# Spring Data Couchbase - Reference Documentation

Michael Nitschinger, Oliver Gierke

Version 1.2.3.RELEASE
2015-07-01

# 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*

```
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*

```
@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*

```
{
    "_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*

```
@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*

```
{
  "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. 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.

## 4.1. 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 17. Annotation-Based Repository Setup*

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

XML-based configuration is also available:

*Example 18. XML-Based Repository Setup*

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

## 4.2. 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 19. 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 2. 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 20. 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.

## 4.3. 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 21. 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 22. 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 23. 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 24. 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 5. 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.

## 5.1. 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 6. Caching

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

## 6.1. 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 25. `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 26. 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