Spring Data Couchbase - Reference Documentation

Michael Nitschinger, Oliver Gierke, Simon Baslé

Version 3.0.1.RELEASE, 2017-10-27

Table of Contents

Preface

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

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

Migrating from Spring Data Couchbase 1.x to 2.x

This chapter is a quick reference of what major changes have been introduced in 2.0.x and gives a high-level overview of things to consider when migrating.

Configuration

The configuration, xml schema, etc… has changed to take the evolution of the 2.x SDK API into account.

Where a single `CouchbaseClient` bean was previously the only bean declarable, you can now declare a `Cluster` bean (`<couchbase:cluster>`), one or more `Bucket` beans (`<couchbase:bucket>`) and even tune the SDK via a `CouchbaseEnvironment` bean (`<couchbase:env>`). All of these can also be created via Java Config method by extending `AbstractCouchbaseConfig`.

The cluster bean lists the nodes to connect through (and references the environment bean if tuning is necessary) while the bucket beans map to bucket names and passwords and actually opens the connections internally.

You can define more beans that are used for internal configuration of the Spring Data Couchbase module (`MappingContext`, `CouchbaseConverter`, `TranslationService`, …).

For more information, see Installation & Configuration.

Repository queries

The view-backed query method has evolved and support for N1QL has been introduced. As a result, there are now 4 ways of doing repository queries:

- Simple View query (to return all elements emitted by a view) - @View annotated without `viewName`
- Intermediate View query by query derivation (to provide some criteria for the view) @View annotated with `viewName`
- N1QL with explicit statements inline - `@Query` annotated with value
- N1QL query derivation - `@Query` annotated without value / no annotation (default)

View backed queries are associated with the `@View` annotation, while N1QL backed queries are associated with the `@Query` annotation.

N1QL query derivation is now the default query method (and there the `@Query` annotation is optional).

See N1QL based querying and Backing Views for more information.

Backing views and view query changes

IMPORTANT	The `all` view is still backing most CRUD operations, but custom repository methods are now by default backed by N1QL.

To instead back them with views, use the `@View` annotation explicitly.

Without a `viewName` specified, the view will be guessed from method name (stripping `count` or `find` prefix). Otherwise, query derivation will be used to parameterize the view query from the method name and parameters.

Passing a ViewQuery object as a parameter to a custom repository method

This behavior has been removed and the recommended approach is now to either use query derivation (if the query parameters are simple enough) or [repositories.single-repository-behaviour].

For instance, for a view emitting user lastNames, the following:

```
@View
List<User> findByLastname(ViewQuery.from("","").key("test").limit(3));
```

is to be replaced by the (more flexible):

```
@View("byLastName")
List<User> findFirst3ByLastnameEquals(String lastName);
```

Reduce in views

Reduce is now supported in view-based querying.

It can be triggered by prefixing the method name with `count` instead of `find`. For example: `countByLastnameContains(String word)` instead of `findByLastnameContains(String word)`.

Alternatively, it can be explicitly be activated by setting `reduce = true` on the `@View` annotation.

Be sure to construct your view correctly:

- specify a reduce function that matches the method return type, which can be anything, eg. long or JSON object
- emit a simple key (not `null` nor a compound key).
- emit a value suitable for the reduce to work (typically `_count` doesn't need any particular value, but `_stats` will need a numerical value, in addition to the key).

Reference Documentation

Chapter 1. Installation & Configuration

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

1.1. Installation

All versions intended 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>2.0.0.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>2.1.0.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.

1.2. 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 subclcass 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> getBootstrapHosts() {
        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 set up, the configuration will be automatically picked up by the context or you need to instantiate your own one. How to manage configurations is not in scope of this manual, please refer to the spring documentation for more information on that topic.

Additionally, the SDK environment can be tuned by overriding the `getEnvironment()` method to return a properly tuned `CouchbaseEnvironment`.

While not immediately obvious, much more things can be customized and overridden as custom beans from this configuration (for example repositories, query consistency, validation and custom converters).

TIP If you use `SyncGateway` and `CouchbaseMobile`, you may run into problem with fields prefixed by `_`. Since Spring Data Couchbase by default stores the type information as a `_class` attribute this can be problematic. Override `typeKey()` (for example to return `MappingCouchbaseConverter.TYPEKEY_SYNCGATEWAY_COMPATIBLE`) to change the name of said attribute.

TIP For generated queries, if you want strong consistency (at the expense of performance), you can override `getDefaultConsistency()` and return `Consistency.READ_YOUR_OWN_WRITES`.

1.3. 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:cluster>
      <couchbase:node>127.0.0.1</couchbase:node>
    </couchbase:cluster>

    <!-- This is needed to probe the server for N1QL support -->
    <!-- Can be either cluster credentials or a bucket credentials -->
    <couchbase:clusterInfo login="beer-sample" password=""/>

    <couchbase:bucket bucketName="beer-sample" bucketPassword=""/>
</beans:beans>
```

This code is equivalent to the java configuration approach shown above. You can customize the SDK `CouchbaseEnvironment` via the `<couchbase:env/>` tag, that supports most tuning parameters as attributes. It is also possible to configure templates and repositories, which is shown in the appropriate sections.

IMPORTANT The XML configuration **must** include the `clusterInfo` credentials, in order to be able to detect N1QL feature.

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 are correct.

Chapter 2. Modeling Entities

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

2.1. 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 from the Couchbase SDK. While this is - strictly speaking - optional, it helps to reduce edge cases and clearly shows the intent and design of the entity. It can also be used to store the field under a different name.

There is also a special `@Id` annotation which needs to be always in place. Best practice is to also name the property `id`.

TIP	Both the Couchbase SDK and Spring Data define their own `@Id` annotation. Either can be used (the Spring Data one will get priority if both are found on different fields).

Here is a very simple `User` entity:

Example 5. A simple Document with Fields

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

@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)`. Alternatively, you can configure the expiry using Spring's property support and the `expiryExpression` parameter, to allow for dynamically changing the expiry value. For example: `@Document(expiryExpression = "${valid.document.expiry}")`. The property must be resolvable to an int value and the two approaches cannot be mixed.

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.

2.2. 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

```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
}
```

Optionally, Date can be converted to and from ISO-8601 compliant strings by setting system property `org.springframework.data.couchbase.useISOStringConverterForDate` to true. 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 overridden `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.
- For dates, reading converters should be able to read from any `Number` (not just `Long`). This is required for N1QL support.

2.3. 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 with 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.

2.4. 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;
```

2.5. Auditing

Entities can be automatically audited (tracing which user created the object, updated the object, and at what times) through Spring Data auditing mechanisms.

First, note that only entities that have a `@Version` annotated field can be audited for creation (otherwise the framework will interpret a creation as an update).

Auditing works by annotating fields with `@CreatedBy`, `@CreatedDate`, `@LastModifiedBy` and `@LastModifiedDate`. The framework will automatically inject the correct values on those fields when persisting the entity. The xxxDate annotations must be put on a `Date` field (or compatible, eg. jodatime classes) while the xxxBy annotations can be put on fields of any class `T` (albeit both fields must be of the same type).

To configure auditing, first you need to have an auditor aware bean in the context. Said bean must be of type `AuditorAware<T>` (allowing to produce a value that can be stored in the xxxBy fields of type `T` we saw earlier). Secondly, you must activate auditing in your `@Configuration` class by using the `@EnableCouchbaseAuditing` annotation.

Here is an example:

Example 17. Sample Auditing Entity

```
@Document
public class AuditedItem {

  @Id
  private final String id;

  private String value;

  @CreatedBy
  private String creator;

  @LastModifiedBy
  private String lastModifiedBy;

  @LastModifiedDate
  private Date lastModification;

  @CreatedDate
  private Date creationDate;

  @Version
  private long version;

  //..omitted constructor/getters/setters/...
}
```

Notice both `@CreatedBy` and `@LastModifiedBy` are both put on a `String` field, so our `AuditorAware` must work with `String`.

Example 18. Sample AuditorAware implementation

```
public class NaiveAuditorAware implements AuditorAware<String> {

  private String auditor = "auditor";

  @Override
  public String getCurrentAuditor() {
    return auditor;
  }

  public void setAuditor(String auditor) {
    this.auditor = auditor;
  }
}
```

To tie all that together, we use the java configuration both to declare an AuditorAware bean and to activate auditing:

Example 19. Sample Auditing Configuration

```
@Configuration
@EnableCouchbaseAuditing //this activates auditing
public class AuditConfiguration extends AbstractCouchbaseConfiguration {

    //... a few abstract methods omitted here

    // this creates the auditor aware bean that will feed the annotations
    @Bean
    public NaiveAuditorAware testAuditorAware() {
      return new NaiveAuditorAware();
    }
```

Chapter 3. Auto generating keys

This chapter describes how couchbase document keys can be auto-generated using builtin mechanisms. There are two types of auto-generation strategies supported.

- Key generation using attributes
- Key generation using uuid

NOTE The maximum key length supported by couchbase is 250 bytes.

3.1. Configuration

Keys to be auto-generated should be annotated with `@GeneratedValue`. The default strategy is `USE_ATTRIBUTES`. Prefix and suffix for the key can be provided as part of the entity itself, these values are not persisted, they are only used for key generation. The prefixes and suffixes are ordered using the `order` value. The default order is `0`, multiple prefixes without order will overwrite the previous. If a value for id is already available, auto-generation will be skipped. The delimiter for concatenation can be provided using `delimiter`, the default delimiter is `.`.

Example 20. Annotation for GeneratedValue

```
@Document
public class User {
     @Id @GeneratedValue(strategy = USE_ATTRIBUTES, delimiter = ".")
     private String id;
     @IdPrefix(order=0)
     private String userPrefix;
     @IdSuffix(order=0)
     private String userSuffix;
     ...
}
```

Common prefix and suffix for all entities keys can be added to `CouchbaseTemplate` directly. Once added to the `CouchbaseTemplate`, they become immutable. These settings are always applied irrespective of the `GeneratedValue` annotation.

Example 21. Common key settings in CouchbaseTemplate

```
@Autowired
CouchbaseTemplate couchbaseTemplate;
...
couchbaseTemplate.keySettings(KeySettings.build().prefix("ApplicationA").suffix("S
erver1").delimiter("::"));
```

Key will be auto-generated only for operations with direct entity input like insert, update, save, delete using entity. For other operations requiring just the key, it can be generated using `CouchbaseTemplate`.

Example 22. Standalone key generation in CouchbaseTemplate

```
@Autowired
CouchbaseTemplate couchbaseTemplate;
...
String id = couchbaseTemplate.getGeneratedId(entity);
...
repo.exists(id);
```

3.2. Key generation using attributes

It is a common practice to generate keys using a combination of the document attributes. Key generation using attributes concatenates all the attribute values annotated with `IdAttribute`, based on the ordering provided similar to prefixes and suffixes.

Example 23. Annotation for IdAttribute

```
@Document
public class User {
     @Id @GeneratedValue(strategy = USE_ATTRIBUTES)
     private String id;
     @IdAttribute
     private String userid;
     ...
}
```

3.3. Key generation using uuid

This auto-generation uses UUID random generator to generate document keys consuming 16 bytes of key space. This mechanism is only recommended for test scaffolding.

Example 24. Annotation for Unique key generation

```
@Document
public class User {
     @Id @GeneratedValue(strategy = UNIQUE)
     private String id;
     ...
}
```

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 25. CrudRepository interface

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

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

  Optional<T> findById(ID primaryKey); ②

  Iterable<T> findAll();               ③

  long count();                        ④

  void delete(T entity);               ⑤

  boolean existsById(ID primaryKey);   ⑥

  // … more functionality omitted.
}
```

① Saves the given entity.

② Returns the entity identified by the given id.

③ Returns all entities.

④ Returns the number of entities.

⑤ Deletes the given entity.

⑥ Indicates whether an entity with the given id exists.

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

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

Example 26. 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 27. Derived Count Query

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

  long countByLastname(String lastname);
}
```

Example 28. Derived Delete Query

```
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<Person, Long> { … }
   ```

2. Declare query methods on the interface.

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

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

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

@EnableJpaRepositories
class Config {}
```

or via XML configuration:

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

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

</beans>
```

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

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

4. Get the repository instance injected and use it.

```
class SomeClient {

  private final PersonRepository repository;

  SomeClient(PersonRepository repository) {
    this.repository = repository;
  }

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

The sections that follow explain each step in detail.

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 29. Selectively exposing CRUD methods

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

  Optional<T> findById(ID id);

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

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

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

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

4.3.2. Null handling of repository methods

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

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

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

Alternatively query methods can choose not to use a wrapper type at all. The absence of a query result will then be indicated by returning `null`. Repository methods returning collections, collection alternatives, wrappers, and streams are guaranteed never to return `null` but rather the corresponding empty representation. See Repository query return types for details.

Nullability annotations

You can express nullability constraints for repository methods using Spring Framework’s nullability annotations. They provide a tooling-friendly approach and opt-in `null` checks during runtime:

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

Spring annotations are meta-annotated with JSR 305 annotations (a dormant but widely spread JSR). JSR 305 meta-annotations allow tooling vendors like IDEA, Eclipse, or Kotlin to provide null-safety support in a generic way, without having to hard-code support for Spring annotations. To enable runtime checking of nullability constraints for query methods, you need to activate non-nullability on package level using Spring’s `@NonNullApi` in `package-info.java`:

Example 30. Declaring non-nullability in `package-info.java`

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

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

Example 31. Using different nullability constraints

```
package com.acme;                                                  ①

import org.springframework.lang.Nullable;

interface UserRepository extends Repository<User, Long> {

  User getByEmailAddress(EmailAddress emailAddress);                 ②

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

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

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

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

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

④ Will return `Optional.empty()` in case the query executed does not produce a result. Will throw an `IllegalArgumentException` in case the `emailAddress` handed to the method is `null`.

Nullability in Kotlin-based repositories

Kotlin has the definition of nullability constraints baked into the language. Kotlin code compiles to bytecode which does not express nullability constraints using method signatures but rather compiled-in metadata. Make sure to include the `kotlin-reflect` JAR in your project to enable introspection of Kotlin's nullability constraints. Spring Data repositories use the language mechanism to define those constraints to apply the same runtime checks:

Example 32. Using nullability constraints on Kotlin repositories

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

  fun findByUsername(username: String): User     ①

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

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

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

4.3.3. Using Repositories with multiple Spring Data modules

Using a unique Spring Data module in your application makes things simple hence, all repository interfaces in the defined scope are bound to the Spring Data module. Sometimes applications require using more than one Spring Data module. In such case, it’s required for a repository definition to distinguish between persistence technologies. Spring Data enters strict repository configuration mode because it detects multiple repository factories on the class path. Strict configuration requires details on the repository or the domain class to decide about Spring Data module binding for a repository definition:

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

Example 33. Repository definitions using Module-specific Interfaces

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

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

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

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

Example 34. Repository definitions using generic Interfaces

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

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

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

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

Example 35. Repository definitions using Domain Classes with Annotations

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

@Entity
class Person {
  …
}

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

@Document
class User {
  …
}
```

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

Example 36. Repository definitions using Domain Classes with mixed Annotations

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

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

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

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

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

The last way to distinguish repositories is scoping repository base packages. Base packages define the starting points for scanning for repository interface definitions which implies to have repository definitions located in the appropriate packages. By default, annotation-driven configuration uses the package of the configuration class. The base package in XML-based configuration is mandatory.

Example 37. Annotation-driven configuration of base packages

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

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 a 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 38. Query creation from method names

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

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

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

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

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

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

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

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

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);
```

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

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 39. Using Pageable, Slice and Sort in query methods

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

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

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

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

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

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

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

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 40. 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 41. 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 42. 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.

4.4.7. Async query results

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

```
@Async
Future<User> findByFirstname(String firstname);               ①

@Async
CompletableFuture<User> findOneByFirstname(String firstname); ②

@Async
ListenableFuture<User> findOneByLastname(String lastname);    ③
```

① Use `java.util.concurrent.Future` as return type.

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

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

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 43. 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 44. 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. [1: JavaConfig in the Spring reference documentation]

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

Example 45. Sample annotation based repository configuration

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

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

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

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 46. Standalone usage of repository factory

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

4.6. Custom implementations for Spring Data repositories

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

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

4.6.1. Customizing individual repositories

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

Example 47. Interface for custom repository functionality

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

Example 48. Implementation of custom repository functionality

```
class CustomizedUserRepositoryImpl implements CustomizedUserRepository {

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

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

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

Example 49. Changes to your repository interface

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

  // Declare query methods here
}
```

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

Spring Data repositories are implemented by using fragments that form a repository composition. Fragments are the base repository, functional aspects such as QueryDsl and custom interfaces along with their implementation. Each time you add an interface to your repository interface, you enhance the composition by adding a fragment. The base repository and repository aspect implementations are provided by each Spring Data module.

Example 50. Fragments with their implementations

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

class HumanRepositoryImpl implements HumanRepository {

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

interface EmployeeRepository {

  void someEmployeeMethod(User user);

  User anotherEmployeeMethod(User user);
}

class ContactRepositoryImpl implements ContactRepository {

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

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

Example 51. Changes to your repository interface

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

  // Declare query methods here
}
```

Repositories may be composed of multiple custom implementations that are imported in the order of their declaration. Custom implementations have a higher priority than the base implementation and repository aspects. This ordering allows you to override base repository and aspect methods and resolves ambiguity if two fragments contribute the same method signature. Repository fragments are not limited to be used in a single repository interface. Multiple repositories may use a fragment interface to reuse customizations across different repositories.

Example 52. Fragments overriding `save(…)`

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

class CustomizedSaveImpl<T> implements CustomizedSave<T> {

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

Example 53. Customized repository interfaces

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

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

Configuration

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

Example 54. Configuration example

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

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

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

Resolution of ambiguity

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

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

Example 55. Resolution of amibiguous implementations

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

class CustomizedUserRepositoryImpl implements CustomizedUserRepository {

  // Your custom implementation
}
```

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

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

  // Your custom implementation
}
```

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

Manual wiring

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

Example 56. Manual wiring of custom implementations

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

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

4.6.2. Customize the base repository

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

Example 57. Custom repository base class

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

  private final EntityManager entityManager;

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

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

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

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

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

Example 58. Configuring a custom repository base class using JavaConfig

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

A corresponding attribute is available in the XML namespace.

Example 59. Configuring a custom repository base class using XML

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

4.7. Publishing events from aggregate roots

Entities managed by repositories are aggregate roots. In a Domain-Driven Design application, these aggregate roots usually publish domain events. Spring Data provides an annotation `@DomainEvents` you can use on a method of your aggregate root to make that publication as easy as possible.

Example 60. Exposing domain events from an aggregate root

```
class AnAggregateRoot {

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

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

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

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

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

4.8. Spring Data extensions

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

4.8.1. Querydsl Extension

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

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

Example 61. QueryDslPredicateExecutor interface

```
public interface QueryDslPredicateExecutor<T> {

  Optional<T> findById(Predicate predicate);  ①

  Iterable<T> findAll(Predicate predicate);   ②

  long count(Predicate predicate);            ③

  boolean exists(Predicate predicate);        ④

  // … more functionality omitted.
}
```

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

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

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

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

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

Example 62. Querydsl integration on repositories

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

}
```

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

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

userRepository.findAll(predicate);
```

4.8.2. Web support

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

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

Example 63. 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 64. 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 65. A Spring MVC controller using domain types in method signatures

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

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

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

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

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

HandlerMethodArgumentResolvers for Pageable and Sort

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

Example 66. Using Pageable as controller method argument

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

  private final UserRepository repository;

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

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

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

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

Table 2. Request parameters evaluated for Pageable instances

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

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

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

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

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

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

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

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

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

Hypermedia support for Pageables

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

Example 67. Using a PagedResourcesAssembler as controller method argument

```
@Controller
class PersonController {

  @Autowired PersonRepository repository;

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

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

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

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

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

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

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

Querydsl web support

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

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

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

can be resolved to

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

using the `QuerydslPredicateArgumentResolver`.

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

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

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

```
@Controller
class UserController {

  @Autowired UserRepository repository;

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

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

    return "index";
  }
}
```

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

The default binding is as follows:

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

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

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

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

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

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

② `QuerydslBinderCustomizer` defined on the repository interface will be automatically picked up and shortcuts `@QuerydslPredicate(bindings=…)`.

③ Define the binding for the `username` property to be a simple contains binding.

④ Define the default binding for `String` properties to be a case insensitive contains match.

⑤ Exclude the *password* property from `Predicate` resolution.

4.8.3. Repository populators

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

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

Example 68. 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 69. Declaring a Jackson repository populator

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

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

</beans>
```

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

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

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

Example 70. 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>
```

4.8.4. Legacy web support

Domain class web binding for Spring MVC

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

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

  private final UserRepository userRepository;

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

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

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

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

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

PropertyEditors

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

```xml
<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.

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

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

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

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.

There are three backing mechanisms in Couchbase for repositories, described in the sections of this chapter:

- N1QL based querying
- View based querying
- Spatial View based querying

CRUD operations are still mostly backed by Couchbase views (see Backing Views). Such views (and, for N1QL, equivalent indexes) can be automatically built, but note this is **discouraged in production** and can be an **expensive operation** (see Automatic Index Management).

Note that you can tune the consistency you want for your queries (see Querying with consistency) and have different repositories backed by different buckets (see Working with multiple buckets)

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 71. Annotation-Based Repository Setup

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

An advanced usage is described in Working with multiple buckets.

XML-based configuration is also available:

Example 72. 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 UserInfo:

Example 73. A UserInfo repository

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

public interface UserRepository extends CrudRepository<UserInfo, 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 `@Autowire` the `UserRepository` to a class that makes use of it. What methods do we have available?

Table 3. Exposed methods on the UserRepository

Method	Description
UserInfo save(UserInfo entity)	Save the given entity.
Iterable<UserInfo> save(Iterable<UserInfo> entity)	Save the list of entities.
UserInfo 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<UserInfo> findAll() (*)	Find all entities by this type in the bucket.
Iterable<UserInfo> 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(UserInfo entity)	Delete the entity.
void delete(Iterable<UserInfo> entities)	Delete all given entities.
void deleteAll() (*)	Delete all entities by type in the bucket.

Now that's 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.

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 requests in the background, as we'll see in the next sections.

5.3. Repositories and Querying

5.3.1. N1QL based querying

As of version `4.0`, Couchbase Server ships with a new query language called `N1QL`. In `Spring-Data-Couchbase 2.0`, N1QL is the default way of doing queries and will allow you to fully derive queries from a method name.

Prerequisite is to have a N1QL-compatible cluster and to have created a PRIMARY INDEX on the bucket where the entities will be stored. DML queries are supported from Couchbase server version `4.1`.

WARNING	If it is detected at configuration time that the cluster doesn't support N1QL while there are `@Query` annotated methods or non-annotated methods in your repository interface, a `UnsupportedCouchbaseFeatureException` will be thrown.

Here is an example:

Example 74. An extended UserInfo repository with N1QL queries

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

    @Query("#{#n1ql.selectEntity} WHERE role = 'admin' AND #{#n1ql.filter}")
    List<UserInfo> findAllAdmins();

    List<UserInfo> findByFirstname(String fname);
}
```

Here we see two N1QL-backed ways of querying.

The first method uses the `Query` annotation to provide a N1QL statement inline. SpEL (Spring Expression Language) is supported by surrounding SpEL expression blocks between `#{` and `}`. A few N1QL-specific values are provided through SpEL:

- `#n1ql.selectEntity` allows to easily make sure the statement will select all the fields necessary to build the full entity (including document ID and CAS value).
- `#n1ql.filter` in the WHERE clause adds a criteria matching the entity type with the field that Spring Data uses to store type information.
- `#n1ql.bucket` will be replaced by the name of the bucket the entity is stored in, escaped in backticks.
- `#n1ql.fields` will be replaced by the list of fields (eg. for a SELECT clause) necessary to reconstruct the entity.
- `#n1ql.delete` will be replaced by the `delete from` statement.
- `#n1ql.returning` will be replaced by returning clause needed for reconstructing entity.

IMPORTANT We recommend that you always use the `selectEntity` SpEL and a WHERE clause with a `filter` SpEL (since otherwise your query could be impacted by entities from other repositories).

String-based queries support parametrized queries. You can either use positional placeholders like “$1”, in which case each of the method parameters will map, in order, to `$1`, `$2`, `$3`… Alternatively, you can use named placeholders using the “$someString” syntax. Method parameters will be matched with their corresponding placeholder using the parameter’s name, which can be overridden by annotating each parameter (except a `Pageable` or `Sort`) with `@Param` (eg. `@Param("someString")`). You cannot mix the two approaches in your query and will get an `IllegalArgumentException` if you do.

Note that you can mix N1QL placeholders and SpEL. N1QL placeholders will still consider all method parameters, so be sure to use the correct index like in the example below:

Example 75. An inline query that mixes SpEL and N1QL placeholders

```
@Query("#{#n1ql.selectEntity} WHERE #{#n1ql.filter} AND #{[0]} = $2")
public List<User> findUsersByDynamicCriteria(String criteriaField, Object
criteriaValue)
```

This allows you to generate queries that would work similarly to eg. `AND name = "someName"` or `AND age = 3`, with a single method declaration.

You can also do single projections in your N1QL queries (provided it selects only one field and returns only one result, usually an aggregation like `COUNT`, `AVG`, `MAX`…). Such projection would have a simple return type like `long`, `boolean` or `String`. This is **NOT** intended for projections to DTOs.

Another example:
`#{#n1ql.selectEntity} WHERE #{#n1ql.filter} AND test = $1`
is equivalent to
`SELECT #{#n1ql.fields} FROM #{#n1ql.bucket} WHERE #{#n1ql.filter} AND test = $1`

A practical application of SpEL with Spring Security

SpEL can be useful when you want to do a query depending on data injected by other Spring components, like Spring Security. Here is what you need to do to extend the SpEL context to get access to such external data.

First, you need to implement an `EvaluationContextExtension` (use the support class as below):

```
class SecurityEvaluationContextExtension extends
EvaluationContextExtensionSupport {

  @Override
  public String getExtensionId() {
    return "security";
  }

  @Override
  public SecurityExpressionRoot getRootObject() {
    Authentication authentication = SecurityContextHolder.getContext()
.getAuthentication();
    return new SecurityExpressionRoot(authentication) {};
  }
}
```

Then all you need to do for Spring Data Couchbase to be able to access associated SpEL values is to declare a corresponding bean in your configuration:

```
@Bean
EvaluationContextExtension securityExtension() {
    return new SecurityEvaluationContextExtension();
}
```

This could be useful to craft a query according to the role of the connected user for instance:

```
@Query("#{#n1ql.selectEntity} WHERE #{#n1ql.filter} AND " +
"role = '?#{hasRole('ROLE_ADMIN') ? 'public_admin' : 'admin'}'")
List<UserInfo> findAllAdmins(); //only ROLE_ADMIN users will see hidden admins
```

Delete query example:

```
@Query("#{#n1ql.delete} WHERE #{#n1ql.filter} AND " +
"username = $1 #{#n1ql.returning}")
UserInfo removeUser(String username);
```

The second method uses Spring-Data's query derivation mechanism to build a N1QL query from the method name and parameters. This will produce a query looking like this: `SELECT … FROM … WHERE firstName = "valueOfFnameAtRuntime"`. You can combine these criteria, even do a count with a name like `countByFirstname` or a limit with a name like `findFirst3ByLastname`…

NOTE	Actually the generated N1QL query will also contain an additional N1QL criteria in order to only select documents that match the repository's entity class.

Most Spring-Data keywords are supported: .Supported keywords inside @Query (N1QL) method names

Keyword	Sample	N1QL WHERE clause snippet
`And`	`findByLastnameAndFirstname`	`lastName = a AND firstName = b`
`Or`	`findByLastnameOrFirstname`	`lastName = a OR firstName = b`
`Is,Equals`	`findByField`,`findByFieldEquals`	`field = a`
`IsNot,Not`	`findByFieldIsNot`	`field != a`
`Between`	`findByFieldBetween`	`field BETWEEN a AND b`
`IsLessThan,LessThan,IsBefore,Before`	`findByFieldIsLessThan`,`findByFieldBefore`	`field < a`
`IsLessThanEqual,LessThanEqual`	`findByFieldIsLessThanEqual`	`field ⇐ a`
`IsGreaterThan,GreaterThan,IsAfter,After`	`findByFieldIsGreaterThan`,`findByFieldAfter`	`field > a`
`IsGreaterThanEqual,GreaterThanEqual`	`findByFieldGreaterThanEqual`	`field >= a`
`IsNull`	`findByFieldIsNull`	`field IS NULL`
`IsNotNull,NotNull`	`findByFieldIsNotNull`	`field IS NOT NULL`
`IsLike,Like`	`findByFieldLike`	`field LIKE "a"` - a should be a String containing % and _ (matching n and 1 characters)
`IsNotLike,NotLike`	`findByFieldNotLike`	`field NOT LIKE "a"` - a should be a String containing % and _ (matching n and 1 characters)
`IsStartingWith,StartingWith,StartsWith`	`findByFieldStartingWith`	`field LIKE "a%"` - a should be a String prefix
`IsEndingWith,EndingWith,EndsWith`	`findByFieldEndingWith`	`field LIKE "%a"` - a should be a String suffix
`IsContaining,Containing,Contains`	`findByFieldContains`	`field LIKE "%a%"` - a should be a String
`IsNotContaining,NotContaining,NotContains`	`findByFieldNotContaining`	`field NOT LIKE "%a%"` - a should be a String
`IsIn,In`	`findByFieldIn`	`field IN array` - note that the next parameter value (or its children if a collection/array) should be compatible for storage in a `JsonArray`)
`IsNotIn,NotIn`	`findByFieldNotIn`	`field NOT IN array` - note that the next parameter value (or its children if a collection/array) should be compatible for storage in a `JsonArray`)

Keyword	Sample	N1QL WHERE clause snippet
`IsTrue,True`	`findByFieldIsTrue`	`field = TRUE`
`IsFalse,False`	`findByFieldFalse`	`field = FALSE`
`MatchesRegex,Matches,Regex`	`findByFieldMatches`	`REGEXP_LIKE(field, "a")` - note that the ignoreCase is ignored here, a is a regular expression in String form
`Exists`	`findByFieldExists`	`field IS NOT MISSING` - used to verify that the JSON contains this attribute
`OrderBy`	`findByFieldOrderByLastnameDesc`	`field = a ORDER BY lastname DESC`
`IgnoreCase`	`findByFieldIgnoreCase`	`LOWER(field) = LOWER("a")` - a must be a String

You can use both counting queries and Limiting query results features with this approach.

With N1QL, another possible interface for the repository is the `PagingAndSortingRepository` one (which extends CRUDRepository). It adds two methods:

Table 4. Exposed methods on the PagingAndSortingRepository

Method	Description
Iterable<T> findAll(Sort sort);	Allows to retrieve all relevant entities while sorting on one of their attributes.
Page<T> findAll(Pageable pageable);	Allows to retrieve your entities in pages. The returned `Page` allows to easily get the next page's `Pageable` as well as the list of items. For the first call, use `new PageRequest(0, pageSize)` as Pageable.

TIP You can also use `Page` and `Slice` as method return types as well with a N1QL backed repository.

NOTE If pageable and sort parameters are used with inline queries, there should not be any order by, limit or offset clause in the inline query itself otherwise the server would reject the query as malformed.

The second way of querying, supported also in older versions of Couchbase Server, is the View-backed one that we'll see in the next section.

5.3.2. Backing Views

This is the historical way of secondary indexing in Couchbase. Views are much more limited in terms of querying flexibility, and each custom method may very well need its own backing view, to be prepared in the cluster beforehand.

We'll only cover views to the extent to 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.

As a rule of thumb, all repository CRUD access methods which are not "by a specific key" still require a single backing view, by default `all`, to find the one or more matching entities.

> **IMPORTANT**
> This is only true for the methods directly defined by the `CrudRepository` interface (the one marked with a `*` in `Table 3.` above), since your additional methods can now be backed by N1QL.

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 uncapitalized (lowercase first letter) entity name as the design document name. So if your entity is named `UserInfo`, then the code expects the `all` view in the `userInfo` design document. It needs to look like this:

Example 76. The all view map function

```
// do not forget the _count reduce function!
function (doc, meta) {
  if (doc._class == "namespace.to.entity.UserInfo") {
    emit(meta.id, 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(meta.id, null)` in the view despite the document id being always sent over to the client implicitly, it is so the view can be queried with a list of ids, eg. in the `findAll(Iterable<ID> ids)` CRUD method.

5.3.3. Automatic Index Management

We've seen that the repositories default methods can be backed by two broad kind of features: views and N1QL (in the case of paging and sorting). In order for the CRUD operations to work, the adequate view must have been created beforehand, and this is usually left for the user to do. First because view creation (and index creation) is an expensive operation that can take quite some time if the quantity of documents is high. Second, because in production it is considered best practice to avoid administration of the cluster elements like buckets, indexes and view by an application code.

In the case where the index creation cost isn't considered too high and you are not in a production environment, it can be triggered automatically instead, in two steps. You will first need to annotate the repositories you want managed with the relevant annotation(s):

- `@ViewIndexed` will create a view like the "all" view previously seen, to list all entities in the bucket.
- `@N1qlPrimaryIndexed` can be used to ensure a general-purpose PRIMARY INDEX is available in N1QL.
- `@N1qlSecondaryIndexed` will create a more specific N1QL index that does the same kind of filtering on entity type that the view does. It'll allow for efficient listing of all documents that correspond to a Repository's associated domain object.

Secondly, you'll need to opt-in to this feature by customizing the `indexManager()` bean of your env-specific `AbstractCouchbaseConfiguration` to take certain types of annotations into account. This is done through the `IndexManager(boolean processViews, boolean processN1qlPrimary, boolean processN1qlSecondary)` constructor. Set the flags for the category of annotations you want processed to true, or false to deactivate the automatic creation feature.

The `@Profile` annotation is one possible Spring annotation to be used to differentiate configurations (or individual beans) per environment.

Example 77. A Dev configuration where only @ViewIndexed annotations will be processed.

```
@Configuration
public class ExampleDevApplicationConfig extends AbstractCouchbaseConfiguration {

  // note a few other overrides are actually needed

  //this is for dev so it is ok to auto-create indexes
  @Override
  public IndexManager indexManager() {
    return new IndexManager(true, false, false);
  }
}
```

5.3.4. View based querying

In `2.0`, since N1QL has been introduced as a more powerful concept, view-backed queries have changed a bit outside of the CRUD methods:

- the `@View` annotation is mandatory.
- if you just want all the results from the view, you can let the framework guess the view name to use by just using the plain annotation `@View`. **You won't be able to customize** the `ViewQuery` (eg. adding limits and specifying a `startkey`) using this method anymore.
- if you want your view query to have restrictions, those can be derived from the method name but in this case you **must** explicitly provide the `viewName` attribute in the annotation.
- View based query derivation is limited to a few keywords and only works on simple keys (not compound keys like `[ age, fname ]`).
- View based query derivation still needs you to include **one** valid property before keywords in

the method name.

Example 78. An extended UserInfo repository with View queries

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

    @View
    List<UserInfo> findAllAdmins();

    @View(viewName="firstNames")
    List<UserInfo> findByFirstnameStartingWith(String fnamePrefix);
}
```

Implementing your custom repository finder methods also needs backing views. The `findAllAdmins` guesses to use the `allAdmins` view in the `userInfo` design document, by convention. 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, unless you plan to call one by prefixing your method with `count` instead of `find`!):

Example 79. The allAdmins map function

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

By now, we've never actually customized our view at query time. This is where the alternative, query derivation, comes along - like in our `findByFirstnameStartingWith(String fnamePrefix)` method.

Example 80. The firstNames view map function

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

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

Example 81. Query a repository method with custom params.

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

// Find all users with first name starting with "Mich"
List<UserInfo> users = repo.findByFirstnameStartingWith("Mich");
```

On all these derived custom finder methods, you have to use the `@View` annotation with at least the view name specified (and you can also override the design document name, otherwise determined by convention).

IMPORTANT	For any other usage and customization of the `ViewQuery` that goes beyond that, recommended approach is to provide an implementation that uses the underlying template, like described in Changing repository behaviour. For more details on behavior, please consult the Couchbase Server and Java SDK documentation directly.

For view-based query derivation, here are the supported keywords (A and B are method parameters in this table):

Table 5. Supported keywords inside @View method names

`Is`,`Equals`	`findAllByUsername`,`findByFieldEquals`	`key=A` **- if only keyword, the method can have no parameter (return all items from the view)**
`Between`	`findByFieldBetween`	`startkey=A&endkey=B`
`IsLessThan`,`LessThan`,`IsBefore`,`Before`	`findByFieldIsLessThan`,`findByFieldBefore`	`endkey=A`
`IsLessThanEqual`,`LessThanEqual`	`findByFieldIsLessThanEqual`	`endkey=A&inclusive_end=true`
`IsGreaterThanEqual`,`GreaterThanEqual`	`findByFieldGreaterThanEqual`	`startkey=A`
`IsStartingWith`,`StartingWith`,`StartsWith`	`findByFieldStartingWith`	`startkey="A"&endkey="A\uefff"` - A should be a String prefix
`IsIn`,`In`	`findByFieldIn`	`keys=[A]` - A should be a `Collection`/`Array` with elements compatible for storage in a `JsonArray` (or a single element to be stored in a `JsonArray`)

Note that both `reduce functions` and Limiting query results are also supported.

TIP	In order to trigger a `reduce`, you can use the `count` prefix instead of `find`. But sometimes is doesn't make much sense (eg. because you actually use the `_stats` built in function, which returns a JSON object). So alternatively you can also explicitly ask for reduce to be executed by setting `reduce = true` in the `@View` annotation. Be sure to specify a return type that make sense for the reduce function of your view.

WARNING	Compound keys are not supported, and neither are Or composition, Ignore Case and Order By. You have to include a valid entity property in the naming of your method.

Last method of querying in Couchbase (from Couchbase Server 4.0, like for N1QL) is querying for dimensional data through **Spatial Views**, as we'll see in the next section.

5.3.5. Spatial View based querying

Couchbase can accommodate multi-dimensional data and query it with the use of special views, the Spatial Views. Such views allows to perform multi-dimensional queries, not only limited to geographical data.

Integration of these views in `Spring Data Couchbase` repositories is done through the `@Dimensional` annotation. Like `@View`, the annotation allows to indicate usage of a Spatial View as the backing mechanism for the annotated method. The annotation requires you to give the name of the `designDocument` and the `spatialViewName` to use. Additionally, you should specify the number of `dimensions` the view works with (unless it is the default classical 2).

Multi-dimensionality concept is interesting, it means you can craft views that allows you to answer questions like "find all shops that are within Manhattan and open between 14:00 and 23:00" (the third dimension of the view being the opening hours).

Couchbase's Spatial View support querying through ranges that represent "lowest" and "highest" values in each dimension, so for 2D it represents a bounding box, with the southwest-most point [x,y] as `startRange` and northeast-most point [x,y] as `endRange`.

TIP	Even though Couchbase Spatial View engine only support Bounding Box querying, the Spring Data Couchbase framework will attempt to remove false positives for you when querying with a `Polygon` or a `Circle` (in TRACE log level each false positive elimination will be logged). Note that a point on the edge of a `Polygon` is **not** considered within (whereas it is when dealing with a `Circle`).

The following query derivation keywords and parameters relative to geographical data in Spring Data are supported for Spatial Views:

Table 6. Supported keywords inside @Dimensional method names

Keyword	Sample	Remarks
`Within,IsWithin`	`findByLocationWithin`	
`Near,IsNear`	`findByLocationNear`	expects a `Point` and a `Distance`, will approximate to bounding box
`Between`	`findByLocationWithinAndOpeningHoursBetween`	useful for dimensions beyond 2, adds two numerical values to the startRange and endRange respectively
`GreaterThan,GreaterThanEqual,After`	`findByLocationWithinAndOpeningHoursAfter`	useful for dimensions beyond 2, adds a numerical value to the startRange
`LessThan,LessThanEqual,Before`	`findByLocationWithinAndOpeningHoursBefore`	useful for dimensions beyond 2, adds a numerical value to the endRange

IMPORTANT	For "within" types of queries, the expected parameters map to geographical 2D data. Classes from the `org.springframework.data.geo` package are usually expected, but Polygon and Boxes can also be expressed as arrays of `Point`s.

Further dimensions are supported through keywords other than Within and Near and require numerical input.

5.3.6. Querying with consistency

One aspect that is often needed and doesn't have a direct equivalent in the Spring Data query derivation mechanism is `query consistency`. In both view-based queries and N1QL, you have this concept that the secondary index can return stale data, because the latest version hasn't been indexed yet. This gives the best performance at the expense of consistency.

Note that weaker consistencies can lead to data being returned that doesn't match the criteria of a derived query. One trickier case is when documents are deleted from Couchbase but views have not yet caught up to the deletion. With weak consistency this can mean that a view would return IDs that are not in the database anymore, leading to null entities. The `CouchbaseTemplate`s `findByView` and `findBySpatialView` methods will remove such stale deleted entities from their result in order to avoid having nulls in the returned collections. Similarly, `CouchbaseRepository's `deleteAll` method will ignore documents that the backing view provided but the SDK remove operation couldn't find.

If one wants to have stronger consistency, there are two possibilities described in the next sections.

Configure it on a global level

A global consistency can be defined using the `Consistency` enumeration (eg. `Consistency.READ_YOUR_OWN_WRITE`):

- in xml, this is done via the `consistency` attribute on `<couchbase:template>`.
- in javaConfig, this is done by overriding the `getDefaultConsistency()` method.

By default it is `Consistency.READ_YOUR_OWN_WRITES` (which means consistency is prioritized over speed, especially when a large number of documents has been created recently).

IMPORTANT	This is **only used in repositories**, either for index-backed methods automatically provided by the repository interface (`findAll()`, `findAll(keys)`, `count()`, `deleteAll()`…) or methods you define in your specific interface using query derivation.

Provide an implementation

Provide the implementation and directly use `queryView` and `queryN1QL` methods on the template with a specific consistency (see Changing repository behaviour).

- one can specify the consistency on those via their respective query classes, according to the Couchbase Java SDK documentation.
- for example for views `ViewQuery.stale(Stale.FALSE)`

* for example for N1QL `Query.simple("SELECT * FROM default", QueryParams.build().consistency(ScanConsistency.REQUEST_PLUS));`

5.4. Working with multiple buckets

The Java Config version allows you to define multiple `Bucket` and `CouchbaseTemplate`, but in order to have different repositories use different underlying buckets/templates, you need to follow these steps:

* in your `AbstractCouchbaseConfiguration` implementation, override the `configureRepositoryOperationsMapping` method.
* mutate the provided `RepositoryOperationsMapping` as needed (it defaults to mapping everything to the default template).
* configure the mapping by chaining calls to `map`, `mapEntity` and `setDefault`.
  * `map` maps a specific repository interface to the `CouchbaseOperations` it should use
  * `mapEntity` maps all unmapped repositories of a domain type / entity class to a common `CouchbaseOperations`
  * `setDefault` maps all remaining unmapped repositories to a default `CouchaseOperations` (the default, using `couchbaseTemplate` bean unless modified).

The idea is that the framework will look for an entry corresponding to the repository's interface when instantiating it. If none is found it will look at the mapping for the repository's domain type. Eventually it will fallback to the default setting. Here is an example:

Example 82. Example of configuring multiple templates and repositories.

```
@Configuration
@EnableCouchbaseRepositories
public class ConcreteCouchbaseConfig extends AbstractCouchbaseConfig {

  //the default bucket and template must be created, implement abstract methods
here to that end

  //we want all User objects to be stored in a second bucket
  //let's define the bucket reference...
  @Bean
  public Bucket userBucket() {
    return couchbaseCluster().openBucket("users", "");
  }

  //... then the template (inspired by couchbaseTemplate() method)...
  @Bean
  public CouchbaseTemplate userTemplate() {
    CouchbaseTemplate template = new CouchbaseTemplate(
        couchbaseClusterInfo(), //reuse the default bean
      userBucket(), //the bucket is non-default
        mappingCouchbaseConverter(), translationService() //default beans here as
well
    );
    template.setDefaultConsistency(getDefaultConsistency());
    return template;
  }

  //... then finally make sure all repositories of Users will use it
  @Override
  public void configureRepositoryOperationsMapping(RepositoryOperationsMapping
baseMapping) {
    baseMapping //this is already using couchbaseTemplate as default
      .mapEntity(User.class, userTemplate()); //every repository dealing with User
will be backed by userTemplate()
  }
}
```

5.5. Changing repository behaviour

Sometimes you don’t simply want the repository to create methods for you, but instead you want to tune the base repository’s behaviour. You can either do that for **all** repositories - by changing the *base class* for them - or just for a single repository - by adding custom implementations for either new or existing methods - (see Custom implementations for Spring Data repositories for a generic introduction to these concepts).

5.5.1. Couchbase specifics about changing the base class

This follows the standard procedure for changing all repositories' base class:

1. Create an generic interface for your base that extends `CouchbaseRepository` (CRUD) or `CouchbasePagingAndSortingRepository`. Declare any method you want to add to all repositories there.
2. Create an implementation (eg. `MyRepositoryImpl`). This should extend one the concrete base classes (`SimpleCouchbaseRepository` or `N1qlCouchbaseRepository`) and you can also override existing methods from the Spring Data interfaces.
3. Declare your repository interfaces as extending `MyRepository` instead of eg. `CRUDRepository` or `CouchbaseRepository`.
4. In the `@EnableCouchbaseRepositories` annotation of your configuration, use the `repositoryBaseClass` parameter.

Here is a complete example that you can find in `RepositoryBaseTest` in the integration tests:

Changing repository base class

```
@NoRepositoryBean ①
public interface MyRepository<T, ID extends Serializable> extends CouchbaseRepository
<T, ID> { ②

  int sharedCustomMethod(ID id); ③
}

public class MyRepositoryImpl<T, ID extends Serializable>
    extends N1qlCouchbaseRepository<T, ID> ④
    implements MyRepository<T, ID> { ⑤

  public MyRepositoryImpl(CouchbaseEntityInformation<T, String> metadata,
CouchbaseOperations couchbaseOperations) { ⑥
    super(metadata, couchbaseOperations);
  }

  @Override
  public int sharedCustomMethod(ID id) {
    //... implement common behavior ⑦
  }
}

@EnableCouchbaseRepositories(repositoryBaseClass = MyRepositoryImpl.class) ⑧
public class MyConfig extends AbstractCouchbaseConfiguration { /** ... */ }
```

① This annotation prevents picking this custom interface as a repository declaration.

② The new base interface extends one from Spring Data Couchbase.

③ This method will be available in all repositories.

④ Custom base implementation relies on the existing bases...

⑤ ...and also implements new interface (so that common methods are exposed).

⑥ Constructors that follow the signature of superconstructor will be picked up by the framework.

⑦ Custom functionality to be implemented by the user (eg. return string's length).

⑧ Weaving it all in by changing the repository base class.

5.5.2. Couchbase specifics about adding methods to a single repository

Again following the standard procedure for custom repository methods, here is a complete example that you can find in `RepositoryCustomMethodTest` in the integration tests:

Adding and overriding methods in a single repository

```
public interface MyRepositoryCustom {
  long customCountItems(); ①
}

public interface MyRepository extends CrudRepository<MyItem, String>,
MyRepositoryCustom { } ②

public class MyRepositoryImpl implements MyRepositoryCustom { ③

  @Autowired
  RepositoryOperationsMapping templateProvider; ④

  @Override
  public long customCountItems() {
    CouchbaseOperations template = templateProvider.resolve(MyRepository.class, Item
.class); ⑤

    CouchbasePersistentEntity<Object> itemPersistenceEntity =
(CouchbasePersistentEntity<Object>)
        template.getConverter()
            .getMappingContext()
            .getPersistentEntity(MyItem.class);

    CouchbaseEntityInformation<? extends Object, String> itemEntityInformation =
        new MappingCouchbaseEntityInformation<Object, String>(itemPersistenceEntity);

    Statement countStatement = N1qlUtils.createCountQueryForEntity( ⑥
        template.getCouchbaseBucket().name(),
        template.getConverter(),
        itemEntityInformation);

    ScanConsistency consistency = template.getDefaultConsistency().n1qlConsistency();
⑦
    N1qlParams queryParams = N1qlParams.build().consistency(consistency);
    N1qlQuery query = N1qlQuery.simple(countStatement, queryParams);

    List<CountFragment> countFragments = template.findByN1QLProjection(query,
```

```
CountFragment.class); ⑧

    if (countFragments == null || countFragments.isEmpty()) {
      return 0L;
    } else {
      return countFragments.get(0).count * -1L; ⑨
    }
  }

  public long count() { ⑩
    return 100;
  }
}
```

① This method is to be added with a user-provided implementation for a single repository.

② This is the declaration of the customized repository, both a CRUD and exposing the custom interface.

③ This is the implementation of the custom interface.

④ The custom implementation doesn't have access to the original base implementation, so use dependency injection to get access to necessary resources.

⑤ Here is a couchbase specificity: if you need to use the `CouchbaseTemplate`, be sure to use the one that would be associated with the customized repository or associated entity type.

⑥ We use `N1QLUtils` to prepare a complete `N1QL` statement for counting. It relies on the information above that we got from the correct template.

⑦ We want to make sure that the default consistency configured in the associated template is used for this query.

⑧ Using `CouchbaseTemplate.findByN1qlProjection`, we execute the count query and store the single aggregation result into a `CountFragment`.

⑨ Now we return this count result with a twist: it is negated.

⑩ **TIP**: You can actually also change implementation of methods from the `CRUDRepository` interface!

By storing 3 items using a `MyRepository` instance and calling `count()` then `customCountItems()`, we'd obtain

```
100
-3
```

5.5.3. DTO Projections

Spring Data Repositories usually return the domain model when using query methods. However, sometimes, you may need to alter the view of that model for various reasons. In this section, you will learn how to define projections to serve up simplified and reduced views of resources.

Look at the following domain model:

```
@Entity
public class Person {

  @Id @GeneratedValue
  private Long id;
  private String firstName, lastName;

  @OneToOne
  private Address address;
  …
}

@Entity
public class Address {

  @Id @GeneratedValue
  private Long id;
  private String street, state, country;

  …
}
```

This `Person` has several attributes:

- `id` is the primary key
- `firstName` and `lastName` are data attributes
- `address` is a link to another domain object

Now assume we create a corresponding repository as follows:

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

  Person findPersonByFirstName(String firstName);
}
```

Spring Data will return the domain object including all of its attributes. There are two options just to retrieve the `address` attribute. One option is to define a repository for `Address` objects like this:

```
interface AddressRepository extends CrudRepository<Address, Long> {}
```

In this situation, using `PersonRepository` will still return the whole `Person` object. Using `AddressRepository` will return just the `Address`.

However, what if you do not want to expose `address` details at all? You can offer the consumer of your repository service an alternative by defining one or more projections.

Example 83. Simple Projection

```
interface NoAddresses {  ①

  String getFirstName(); ②

  String getLastName();  ③
}
```

This projection has the following details:

① A plain Java interface making it declarative.

② Export the `firstName`.

③ Export the `lastName`.

The `NoAddresses` projection only has getters for `firstName` and `lastName` meaning that it will not serve up any address information. The query method definition returns in this case `NoAdresses` instead of `Person`.

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

  NoAddresses findByFirstName(String firstName);
}
```

Projections declare a contract between the underlying type and the method signatures related to the exposed properties. Hence it is required to name getter methods according to the property name of the underlying type. If the underlying property is named `firstName`, then the getter method must be named `getFirstName` otherwise Spring Data is not able to look up the source property.

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. Similarly, find methods using N1QL are provided in `findByN1QL` and `queryN1QL`. Additionally, since N1QL allows you to select specific fields in documents (or even across documents using joins), `findByN1QLProjection` will allow you to skip full `Document` conversion and map these fields to an ad-hoc class.

WARNING	If it is detected at runtime that the cluster doesn't support N1QL, these methods will throw a `UnsupportedCouchbaseFeatureException`.

If you really need low-level semantics, the `couchbaseBucket` is also always in scope through `getCouchbaseBucket()`.

6.2. Xml Configuration

The template can be configured via xml, including setting a custom `TranslationService`.

Example 84. XML Based Template Declaration

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

    <couchbase:env/>
    <couchbase:cluster/>
    <couchbase:clusterInfo/>
    <couchbase:bucket/>

    <couchbase:template translation-service-ref="myCustomTranslationService"/>

    <bean id="myCustomTranslationService" class=
"org.springframework.data.couchbase.core.convert.translation.JacksonTranslationSer
vice"/>

</beans>
```

NOTE In the example above most tags assume their default values, that is a localhost cluster and bucket "default". In production you would have to also provide specifics to these tags.

Appendix

Appendix A: Namespace reference

The <repositories /> element

The `<repositories />` element triggers the setup of the Spring Data repository infrastructure. The most important attribute is `base-package` which defines the package to scan for Spring Data repository interfaces. [3: see XML configuration]

Table 7. Attributes

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

Appendix B: Populators namespace reference

The <populator /> element

The `<populator />` element allows to populate the a data store via the Spring Data repository infrastructure. [4: see XML configuration]

Table 8. 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 9. Query keywords

Logical keyword	Keyword expressions
`AND`	`And`
`OR`	`Or`
`AFTER`	`After`, `IsAfter`
`BEFORE`	`Before`, `IsBefore`
`CONTAINING`	`Containing`, `IsContaining`, `Contains`
`BETWEEN`	`Between`, `IsBetween`
`ENDING_WITH`	`EndingWith`, `IsEndingWith`, `EndsWith`
`EXISTS`	`Exists`
`FALSE`	`False`, `IsFalse`
`GREATER_THAN`	`GreaterThan`, `IsGreaterThan`
`GREATER_THAN_EQUALS`	`GreaterThanEqual`, `IsGreaterThanEqual`
`IN`	`In`, `IsIn`
`IS`	`Is`, `Equals`, (or no keyword)
`IS_EMPTY`	`IsEmpty`, `Empty`
`IS_NOT_EMPTY`	`IsNotEmpty`, `NotEmpty`
`IS_NOT_NULL`	`NotNull`, `IsNotNull`
`IS_NULL`	`Null`, `IsNull`
`LESS_THAN`	`LessThan`, `IsLessThan`
`LESS_THAN_EQUAL`	`LessThanEqual`, `IsLessThanEqual`
`LIKE`	`Like`, `IsLike`
`NEAR`	`Near`, `IsNear`
`NOT`	`Not`, `IsNot`
`NOT_IN`	`NotIn`, `IsNotIn`
`NOT_LIKE`	`NotLike`, `IsNotLike`
`REGEX`	`Regex`, `MatchesRegex`, `Matches`
`STARTING_WITH`	`StartingWith`, `IsStartingWith`, `StartsWith`
`TRUE`	`True`, `IsTrue`
`WITHIN`	`Within`, `IsWithin`

Appendix D: Repository query return types

Supported query return types

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

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

Table 10. Query return types

Return type	Description
void	Denotes no return value.
Primitives	Java primitives.
Wrapper types	Java wrapper types.
T	An unique entity. Expects the query method to return one result at most. In case no result is found null is returned. More than one result will trigger an IncorrectResultSizeDataAccessException.
Iterator<T>	An Iterator.
Collection<T>	A Collection.
List<T>	A List.
Optional<T>	A Java 8 or Guava Optional. Expects the query method to return one result at most. In case no result is found Optional.empty() /Optional.absent() is returned. More than one result will trigger an IncorrectResultSizeDataAccessException.
Option<T>	An either Scala or JavaSlang Option type. Semantically same behavior as Java 8's Optional described above.
Stream<T>	A Java 8 Stream.
Future<T>	A Future. Expects method to be annotated with @Async and requires Spring's asynchronous method execution capability enabled.
CompletableFuture<T>	A Java 8 CompletableFuture. Expects method to be annotated with @Async and requires Spring's asynchronous method execution capability enabled.
ListenableFuture	A org.springframework.util.concurrent.ListenableFuture. Expects method to be annotated with @Async and requires Spring's asynchronous method execution capability enabled.
Slice	A sized chunk of data with information whether there is more data available. Requires a Pageable method parameter.
Page<T>	A Slice with additional information, e.g. the total number of results. Requires a Pageable method parameter.
GeoResult<T>	A result entry with additional information, e.g. distance to a reference location.

Return type	Description
`GeoResults<T>`	A list of `GeoResult<T>` with additional information, e.g. average distance to a reference location.
`GeoPage<T>`	A `Page` with `GeoResult<T>`, e.g. average distance to a reference location.