Spring Data Couchbase - Reference Documentation

Michael Nitschinger, Oliver Gierke

Version 1.3.0.RELEASE
2015-03-23

Table of Contents

Preface

This reference documentation describes the general usage of the Spring Data Couchbase library.

Chapter 1. Project Information

- Version control - https://github.com/spring-projects/spring-data-couchbase
- Bugtracker - https://jira.springsource.org/browse/DATACOUCH
- Release repository - https://repo.spring.io/libs-release
- Milestone repository - https://repo.spring.io/libs-milestone
- Snapshot repository - https://repo.spring.io/libs-snapshot

Reference Documentation

Chapter 2. Installation & Configuration

This chapter describes the common installation and configuration steps needed when working with the library.

2.1. Installation

All versions intented for production use are distributed across Maven Central and the Spring release repository. As a result, the library can be included like any other maven dependency:

Example 1. Including the dependency through maven

```
<dependency>
    <groupId>org.springframework.data</groupId>
    <artifactId>spring-data-couchbase</artifactId>
    <version>1.0.0.RELEASE</version>
</dependency>
```

This will pull in several dependencies, including the underlying Couchbase Java SDK, common Spring dependencies and also Jackson as the JSON mapping infrastructure.

You can also grab snapshots from the spring snapshot repository and milestone releases from the milestone repository. Here is an example on how to use the current SNAPSHOT dependency:

Example 2. Using a snapshot version

```
<dependency>
  <groupId>org.springframework.data</groupId>
  <artifactId>spring-data-couchbase</artifactId>
  <version>1.1.0.BUILD-SNAPSHOT</version>
</dependency>

<repository>
  <id>spring-libs-snapshot</id>
  <name>Spring Snapshot Repository</name>
  <url>https://repo.spring.io/libs-snapshot</url>
</repository>
```

Once you have all needed dependencies on the classpath, you can start configuring it. Both Java and XML config are supported. The next sections describe both approaches in detail.

2.2. Annotation-based Configuration ("JavaConfig")

The annotation based configuration approach is getting more and more popular. It allows you to get rid of XML configuration and treat configuration as part of your code directly. To get started, all you need to do is sublcass the `AbstractCouchbaseConfiguration` and implement the abstract methods.

Please make sure to have cglib support in the classpath so that the annotation based configuration works.

Example 3. Extending the `AbstractCouchbaseConfiguration`

```
@Configuration
public class Config extends AbstractCouchbaseConfiguration {

    @Override
    protected List<String> bootstrapHosts() {
        return Collections.singletonList("127.0.0.1");
    }

    @Override
    protected String getBucketName() {
        return "beer-sample";
    }

    @Override
    protected String getBucketPassword() {
        return "";
    }
}
```

All you need to provide is a list of Couchbase nodes to bootstrap into (without any ports, just the IP address or hostname). Please note that while one host is sufficient in development, it is recommended to add 3 to 5 bootstrap nodes here. Couchbase will pick up all nodes from the cluster automatically, but it could be the case that the only node you've provided is experiencing issues while you are starting the application.

The `bucketName` and `password` should be the same as configured in Couchbase Server itself. In the example given, we are connecting to the `beer-sample` bucket which is one of the sample buckets shipped with Couchbase Server and has no password set by default.

Depending on how your environment is setup, the configuration will be automatically picked up by the context or you need to instantiate your own one. How to manage configurations is not scope of this manual, please refer to the spring documentation for more information on that topic.

While not immediately obvious, much more things can be customized and overridden as custom beans from this configuration - we'll touch them in the individual manual sections as needed (for example repositories, validation and custom converters).

2.3. XML-based Configuration

The library provides a custom namespace that you can use in your XML configuration:

Example 4. Basic XML configuration

```
<?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/couchbase
  xsi:schemaLocation="http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans.xsd
    http://www.springframework.org/schema/data/couchbase
    http://www.springframework.org/schema/data/couchbase/spring-couchbase.xsd">

    <couchbase:couchbase bucket="beer-sample" password="" host="127.0.0.1" />

</beans:beans>
```

This code is equivalent to the java configuration approach shown above. It is also possible to configure templates and repositories, which is shown in the appropriate sections.

If you start your application, you should see Couchbase INFO level logging in the logs, indicating that the underlying Couchbase Java SDK is connecting to the database. If any errors are reported, make sure that the given credentials and host information is correct.

Chapter 3. Modeling Entities

This chapter describes how to model Entities and explains their counterpart representation in Couchbase Server itself.

3.1. Documents and Fields

All entities should be annotated with the `@Document` annotation. Also, every field in the entity should be annotated with the `@Field` annotation. While this is - strictly speaking - optional, it helps to reduce edge cases and clearly shows the intent and design of the entity.

There is also a special `@Id` annotation which needs to be always in place. Best practice is to also name the property `id`. Here is a very simple `User` entity:

Example 5. A simple Document with Fields

```java
import org.springframework.data.annotation.Id;
import org.springframework.data.couchbase.core.mapping.Document;
import org.springframework.data.couchbase.core.mapping.Field;

@Document
public class User {

    @Id
    private String id;

    @Field
    private String firstname;

    @Field
    private String lastname;

    public User(String id, String firstname, String lastname) {
        this.id = id;
        this.firstname = firstname;
        this.lastname = lastname;
    }

    public String getId() {
        return id;
    }

    public String getFirstname() {
        return firstname;
    }

    public String getLastname() {
        return lastname;
    }
}
```

Couchbase Server supports automatic expiration for documents. The library implements support for it through the `@Document` annotation. You can set a `expiry` value which translates to the number of seconds until the document gets removed automatically. If you want to make it expire in 10 seconds after mutation, set it like `@Document(expiry = 10)`.

If you want a different representation of the field name inside the document in contrast to the field name used in your entity, you can set a different name on the `@Field` annotation. For example if you want to keep your documents small you can set the firstname field to `@Field("fname")`. In the JSON document, you'll see `{"fname": ".."}` instead of `{"firstname": ".."}`.

The `@Id` annotation needs to be present because every document in Couchbase needs a unique key. This key needs to be any string with a length of maximum 250 characters. Feel free to use whatever fits your use case, be it a UUID, an email address or anything else.

3.2. Datatypes and Converters

The storage format of choice is JSON. It is great, but like many data representations it allows less datatypes than you could express in Java directly. Therefore, for all non-primitive types some form of conversion to and from supported types needs to happen.

For the following entity field types, you don't need to add special handling:

Table 1. Primitive Types

Java Type	JSON Representation
string	string
boolean	boolean
byte	number
short	number
int	number
long	number
float	number
double	number
null	Ignored on write

Since JSON supports objects ("maps") and lists, `Map` and `List` types can be converted naturally. If they only contain primitive field types from the last paragraph, you don't need to add special handling too. Here is an example:

Example 6. A Document with Map and List

```java
@Document
public class User {

    @Id
    private String id;

    @Field
    private List<String> firstnames;

    @Field
    private Map<String, Integer> childrenAges;

    public User(String id, List<String> firstnames, Map<String, Integer>
childrenAges) {
        this.id = id;
        this.firstnames = firstnames;
        this.childrenAges = childrenAges;
    }

}
```

Storing a user with some sample data could look like this as a JSON representation:

Example 7. A Document with Map and List - JSON

```json
{
    "_class": "foo.User",
    "childrenAges": {
        "Alice": 10,
        "Bob": 5
    },
    "firstnames": [
        "Foo",
        "Bar",
        "Baz"
    ]
}
```

You don’t need to break everything down to primitive types and Lists/Maps all the time. Of course, you can also compose other objects out of those primitive values. Let’s modify the last example so that we want to store a `List` of `Children`:

Example 8. A Document with composed objects

```
@Document
public class User {

    @Id
    private String id;

    @Field
    private List<String> firstnames;

    @Field
    private List<Child> children;

    public User(String id, List<String> firstnames, List<Child> children) {
        this.id = id;
        this.firstnames = firstnames;
        this.children = children;
    }

    static class Child {
        private String name;
        private int age;

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

    }

}
```

A populated object can look like:

Example 9. A Document with composed objects - JSON

```
{
  "_class": "foo.User",
  "children": [
    {
      "age": 4,
      "name": "Alice"
    },
    {
      "age": 3,
      "name": "Bob"
    }
  ],
  "firstnames": [
    "Foo",
    "Bar",
    "Baz"
  ]
}
```

Most of the time, you also need to store a temporal value like a `Date`. Since it can't be stored directly in JSON, a conversion needs to happen. The library implements default converters for `Date`, `Calendar` and JodaTime types (if on the classpath). All of those are represented by default in the document as a unix timestamp (number). You can always override the default behavior with custom converters as shown later. Here is an example:

Example 10. A Document with Date and Calendar

```java
@Document
public class BlogPost {

    @Id
    private String id;

    @Field
    private Date created;

    @Field
    private Calendar updated;

    @Field
    private String title;

    public BlogPost(String id, Date created, Calendar updated, String title) {
        this.id = id;
        this.created = created;
        this.updated = updated;
        this.title = title;
    }

}
```

A populated object can look like:

Example 11. A Document with Date and Calendar - JSON

```json
{
  "title": "a blog post title",
  "_class": "foo.BlogPost",
  "updated": 1394610843,
  "created": 1394610843897
}
```

If you want to override a converter or implement your own one, this is also possible. The library implements the general Spring Converter pattern. You can plug in custom converters on bean creation time in your configuration. Here's how you can configure it (in your overriden `AbstractCouchbaseConfiguration`):

Example 12. Custom Converters

```
@Override
public CustomConversions customConversions() {
    return new CustomConversions(Arrays.asList(FooToBarConverter.INSTANCE,
BarToFooConverter.INSTANCE));
}

@WritingConverter
public static enum FooToBarConverter implements Converter<Foo, Bar> {
    INSTANCE;

    @Override
    public Bar convert(Foo source) {
        return /* do your conversion here */;
    }

}

@ReadingConverter
public static enum BarToFooConverter implements Converter<Bar, Foo> {
    INSTANCE;

    @Override
    public Foo convert(Bar source) {
        return /* do your conversion here */;
    }

}
```

There are a few things to keep in mind with custom conversions:

- To make it unambiguous, always use the `@WritingConverter` and `@ReadingConverter` annotations on your converters. Especially if you are dealing with primitive type conversions, this will help to reduce possible wrong conversions.
- If you implement a writing converter, make sure to decode into primitive types, maps and lists only. If you need more complex object types, use the `CouchbaseDocument` and `CouchbaseList` types, which are also understood by the underlying translation engine. Your best bet is to stick with as simple as possible conversions.
- Always put more special converters before generic converters to avoid the case where the wrong converter gets executed.

3.3. Optimistic Locking

Couchbase Server does not support multi-document transactions or rollback. To implement optimistic locking, Couchbase uses a CAS (compare and swap) approach. When a document is mutated, the CAS value also changes. The CAS is opaque to the client, the only thing you need to know is that it changes when the content or a meta information changes too.

In other datastores, similar behavior can be achieved through an arbitrary version field whith a incrementing counter. Since Couchbase supports this in a much better fashion, it is easy to implement. If you want automatic optimistic locking support, all you need to do is add a `@Version` annotation on a long field like this:

Example 13. A Document with optimistic locking.

```
@Document
public class User {

        @Version
        private long version;

        // constructor, getters, setters...
}
```

If you load a document through the template or repository, the version field will be automatically populated with the current CAS value. It is important to note that you shouldn't access the field or even change it on your own. Once you save the document back, it will either succeed or fail with a `OptimisticLockingFailureException`. If you get such an exception, the further approach depends on what you want to achieve application wise. You should either retry the complete load-update-write cycle or propagate the error to the upper layers for proper handling.

3.4. Validation

The library supports JSR 303 validation, which is based on annotations directly in your entities. Of course you can add all kinds of validation in your service layer, but this way its nicely coupled to your actual entities.

To make it work, you need to include two additional dependencies. JSR 303 and a library that implements it, like the one supported by hibernate:

Example 14. Validation dependencies

```
<dependency>
  <groupId>javax.validation</groupId>
  <artifactId>validation-api</artifactId>
</dependency>
<dependency>
  <groupId>org.hibernate</groupId>
  <artifactId>hibernate-validator</artifactId>
</dependency>
```

Now you need to add two beans to your configuration:

Example 15. Validation beans

```
@Bean
public LocalValidatorFactoryBean validator() {
    return new LocalValidatorFactoryBean();
}

@Bean
public ValidatingCouchbaseEventListener validationEventListener() {
    return new ValidatingCouchbaseEventListener(validator());
}
```

Now you can annotate your fields with JSR303 annotations. If a validation on `save()` fails, a `ConstraintViolationException` is thrown.

Example 16. Sample Validation Annotation

```
@Size(min = 10)
@Field
private String name;
```

Chapter 4. Working with Spring Data Repositories

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

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

4.1. Core concepts

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

Example 17. CrudRepository interface

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

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

    T findOne(ID primaryKey);       <2>

    Iterable<T> findAll();          <3>

    Long count();                   <4>

    void delete(T entity);          <5>

    boolean exists(ID primaryKey);  <6>

    //    more functionality omitted.
}
```

① Saves the given entity.

② Returns the entity identified by the given id.

③ Returns all entities.

④ Returns the number of entities.

⑤ Deletes the given entity.

⑥ Indicates whether an entity with the given id exists.

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

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

Example 18. PagingAndSortingRepository

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

  Iterable<T> findAll(Sort sort);

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

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

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

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

Example 19. Derived Count Query

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

  Long countByLastname(String lastname);
}
```

Example 20. Derived Delete Query

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

  Long deleteByLastname(String lastname);

  List<User> removeByLastname(String lastname);

}
```

4.2. Query methods

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

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

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

2. Declare query methods on the interface.

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

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

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

@EnableJpaRepositories
class Config {}
```

or via XML configuration:

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

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

</beans>
```

The JPA namespace is used in this example. If you are using the repository abstraction for any other store, you need to change this to the appropriate namespace declaration of your store module which should be exchanging `jpa` in favor of, for example, `mongodb`. Also, note that the JavaConfig variant doesn't configure a package explictly as the package of the annotated class is used by default. To customize the package to scan

4. Get the repository instance injected and use it.

```
public class SomeClient {

  @Autowired
  private PersonRepository repository;

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

The sections that follow explain each step in detail.

4.3. Defining repository interfaces

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

4.3.1. Fine-tuning repository definition

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

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

Example 21. Selectively exposing CRUD methods

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

  T findOne(ID id);

  T save(T entity);
}

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

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

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

4.4. Defining query methods

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

4.4.1. Query lookup strategies

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

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

4.4.2. Query creation

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

Example 22. Query creation from method names

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

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

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

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

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

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

- The expressions are usually property traversals combined with operators that can be concatenated. You can combine property expressions with `AND` and `OR`. You also get support for operators such as `Between`, `LessThan`, `GreaterThan`, `Like` for the property expressions. The supported operators can vary by datastore, so consult the appropriate part of your reference documentation.

- The method parser supports setting an `IgnoreCase` flag for individual properties (for example, `findByLastnameIgnoreCase(…)`) or for all properties of a type that support ignoring case (usually `String` instances, for example, `findByLastnameAndFirstnameAllIgnoreCase(…)`). Whether ignoring cases is supported may vary by store, so consult the relevant sections in the reference documentation for the store-specific query method.

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

4.4.3. Property expressions

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

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

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

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

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

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

If your property names contain underscores (e.g. `first_name`) you can escape the underscore in the method name with a second underscore. For a `first_name` property the query method would have to be named `findByFirst__name(…)`.

4.4.4. Special parameter handling

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

Example 23. Using Pageable, Slice and Sort in query methods

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

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

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

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

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

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

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

4.4.5. Limiting query results

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

Example 24. Limiting the result size of a query with `Top` and `First`

```
User findFirstByOrderByLastnameAsc();

User findTopByOrderByAgeDesc();

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

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

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

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

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

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

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

4.4.6. Streaming query results

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

Example 25. Stream the result of a query with Java 8 `Stream<T>`

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

Stream<User> readAllByFirstnameNotNull();

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

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

Example 26. Working with a `Stream<T>` result in a try-with-resources block

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

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

4.5. Creating repository instances

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

4.5.1. XML configuration

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

Example 27. Enabling Spring Data repositories via XML

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

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

</beans:beans>
```

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

Using filters

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

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

Example 28. Using exclude-filter element

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

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

4.5.2. JavaConfig

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

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

Example 29. Sample annotation based repository configuration

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

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

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

4.5.3. Standalone usage

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

Example 30. Standalone usage of repository factory

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

4.6. Custom implementations for Spring Data repositories

Often it is necessary to provide a custom implementation for a few repository methods. Spring Data repositories easily allow you to provide custom repository code and integrate it with generic CRUD abstraction and query method functionality.

4.6.1. Adding custom behavior to single repositories

To enrich a repository with custom functionality you first define an interface and an implementation for the custom functionality. Use the repository interface you provided to extend the custom interface.

Example 31. Interface for custom repository functionality

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

Example 32. Implementation of custom repository functionality

```
class UserRepositoryImpl implements UserRepositoryCustom {

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

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

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

Example 33. Changes to the your basic repository interface

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

  // Declare query methods here
}
```

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

Configuration

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

Example 34. Configuration example

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

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

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

Manual wiring

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

Example 35. Manual wiring of custom implementations

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

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

4.6.2. Adding custom behavior to all repositories

The preceding approach is not feasible when you want to add a single method to all your repository interfaces.

1. To add custom behavior to all repositories, you first add an intermediate interface to declare the shared behavior.

   Example 36. An interface declaring custom shared behavior

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

     void sharedCustomMethod(ID id);
   }
   ```

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

3. Next, create an implementation of the intermediate interface that extends the persistence technology-specific repository base class. This class will then act as a custom base class for the repository proxies.

Example 37. Custom repository base class

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

  private final EntityManager entityManager;

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

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

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

The default behavior of the Spring `<repositories />` namespace is to provide an implementation for all interfaces that fall under the `base-package`. This means that if left in its current state, an implementation instance of `MyRepository` will be created by Spring. This is of course not desired as it is just supposed to act as an intermediary between `Repository` and the actual repository interfaces you want to define for each entity. To exclude an interface that extends `Repository` from being instantiated as a repository instance, you can either annotate it with `@NoRepositoryBean` (as seen above) or move it outside of the configured `base-package`.

4. Then create a custom repository factory to replace the default `RepositoryFactoryBean` that will in turn produce a custom `RepositoryFactory`. The new repository factory will then provide your `MyRepositoryImpl` as the implementation of any interfaces that extend the `Repository` interface, replacing the `SimpleJpaRepository` implementation you just extended.

Example 38. Custom repository factory bean

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

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

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

    private final EntityManager em;

    public MyRepositoryFactory(EntityManager em) {

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

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

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

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

Example 39. Using the custom factory with the namespace

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

Example 40. Using the custom factory with the `@Enable` annotation

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

4.7. Spring Data extensions

This section documents a set of Spring Data extensions that enable Spring Data usage in a variety of contexts. Currently most of the integration is targeted towards Spring MVC.

4.7.1. Web support

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

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

Example 41. Enabling Spring Data web support

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

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

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

Example 42. Enabling Spring Data web support in XML

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

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

Basic web support

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

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

DomainClassConverter

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

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

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

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

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

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

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

HandlerMethodArgumentResolvers for Pageable and Sort

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

Example 44. Using Pageable as controller method argument

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

  @Autowired UserRepository repository;

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

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

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

Table 2. Request parameters evaluated for Pageable instances

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

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

In case you need multiple `Pageable` or `Sort` instances to be resolved from the request (for multiple tables, for example) you can use Spring's `@Qualifier` annotation to distinguish one from another. The request parameters then have to be prefixed with `${qualifier}_`. So for a method signature like this:

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

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

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

Hypermedia support for Pageables

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

Example 45. Using a PagedResourcesAssembler as controller method argument

```
@Controller
class PersonController {

  @Autowired PersonRepository repository;

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

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

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

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

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

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

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

4.7.2. Repository populators

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

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

Example 46. Data defined in JSON

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

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

Example 47. Declaring a Jackson repository populator

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

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

</beans>
```

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

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

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

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

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

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

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

</beans>
```

4.7.3. Legacy web support

Domain class web binding for Spring MVC

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

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

  private final UserRepository userRepository;

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

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

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

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

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

PropertyEditors

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

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

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

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

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

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

ConversionServiceIn Spring 3.0 and later the `PropertyEditor` support is superseded by a new conversion infrastructure that eliminates the drawbacks of `PropertyEditors` and uses a stateless X to Y conversion approach. Spring Data now ships with a `DomainClassConverter` that mimics the behavior of `DomainClassPropertyEditorRegistrar`. To configure, simply declare a bean instance and pipe the `ConversionService` being used into its constructor:

```
<mvc:annotation-driven conversion-service="conversionService" />

<bean class="org.springframework.data.repository.support.DomainClassConverter">
  <constructor-arg ref="conversionService" />
</bean>
```

If you are using JavaConfig, you can simply extend Spring MVC's `WebMvcConfigurationSupport` and hand the `FormatingConversionService` that the configuration superclass provides into the `DomainClassConverter` instance you create.

```
class WebConfiguration extends WebMvcConfigurationSupport {

  // Other configuration omitted

  @Bean
  public DomainClassConverter<?> domainClassConverter() {
    return new DomainClassConverter<FormattingConversionService>(mvcConversionService());
  }
}
```

Web pagination

When working with pagination in the web layer you usually have to write a lot of boilerplate code yourself to extract the necessary metadata from the request. The less desirable approach shown in the example below requires the method to contain an `HttpServletRequest` parameter that has to be parsed manually. This example also omits appropriate failure handling, which would make the code even more verbose.

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

  // DI code omitted

  @RequestMapping
  public String showUsers(Model model, HttpServletRequest request) {

    int page = Integer.parseInt(request.getParameter("page"));
    int pageSize = Integer.parseInt(request.getParameter("pageSize"));

    Pageable pageable = new PageRequest(page, pageSize);

    model.addAttribute("users", userService.getUsers(pageable));
    return "users";
  }
}
```

The bottom line is that the controller should not have to handle the functionality of extracting pagination information from the request. So Spring Data ships with a `PageableHandlerMethodArgumentResolver` that will do the work for you. The Spring MVC JavaConfig support exposes a `WebMvcConfigurationSupport` helper class to customize the configuration as follows:

```java
@Configuration
public class WebConfig extends WebMvcConfigurationSupport {

  @Override
  protected void addArgumentResolvers(List<HandlerMethodArgumentResolver>
argumentResolvers) {
    argumentResolvers.add(new PageableHandlerMethodArgumentResolver());
  }
}
```

If you’re stuck with XML configuration you can register the resolver as follows:

```xml
<bean class="  .web.servlet.mvc.method.annotation.RequestMappingHandlerAdapter">
  <property name="customArgumentResolvers">
    <list>
      <bean class="org.springframework.data.web.PageableHandlerMethodArgumentResolver" />
    </list>
  </property>
</bean>
```

Once you’ve configured the resolver with Spring MVC it allows you to simplify controllers down to something like this:

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

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

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

The `PageableArgumentResolver` automatically resolves request parameters to build a `PageRequest` instance. By default it expects the following structure for the request parameters.

Table 3. Request parameters evaluated by PageableHandlerMethodArgumentResolver

`page`	Page you want to retrieve, 0 indexed and defaults to 0.

`size`	Size of the page you want to retrieve, defaults to 20.
`sort`	A collection of sort directives in the format `($propertyname,)[asc|desc]?`.

To retrieve the third page with a maximum page size of 100 with the data sorted by the email property in ascending order use the following url parameter:

```
?page=2&size=100&sort=email,asc
```

To sort the data by multiple properties in different sort order use the following URL parameter:

```
?sort=foo,asc&sort=bar,desc
```

In case you need multiple `Pageable` instances to be resolved from the request (for multiple tables, for example) you can use Spring's `@Qualifier` annotation to distinguish one from another. The request parameters then have to be prefixed with `${qualifier}_`. So for a method signature like this:

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

you have to populate `foo_page` and `bar_page` and the related subproperties.

Configuring a global default on bean declaration the `PageableArgumentResolver` will use a `PageRequest` with the first page and a page size of 10 by default. It will use that value if it cannot resolve a `PageRequest` from the request (because of missing parameters, for example). You can configure a global default on the bean declaration directly. If you might need controller method specific defaults for the `Pageable`, annotate the method parameter with `@PageableDefaults` and specify page (through `pageNumber`), page size (through `value`), `sort` (list of properties to sort by), and `sortDir` (the direction to sort by) as annotation attributes:

```
public String showUsers(Model model,
  @PageableDefaults(pageNumber = 0, value = 30) Pageable pageable) {    }
```

Chapter 5. Couchbase repositories

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

5.1. Configuration

While support for repositories is always present, you need to enable them in general or for a specific namespace. If you extend `AbstractCouchbaseConfiguration`, just use the `@EnableCouchbaseRepositories` annotation. It provides lots of possible options to narrow or customize the search path, one of the most common ones is `basePackages`.

Example 49. Annotation-Based Repository Setup

```
@Configuration
@EnableCouchbaseRepositories(basePackages = {"com.couchbase.example.repos"})
public class Config extends AbstractCouchbaseConfiguration {
    //...
}
```

XML-based configuration is also available:

Example 50. XML-Based Repository Setup

```
<couchbase:repositories base-package="com.couchbase.example.repos" />
```

5.2. Usage

In the simplest case, your repository will extend the `CrudRepository<T, String>`, where T is the entity that you want to expose. Let's look at a repository for a user:

Example 51. A User repository

```
import org.springframework.data.repository.CrudRepository;

public interface UserRepository extends CrudRepository<User, String> {
}
```

Please note that this is just an interface and not an actual class. In the background, when your context gets initialized, actual implementations for your repository descriptions get created and you can access them through regular beans. This means you will save lots of boilerplate code while still exposing full CRUD semantics to your service layer and application.

Now, let's imagine we `@Autowrie` the `UserRepository` to a class that makes use of it. What methods do we have available?

Table 4. Exposed methods on the UserRepository

Method	Description
User save(User entity)	Save the given entity.
Iterable<User> save(Iterable<User> entity)	Save the list of entities.
User findOne(String id)	Find a entity by its unique id.
boolean exists(String id)	Check if a given entity exists by its unique id.
Iterable<User> findAll() (*)	Find all entities by this type in the bucket.
Iterable<User> findAll(Iterable<String> ids)	Find all entities by this type and the given list of ids.
long count() (*)	Count the number of entities in the bucket.
void delete(String id)	Delete the entity by its id.
void delete(User entity)	Delete the entity.
void delete(Iterable<User> entities)	Delete all given entities.
void deleteAll() (*)	Delete all entities by type in the bucket.

Now thats awesome! Just by defining an interface we get full CRUD functionality on top of our managed entity. All methods suffixed with (*) in the table are backed by Views, which is explained later.

If you are coming from other datastore implementations, you might want to implement the `PagingAndSortingRepository` as well. Note that as of now, it is not supported but will be in the future.

While the exposed methods provide you with a great variety of access patterns, very often you need to define custom ones. You can do this by adding method declarations to your interface, which will be automatically resolved to view requests in the background. Here is an example:

Example 52. An extended User repository

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

    List<User> findAllAdmins();

    List<User> findByFirstname(Query query);
}
```

Since we've came across views now multiple times and the `findByFirstname(Query query)` exposes a yet unknown parameter, let's cover that next.

5.3. Backing Views

As a rule of thumb, all repository access methods which are not "by a specific key" require a backing view to find the one or more matching entities. We'll only cover views to the extend which they are needed, if you need in-depth information about them please refer to the official Couchbase Server manual and the Couchbase Java SDK manual.

To cover the basic CRUD methods from the `CrudRepository`, one view needs to be implemented in Couchbase Server. It basically returns all documents for the specific entity and also adds the optional reduce function `_count`.

Since every view has a design document and view name, by convention we default to `all` as the view name and the lower-cased entity name as the design document name. So if your entity is named `User`, then the code expects the `all` view in the `user` design document. It needs to look like this:

Example 53. The all view map function

```
// do not forget the _count reduce function!
function (doc, meta) {
  if (doc._class == "namespace.to.entity.User") {
    emit(null, null);
  }
}
```

Note that the important part in this map function is to only include the document IDs which correspond to our entity. Because the library always adds the `_class` property, this is a quick and easy way to do it. If you have another property in your JSON which does the same job (like a explicit `type` field), then you can use that as well - you don't have to stick to `_class` all the time.

Also make sure to publish your design documents into production so that they can be picked up by the library! Also, if you are curious why we use `emit(null, null)` in the view: the document id is always sent over to the client implicitly, so we can shave off a view bytes in our view by not duplicating the id. If you use `emit(meta.id, null)` it won't hurt much too.

Implementing your custom repository finder methods works the same way. The `findAllAdmins` calls the `allAdmins` view in the `user` design document. Imagine we have a field on our entity which looks like `boolean isAdmin`. We can write a view like this to expose them (we don't need a reduce function for this one):

Example 54. A custom view map function

```
function (doc, meta) {
  if (doc._class == "namespace.to.entity.User" && doc.isAdmin) {
    emit(null, null);
  }
}
```

By now, we've never actually customized our view at query time. This is where the special `Query` argument comes along - like in our `findByFirstname(Query query)` method.

Example 55. A parameterized view map function

```
function (doc, meta) {
  if (doc._class == "namespace.to.entity.User") {
    emit(doc.firstname, null);
  }
}
```

This view not only emits the document id, but also the firstname of every user as the key. We can now run a `Query` which returns us all users with a firstname of "Michael" or "Thomas".

Example 56. Query a repository method with custom params.

```
// Load the bean, or @Autowire it
UserRepository repo = ctx.getBean(UserRepository.class);

// Create the CouchbaseClient Query object
Query query = new Query();

// Filter on those two keys
query.setKeys(ComplexKey.of("Michael", "Thomas"));

// Run the query and get all matching users returned
List<User> users = repo.findByFirstname(query));
```

On all custom finder methods, you can use the `@View` annotation to both customize the design document and view name (to override the conventions).

Please keep in mind that by default, the `Stale.UPDATE_AFTER` mechanism is used. This means that whatever is in the index gets returned, and then the index gets updated. This strikes a good balance between performance and data freshness. You can tune the behavior through the `setStale()` method on the query object. For more details on behavior, please consult the Couchbase Server and Java SDK documentation directly.

Chapter 6. Template & direct operations

The template provides lower level access to the underlying database and also serves as the foundation for repositories. Any time a repository is too high-level for you needs chances are good that the templates will serve you well.

6.1. Supported operations

The template can be accessed through the `couchbaseTemplate` bean out of your context. Once you've got a reference to it, you can run all kinds of operations against it. Other than through a repository, in a template you need to always specify the target entity type which you want to get converted.

To mutate documents, you'll find `save`, `insert` and `update` methods exposed. Saving will insert or update the document, insert will fail if it has been created already and update only works against documents that have already been created.

Since Couchbase Server has different levels of persistence (by default you'll get a positive response if it has been acknowledged in the managed cache), you can provide higher durability options through the overloaded `PersistTo` and/or `ReplicateTo` options. The behaviour is part of the Couchbase Java SDK, please refer to the official documentation for more details.

Removing documents through the `remove` methods works exactly the same.

If you want to load documents, you can do that through the `findById` method, which is the fastest and if possible your tool of choice. The find methods for views are `findByView` which converts it into the target entity, but also `queryView` which exposes lower level semantics.

If you really need low-level semantics, the `couchbaseClient` bean is also always in scope.

Chapter 7. Caching

This chapter describes additional support for caching and `@Cacheable`.

7.1. Configuration & Usage

Technically, caching is not part of spring-data, but is implemented directly in the spring core. Most database implementations in the spring-data package can't support `@Cacheable`, because it is not possible to store arbitrary data.

Couchbase supports both binary and JSON data, so you can get both out of the same database.

To make it work, you need to add the `@EnableCaching` annotation and configure the `cacheManager` bean:

Example 57. `AbstractCouchbaseConfiguration` for Caching

```
@Configuration
@EnableCaching
public class Config extends AbstractCouchbaseConfiguration {
    // general methods

    @Bean
    public CouchbaseCacheManager cacheManager() throws Exception {
        HashMap<String, CouchbaseClient> instances = new HashMap<String,
CouchbaseClient>();
        instances.put("persistent", couchbaseClient());
        return new CouchbaseCacheManager(instances);
    }
}
```

The `persistent` identifier can then be used on the `@Cacheable` annotation to identify the cache manager to use (you can have more than one configured).

Once it is set up, you can annotate every method with the `@Cacheable` annotation to transparently cache it in your couchbase bucket. You can also customize how the key is generated.

Example 58. Caching example

```
@Cacheable(value="persistent", key="'longrunsim-'+#time")
public String simulateLongRun(long time) {
    try {
        Thread.sleep(time);
    } catch(Exception ex) {
        System.out.println("This shouldnt happen...");
    }
    return "I've slept " + time + " miliseconds.;
}
```

If you run the method multiple times, you'll see a set operation happening first, followed by multiple get operations and no sleep time (which fakes the expensive execution). You can store whatever you want, if it is JSON of course you can access it through views and look at it in the Web UI.

Appendix

Appendix A: Namespace reference

The <repositories /> element

The `<repositories />` element triggers the setup of the Spring Data repository infrastructure. The most important attribute is `base-package` which defines the package to scan for Spring Data repository interfaces. [see XML configuration]

Table 5. Attributes

Name	Description
`base-package`	Defines the package to be used to be scanned for repository interfaces extending *Repository (actual interface is determined by specific Spring Data module) in auto detection mode. All packages below the configured package will be scanned, too. Wildcards are allowed.
`repository-impl-postfix`	Defines the postfix to autodetect custom repository implementations. Classes whose names end with the configured postfix will be considered as candidates. Defaults to `Impl`.
`query-lookup-strategy`	Determines the strategy to be used to create finder queries. See Query lookup strategies for details. Defaults to `create-if-not-found`.
`named-queries-location`	Defines the location to look for a Properties file containing externally defined queries.
`consider-nested-repositories`	Controls whether nested repository interface definitions should be considered. Defaults to `false`.

Appendix B: Populators namespace reference

The <populator /> element

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

Table 6. Attributes

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

Appendix C: Repository query keywords

Supported query keywords

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

Table 7. Query keywords

Logical keyword	Keyword expressions
`AND`	`And`
`OR`	`Or`
`AFTER`	`After`, `IsAfter`
`BEFORE`	`Before`, `IsBefore`
`CONTAINING`	`Containing`, `IsContaining`, `Contains`
`BETWEEN`	`Between`, `IsBetween`
`ENDING_WITH`	`EndingWith`, `IsEndingWith`, `EndsWith`
`EXISTS`	`Exists`
`FALSE`	`False`, `IsFalse`
`GREATER_THAN`	`GreaterThan`, `IsGreaterThan`
`GREATER_THAN_EQUALS`	`GreaterThanEqual`, `IsGreaterThanEqual`
`IN`	`In`, `IsIn`
`IS`	`Is`, `Equals`, (or no keyword)
`IS_NOT_NULL`	`NotNull`, `IsNotNull`
`IS_NULL`	`Null`, `IsNull`
`LESS_THAN`	`LessThan`, `IsLessThan`
`LESS_THAN_EQUAL`	`LessThanEqual`, `IsLessThanEqual`
`LIKE`	`Like`, `IsLike`
`NEAR`	`Near`, `IsNear`
`NOT`	`Not`, `IsNot`
`NOT_IN`	`NotIn`, `IsNotIn`
`NOT_LIKE`	`NotLike`, `IsNotLike`

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

Appendix D: Repository query return types

Supported query return types

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

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

Table 8. Query return types

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