrastructue is useful to build constraining queries over entities of the repository. We will strip the prefixes `findBy`, `find`, `readBy`, `read`, `getBy` as well as `get` from the method and start parsing the rest of it. At a very basic level you can define conditions on entity

properties and concatenate them with `AND` and `OR`.

Example 4.3. Query creation from method names

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

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

The actual result of parsing that method will of course depend on the persistence store we create the query for. However there are some general things to notice. The expression are usually property traversals combined with operators that can be concatenated. As you can see in the example you can combine property expressions with And and Or. Beyond that you will get support for various operators like `Between`, `LessThan`, `GreaterThan`, `Like` for the property expressions. As the operators supported can vary from datastore to datastore please consult the according part of the reference documentation.

4.3.2.2.1. Property expressions

Property expressions can just refer to a direct property of the managed entity (as you just saw in the example above. On query creation time we already make sure that the parsed property is at a property of the managed domain class. However you can also traverse nested properties to define constraints on. Assume `Persons` have `Addresses` with `ZipCodes`. In that case a method name of

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

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

Now although this should work for most cases, there might be cases where the algorithm could select the wrong property. Suppose our `Person` class has a `addressZip` property as well. Then our 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);
```

4.3.2.3. Special parameter handling

To hand parameters to your query you simply define method parameters as already seen in in examples above. Besides that we will recognizes certain specific types to apply pagination and sorting to your queries dynamically.

Example 4.4. Using Pageable and Sort in query methods

```
Page<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 a `Pageable` instance to the query method to dynamically add paging to your statically defined query. `Sorting` options are handed via the `Pageable` instance, too. If you only need sorting, simply add a `Sort` parameter to your method. As you also can see, simply returning a `List` is possible as well. We will then not retrieve the additional metadata required to build the actual `Page` instance but rather simply restrict the query to lookup only the given range of entities.

> **Note**
>
> To find out how many pages you get for a query entirely we have to trigger an additional count query. This will be derived from the query you actually trigger by default.

4.3.3. Creating repository instances

So now the question is how to create instances and bean definitions for the repository interfaces defined.

4.3.3.1. Spring

The easiest way to do so is by using the Spring namespace that is shipped with each Spring Data module that supports the repository mechanism. Each of those includes a repositories element that allows you to simply define a base packge Spring shall scan for you.

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

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

</beans:beans>
```

In this case we instruct Spring to scan com.acme.repositories and all it's sub packages for interfaces extending the appropriate `Repository` sub-interface (in this case `JpaRepository`). For each interface found it will register the presistence technology specific `FactoryBean` to create the according proxies that handle invocations of the query methods. Each of these beans will be 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 to use wildcards, so that you can have a pattern of packages parsed.

Using filters

By default we will pick up every interface extending the persistence technology specific `Repository` sub-interface located underneath the configured base package and create a bean instance for it. However, you might want to gain finer grained control over which interfaces bean instances get created for. To do this we support the use of `<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.

E.g. to exclude certain interfaces from instantiation as repository, you could use the following configuration:

Example 4.5. Using exclude-filter element

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

This would exclude all interface ending on `SomeRepository` from being instantiated.

Manual configuration

If you'd rather like to manually define which repository instances to create you can do this with nested `<repository />` elements.

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

4.3.3.2. Standalone usage

You can also use the repository infrastructure outside of a Spring container usage. You will still need to have some of the Spring libraries on your classpath but you can generally setup repositories programatically as well. The Spring Data modules providing repository support ship a persistence technology specific RepositoryFactory that can be used as follows:

Example 4.6. Standalone usage of repository factory

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

4.4. Custom implementations

4.4.1. Adding behaviour to single repositories

Often it is necessary to provide a custom implementation for a few repository methods. Spring Data repositories easily allow provide custom repository code and integrate it with generic CRUD abstraction and query method functionality. To enrich a repository with custom functionality you have to define an interface and an implementation for that functionality first and let the repository interface you provided so far extend that custom interface.

Example 4.7. Interface for custom repository functionality

```
interface UserRepositoryCustom {

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

Example 4.8. Implementation of custom repository functionality

```
class UserRepositoryImpl implements UserRepositoryCustom {

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

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

Example 4.9. Changes to the your basic repository interface

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

  // Declare query methods here
}
```

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

Configuration

If you use namespace configuration the repository infrastructure tries to autodetect custom implementations by looking up classes in the package we found a repository using the naming conventions appending the namespace element's attribute `repository-impl-postfix` to the classname. This suffix defaults to `Impl`.

Example 4.10. Configuration example

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

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

The first configuration example will try to lookup a class `com.acme.repository.UserRepositoryImpl` to act as custom repository implementation, where the second example will try to lookup `com.acme.repository.UserRepositoryFooBar`.

Manual wiring

The approach above works perfectly well if your custom implementation uses annotation based configuration and autowring entirely as will be trated as any other Spring bean. If your customly implemented bean needs some special wiring you simply declare the bean and name it after the conventions just descibed. We will then pick up the custom bean by name rather than creating an own instance.

Example 4.11. Manual wiring of custom implementations (I)

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

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

This also works if you use automatic repository lookup without defining single `<repository />` elements.

In case you are not in control of the implementation bean name (e.g. if you wrap a generic repository facade around an existing repository implementation) you can explicitly tell the `<repository />` element which bean to use as custom implementation by using the `repository-impl-ref` attribute.

Example 4.12. Manual wiring of custom implementations (II)

```
<repositories base-package="com.acme.repository">
  <repository id="userRepository" repository-impl-ref="customRepositoryImplementation" />
</repositories>

<bean id="customRepositoryImplementation" class="…">
  <!-- further configuration -->
</bean>
```

4.4.2. Adding custom behaviour to all repositories

In other cases you might want to add a single method to all of your repository interfaces. So the approach just shown is not feasible. The first step to achieve this is adding and intermediate interface to declare the shared behaviour

Example 4.13. An interface declaring custom shared behaviour

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

  void sharedCustomMethod(ID id);
}
```

Now your individual repository interfaces will extend this intermediate interface to include the functionality declared. The second step is to create an implementation of this interface that extends the persistence technology specific repository base class which will act as custom base class for the repository proxies then.

Note

If you're using automatic repository interface detection using the Spring namespace using the interface just as is will cause Spring trying to create an instance of `MyRepository`. This is of course not desired as it just acts as indermediate between `Repository` and the actual repository interfaces

you want to define for each entity. To exclude an interface extending `Repository` from being instantiated as repository instance annotate it with `@NoRepositoryBean`.

Example 4.14. Custom repository base class

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

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

The last step to get this implementation used as base class for Spring Data repositores is replacing the standard `RepositoryFactoryBean` with a custom one using a custom `RepositoryFactory` that in turn creates instances of your `MyRepositoryImpl` class.

Example 4.15. Custom repository factory bean

```
public class MyRepositoryFactoryBean<T extends JpaRepository<?, ?>
  extends JpaRepositoryFactoryBean<T> {

  protected RepositoryFactorySupport getRepositoryFactory(…) {
    return new MyRepositoryFactory(…);
  }

  private static class MyRepositoryFactory extends JpaRepositoryFactory{

    public MyRepositoryImpl getTargetRepository(…) {
      return new MyRepositoryImpl(…);
    }

    public Class<? extends RepositorySupport> getRepositoryClass() {
      return MyRepositoryImpl.class;
    }
  }
}
```

Finally you can either declare beans of the custom factory directly or use the `factory-class` attribute of the Spring namespace to tell the repository infrastructure to use your custom factory implementation.

Example 4.16. Using the custom factory with the namespace

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

Part II. Reference Documentation

Document Structure

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

Chapter 5, *MongoDB support* introduces the MongoDB module feature set.

Chapter 6, *Mongo repositories* introduces the repository support for MongoDB.

Chapter 5. 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
- Low-level mapping using MongoReader/MongoWriter abstractions
- Java based Query, Criteria, and Update DSLs
- Automatic implementatin of Repository interfaces including support for custom finder methods.
- QueryDSL integration to support type-safe queries.
- Cross-store persistance - support for JPA Entities with fields transparently persisted/retrieved using MongoDB
- Log4j log appender
- 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 `org.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.

5.1. Getting Started

Spring MongoDB support requires MongoDB 1.4 or higher and Java SE 5 or higher. The latest production release (1.8.x as of this writing) is recommended. 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 Mongo instance. Once installed starting Mongo 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>1.0.0.M2</version>
  </dependency>

  <dependency>
    <groupId>cglib</groupId>
    <artifactId>cglib</artifactId>
    <version>2.2</version>
  </dependency>

</dependencies>
```

The cglib dependency is there as we will use Spring's Java configuration style. Also change the version of Spring in the pom.xml to be

```
<spring.framework.version>3.0.5.RELEASE</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://maven.springframework.org/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 and add

```
log4j.category.org.springframework.data.document.mongodb=DEBUG
```

Next, in the org.spring.mongodb package in the `src/test/java` directory create a class as shown below.

```
package org.spring.mongodb;

import org.springframework.context.annotation.Configuration;
import org.springframework.data.document.mongodb.MongoTemplate;
import org.springframework.data.document.mongodb.config.AbstractMongoConfiguration;

import com.mongodb.Mongo;

@Configuration
public class MongoConfig extends AbstractMongoConfiguration {

  @Override
  public Mongo mongo() throws Exception {
    return new Mongo("localhost");
  }

  @Override
  public MongoTemplate mongoTemplate() throws Exception {
    return new MongoTemplate(mongo() , "database", "mongoexample");
  }

}
```

Then create a simple Person class to persist

```
package org.spring.mongodb;

public class Person {

  private String id;

  private String name;

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

  public String getId() {
    return id;
  }

  public String getName() {
    return name;
  }

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

}
```

And a main application to run

```
package org.spring.mongodb;

import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;
import org.springframework.context.ApplicationContext;
import org.springframework.context.annotation.AnnotationConfigApplicationContext;
import org.springframework.data.document.mongodb.MongoOperations;
import org.springframework.data.document.mongodb.query.Criteria;
import org.springframework.data.document.mongodb.query.Query;

public class MongoApp {

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

  public static void main(String[] args) {
    ApplicationContext ctx = new AnnotationConfigApplicationContext(MongoConfig.class);
    MongoOperations mongoOps = ctx.getBean(MongoOperations.class);

    mongoOps.insert(new Person("1234", "Joe"));

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

}
```

This will produce the following output

```
MongoPersistentEntityIndexCreator] - <Analyzing class class org.spring.mongodb.Person for index information.>
LoggingEventListener] - <onBeforeConvert: Person [id=1234, name=Joe]>
LoggingEventListener] - <onBeforeSave: Person [id=1234, name=Joe], { "_id" : "1234" , "name" : "Joe"}>
MongoTemplate] - <insert DBObject: { "_id" : "1234" , "name" : "Joe"}>
LoggingEventListener] - <onAfterSave: Person [id=1234, name=Joe], { "_id" : "1234" , "name" : "Joe"}>
MongoTemplate] - <findOne using query: { "name" : "Joe"} in db.collection: database.person>
LoggingEventListener] - <onAfterLoad: { "_id" : "1234" , "name" : "Joe"}>
LoggingEventListener] - <onAfterConvert: { "_id" : "1234" , "name" : "Joe"}, Person [id=1234, name=Joe]>
MongoApp] - <Person [id=1234, name=Joe]>
```

5.2. 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.

5.3. Connecting to MongoDB

One of the first tasks when using MongoDB and Spring is to create a `com.mongodb.Mongo` 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.

5.3.1. Using Java based metadata

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

Example 5.1. Registering a com.mongodb.Mongo object using Java based bean metadata

```
@Configuration
public class AppConfig {

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

This approach allows you to use the standard `com.mongodb.Mongo` API that you may already be used to using but also pollutes the code with the UnknownHostException checked exception.

You may also register an instance of `com.mongodb.Mongo` instance with the container using Spring's `MongoFactoryBean`. As compared to instantiating a `com.mongodb.Mongo` instance directly, the FactoryBean approach has the added advantage of also providing the container with an ExceptionTranslator that can translate Mongo exceptions to exceptions in Spring's portable `DataAccessException` hierarchy. This hierarchy is described in Spring's DAO support features. An example is shown below

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

Example 5.2. Registering a com.mongodb.Mongo object using Spring's MongoFactoryBean and enabling Spring's exception translation support

```
@Configuration
public class AppConfig {
```

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

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

5.3.2. Using XML based metadata

While you can use Spring's traditional <beans/> XML namespace to register an instance of `com.mongodb.Mongo` with the container, the XML can be quite verbose, does not easily support the configuration of public instance variables used with the driver's MongoOptions class, and constructor arguments/names are not the most effective means to distinguish between configuration of replica sets and replica pairs. o address these issues a XML namespace is available to simplify the configuration of a com.mongodb.Mongo instance in XML.

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

Example 5.3. 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 host="localhost" port="27017"/>

    <!-- To translate any MongoExceptions thrown in @Repository annotated classes -->
    <context:annotation-config/>

</beans>
```

A more advanced configuration with MongoOptions is shown below

Example 5.4. XML schema to configure MongoOptinos in MongoDB

```
<beans>

  <mongo:mongo host="localhost" port="27017">
    <mongo:options connectionsPerHost="10"
                   threadsAllowedToBlockForConnectionMultiplier="5"
                   maxWaitTime="12000"
                   connectTimeout="0"
                   socketTimeout="0"
                   autoConnectRetry="0"/>
```

```
  </mongo:mongo/>

</beans>
```

A configuration using replica sets within the XML schema is not yet available. If you would like to configure ReplicaSets use Spring's Java based bean metadata shown below:

Example 5.5. Configuring a com.mongodb.Mongo object with Replica Sets using Java based bean metadata

```
@Configuration
public class AppConfig {

  /*
   * Use the standard Mongo driver API to create a com.mongodb.Mongo instance that supports replica sets
   */
  @Bean
  public Mongo mongo() throws Exception {

    List<ServerAddress> serverAddresses = new ArrayList<ServerAddress>();
    serverAddresses.add( new ServerAddress( "127.0.0.1", 27017 ) );
    serverAddresses.add( new ServerAddress( "127.0.0.1", 27018 ) );

    MongoOptions options = new MongoOptions();
    options.autoConnectRetry = true;

    Mongo mongo  = new Mongo( serverAddresses, options );
    mongo.slaveOk();

    return mongo;
  }
}
```

5.4. Introduction to MongoTemplate

The class `MongoTemplate`, located in the package `org.springframework.data.document.mongodb`, is the central class of the Spring's MongoDB support providng a rich feature set. The template offers convenience operations to create, update, delete and query for MongoDB document and provide 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 Mongo documents and domain classes is done by delegating to an implementation of the interface `MongoConverter`. Spring provides two implementations, `SimpleMappingConverter` and `MongoMappingConverter`, but you can also write your own converter. Please refer to the section on MongoCoverters 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. 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`. The difference in betwee the two is that MongOperations can be passed domain objects instead of `DBObject` and there are fluent APIs for `Query`, `Criteria`, and `Update`

operations instead of `DBObject`..

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

The default converter implementation used by `MongoTemplate` is `SimpleMappingConverter`, which as the name implies, is simple. `SimpleMapingConverter` does not use any additional mapping metadata to converter a domain object to a MongoDB document. As such, it does not support functionality such as DBRefs or creating indexes using annotations on domain classes. For a detailed description of the MongoMappingConverter read the section on Mapping Support.

Another central feature of MongoTemplate is exception translation of exceptions thrown in the Mongo 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 Mongo driver API directly to access functionality not explicitly exposed by the MongoTemplate you can use one of several Execute callback methods. These will give you a reference to a Mongo Collection or DB object. Please see the section Execution Callbacks for more information.

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

5.4.1. Instantiating MongoTemplate

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

Example 5.6. Registering a com.mongodb.Mongo object and enabling Spring's exception translation support

```
@Configuration
public class AppConfig {

    public @Bean Mongo mongo() throws Exception {
        return new Mongo("localhost");
    }

    public @Bean MongoTemplate mongoTemplate() throws Exception {
        return new MongoTemplate(mongo(), "mydatabase", "mycollection");
    }
}
```

There are several overloaded constructors of MongoTemplate. These are

- **MongoTemplate** `(Mongo mongo, String databaseName)` - takes the default database name to operate against
- **MongoTemplate** `(Mongo mongo, String databaseName, String defaultCollectionName)` - adds the default collection name to operate against.
- **MongoTemplate** `(Mongo mongo, String databaseName, String defaultCollectionName, MongoConverter mongoConverter)` - override with a provided MongoConverter. Default is

SimpleMongoConverter

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

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

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

Other properties that you might like to set when creating a MongTemplate are WriteResultCheckingPolicy and the default WriteConcern.

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

5.4.1.1. WriteResultChecking Policy

When in development it is very handy to either log or throw an exception if the 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 ran successfully but the database was not modified according to your expectations. Setting the WriteResultChecking is an enum with the following values, NONE, LOG, EXCEPTION.

The default is to use a WriteResultChecking of NONE.

5.4.1.2. WriteConcern

You can set the WriteConcern property that the MongoTemplate will use for write operations if it has not yet been specifid with the driver. If not set, it will default to the one set in the MongoDB driver's DB or Collection setting.

> **Note**
>
> Setting the WriteConcern to different values when saving an object will be provided in a future release. This will most likely be handled using mapping metadata provided either in the form of annotations on the domain object or by an external fluent DSL.

5.4.2. Configuring the MongoConverter

The SimpleMongoConverter is used by default but if you want to use the more feature rich MappingMongoConverter there are a few steps. Please refer to the mapping section for more information.

5.5. 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 firstName;
  private int age;

  // getters and setter omitted

}
```

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

Note

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

```
public class PersonExample {

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

  @Autowired
  private MongoOperations mongoOps;

  public void doWork() {

    Person p = new Person();
    p.setFirstName("Sven");
    p.setAge(22);

    // Save
    mongoOps.save(p);
    log.debug("Saved: " + p);

    // Find
    p = mongoOps.findOne(query(whereId().is(p.getId())), Person.class);
    log.debug("Found: " + p);

    // Update age to 24 for Sven
    mongoOps.updateFirst(query(where("firstName").is("Sven")), update("age", 24));
    p = mongoOps.findOne(query(whereId().is(p.getId())), Person.class);
    log.debug("Updated: " + p);

    // Delete
    mongoOps.remove(p);

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

}
```

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

```
Saved: PersonWithIdPropertyOfTypeString [id=4d9e82ac94fa72c65a9e7d5f, firstName=Sven, age=22]
findOne using query: { "_id" : { "$oid" : "4d9e82ac94fa72c65a9e7d5f"}} in db.collection: database.personexample
Found: PersonWithIdPropertyOfTypeString [id=4d9e82ac94fa72c65a9e7d5f, firstName=Sven, age=22]
findOne using query: { "_id" : { "$oid" : "4d9e82ac94fa72c65a9e7d5f"}} in db.collection: database.personexample
Updated: PersonWithIdPropertyOfTypeString [id=4d9e82ac94fa72c65a9e7d5f, firstName=Sven, age=24]
remove using query: { "_id" : { "$oid" : "4d9e82ac94fa72c65a9e7d5f"}}
Number of people = : 0
```

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 stynax used in the example is explained in more detail in the section Querying Documents.

5.5.1. Methods for saving and inserting documents

There are several convenient methods on MongoTemplate for saving and inserting your objects. In addition to using a `MongoCoverter` to converter your domain object to the database, you can also use an implementation of the `MongoWriter` interface to have very fine grained control over the conversion process.

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 pass in as an argument only the object to save. In this case the default collection assigned to the template will be used unless the converter overrides this default through the use of more specific mapping metadata. You may also call the save operation with a specific collection name.

When inserting or saving, if the Id property is not set, the assumption is that its value will be autogenerated by the database. As such, for autogeneration 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 5.7. Inserting and retrieving documents using the MongoTemplate

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

    ...

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

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

The four save operations available to you are listed below.

- `void` **save** `(Object objectToSave)` Save the object to the default collection.
- `void` **save** `(String collectionName, Object objectToSave)` Save the object to the specified collection.
- `<T> void` **save** `(T objectToSave, MongoWriter<T> writer)` Save the object into the default collection using the provided writer.
- `<T> void` **save** `(String collectionName, T objectToSave, MongoWriter<T> writer)` Save the object

into the specified collection using the provided writer.

A similar set of insert operations is listed below

- `void` **insert** `(Object objectToSave)` Insert the object to the default collection.
- `void` **insert** `(String collectionName, Object objectToSave)` Insert the object to the specified collection.
- `<T> void` **insert** `(T objectToSave, MongoWriter<T> writer)` Insert the object into the default collection using the provided writer.
- `<T> void` **insert** `(String collectionName, T objectToSave, MongoWriter<T> writer)` Insert the object into the specified collection using the provided writer.

Unless and explicit MongoWriter is passed into the save or insert method, the the template's MongoConverter will be used.

5.5.1.1. Saving using MongoWriter

The MongoWriter interface allows you to have lower level control over the mapping of an object into a DBObject. The MongoWriter interface is

```
/**
 * A MongoWriter is responsible for converting an object of type T to the native MongoDB
 * representation DBObject.
 *
 * @param <T> the type of the object to convert to a DBObject
 */
public interface MongoWriter<T> {

  /**
   * Write the given object of type T to the native MongoDB object representation DBObject.
   *
   * @param t   The object to convert to a DBObject
   * @param dbo The DBObject to use for writing.
   */
  void write(T t, DBObject dbo);

}
```

5.5.1.2. Inserting Lists of objects in 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

- `void` **insertList** `(List<? extends Object> listToSave)` Insert a list of objects into the default collection in a single batch write to the database.
- `void` **insertList** `(String collectionName, List<? extends Object> listToSave)` Insert a list of objects into the specified collection in a single batch write to the database.
- `<T> void` **insertList** `(List<? extends T> listToSave, MongoWriter<T> writer)` Insert the object into the default collection using the provided writer.
- `<T> void` **insertList**`(String collectionName, List<? extends T> listToSave, MongoWriter<T>`

`writer)` Insert a list of objects into the specified collection using the provided MongoWriter instance

5.5.2. 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 5.8. Updating documents using the MongoTemplate

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

          ...

  WriteResult wr = mongoTemplate.updateMulti(query(where("accounts.accountType").is(Account.Type.SAVINGS)),
                                             update.inc("accounts.$.balance", 50.00));
```

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.

5.5.2.1. Methods for executing updates for documents

- `WriteResult` **updateFirst** `(Query query, Update update)` Updates the first object that is found in the default collection that matches the query document with the provided updated document.
- `WriteResult` **updateFirst** `(String collectionName, Query query, Update update)` Updates the first object that is found in the specified collection that matches the query document criteria with the provided updated document.
- `WriteResult` **updateMulti** `(Query query, Update update)` Updates all objects that are found in the default collection that matches the query document criteria with the provided updated document.
- `WriteResult` **updateMulti** `(String collectionName, Query query, Update update)` Updates all objects that are found in the specified collection that matches the query document criteria with the provided updated document.

5.5.2.2. Methods for the Update class

- `Update` **addToSet** `(String key, Object value)` Update using the `$addToSet` update modifier
- `Update` **inc** `(String key, Number inc)` Update using the `$inc` 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` **unset** `(String key)` Update using the `$unset` update modifier

5.5.3. Methods for removing documents

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

- `void` **remove** `(Object object)` Remove the given object from the collection by Id
- `void` **remove** `(Query query)` Remove all documents from the default collection that match the provided query document criteria.
- `void` **remove** `(String collectionName, Query query)` Remove all documents from the specified collection that match the provided query document criteria.
- `<T> void` **remove** `(Query query, Class<T> targetClass)` Same behavior as the remove(Query) method but the Class parameter is used to help convert the Id of the object if it is present in the query.
- `<T> void` **remove** `(String collectionName, Query query, Class<T> targetClass)`Same behavior as remove(Query, Class) but the Class parameter is used to help convert the Id of the object if it is present in the uqery

5.6. Querying Documents

You can express you 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 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.

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

5.6.1. Querying documents in a collection

We saw how to retrieve a single document. We can also query for a collection of documents to be returned as domain objects in a list. 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 5.9. Querying for documents using the MongoTemplate

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

    ...

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

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.document.mongodb.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.

5.6.1.1. Methods for the Criteria class

- `Criteria` **all** `(Object o)`Creates a criterion using the `$all` operator
- `Criteria` **elemMatch** `(Criteria c)` Creates a criterion using the `$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
- `Criteria` **is** `(Object o)`Creates a criterion using the `$is` operator
- `Criteria` **lt** `(Object o)`Creates a criterion using the `$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` **nin** `(Object... o)` Creates a criterion using the `$nin` operator
- `Criteria` **not** `()`Creates a criterion using the `$not` meta operator which affects the clause directly following
- `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

- `Criteria` **and** `(String key)` Adds a chained `Criteria` with the specified `key` to the current `Criteria` and retuns the newly created one

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

- `Criteria` **withinCenter** `(Circle circle)`Creates a geospatial criterion using `$within $center` operators
- `Criteria` **withinCenterSphere** `(Circle circle)` Creates a geospatial criterion using `$within $center` operators. This is only available for Mongo 1.7 and higher.
- `Criteria` **withinBox** `(Box box)`Creates a geospatial criterion using a `$within $box` operation
- `Criteria` **near** `(Point point)`Creates a geospatial criterion using a `$near` operation
- `Criteria` **nearSphere** `(Point point)` Creates a geospatial criterion using `$nearSphere$center` operations. This is only available for Mongo 1.7 and higher.

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

5.6.1.2. Methods for the Query class

- `Query` **addCriteria** `(Criteria criteria)` used to add additional criteria to the query
- `void` **or** `(List<Query> queries)` Creates an or query using the `$or` operator for all of the provided queries
- `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)
- `Sort` **sort** `()` used to provide sort definition for the results

5.6.2. Methods for querying for documents

- `<T> List<T>` **getCollection** `(Class<T> targetClass)` Query for a list of objects of type T from the default collection.
- `<T> List<T>` **getCollection** `(String collectionName, Class<T> targetClass)` Query for a list of objects of type T from the specified collection.
- `<T> List<T>` **getCollection** `(String collectionName, Class<T> targetClass, MongoReader<T> reader)` Query for a list of objects of type T from the specified collection, mapping the DBObject using the provided MongoReader.
- `<T> T` **findOne** `(Query query, Class<T> targetClass)` Map the results of an ad-hoc query on the default MongoDB collection to a single instance of an object of the specified type.
- `<T> T` **findOne** `(Query query, Class<T> targetClass, MongoReader<T> reader)` Map the results of an

ad-hoc query on the default MongoDB collection to a single instance of an object of the specified type.

- `<T> T` **findOne** `(String collectionName, Query query, Class<T> targetClass)` Map the results of an ad-hoc query on the specified collection to a single instance of an object of the specified type.
- `<T> T` **findOne** `(String collectionName, Query query, Class<T> targetClass, MongoReader<T> reader)` Map the results of an ad-hoc query on the specified collection to a single instance of an object of the specified type.
- `<T> List<T>` **find** `(Query query, Class<T> targetClass)` Map the results of an ad-hoc query on the default MongoDB collection to a List of the specified type.
- `<T> List<T>` **find** `(Query query, Class<T> targetClass, MongoReader<T> reader)` Map the results of an ad-hoc query on the default MongoDB collection to a List of the specified type.
- `<T> List<T>` **find** `(String collectionName, Query query, Class<T> targetClass)` Map the results of an ad-hoc query on the specified collection to a List of the specified type.
- `<T> List<T>` **find** `(String collectionName, Query query, Class<T> targetClass, CursorPreparer preparer)` Map the results of an ad-hoc query on the specified collection to a List of the specified type.
- `<T> List<T>` **find** `(String collectionName, Query query, Class<T> targetClass, MongoReader<T> reader)` Map the results of an ad-hoc query on the specified collection to a List of the specified type.
- `<T> T` **findAndRemove** `(Query query, Class<T> targetClass)` Map the results of an ad-hoc query on the default MongoDB 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.
- `<T> T` **findAndRemove** `(Query query, Class<T> targetClass, MongoReader<T> reader)` Map the results of an ad-hoc query on the default MongoDB 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.
- `<T> T` **findAndRemove** `(String collectionName, Query query, Class<T> targetClass)` Map the results of an ad-hoc query on the specified 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.
- `<T> T` **findAndRemove** `(String collectionName, Query query, Class<T> targetClass, MongoReader<T> reader)` Map the results of an ad-hoc query on the specified 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.

The MongoReader can be used

5.6.2.1. Reading using MongoWriter

The MongoReader interface allows you to have lower level control over the mapping of an DBObject into a Java object. This is similar to the role of RowMapper in JdbcTemplate. The MongoReader interface is

```
/**
 * A MongoWriter is responsible for converting a native MongoDB DBObject to an object of type T.
 *
 * @param <T> the type of the object to convert from a DBObject
 */
public interface MongoReader<T> {

  /**
```

```
   * Ready from the native MongoDB DBObject representation to an instance of the class T.
   * The given type has to be the starting point for marshalling the {@link DBObject}
   * into it. So in case there's no real valid data inside {@link DBObject} for the
   * given type, just return an empty instance of the given type.
   *
   * @param clazz the type of the return value
   * @param dbo   theDBObject
   * @return the converted object
   */
  <S extends T> S read(Class<S> clazz, DBObject dbo);
}
```

5.6.3. GeoSpatial Queries

MongoDB supports GeoSpatial queries through the use of operators such as `$near`, `$within`, 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").withinCenter(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").withinCenterSphere(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").withinBox(box)), Venue.class);
```

To find venues near a Point, the following query 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);
```

To find venues near a Point using Spherical coordines 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);
```

Note

Support for the GeoNear command will be provided in the RC release.

5.7. Index and Collection managment

MongoTemplate provides a few methods for managing indexes and collections.

5.7.1. Methods for creating an Index

We can create an index on a collection to improve query performance.

Example 5.10. Creating an index using the MongoTemplate

```
mongoTemplate.ensureIndex("MyCollection", new Index().on("name",Order.ASCENDING));
```

- `void` **ensureIndex** `(IndexDefinition indexDefintion)` Ensure that an index for the provided IndexDefinition exists for the default collection.
- `void` **ensureIndex** `(String collectionName, IndexDefinition indexSpecification)` Ensure that an index for the provided IndexDefinition exists.

You can create both standard indexes and geospatial indexes using the classes `IndexDefinition` and `GeoSpatialIndex` respectfully. For example, given the Venue class defined in a previous section, you would declare a geospatial query as shown below

```
mongoTemplate.ensureIndex(new GeospatialIndex("location"));
```

5.7.2. 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 5.11. Working with collections using the MongoTemplate

```
DBCollection collection = null;
if (!mongoTemplate.getCollectionNames().contains("MyNewCollection")) {
    collection = mongoTemplate.createCollection("MyNewCollection");
}

mongoTemplate.dropCollection("MyNewCollection");
```

- `Set<String>` **getCollectionNames** `()` A set of collection names.
- `boolean` **collectionExists** `(String collectionName)` Check to see if a collection with a given name exists.
- `DBCollection` **createCollection** `(String collectionName)` Create an uncapped collection with the provided name.
- `DBCollection` **createCollection** `(String collectionName, CollectionOptions collectionOptions)` Create a collect with the provided name and options.
- `void` **dropCollection** `(String collectionName)` Drop the collection with the given name.
- `DBCollection` **getCollection** `(String collectionName)` Get a collection by name, creating it if it doesn't exist.
- `DBCollection` **getDefaultCollection** `()` The default collection used by this template.
- `String` **getDefaultCollectionName** `()` The default collection name used by this template.

5.8. Executing Commands

You can also get at the Mongo driver's collection.command( ) method using the executeCommand methods on MongoTemplate. These will also perform exception translation into Spring's Data Access Exception hierarchy.

5.8.1. Methods for executing commands

- `CommandResult` **executeCommand** `(DBObject command)` Execute a MongoDB command.
- `CommandResult` **executeCommand** `(String jsonCommand)` Execute the a MongoDB command expressed as a JSON string.

5.9. 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 infastructure 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 `com.mongodb.DBObject`), you'd register a subclass of `org.springframework.data.document.mongodb.mapping.event.AbstractMappingEventListener` 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.

Example 5.12.

```
public class BeforeConvertListener<BeforeConvertEvent, Person> extends AbstractMappingEventListener {
  @Override
  public void onBeforeConvert(Person p) {
    ... 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.document.mongodb.mapping.event.AbstractMappingEventListener` 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.DBObject`.

Example 5.13.

```
public class BeforeSaveListener<BeforeSaveEvent, Person> extends AbstractMappingEventListener {
  @Override
  public void onBeforeSave(Person p, DBObject dbo) {
    ... 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 DBObject using a MongoConveter.
- `onBeforeSave` - called in MongoTemplate insert, insertList and save operations *before* inserting/saving the DBObject in the database.
- `onAfterSave` - called in MongoTemplate insert, insertList and save operations *after* inserting/saving the

DBObject in the database.

- `onAfterLoad` - called in MongoTempnlate find, findAndRemove, findOne and getCollection methods after the DBObject is retrieved from the database.
- `onAfterConvert` - called in MongoTempnlate find, findAndRemove, findOne and getCollection methods after the DBObject retrieved from the database was converted to a POJO.

5.. 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 Mongo 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.

5.11. Execution Callback

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 thexe execute callback is the preferred way to access the Mongo driver's DB and Collection objects to perform uncommon operations that were not exposed as methods on `MongoTemplate`.

Here is a list of execute callback methods.

- `<T> T` **execute** `(CollectionCallback<T> action)` Executes the given CollectionCallback on the default collection.
- `<T> T` **execute** `(String collectionName, CollectionCallback<T> action)` Executes the given CollectionCallback on the collection of the given name.update using the $addToSet update modifier
- `<T> T` **execute** `(DbCallback<T> action)` Executes a DbCallback 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 CollectionCallback<Boolean>() {
      public Boolean doInCollection(DBCollection collection) throws MongoException, DataAccessException {
        List<DBObject> indexes = collection.getIndexInfo();
        for (DBObject dbo : indexes) {
          if ("location_2d".equals(dbo.get("name"))) {
            return true;
          }
        }
        return false;
      }
    });
```

Chapter 6. Mongo repositories

6.1. Introduction

This chapter will point out the specialties for repository support for MongoDB. This builds on the core repository support explained inChapter 4, *Repositories*. So make sure you've got a sound understanding of the basic concepts explained there.

6.2. 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 6.1. Sample Person entity

```
public class Person {

      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 6.2. Basic repository interface to persist Person entities

```
public interface PersonRepository extends MongoRepository<Person, Long> {

   // additional custom finder methods go here
}
```

The central MongoDB CRUD repository interface is `MongoRepository`. Right now this interface simply serves typing purposes but we will add additional methods to it later. In your Spring configuration simply add

Example 6.3. General mongo 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"
      xmlns:context="http://www.springframework.org/schema/context"
      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
```

```
      http://www.springframework.org/schema/context
      http://www.springframework.org/schema/context/spring-context-3.0.xsd">

  <mongo:mongo id="mongo" />

  <bean id="mongoTemplate" class="org.springframework.data.document.mongodb.MongoTemplate">
    <constructor-arg ref="mongo" />
    <constructor-arg value="database" />
    <constructor-arg value="collection" />
    <constructor-arg>
      <mongo:mapping-converter />
    </constructor-arg>
  </bean>

  <mongo:repositories base-package="com.acme.*.repositories" mongo-template-ref="myMongoTemplate" />
    …

</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.

`MongoRepository` extends `PagingAndSortingRepository` which you can read about inExample 4.1, “Repository interface”. In general 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 `Persons` at a page size of 10 would simply look something like this:

Example 6.4. Paging access to Person entities

```
@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration
public class PersonRepositoryTests {

    @Autowired PersonRepository repository;

    @Test
    public void readsFirstPageCorrectly() {

      Page<Person> persons = repository.findAll(new PageRequest(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.

6.3. Query methods

Most of the data access operations you usually trigger on a repository result a query being executed against the Mongo databases. Defining such a query is just a matter of declaring a method on the repository interface

Example 6.5. PersonRepository with query methods

```
public interface PersonRepository extends MongoRepository<Person, String> {
```

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

    Page<Person> findByFirstname(String firstname, Pageable pageable);

    Person findByShippingAddresses(Address address);

}
```

The first 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}`. The second example shows how pagination is applied 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. The third examples shows that you can query based on properties which are not a primitive type.

Table 6.1. Supported keywords for query methods

Keyword	Sample	Logical result
`GreaterThan`	`findByAgeGreaterThan(int age)`	`{"age" : {"$gt" : age}}`
`LessThan`	`findByAgeLessThan(int age)`	`{"age" : {"$lt" : age}}`
`Between`	`findByAgeBetween(int from, int to)`	`{"age" : {"$gt" : from, "$lt" : to}}`
`IsNotNull`, `NotNull`	`findByFirstnameNotNull()`	`{"age" : {"$ne" : null}}`
`IsNull`, `Null`	`findByFirstnameNull()`	`{"age" : null}`
`Like`	`findByFirstnameLike(String name)`	`{"age" : age}` (age as regex)
(No keyword)	`findByFirstname(String name)`	`{"age" : name}`
`Not`	`findByFirstnameNot(String name)`	`{"age" : {"$ne" : name}}`

6.3.1. Mongo JSON based query methods and field restriction

By adding the annotation `org.springframework.data.document.mongodb.repository.Query` repository finder methods you can specify a Mongo 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.

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,lastname")
  List<Person> findByThePersonsFirstname(String firstname);

}
```

This will return only the firstname, lastname and Id properties of the Person objects, for example age will be null.

6.3.2. Type-safe Query methods

Mongo 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 using 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"),
                                       new PageRequest(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 additiion 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.

Chapter 7. Mapping support

Rich maping support is provided by the `MongoMappingConverter`. `MongoMappingConverter` has a rich metadata model that provides a full feature set of functionality to map domain objects into MongoDB documents. The mapping metadata model is populated using annotations on your domain objects but is not limited to using annotations as the only source of metadata information. In this section we will describe the features of the MongoMappingConverter and how to specify annotation based mapping metadata.

7.1. MongoDB Mapping Configuration

You can configure the MongoMappingConverter as well as Mongo and MongoTemplate eithe using Java or XML based metadata.

Here is an example using Spring's Java based configuration

Example 7.1. @Configuration class to configure MongoDB mapping support

```
@Configuration
public class GeoSpatialAppConfig extends AbstractMongoConfiguration {

  @Bean
  public Mongo mongo() throws Exception {
    return new Mongo("localhost");
  }

  @Bean
  public MongoTemplate mongoTemplate() throws Exception {
    return new MongoTemplate(mongo(), "geospatial", "newyork", mappingMongoConverter());
  }

  // specify which package to scan for @Document objects.
  public String getMappingBasePackage() {
    return "org.springframework.data.document.mongodb";
  }

  // optional
  @Bean
  public LoggingEventListener<MongoMappingEvent> mappingEventsListener() {
    return new LoggingEventListener<MongoMappingEvent>();
  }
}
```

`AbstractMongoConfiguration` requires you to implement methods that define a `Mongo` as well as a `MongoTemplate` object to the container. `AbstractMongoConfiguration` also has a method you can override named '`getMappingBasePackage`' which tells the configuration where to scan for classes annotated with the `@org.springframework.data.document.mongodb.mapping.Document` annotation.

Spring's Mongo namespace enables you to easily enable mapping functionality in XML

Example 7.2. 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/c
           http://www.springframework.org/schema/data/mongo http://www.springframework.org/schema/data/mongo/spr
```

```
          http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans-

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

  <!-- by default look for a Mongo object named 'mongo' - default name used for the converter is 'mappingConvert
  <mongo:mapping-converter base-package="com.mycompany.domain"/>

  <!-- set the mapping converter to be used by the MongoTemplate -->
  <bean id="mongoTemplate" class="org.springframework.data.document.mongodb.MongoTemplate">
    <constructor-arg name="mongo" ref="mongo" />
    <constructor-arg name="databaseName" value="test" />
    <constructor-arg name="defaultCollectionName" value="myCollection" />
    <constructor-arg name="mongoConverter" ref="mappingConverter"/>
  </bean>


</beans
```

This sets up the right objects in the ApplicationContext to perform the full gamut of mapping operations. The `base-package` property tells it where to scan for classes annotated with the `@org.springframework.data.document.mongodb.mapping.Document` annotation.

7.2. Mapping Framework Usage

To take full advantage of the object mapping functionality inside the Spring Data/MongoDB support, you should annotate your mapped objects with the `@org.springframework.data.document.mongodb.mapping.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 7.3. 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 `ensureIndex` on that property of your document, making searches faster.

7.2.1. Mapping annotation overview

The MappingMongoConverter relies on 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 identiy 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.
- `@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 DBObject.
- `@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.

The mapping metadata infrastructure is defined in a seperate spring-data-commons project that is technology agnostic. Specific subclasses are using in the Mongo 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;
        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 ommitted

}
```

7.2.2. Id fields

The @Id annotation is applied to fields. MongoDB lets you store any type as the _id field in the database, including long and string. It is of course common to use ObjectId for this purpose. If the value on the @Id field is not null, it is stored into the database as-is. If it is null, then the converter will assume you want to store an ObjectId in the database. For this to work the field type should be either ObjectId, String, or BigInteger.

7.2.3. Compound Indexes

Compound indexes are also supported. They are defined at the class level, rather than on indidvidual properties. Here's an example that creates a compound index of `lastName` in ascending order and `age` in descending order:

Example 7.4. 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;

}
```

7.2.4. 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):

Example 7.5.

```
@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're wanting 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`.

7.2.5. 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.

7.2.6. 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 DBObjects. However, sometimes you may want the `MongoConverter`'s do most of the work but allow you to selectivly handle the conversion for a particular type. To do this, 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 3 Type Conversion.

The `setConverters` method on `SimpleMongoConverter` and `MappingMongoConverter` should be used for this purpose. The method `afterMappingMongoConverterCreation` in `AbstractMongoConfiguration` can be overriden to configure a MappingMongoConverter.

Chapter 8. Cross Store support

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 implemenatation 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.

8.1. 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 `spring-data-mongodb-cross-store` module. Using Maven this is done by adding a dependency to your pom:

Example 8.1. 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 8.2. 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 8.3. 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 host="localhost" port="27017"/>

  <bean id="mongoTemplate" class="org.springframework.data.document.mongodb.MongoTemplate">
```

```
    <constructor-arg name="mongo" ref="mongo"/>
    <constructor-arg name="databaseName" value="test"/>
    <constructor-arg name="defaultCollectionName" value="cross-store"/>
  </bean>

  <bean class="org.springframework.data.document.mongodb.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>
```

8.2. 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 persited. 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 succesfull 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 @RelatedEntity.

Example 8.4. 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 8.5. 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 8.6. 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 8.7. 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 9. Logging support

An appender for Log4j is provided in the maven module "spring-data-mongodb-log4j". Note, there is no dependency on other Spring Mongo modules, only the MongoDB driver.

9.1. MongoDB Log4j Configuration

Here is an example configuration

```
log4j.rootCategory=INFO, stdout

log4j.appender.stdout=org.springframework.data.document.mongodb.log4j.MongoLog4jAppender
log4j.appender.stdout.layout=org.apache.log4j.PatternLayout
log4j.appender.stdout.layout.ConversionPattern=%d %p [%c] - <%m>%n
log4j.appender.stdout.host = localhost
log4j.appender.stdout.port = 27017
log4j.appender.stdout.database = logs
log4j.appender.stdout.collectionPattern = %X{year}%X{month}
log4j.appender.stdout.applicationId = my.application
log4j.appender.stdout.warnOrHigherWriteConcern = FSYNC_SAFE

log4j.category.org.apache.activemq=ERROR
log4j.category.org.springframework.batch=DEBUG
log4j.category.org.springframework.data.document.mongodb=DEBUG
log4j.category.org.springframework.transaction=INFO
```

The important configuration to look at aside from host and port is the database and collectionPattern. The variables year, month, day and hour are available for you to use in forming a collection name. This is to support the common convention of grouping log information in a collection that corresponds to a specific time period, for example a collection per day.

There is also an applicationId which is put into the stored message. The document stored from logging as the following keys: level, name, applicationId, timestamp, properties, traceback, and message.

Chapter 10. 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.

10.1. MongoDB JMX Configuration

Spring's Mongo namespace enables you to easily enable JMX functionality

Example 10.1. XML schmea 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

<beans>

    <!-- Default bean name is 'mongo' -->
    <mongo:mongo 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