Spring Data Elasticsearch

BioMed CentralDevelopment Team

Copyright © 2013The original author(s)

Table of Contents

Preface

The Spring Data Elasticsearch project applies core Spring concepts to the development of solutions using the Elasticsearch Search Engine. We have povided a "template" as a high-level abstraction for storing,querying,sorting and faceting documents. You will notice similarities to the Spring data solr and mongodb support in the Spring Framework.

1 Project Metadata

- Version Control - git://github.com/BioMedCentralLtd/spring-data-elasticsearch.git

2 Requirements

Requires Elasticsearch 0.20.2 and above or optional dependency or not even that if you are using Embedded Node Client

Part I. Reference Documentation

1. Repositories

1.1 Introduction

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

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

The following chapters will introduce the core concepts and interfaces of Spring Data repositories in general for detailled information on the specific features of a particular store consult the later chapters of this document.

Note

As this part of the documentation is pulled in from Spring Data Commons we have to decide for a particular module to be used as example. The configuration and code samples in this chapter are using the JPA module. Make sure you adapt e.g. the XML namespace declaration, types to be extended to the equivalents of the module you're actually using.

1.2 Core concepts

The central interface in Spring Data repository abstraction is `Repository` (probably not that much of a surprise). It is typeable to the domain class to manage as well as the id type of the domain class. This interface mainly acts as marker interface to capture the types to deal with and help us when discovering interfaces that extend this one. Beyond that there's `CrudRepository` which provides some sophisticated functionality around CRUD for the entity being managed.

```
public interface CrudRepository<T, ID extends Serializable>
    extends Repository<T, ID> {
                                                                        ❶
    T save(T entity);
                                                                        ❷
    T findOne(ID primaryKey);
                                                                        ❸
    Iterable<T> findAll();

    Long count();
                                                                        ❹
    void delete(T entity);
                                                                        ❺
    boolean exists(ID primaryKey);
                                                                        ❻
    // … more functionality omitted.
}
```

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

Example 1.1 `CrudRepository` interface

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

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

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

    Iterable<T> findAll(Sort sort);

    Page<T> findAll(Pageable pageable);
}
```
Example 1.2 PagingAndSortingRepository

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));
```

1.3 Query methods

Next to standard CRUD functionality repositories are usually 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 sub-interfaces and type it to the domain class it shall handle.

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

2. Declare query methods on the interface.

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

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

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

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

</beans>
```

Note

Note that we use the JPA namespace here just by example. If you're 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 e.g. `mongodb`.

4. Get the repository instance injected and use it.

```
public class SomeClient {

  @Autowired
  private PersonRepository repository;

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

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

Defining repository interfaces

As a very first step you define a domain class specific repository interface. It's got to 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`.

Fine tuning repository definition

Usually you will have your repository interface extend `Repository`, `CrudRepository` or `PagingAndSortingRepository`. If you don't like extending Spring Data interfaces at all you can also annotate your repository interface with `@RepositoryDefinition`. Extending `CrudRepository` will expose a complete set of methods to manipulate your entities. If you would rather be selective about

the methods being exposed, simply copy the ones you want to expose from `CrudRepository` into your domain repository.

```
interface MyBaseRepository<T, ID extends Serializable> extends Repository<T, ID> {
  T findOne(ID id);
  T save(T entity);
}

interface UserRepository extends MyBaseRepository<User, Long> {

  User findByEmailAddress(EmailAddress emailAddress);
}
```

Example 1.3 Selectively exposing CRUD methods

In the first step we define a common base interface for all our domain repositories and expose `findOne(…)` as well as `save(…)`.These methods will be routed into the base repository implementation of the store of your choice because they are matching the method signatures in `CrudRepository`. So our `UserRepository` will now be able to save users, find single ones by id as well as triggering a query to find `User`s by their email address.

Defining query methods

Query lookup strategies

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

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

CREATE

This strategy will try to construct a store specific query from the query method's name. The general approach is to remove a given set of well-known prefixes from the method name and parse the rest of the method. Read more about query construction in the section called "Query creation".

USE_DECLARED_QUERY

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

CREATE_IF_NOT_FOUND (default)

This strategy is actually a combination of `CREATE` and `USE_DECLARED_QUERY`. It will try to lookup a declared query first but create a custom method name based query if no declared query was found. This is the default lookup strategy and thus will be used if you don't configure anything explicitly. It allows quick query definition by method names but also custom tuning of these queries by introducing declared queries as needed.

Query creation

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

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

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

Example 1.4 Query creation from method names

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

Property expressions

Property expressions can just refer to a direct property of the managed entity (as you just saw in the example above). On query creation time we already make sure that the parsed property is at a property of the managed domain class. However, you can also define constraints by traversing nested properties. Assume `Person`s have `Address`es with `ZipCode`s. In that case a method name of

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

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

Although this should work for most cases, there might be cases where the algorithm could select the wrong property. Suppose our `Person` class has an `addressZip` property as well. Then our algorithm would match in the first split round already and essentially choose the wrong property and finally fail (as the type of `addressZip` probably has no code property). To resolve this ambiguity you can use _ inside your method name to manually define traversal points. So our method name would end up like so:

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

Special parameter handling

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

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

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

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

Example 1.5 Using Pageable and Sort in query methods

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

🔵 Note

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

Creating repository instances

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

XML Configuration

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

```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 this case we instruct Spring to scan com.acme.repositories and all its sub packages for interfaces extending `Repository` or one of its sub-interfaces. For each interface found it will register the persistence technology specific `FactoryBean` to create the according proxies that handle invocations of the query methods. Each of these beans will be registered under a bean name that is derived from the interface name, so an interface of `UserRepository` would be registered under `userRepository`. The `base-package` attribute allows the use of wildcards, so that you can have a pattern of scanned packages.

Using filters

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

inside `<repositories />`. The semantics are exactly equivalent to the elements in Spring's context namespace. For details see [Spring reference documentation](#) on these elements.

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

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

This would exclude all interfaces ending in `SomeRepository` from being instantiated.
Example 1.6 Using exclude-filter element

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 please have a look at the reference documentation.[2]

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

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

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

Example 1.7 Sample annotation based repository configuration

Note that the sample uses the JPA specific annotation which would have to be exchanged dependingon which store module you actually use. The same applies to the definition of the `EntityManagerFactory` bean. Please consult the sections covering the store-specific configuration.

Standalone usage

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

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

Example 1.8 Standalone usage of repository factory

1.4 Custom implementations

Adding behaviour to single repositories

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

---

[2]JavaConfig in the Spring reference documentation - http://static.springsource.org/spring/docs/3.1.x/spring-framework-reference/html/beans.html#beans-java

abstraction and query method functionality. To enrich a repository with custom functionality you have to define an interface and an implementation for that functionality first and let the repository interface you provided so far extend that custom interface.

```
interface UserRepositoryCustom {

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

Example 1.9 Interface for custom repository functionality

```
class UserRepositoryImpl implements UserRepositoryCustom {

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

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

Example 1.10 Implementation of custom repository functionality

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

  // Declare query methods here
}
```

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

Example 1.11 Changes to the your basic repository interface

Configuration

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

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

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

Example 1.12 Configuration example

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

Manual wiring

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

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

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

Example 1.13 Manual wiring of custom implementations (I)

Adding custom behaviour to all repositories

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

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

  void sharedCustomMethod(ID id);
}
```

Example 1.14 An interface declaring custom shared behaviour

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

Note

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

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

  private EntityManager entityManager;

  // There are two constructors to choose from, either can be used.
  public MyRepositoryImpl(Class<T> domainClass, EntityManager entityManager) {
    super(domainClass, entityManager);

    // This is the recommended method for accessing inherited class dependencies.
    this.entityManager = entityManager;
  }

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

Example 1.15 Custom repository base class

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

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

  protected RepositoryFactorySupport createRepositoryFactory(EntityManager entityManager)
 {

    return new MyRepositoryFactory(entityManager);
  }

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

    private EntityManager entityManager;

    public MyRepositoryFactory(EntityManager entityManager) {
      super(entityManager);

      this.entityManager = entityManager;
    }

    protected Object getTargetRepository(RepositoryMetadata metadata) {

      return new MyRepositoryImpl<T, I>((Class<T>) metadata.getDomainClass(),
entityManager);
    }

    protected Class<?> getRepositoryBaseClass(RepositoryMetadata metadata) {

      // The RepositoryMetadata can be safely ignored, it is used by the
JpaRepositoryFactory
      //to check for QueryDslJpaRepository's which is out of scope.
      return MyRepository.class;
    }
  }
}
```

Example 1.16 Custom repository factory bean

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

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

Example 1.17 Using the custom factory with the namespace

1.5 Extensions

This chapter 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.

Domain class web binding for Spring MVC

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

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

  private final UserRepository userRepository;

  public UserController(UserRepository userRepository) {
    userRepository = userRepository;
  }

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

    // Do null check for id
    User user = userRepository.findOne(id);
    // Do null check for user
    // Populate model
    return "user";
  }
}
```

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

PropertyEditors

For versions up to Spring 3.0 simple Java PropertyEditors had to be used. Thus, we offer a DomainClassPropertyEditorRegistrar, that will look up all Spring Data repositories registered in the ApplicationContext and register 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 like this you can turn your controller into the following that reduces a lot of the clutter and boilerplate.

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

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

    // Do null check for user
    // Populate model
    return "userForm";
  }
}
```
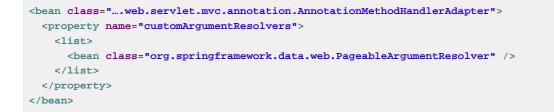
ConversionService

As of Spring 3.0 the `PropertyEditor` support is superseeded by a new conversion infrstructure that leaves all the drawbacks of `PropertyEditor`s behind and uses a stateless X to Y conversion approach. We now ship with a `DomainClassConverter` that pretty much mimics the behaviour of `DomainClassPropertyEditorRegistrar`. To register the converter you have to declare `ConversionServiceFactoryBean`, register the converter and tell the Spring MVC namespace to use the configured conversion service:

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

<bean id="conversionService" class="….context.support.ConversionServiceFactoryBean">
  <property name="converters">
    <list>
      <bean class="org.springframework.data.repository.support.DomainClassConverter">
        <constructor-arg ref="conversionService" />
      </bean>
    </list>
  </property>
</bean>
```

Web pagination

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

  // DI code omitted

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

    int page = Integer.parseInt(request.getParameter("page"));
    int pageSize = Integer.parseInt(request.getParameter("pageSize"));
    model.addAttribute("users", userService.getUsers(pageable));
    return "users";
  }
}
```

As you can see the naive approach requires the method to contain an `HttpServletRequest` parameter that has to be parsed manually. We even omitted an appropriate failure handling which would make the code even more verbose. The bottom line is that the controller actually shouldn't have to handle the functionality of extracting pagination information from the request. So we include a `PageableArgumentResolver` that will do the work for you.

```
<bean class="….web.servlet.mvc.annotation.AnnotationMethodHandlerAdapter">
  <property name="customArgumentResolvers">
    <list>
      <bean class="org.springframework.data.web.PageableArgumentResolver" />
    </list>
  </property>
</bean>
```

This configuration allows you to simplify controllers down to something like this:

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

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

    model.addAttribute("users", userDao.readAll(pageable));
    return "users";
  }
}
```

The `PageableArgumentResolver` will automatically resolve request parameters to build a `PageRequest` instance. By default it will expect the following structure for the request parameters:

Table 1.1. Request parameters evaluated by `PageableArgumentResolver`

page	The page you want to retrieve
page.size	The size of the page you want to retrieve
page.sort	The property that should be sorted by
page.sort.dir	The direction that should be used for sorting

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

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

you'd have to populate `foo_page` and `bar_page` and the according subproperties.

Defaulting

The `PageableArgumentResolver` will use a `PageRequest` with the first page and a page size of 10 by default and will use that in case it can't resolve a `PageRequest` from the request (because of missing parameters e.g.). You can configure a global default on the bean declaration directly. In case you might need controller method specific defaults for the `Pageable` simply annotate the method parameter with `@PageableDefaults` and specify page and page size as annotation attributes:

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

Repository populators

If you have been working with the JDBC module of Spring you're probably familiar with the support to populate a DataSource using SQL scripts. A similar abstraction is available on the repositories level although we don't use SQL as data definition language as we need to be store independent of course. Thus the populators support XML (through Spring's OXM abstraction) and JSON (through Jackson) to define data for the repositories to be populated with.

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

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

Example 1.18 Data defined in JSON

You can easily populate you repositories by using the populator elements of the repository namespace provided in Spring Data Commons. To get the just shown data be populated to your `PersonRepository` all you need to do is the following:

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

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

</beans>
```

Example 1.19 Declaring a Jackson repository populator

This declaration causes the data.json file being read, deserialized by a Jackson `ObjectMapper`. The type the JSON object will be unmarshalled to will be determined by inspecting the `_class` attribute of the JSON document. We will eventually select the appropriate repository being able to handle the object just deserialized.

To rather use XML to define the repositories shall be populated with you can use the unmarshaller-populator you hand one of the marshaller options Spring OXM provides you with.

```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: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 location="classpath:data.json" unmarshaller-
ref="unmarshaller" />

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

</beans>
```

Example 1.20 Declaring an unmarshalling repository populator (using JAXB)

2. Elasticsearch Repositories

This chapter includes details of the Elasticsearch repository implementation.

2.1 Introduction

Spring Namespace

The Spring Data Elasticsearch module contains a custom namespace allowing definition of repository beans as well as elements for instantiating a `ElasticsearchServer` .

Using the `repositories` element looks up Spring Data repositories as described in the section called "Creating repository instances" .

```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:elasticsearch="http://www.springframework.org/schema/data/elasticsearch"
xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans-3.1.xsd
http://www.springframework.org/schema/data/elasticsearch
http://www.springframework.org/schema/data/elasticsearch/spring-elasticsearch-1.0.xsd">

<elasticsearch:repositories base-package="com.acme.repositories" />
</beans>
```
Example 2.1 Setting up Elasticsearch repositories using Namespace

Using the `Transport Client` or `Node Client` element registers an instance of `Elasticsearch Server` in the context.

```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:elasticsearch="http://www.springframework.org/schema/data/elasticsearch"
xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans-3.1.xsd
http://www.springframework.org/schema/data/elasticsearch
http://www.springframework.org/schema/data/elasticsearch/spring-elasticsearch-1.0.xsd">

<elasticsearch:transport-client id="client" cluster-nodes="localhost:9300,someip:9300" />
</beans>
```
Example 2.2 Transport Client using Namespace

```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:elasticsearch="http://www.springframework.org/schema/data/elasticsearch"
xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans-3.1.xsd
http://www.springframework.org/schema/data/elasticsearch
http://www.springframework.org/schema/data/elasticsearch/spring-elasticsearch-1.0.xsd">

<elasticsearch:node-client id="client" local="true" />
</beans>
```
Example 2.3 Node Client using Namespace

Annotation based configuration

The Spring Data Elasticsearch repositories support cannot only be activated through an XML namespace but also using an annotation through JavaConfig.

```
@Configuration
@EnableElasticsearchRepositories(basePackages = "org/springframework/data/elasticsearch/
repositories")
static class Config {

@Bean
public ElasticsearchOperations elasticsearchTemplate() {
return new ElasticsearchTemplate(nodeBuilder().local(true).node().client());
 }
}
```

The configuration above sets up an `Embedded Elasticsearch Server` which is used by the `ElasticsearchTemplate` . Spring Data Elasticsearch Repositories are activated using the `@EnableElasticsearchRepositories` annotation, which essentially carries the same attributes as the XML namespace does. If no base package is configured, it will use the one the configuration class resides in.

Example 2.4 Spring Data Elasticsearch repositories using JavaConfig

Elasticsearch Repositores using CDI

The Spring Data Elasticsearch repositories can also be set up using CDI functionality.

```
class ElasticsearchTemplateProducer {

@Produces
@ApplicationScoped
public ElasticsearchOperations createElasticsearchTemplate() {
    return new ElasticsearchTemplate(nodeBuilder().local(true).node().client());
  }
}

class ProductService {

private ProductRepository repository;

public Page<Product> findAvailableBookByName(String name, Pageable pageable) {
   return repository.findByAvailableTrueAndNameStartingWith(name, pageable);
}

@Inject
public void setRepository(ProductRepository repository) {
   this.repository = repository;
 }
}
```

Example 2.5 Spring Data Elasticsearch repositories using JavaConfig

2.2 Query methods

Query lookup strategies

The Elasticsearch module supports all basic query building feature as String,Abstract,Criteria or have it being derived from the method name.

Declared queries

Deriving the query from the method name is not always sufficient and/or may result in unreadable method names. In this case one might make either use of `@Query` annotation (see the section called "Using @Query Annotation").

Query creation

Generally the query creation mechanism for Elasticsearch works as described in Section 1.3, "Query methods" . Here's a short example of what a Elasticsearch query method translates into:

```java
public interface BookRepository extends Repository<Book, String> {
  List<Book> findByNameAndPrice(String name, Integer price);
}
```

The method name above will be translated into the following Elasticsearch json query

```
{ "bool" :
{ "must" :
[
{ "field" : {"name" : "?"} },
{ "field" : {"price" : "?"} }
] } }
```

Example 2.6 Query creation from method names

A list of supported keywords for Elasticsearch is shown below.

Table 2.1. Supported keywords inside method names

Keyword	Sample	Elasticsearch Query String
And	findByNameAndPrice	{"bool" : {"must" : [{"field" : {"name" : "?"}}, {"field" : {"price" : "?"}}]}}
Or	findByNameOrPrice	{"bool" : {"should" : [{"field" : {"name" : "?"}}, {"field" : {"price" : "?"}}]}}
Is	findByName	{"bool" : {"must" : {"field" : {"name" : "?"}}}}
Not	findByNameNot	{"bool" : {"must_not" : {"field" : {"name" : "?"}}}}
Between	findByPriceBetween	{"bool" : {"must" : {"range" : {"price" :

Keyword	Sample	Elasticsearch Query String
		`{"from" : ?,"to" : ?,"include_lower" : true,"include_upper" : true}}}}`
LessThanEqual	findByPriceLessThan	`{"bool" : {"must" : {"range" : {"price" : {"from" : null,"to" : ?,"include_lower" : true,"include_upper" : true}}}}`
GreaterThanEqual	findByPriceGreaterThan	`{"bool" : {"must" : {"range" : {"price" : {"from" : ?,"to" : null,"include_lower" : true,"include_upper" : true}}}}`
Before	findByPriceBefore	`{"bool" : {"must" : {"range" : {"price" : {"from" : null,"to" : ?,"include_lower" : true,"include_upper" : true}}}}`
After	findByPriceAfter	`{"bool" : {"must" : {"range" : {"price" : {"from" : ?,"to" : null,"include_lower" : true,"include_upper" : true}}}}`
Like	findByNameLike	`{"bool" : {"must" : {"field" : {"name" : {"query" : "?*","analyze_wildcard" : true}}}}`
StartingWith	findByNameStartingWith	`{"bool" : {"must" : {"field" : {"name" : {"query" : "?*","analyze_wildcard" : true}}}}`
EndingWith	findByNameEndingWith	`{"bool" : {"must" : {"field" : {"name" : {"query" : "*?","analyze_wildcard" : true}}}}`
Contains/Containing	findByNameContaining	`{"bool" : {"must" : {"field" : {"name" : {"query" : "*?*","analyze_wildcard" : true}}}}`
In	findByNameIn(Collection<String>names)	`{"bool" : {"must" : {"bool" : {"should" : [ {"field" : {"name" : "?"}}, {"field" : {"name" : "?"}} ]}}}}`
NotIn	findByNameNotIn(Collection<String>names)	`{"bool" : {"must_not" : {"bool" : {"should" : {"field" : {"name" : "?"}}}}}}}`
Near	findByStoreNear	Not Supported Yet !
True	findByAvailableTrue	`{"bool" : {"must" : {"field" : {"available" : true}}}}`

Keyword	Sample	Elasticsearch Query String
False	findByAvailableFalse	`{"bool" : {"must" : {"field" : {"available" : false}}}}`
OrderBy	findByAvailableTrueOrderByNameDesc	`{"sort" : [{ "name" : {"order" : "desc"} }],"bool" : {"must" : {"field" : {"available" : true}}}}`

Using @Query Annotation

```
public interface BookRepository extends ElasticsearchRepository<Book, String> {
    @Query("{"bool" : {"must" : {"field" : {"name" : "?0"}}}}")
    Page<Book> findByName(String name,Pageable pageable);
}
```

Example 2.7 Declare query at the method using the `@Query` annotation.

3. Miscellaneous Elasticsearch Operation Support

This chapter covers additional support for Elasticsearch operations that cannot be directly accessed via the repository interface. It is recommended to add those operations as custom implementation as described in Section 1.4, "Custom implementations" .

3.1 Filter Builder

Filter Builder improves query speed.

```
private ElasticsearchTemplate elasticsearchTemplate;
SearchQuery searchQuery = new NativeSearchQueryBuilder()
.withQuery(matchAllQuery())
.withFilter(boolFilter().must(termFilter("id", documentId)))
.build();
Page<SampleEntity> sampleEntities =
 elasticsearchTemplate.queryForPage(searchQuery,SampleEntity.class);
```

Example 3.1

3.2 Using Scan And Scroll For Big Result Set

Elasticsearch has scan and scroll feature for getting big result set in chunks. `ElasticsearchTemplate` has scan and scroll methods that can be used as below.

```
SearchQuery searchQuery = new NativeSearchQueryBuilder()
.withQuery(matchAllQuery())
.withIndices("test-index")
.withTypes("test-type")
.withPageable(new PageRequest(0,1))
.build();
String scrollId = elasticsearchTemplate.scan(searchQuery,1000,false);
List<SampleEntity> sampleEntities = new ArrayList<SampleEntity>();
boolean hasRecords = true;
while (hasRecords){
Page<SampleEntity> page = elasticsearchTemplate.scroll(scrollId, 5000L , new
 ResultsMapper<SampleEntity>() {
@Override
public Page<SampleEntity> mapResults(SearchResponse response) {
List<SampleEntity> chunk = new ArrayList<SampleEntity>();
for(SearchHit searchHit : response.getHits()){
 if(response.getHits().getHits().length <= 0) {
   return null;
 }
 SampleEntity user = new SampleEntity();
 user.setId(searchHit.getId());
 user.setMessage((String)searchHit.getSource().get("message"));
 chunk.add(user);
 }
 return new PageImpl<SampleEntity>(chunk);
}
 });
   if(page != null) {
   sampleEntities.addAll(page.getContent());
   hasRecords = page.hasNextPage();
  }
  else{
   hasRecords = false;
  }
 }
}
```

Example 3.2 Using Scan and Scroll

Part II. Appendix

Appendix A. Namespace reference

A.1 The `<repositories />` element

The `<repositories />` 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.[1]

Table A.1. 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 also 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 the section called "Query lookup strategies" for details. Defaults to `create-if-not-found`.

---

[1]see the section called "XML Configuration"

Appendix B. Repository query keywords

B.1 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 as some of the ones listed here might not be supported in a particular store.

Table B.1. Query keywords

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

Logical keyword	Keyword expressions
TRUE	True, IsTrue
WITHIN	Within, IsWithin