Elasticsearch Plugin - Reference Documentation

Noam Y. Tenne

Version 1.0.0.1

Table of Contents

Chapter 1. Introduction

The ElasticSearch plugin intends to implement a simple integration with Grails of the Open Source Search Engine ElasticSearch, which is based on Lucene and provide distributed capabilities.

The plugin focus on exposing Grails domain classes for the moment. It highly takes the existing Searchable Plugin as reference for its syntax and behaviour.

Note that the plugin is still under development, so you may not be able to use all the features of ElasticSearch yet.

In addition to this document, you may want to read the official ElasticSearch documentation here.

1.1. Features

- Maps domain classes to their corresponding index in ElasticSearch
- Provides an ElasticSearch service for cross-domain searching
- Injects domain class methods for specific domain searching, indexing and unindexing
- Automatically mirrors any changes made through GORM to the index
- Allow to use the Groovy Content Builder DSL for search queries
- Support for term highlighting

1.2. History

1.2.1. Grails 3.x version

- April , 2016
  - 1.0.1
    - Add ability to change search method name in domain class via config
    - Updated documentation to asciidoc
- 2016
  - 1.0.0
    - Upgrade for Grails 3.1.1 *

1.2.2. Grails 2.x version

- April , 2016
  - 0.1.0
    - Upgrade to Elasticsearch 2.1.2 *
- June 30, 2015

- 0.0.4.5
    - Upgrade to ElasticSearch 1.6.0
    - Support the return of aggregation results

- June 15, 2015
    - 0.0.4.5
        - Add the ability to define property names that are excluded by default
        - Fix NPE
        - Add the attachment type

- March 5, 2015
    - 0.0.4.4
        - Upgrade to Elasticsearch-Groovy 1.4.4

- February 22, 2015
    - 0.0.4.3
        - Add mapping configuration support for '_all'
        - Fix issue with indexing nested GeoPoint
        - Add support for transient properties

- February 10, 2015
    - 0.0.4.2
        - Reduce severity of non-searchable property in index document when unmarshalling domain

- February 03, 2015
    - 0.0.4.1
        - Upgrade to Elasticsearch 1.4.2
        - Enable configuration of the number of replicas created per shard

- January 28, 2015
    - 0.0.4.0
        - Included Mapping migrations
        - Included read and write aliases to indices to deal with migrations on multinode deployments

- December 14, 2014
    - 0.0.3.8

        - Upgrade to ElasticSearch 1.4.1
        - Support the min_score query parameter.
        - Try to detect the MongoDB without using the plugin manager.

- December 01, 2014
    - 0.0.3.7
        - Create separate SimpleTypeConverter per-thread

- November 06, 2014
    - 0.0.3.6
        - Upgrade to ElasticSearch 1.4.0

- October 28, 2014
    - 0.0.3.5
        - Fix the bulk index query iteration.

- October 14, 2014
    - 0.0.3.4
        - Upgrade to latest version of ElasticSearch and remove the Groovy client dependency.

- August 28, 2014
    - 0.0.3.3
        - Configure a component field to act as an inner object instead of a nested object.

- August 3, 2014
    - 0.0.3.2
        - Add the ability to mark fields with aliases
        - Support ES client HTTP configuration parameters
        - Improve Hibernate 4 support

- June 9, 2014
    - 0.0.3.1
        - Upgrade to ElasticSearch 1.2.x
        - Add special treatment for MongoDB ObjectId data types
        - Return raw result objects when now class mapping is found
        - Fix integration-test NPE

- May 25, 2014

- 0.0.3.0
  - Upgrade to Grails dependency 2.2.x
  - Upgrade to Grails runtime 2.3.x
  - Upgrade to ElasticSearch 1.x
  - Apply ElasticSearch 1.x compatibility fixes
  - Enable customization of index name types when mapping classes

- May 15, 2014
  - 0.0.2.6
    - Use 'grails.util.Holders' instead of ApplicationHolder

- April 2, 2014
  - 0.0.2.5
    - Start releasing the plugin as 'elasticsearch' instead of 'elasticsearch-gorm'
    - Fix NPE when marshalling JSONObject fields

- March 24, 2014
  - 0.0.2.4
    - GeoPoint mapping
    - Injected service now supports filters (e.g. geo_reference) and sort builders (e.g. for geo_distance sorting)
    - Marshalled date values are now with correct time zone
    - Removed dependency on Java 7
    - Fix support of BigDecimal
    - Searchable mapping property name and Elasticsearch plugin path are now configurable.

- February 4, 2014
  - 0.0.2.3 Bugfix release
- January 19, 2014
  - 0.0.2.2 Bugfix release
- November 24, 2013
  - 0.0.2.1 Bugfix release
- November 12, 2013
  - 0.0.2 release
- November 2, 2013
  - initial 0.0.1 release

1.3. Acknowledgments

Many thanks to all the users who reported issues and sent me pull requests.

1.3.1. Authors and Contributors

- Noam Y. Tenne
- Stefan Rother-Stübs (Dating Cafe)
- Sven Kiesewetter (Dating Cafe)
- Michael Schwartz (Dating Cafe)
- Puneet Behl

1.3.2. Authors and Contributors of the original plugin

- Manuarii Stein (doc4web consulting)
- Stephane Maldini (doc4web consulting)
- Serge P. Nekoval

1.3.3. Previous work

Graeme Rocher started the first draft which this plugin is based on.

Get the full and updated list of contributors on the github repository.

1.4. Plugin Versioning

The versioning model has changed. The version number of the plugin will reflect the one of the underline integrated Elasticsearch. If necessary a 4th level point release number will be used for successive changes on the plugin's code with same version of Elasticsearch.

`<GRAILS_VERSION>.<ES_VERSION>.<FEATURE/PATCH_VERSION>`, where there isn't really a 1-to-1 plugin version to grails or es version, but we just increase our major or minor version by one, whenever there are breaking changes on either Grails or ES. Therefore have something that looks like:

Plugin Version	Grails	Elasticsearch
0.0.4	2.4.x	1.x
0.1.x	2.4.x	2.1.x
0.2.x	2.4.x	2.2.x
0.3.x	2.4.x	(hypothetical) 3.0
1.0.x	3.1.x	1.x
1.1.x	3.1.x	2.1.x

1.2.x	3.1.x	2.2.x
2.2.x	(hypothetical)3.2.x	2.2.x
3.2.x	(hypothetical) 4.0.x	2.2.x

Current version is **1.0.0.1** (for Grails 2.x the latest version is **0.1.0**)

1.5. License

This plugin is released under the Apache License, Version 2.0

Chapter 2. Configuration

The plugin provide a default configuration, but you may add your own settings in your **Config.groovy** for Grails 2.x and **application.groovy** or **application.yml** for Grails 3.x.

2.1. Client mode

You can set the plugin in 3 different modes, detailed on the official ElasticSearch doc. The mode is defined with the following config key:

applicaiton.groovy or Config.groovy

```
elasticSearch.client.mode = '<mode>'
```

application.yml

```
elasticSearch:
    client:
        node: <mode>
```

Table 1. Possible values for client node

Value	Description
node	The plugin create its own node and join the ElasticSearch cluster as a client node (`node.client = true`). This setting requires that you have an ElasticSearch instance running and available on your network (use the discovery feature)
dataNode	The plugin create its own node and join the ElasticSearch cluster as a node that can hold data. This setting requires that you have an ElasticSearch instance running and available on your network (use the discovery feature)
local	The plugin create its own local (to the JVM) node. Does not require any running ElasticSearch instance. Useful for development or testing.
transport	The plugin create a transport client that will connect to a remote ElasticSearch instance without joining the cluster.

"Transport" mode needs you to provide the host address and port. You can define one or multiple hosts with the following config key:

application.groovy or Config.groovy

```
elasticSearch.client.hosts = [
       [host:'192.168.0.3', port:9300],
       [host:'228.168.0.4', port:9300]
]
```

application.yml

```
elasticSearch:
    client:
        hosts:
        - {host: 192.168.0.3, port: 9300}
        - {host: 228.168.0.4, port: 9300}
```

If no host is defined, `localhost:9300` will be used by the transport client.

2.2. Mapping Migration properties

Define the application's behaviour when a conflict is found while installing Elasticsearch mappings on startup. For a detailed explanation, see Mapping Migrations.

2.2.1. `elasticSearch.migration.strategy`

Defines the behaviour to follow if an error occurs on startup when the application is installing new mappings on ElasticSearch due to conflicting mappings.

Table 2. Possible Values for migration strategy

Value	Description
`'none'`	No changes on the indices or mappings will happen, the merge problem will be logged and a MappingException will be thrown.
`'delete'`	The conflicting mapping will be deleted (along with all indexed content of that type) and replaced with a new mapping. Deleted content can be automatically reindexed on startup by using this in combination the `elasticSearch.bulkIndexOnStartup` config option

Value	Description
`'alias'`	Applies Elasticsearch recommended approach for migrating conflicting mappings. A new numbered index will be created (`<indexName>_vX`) where new mappings will installed for all the types included on the original index. An Elasticsearch alias called `<indexName>` will point to the new index. As content won't be available on the new index, content can be automatically reindexed on startup by using this in combination the `elasticSearch.bulkIndexOnStartup` config option. It is recommended to set `elasticSearch.aliasReplacesIndex` to deal with potential index/alias conflicts.

The default is `'alias'`.

2.2.2. `elasticSearch.migration.aliasReplacesIndex`

Deals with a special conflict case using the `'alias'` strategy. When the `'alias'` migration strategy is chosen and there's a mapping conflict on an index, defines whether to replace the index with a versioned index (`<indexName>_vX`) and an alias (`<indexName>`). This is required when applying the alias strategy on top of existing indices for the first time as indices cannot be renamed (from `<indexName>` to `<indexName>_vX`) and an alias cannot exist with the same name as an index.

Table 3. Possible Values for aliasReplacesIndex

Value	Description
`true`	The index and it's content will be deleted and a versioned index and an alias will be created. Deleted content can be automatically reindexed on startup by using this in combination the `elasticSearch.bulkIndexOnStartup` config option
`false`	Falls back to the `'none'` strategy. Event will be logged and a MappingException will be thrown.

The default is `true`.

2.2.3. `elasticSearch.migration.disableAliasChange`

In some cases the developer may prefer not to upgrade the alias to the new version of the index until some other tasks are performed. This allows them to disable automatically pointing the alias to a new version of the index when this is created. Aliases can be changed later on manually or programatically using `elasticSearchAdminService`

Table 4. Possible Values for disableAliasChange

Value	Description
`false`	Standard behaviour
`true`	Prevents the aliases to be changed to point to a new index

The default is `false`.

2.3. Others properties

2.3.1. `elasticSearch.datastoreImpl`

Only required when enabling the auto-index feature. This property specifies which GORM datastore implementation should be watched for storage events. The value should be the name of the datastore bean as it is configured in the Spring context; some possible values:

Table 5. Possible Values for datastoreImpl

Value	Description
mongoDatastore	The name of the MongoDB datastore bean.
hibernateDatastore	The name of the Hibernate datastore bean.

2.3.2. `elasticSearch.bootstrap.config.file`

When using then plugin to construct a local node, the default Elasticsearch configuration is used by default. If you use a modified Elasticsearch configuration, you can use this property to specify the location of the file (as an application resource).

2.3.3. `elasticSearch.bootstrap.transportSettings.file`

When choosing transport mode this configuration will be used to set up the TransportClient settings (used by some cloud providers).

2.3.4. `elasticSearch.client.transport.sniff`

Only usable in with a transport client. Allows to sniff the rest of the cluster, and add those into its list of machines to use. In this case, the ip addresses used will be the ones that the other nodes were started with (the “publish” address)

2.3.5. `elasticSearch.cluster.name`

The name of the cluster for the client to join.

2.3.6. `elasticSearch.date.formats`

List of date formats used by the JSON unmarshaller to parse any date field properly. Note : future version of the plugin may change how formats are manipulated.

2.3.7. elasticSearch.defaultExcludedProperties

List of domain class properties to automatically ignore (will not be indexed) if their name match one of those. This will apply to both the default-mapped domain class, with the static `searchable` property set to "true", and when using closure mapping. To override this setting on a specific class, it can be added to the `only` property of the `searchable` closure.

2.3.8. elasticSearch.disableAutoIndex

A boolean determining if the plugin should reflect any database save/update/delete automatically on the indices. Default to `false`.

2.3.9. elasticSearch.bulkIndexOnStartup

Determines whether the application should launch a bulk index operation upon startup.

Table 6. Possible Values for bulkIndexOnStartup

Value	Description
`false`	No indexing will happen on startup.
`true`	All content will be indexed on startup.
`'deleted'`	This value is related to the mapping migration strategy chosen. If any migration is required and any content is deleted due to it, on startup only indices and mappings lost will be indexed. More on Mapping Migrations.

Default to `true`.

2.3.10. elasticSearch.index.name

A string indicating which ElasticSearch index should be used. If not present, will default to the package name of the domain in question.

2.3.11. elasticSearch.index.compound_format

Should the compound file format be used (boolean setting). Set to `false` by default (really applicable for file system based index storage). More details on this setting on the ElasticSearch Documentation.

2.3.12. elasticSearch.index.store.type

Determine how the indices will be stored. More details on the possible values on the ElasticSearch Documentation.

Table 7. Possible value for index store type

Value	Description
memory	Stores the index in memory. Useful for testing.
mmapfs	Stores the shard index on the file system (maps to Lucene MMapDirectory) using mmap.
niofs	Stores the shard index on the file system (maps to Lucene NIOFSDirectory) and allows for multiple threads to read from the same file concurrently.
simplefs	Stores using a plain forward implementation of file system storage (maps to Lucene SimpleFsDirectory) using random access file.

2.3.13. `elasticSearch.index.numberOfReplicas`

Sets the number of replicas created for each shard of the index. If not present, will default to zero.

2.3.14. `elasticSearch.gateway.type`

Determine the gateway type to be used. More details on the possible values are in the ElasticSearch Documentation. Using a setting of "none" (possibly in combination with index.store.type set to "memory") can be useful for tests.

2.3.15. `elasticSearch.maxBulkRequest`

Max number of requests to process at once. Reduce this value if you have memory issue when indexing a big amount of data at once. If this setting is not specified, 500 will be use by default.

2.3.16. `elasticSearch.path.data`

The location of the data files of each index / shard allocated on the node.

2.3.17. `elasticSearch.path.plugins`

The location of plugin files such as native scripts. Each plugin will be contained in a subdirectory.

2.3.18. `elasticSearch.searchableProperty.name`

The name of the ElasticSearch mapping configuration property that annotates domain classes. The default is 'searchable'.

2.3.19. `elasticSearch.includeTransients`

Whether to index and search all non excluded transient properties. All explicitly included transients in `only` will be indexed regardless.

Default is `false`.

2.3.20. `elasticSearch.searchMethodName`

Change the name of search method in domain class. By default it's `search`.

For example

```
MyDomain.search("${params.query}")
```

In order to change the method name to `esSearch` just update the `elasticSearch.searchMethodName='esSearch'` in application.groovy

2.4. Default configuration script

2.4.1. Grails 2.x

Below is the default configuration loaded by the plugin (any of your settings in the Config.groovy script overwrite those).

Config.groovy

```
elasticSearch {
  /**
   * Date formats used by the unmarshaller of the JSON responses
   */
  date.formats = ["yyyy-MM-dd'T'HH:mm:ss'Z'"]

  /**
   * Hosts for remote ElasticSearch instances.
   * Will only be used with the "transport" client mode.
   * If the client mode is set to "transport" and no hosts are defined, ["localhost",
9300] will be used by default.
   */
  client.hosts = [
          [host:'localhost', port:9300]
  ]

  /**
   * Default mapping property exclusions
   *
   * No properties matching the given names will be mapped by default
   * i.e., when using "searchable = true"
   *
   * This does not apply for classes using mapping by closure
   */
  defaultExcludedProperties = ["password"]

  /**
   * Determines if the plugin should reflect any database save/update/delete
```

```
automatically
   * on the ES instance. Default to false.
   */
  disableAutoIndex = false

  /**
   * Should the database be indexed at startup.
   *
   * The value may be a boolean true|false.
   * Indexing is always asynchronous (compared to Searchable plugin) and executed
after BootStrap.groovy.
   */
  bulkIndexOnStartup = true

  /**
   *  Max number of requests to process at once. Reduce this value if you have memory
issue when indexing a big amount of data
   *  at once. If this setting is not specified, 500 will be use by default.
   */
  maxBulkRequest = 500


  /**
   * The name of the ElasticSearch mapping configuration property that annotates
domain classes. The default is 'searchable'.
   */
  searchableProperty.name = 'searchable'
}

environments {
  development {
    /**
     * Possible values : "local", "node", "dataNode", "transport"
     * If set to null, "node" mode is used by default.
     */
    elasticSearch.client.mode = 'local'
  }
  test {
      elasticSearch {
          client.mode = 'local'
          index.store.type = 'memory' // store local node in memory and not on disk
      }
  }
  production {
    elasticSearch.client.mode = 'node'
  }
}
```

2.4.2. Grails 3.x

Below is the default configuration loaded by the plugin (any of your settings in the application.yml script overwrite those).

application.yml

```
elasticSearch:
    date:
        formats: ["yyyy-MM-dd'T'HH:mm:ss.SSS'Z'"]
    client.hosts:
        - {host: localhost, port: 9300}
    defaultExcludedProperties: ['password']
    disableAutoIndex: false
    index:
        compound_format: true
    unmarshallComponents: true
    searchableProperty:
        name: searchable
    includeTransients: false
environments:
    development:
        elasticSearch:
            client:
                mode: local
                transport.sniff: true
            bulkIndexOnStartup: true
    test:
        elasticSearch:
            client:
                mode: local
                transport.sniff: true
            datastoreImpl: hibernateDatastore
            index:
                store.type: memory
                analysis:
                    filter:
                        replace_synonyms:
                            type: synonym
                            synonyms: ['abc => xyz']
                    analyzer:
                        test_analyzer:
                            tokenizer: standard
                            filter: ['lowercase']
                        repl_analyzer:
                            tokenizer: standard
                            filter: ['lowercase', 'replace_synonyms']

    production:
        elasticSearch:
            client:
                mode: node
```

Chapter 3. Mapping

From version 0.0.4.0 in addition to the indices generated by the plugin based on the domain objects package names or configuration, two new aliases are created for every index: `<indexName>_read` and `<indexName>_write`. These two aliases are used by the plugin to index and query from Elasticsearch and are needed to centralise the choice of index to use during mapping migrations when the `'alias'` strategy is used and there are multiple instances of the application.

3.1. Quick Start

3.1.1. Default mapping

To declare a domain class to be searchable, the simplest way is to define the following static property in the code:

```
static searchable = true
```

The plugin will generate a default mapping for each properties of the domain.

3.1.2. Custom mapping

You can customize how each properties are mapped to the index using a closure. The syntax is similar to GORM's mapping DSL.

```
static searchable = {
    // mapping DSL...
}
```

See below for more details on the mapping DSL.

3.1.3. Limit properties with only/except

`only` and `except` are used to limit the properties that are made searchable. You may not define both except & only settings at the same time.

The following code will only map the 'message' property, any others will be ignored.

```
class Tweet {
    static searchable = {
        only = 'message'
    }
    String message
    String someUselessField
}
```

The following code will map all properties except the one specified.

```
class Tweet {
    static searchable = {
        except = 'someUselessField'
    }
    String message
    String someUselessField
}
```

You can use a Collection to specify several properties.

```
class Tweet {
    static searchable = {
        except = ['someUselessField', 'userName']
    }
    String message
    String userName
    String someUselessField
}
```

The properties that are ignored will not be sent to ElasticSearch. It also means that when you will get back a domain from ElasticSearch, some fields that are not supposed to be null, may still be null.

3.1.4. Including transients

How the plugin manages transient properties is controlled by the `elasticSearch.includeTransients` configuration property. If this is set to `false` only transient properties explicitly included in `only` will be mapped and searchable, if set to `true`, all domain class properties will be mapped, including `transients`.

The following are valid examples

```
//assert grailsApplication.config.elasticSearch.includeTransients == false
class Person {
    String firstName
    String lastName
    String getFullName() {
        firstName + " " + lastName
    }
    static transients = ['fullName']
    static searchable = {
        only = ['fullName']
    }
}

// new Person(firstNameme: "Nikola", lastName: "Tesla")
// can be found using:
// def tesla = Person.search("Nikola Tesla").searchResults.first()
```

```
//assert grailsApplication.config.elasticSearch.includeTransients == true
class Multiplication {
    int opA
    int opB
    int getResult() {
        opA * opB
    }
    static transients = ['result']
    static searchable = true
}
// new Multiplication(opA: 2, opB: 3)
// can be found using:
// def multiplication = Multiplication.search("2").searchResults.first()
// def multiplication = Multiplication.search("3").searchResults.first()
// def multiplication = Multiplication.search("6").searchResults.first()
```

From the examples above, once the domain object is found, its transient values will be calculated from the information stored on ElasticSearch: `multiplication.result == 6`, but `tesla.fullName == "null null"`, as firstName and lastName where not indexed. This behaviour can be prevented by creating convenient setters for the transient properties.

3.1.5. Transients and collections

When transient properties are collections the only way the plugin can define the correct ElasticSearch mapping during boot is if the element types are explicitly defined on the grails domain object. For instances of `Collection` this can be achieved by defining its type on the `hasMany` property (otherwise the ElasticSearch type will be defined as `object`). This is not required for arrays.

Some valid examples:

```
class Tweet {
    String message
    List getHashtags() { ... }
    static transients = ['hashtags']
    static hasMany = [hashtags: String]
    static searchable = {only = 'hashtags' }
}
```

```
class FamilyGuy {
    String wife
    String son
    String daughter
    String baby
    String[] getRelatives() { ... }
    static transients = ['relatives']
    static searchable = { only = 'relatives' }
}
```

3.2. Class Mapping

3.2.1. root

Determine if the domain class will have its own index or not. Take a boolean as parameter, and is set to `true` by default.

```
class Preference {
    static searchable = {
        root false
    }
    // ...
}

class Tag {
    static searchable = true
    // ...
}

class Tweet {
    static searchable = {
        message boost:2.0
    }
    // ...
}
```

In this code, the classes `Tweet` and `Tag` are going to have their own index. The class `Preference` will not. It also mean that any search request will never return a Preference-type hit. The dynamic method `search` will not be injected in the `Preference` domain class.

The domains not root-mapped can still be considered searchable, as they can be components of another domain which is root-mapped. For example, considered the following domain:

```
class User {
    static searchable = {
        userPreferences component:true
    }

    Preference userPreferences
}
```

When searching, any matches in the `userPreferences` property will be considered as a `User` match.

3.2.2. all

Set default analyzer for all domain class fields.

```
static searchable = {
  all = [analyzer: 'russian_morphology']
}
```

```
static searchable = {
  all = false
}
```

When disabling the all field, it is a good practice to set index.query.default_field to a different value (for example, if you have a main 'message' field in your data, set it to message).

3.3. Properties Mapping

You can customize the mapping for each domain properties using the closure mapping. The syntax is simple:

```
static searchable = {
    propertyName option1:value, option2:value, ...
}
```

3.3.1. Available options

Option Name	Values	Description
boost	Number	A decimal boost value. With a positive value, promotes search results for hits in this property; with a negative value, demotes search results that hit this property.
component	`true`, `false`	To use only on domain (or collection of domains), make the property a searchable component.
converter	A `Class`	A `Class` to use as a converter during the marshalling/un-marshalling process for that peculiar property. That class must extends the `PropertyEditorSupport` java class.
excludeFromAll	`true`, `false`	determines if the property is to append in the `"_all"` field. Default to `true`.
index	`"no"`, `"not_analyzed"`, `"analyzed"`.	How or if the property is made into searchable element. One of `"no"`, `"not_analyzed"` or `"analyzed"`.
reference	`true`, `false`	To use only on domain (or collection of domains), make the property a searchable reference.
parent	`true`, `false`	A boolean value to be used in conjunction with the `reference` or `component` property . Set to `true` if the referenced field should be mapped as the parent of this document. Default set to `false`.
multi_field	`true`, `false`	A boolean value. Maps the value of the field twice; Once with it being analyzed, and once with it being not_analyzed under untouched. Default set to `false`.
geoPoint	`true`, `false`	Maps the field to a geo_point. Default: `false`

Option Name	Values	Description
alias	String	A string value. The field noted with this parameter will be duplicated to an alias
dynamic	`true`, `false`	Only available for String properties. Determines whether this field should be dynamically mapped by elasticsearch.

3.4. Parent Child

To map a parent/child relationship, the child element must either contain the parent element as a component or reference it as a referenced document. This component must be mapped as a parent in the child element.

Example

```
class ParentElement {
...
}

class EmbeddingChild {
    ParentElement parentElement

    static searchable = {
        parentElement parent: true, component: true
    }
}

class ReferencingChild {
    ParentElement parentElement

    static searchable = {
        parentElement parent: true, reference: true
    }
}
```

3.5. Geo Point

A geographic location can be mapped to a geo_point. The field for the longitude has to be named `lon` and the field for the latitude has to be named `lat`

Example

```
class GeoPoint {

    Double lat
    Double lon

    static searchable = {
        root false
    }
}

class Building {

    String name
    GeoPoint location

    static searchable = {
        location geoPoint: true, component: true
    }
}
```

3.6. Alias

A field can be aliased. This is useful in situations where another service may expect certain tags.

For example, Kibana uses an `\@timestamp` field to filter report records by date.

Example

```
class Session {

    Date loginTime

    static searchable = {
        loginTime alias:'@timestamp'
    }
}
```

3.7. Dynamic

Elasticsearch can map field contents as dynamic objects.

This is especially useful if you store JSON Strings in your database and want to make those objects searchable in elasticsearch.

Example

```
class Session {

    String jsonData

    static searchable = {
        dynamic: true
    }
}

Session session = new Session()
session.jsonData = ([foo: 'bar'] as JSON).toString()
```

The default mapping would make the jsonData field an escaped String field and a search for jsonData.foo = bar would result in no result. With dynamic mapping enabled for this field, we enable JSON handling of this field and tell elasticsearch to map this field dynamically. The result is that a search for `jsonData.foo=bar` would result in a search hit.

This will only work on String fields and will result in an error if the String is no valid json

3.8. Searchable Component Reference

The plugin support a similar searchable-component & searchable-reference behaviour from Compass when you are dealing with domain association. See below to find out about the difference between both mapping modes.

3.8.1. Searchable Reference

The searchable-reference mapping mode is the default mode used for association, and requires the searchable class of the association to be root-mapped in order to have its own index. With this mode, the associated domains are not completely marshalled in the resulting JSON document: only the id and the type of the instances are kept. When the document is retrieved from the index, the plugin will automatically rebuild the association from the indices using the stored id.

Example

```
class MyDomain {
    // odom is an association with the OtherDomain class, set as a reference
    OtherDomain odom

    static searchable = {
        odom reference:true
    }
}

// The OtherDomain definition, with default searchable configuration
class OtherDomain {
    static searchable = true

    String field1 = "val1"
    String field2 = "val2"
    String field3 = "val3"
    String field4 = "val4"
}
```

When indexing an instance of MyDomain, the resulting JSON documents will be sent to ElasticSearch:

```
{
    "mydomain": {
        "_id":1,
        "odom": { "id":1 }
    }
}

{
    "otherdomain": {
        "_id":1,
        "field1":"val1",
        "field2":"val2",
        "field3":"val3",
        "field4":"val4"
    }
}
```

3.8.2. Searchable Component

The searchable-component mapping mode must be explicitly set, and does not require the searchable class of the association to be root-mapped.

With this mode, the associated domains are nested in the parent document.

Example

```
class MyDomain {
    // odom is an association with the OtherDomain class, set as a reference
    OtherDomain odom

    static searchable = {
        odom component:true
    }
}

// The OtherDomain definition, with default searchable configuration
class OtherDomain {
    static searchable = true

    String field1 = "val1"
    String field2 = "val2"
    String field3 = "val3"
    String field4 = "val4"
}
```

When indexing an instance of MyDomain, the resulting JSON document will be sent to ElasticSearch:

```
{
    "mydomain": {
        "_id":1,
        "odom": {
            "_id":1,
            "field1":"val1",
            "field2":"val2",
            "field3":"val3",
            "field4":"val4"
        }
    }
}
```

If you'd rather that the reference object be mapped with type 'inner' rather than the default 'nested', set the 'component' key with a value of 'inner' rather than 'true':

```
class MyDomain {
    // odom is an association with the OtherDomain class, set as a reference
    OtherDomain odom

    static searchable = {
        odom component: 'inner'
    }
}
```

3.9. Mapping Migrations

During the application startup the application will attempt to create the needed indices on Elasticsearch and create the type mappings defined by the user. If these indices and mappings already existed on the Elasticsearch cluster (ie. an older version of the application was running against it) and the new mapping definitions differ with the existing ones there's the potential for a Mapping conflict. This section describes how to configure the application to deal with this scenario.

It is important to highlight that not all type mapping changes will result on a conflict. Ie. adding a new field to a mapping does not result in a conflict whilst changing a property from component:'inner' to nested or vice-versa, will. These strategies will only be needed and applied when a **conflicting** mapping is found.

3.9.1. Migration Strategies

The migration strategy is defined by the `elasticSearch.migration.strategy` configuration property and it accepts three values:

- `'none'`
- `'delete'`
- `'alias'`

The default strategy is `'alias'` as it is the only strategy that can achieve zero-downtime migrations and thus recommended by Elasticsearch

These values are described on more detail further ahead

3.9.2. Migration Strategy 'none'

This option keeps the original behaviour the plugin used before the Migration Strategies were implemented. When a Mapping Merge conflict id identified the event will be logged and an Exception will be logged. It will be responsibility for the application administrator to manually fix the problem.

This configuration was left as a backwards compatibility and it will prevent the application from booting successfully, therefore we **discourage teams from using this option**.

3.9.3. Migration Strategy 'delete'

When choosing this option, when a conflict occurs installing mapping, the application will delete the existing mapping for the type, alongside with all content indexed on that index and type and recreated the mapping. There are a couple of important details on this information:

- Only documents indexed on the conflicting mapping will be deleted, any other document on a different mapping on the same (or other) index will remain untouched.
- Deleted documents can be automatically reindexed on startup by using the `elasticSearch.bulkIndexOnStartup` configuration property (See below)
- Using this configuration there will always be a time window (between deletion and

reindexation) where documents can't be found by search, therefore this option cannot achieve a **zero-downtime** deployment

See Dealing with deleted content below for more details on automatic indexing.

3.9.4. Migration Strategy 'alias'

This is the migration strategy recommended by Elasticsearch.

To better understand this strategy we will describe a typical `'alias'` migration.

```
Elasticsearch contains
  index 'myapplication.store_v27' with types 'car' and 'motorbike'
  alias 'myapplication.store' pointing to 'myapplication.store_v27'
  'myapplication.store_v27/car' contains 520 documents
  'myapplication.store_v27/motorbike' contains 12 documents
  index 'myapplication.admin_v0' with type 'quote'
  alias 'myapplication.admin' pointing to 'myapplication.admin_v0'
  'myapplication.admin_v0/quote' contains 3200 documents

The application is configured to use indexes based on package names
'myapplication.store' and 'myapplication.admin'
(which as we already explained are actually aliases that point to versioned indices)

The team introduced a change on the Car domain that results in a conflict on the 'car'
mapping

The application starts up
    Tries to install the mapping for 'motorbike', it detects the conflict
    Creates a new index called 'myapplication.store_v28'
    Creates mappings 'myapplication.store_v28/car' and
'myapplication.store_v28/motorbike'
    Points all indexing requests for Car and Motorbike to the new index, while queries
still happen on 'myapplication.store'
On Boostrap (bulkIndexOnStartup)
    It indexes 520 cars into 'myapplication.store_v28/car'
    It indexes 12 motorbikes into 'myapplication.store_v28/motorbike'
    Switches the 'myapplication.store' alias to point to 'myapplication.store_v28'
    Now all cars are indexed according to the new mapping
    Now all motorbikes are indexed according to the new mapping
```

All content can be queried at all times, during Bootstrap bulkIndexOnStartup content will be retrieved from the old index.

Even though there wasn't a conflict on 'car', all cars needed to be reindexed as they lived on the same index.

There are three potential scenarios when using the `'alias'` strategy:

Scenario	Behaviour
The index (ie. 'myapplication.store') does not exist	On this case there is not possibility of conflicts, as no previous mapping exist. However the application will behave slightly different than on the other to scenarios. Instead of creating the index (ie. 'myapplication.store'), it will create version 0 of it (ie. 'myapplication.store_v0') and an alias pointing to it. This is to facilitate the creation of future versions in case of conflict.
Alias exists pointing to a version (ie. 'myapplication.store' → 'myapplication.store_v27')	If there's a conflict on a mapping on the index, it will create a new version (ie. 'myapplication.store_v28'), reindex the content or not depending on the value of the `elasticSearch.bulkIndexOnStartup` configuration property and point the alias to the new version once done.
Index already exists (ie. 'myapplication.store')	Elasticsearch cannot rename an index or create an alias with the same name as an index. The two alternatives here are to delete the index or fail the migration. This is controlled by the `elasticSearch.migration.aliasReplacesIndex` configuration property, if set to true, it will delete the index and proceed the same way as when the index did not exist. The deleted documents will be reindexed or not depending on the value of the `elasticSearch.bulkIndexOnStartup`. **This is the only scenario where there is content loss/downtime using the `'alias'` strategy.**

In the case you wanted to create a new version of an index, but not change where the alias points to (ie. for testing or if you wanted to perform extra tasks on the index before updating the alias), the `elasticSearch.migration.disableAliasChange` configuration property can be used

Aliases will only point to the new version of the index once all content is reindexed (if chosen to). Meanwhile, all index requests, either by `elasticSearchService` or using dynamic finders will go to the new version of the index, whilst queries will go to the old version of the index.

See Dealing with deleted content below for more details on automatic indexing.

3.9.5. Dealing with deleted content

Using the `'delete'` or `'alias'` strategy may lead to deleting content stored on Elasticsearch. This content can be automatically reindexed using the `elasticSearch.bulkIndexOnStartup`. The duration of this process will depend on the amount of content to index.

When this property is set to `true` all content will be deleted. When set to `'deleted'` only the domain classes which documents where deleted will be indexed. In either case, when using the `'alias'`

strategy, once all content is indexed all aliases will point to the latest version of the index.

Chapter 4. Indexing

With its default configuration (with the `disableAutoIndex` configuration key set to `false`), the plugin is indexing automatically any searchable domains when GORM/Hibernate do a save or an update in the database.

It also delete automatically from the index any document corresponding to a domain that is deleted from the database. You normally shouldn't have to worry about indexing, but sometimes you may have to do it by yourself, for example on dirty domain object that you may not want to save right now.

The plugin is providing a few injected methods in the domain or in the `ElasticSearchService` to allow that.

4.1. Index examples

```
// Index all searchable instances
elasticSearchService.index()

// Index a specific domain instance
MyDomain md = new MyDomain(value:'that')
md.save()
elasticSearchService.index(md)

// Index a collection of domain instances
def ds = [new MyDomain(value:'that'), new MyOtherDomain(name:'this'), new
MyDomain(value:'thatagain')]
ds*.save()
elasticSearchService.index(ds)

// Index all instances of the specified domain class
elasticSearchService.index(MyDomain)
elasticSearchService.index(class:MyDomain)
elasticSearchService.index(MyDomain, MyOtherDomain)
elasticSearchService.index([MyDomain, MyOtherDomain])
```

4.2. Unindex examples

```
// Unindex all searchable instances
elasticSearchService.unindex()

// Unindex a specific domain instance
MyDomain md = new MyDomain(value:'that')
md.save()
elasticSearchService.unindex(md)

// Unindex a collection of domain instances
def ds = [new MyDomain(value:'that'), new MyOtherDomain(name:'this'), new
MyDomain(value:'thatagain')]
ds*.save()
elasticSearchService.unindex(ds)

// Unindex all instances of the specified domain class
elasticSearchService.unindex(MyDomain)
elasticSearchService.unindex(class:MyDomain)
elasticSearchService.unindex(MyDomain, MyOtherDomain)
elasticSearchService.unindex([MyDomain, MyOtherDomain])
```

Chapter 5. Searching

The plugin provides 2 ways to send search requests.

- You can use the `elasticSearchService` and its public `search` method for cross-domain searching, meaning that ElasticSearch may analyze multiple indices and return hits of different types (=different domains).

```
def res = elasticSearchService.search("${params.query}")
// 'res' search results may contains multiple types of results
```

- You can use the injected dynamic method in the domain for domain-specific searching.

```
def res = Tweet.search("${params.query}")
// 'res' search results contains only Tweet instances
```

These search methods return a `Map` containing 3 entries:

- a `total` entry, representing the total number of hits found
- a `searchResults` entry, containing the hits
- a `scores` entry, containing the hits scores

Example

```
def res = Tweet.search("${params.query}")
println "Found ${res.total} result(s)"
res.searchResults.each {
    println it.message
}

def res = elasticSearchService.search("${params.query}")
println "Found ${res.total} result(s)"
res.searchResults.each {
    if(it instanceof Tweet) {
        println it.message
    } else {
        println it.toString()
    }
}
```

If you're willing to retrieve only the number of hits for a peculiar query, you can use the `countHits()` method. It will only return an `Integer` representing the total hits matching your query.

Example

```
def res = Tweet.countHits("${params.query}")
println "Found ${res} result(s)"

def res = elasticSearchService.countHits("${params.query}", [indices:'test'])
println "Found ${res} result(s)"
```

5.1. Query Strings

The search method injected in the domain or the `ElasticSearchService` has multiple signatures available. You can pass it a simple `String` to compute your search request. That string will be parsed by the **Lucene query parser** so feel free to use its syntax to do more specific search query.

You can find out about the syntax on the Apache Lucene website.

Example

```
def results = elasticSearchService.search("${params.query}")
def resultsTweets = Tweet.search("message:${params.query}")
```

5.2. Query Closure

You can use the Groovy Query DSL to build your search query as a `Closure`.

The format of the search `Closure` follow the same JSON syntax as the ElasticSearch REST API and the Java Query DSL.

Example

```
def result = elasticSearchService.search(searchType:'dfs_query_and_fetch') {
  bool {
      must {
          query_string(query: params.query)
      }
      if (params.firstname) {
          must {
              term(firstname: params.firstname)
          }
      }
  }
}
```

5.3. Query Builder

A `QueryBuilder` can be passed to the search method.

Example

```
QueryBuilder query = QueryBuilders.matchAllQuery()
def result = elasticSearchService.search(query)
```

5.4. Filter Closure

A filter closure can be passed as a second argument after the search closure to the search method.

Example

```
def result = elasticSearchService.search(
    [indices: Building, types: Building, sort: sortBuilder],
    null as Closure,
    {
        geo_distance(
            'distance': '5km',
            'location': [lat: 48.141, lon: 11.57]
        )
    })
```

5.5. Filter Builder

A `FilterBuilder` filter can be passed as a second argument after the search parameter to the search method.

Example

```
FilterBuilder filter = FilterBuilders.rangeFilter("price").gte(1.99).lte(2.3)
def result = elasticSearchService.search(
    [indices: Building, types: Building, sort: sortBuilder],
    null as Closure,
    filter)
```

5.6. Highlighting

The search method support highlighting: automatic wrapping of the matching terms in the search results with HTML/XML/Whatever tags.

You can activate this with a `Closure` containing the highlight settings in the search method `highlight` parameter.

The format of the `Closure` for defining the highlight settings is the same as the ElasticSearch REST API.

Example

```
// Define the pre & post tag that will wrap each term matched in the document.
def highlighter = {
  field 'message'
  field 'tags.name'
  preTags '<strong>'
  postTags '</strong>'
}

def results = Tweet.search("${params.query}", [highlight: highlightSettings])
```

5.6.1. Highlight results

If a search result is found, the `search` method will add a `highlight` entry in the map result.

That entry contains a `List`with every highlighted fragments/fields found for each hit.

```
def results = Tweet.search("${params.query}", [highlight: { field 'message' }])
def highlighted = results.highlight

results?.searchResults?.eachWithIndex { hit, index ->
    // Retrieve the 'message' field fragments for the current hits
    def fragments = highlighted[index].message?.fragments

    // Print the fragment
    println fragments?.size() ? fragments[0] : ''
}
```

5.6.2. Highlighted fields

To determine which fields are to be processed by ElasticSearch, use the `field` setting.

You can call the `field` setting as many time as you want to add any field.

Signature

```
field <fieldName>, <fragmentSize>, <numberOfFragment>
```

Examples

```
def highlightSettings = {

    field 'message'                    // Add the 'message' field in the highlighted
fields list

    field 'tags.name'                  // Add the 'name' field contained in the 'tags'
field of
                                       // the document in the highlighted fields list

    field 'thatAwesomeField', 0, 20    // Add the 'thatAwesomeField' field with
                                       // some values fixed for fragmentSize and
                                       // numberOfFragment parameters
}

def highlightSettings2 = {
    field '_all'                       // Add the special '_all' field in the
highlighted
                                       // fields list
}

def results = Tweet.search("${params.query}", [highlight: highlightSettings])
def results2 = Tweet.search("${params.query}", [highlight: highlightSettings2])
```

5.6.3. Highlighting tags

By default, ElasticSearch will use emphasis tag “…” to wrap the matching text.

You can customize the tags with the `preTags` and `postTags` settings.

```
def highlightSettings = {
    field 'message'
    preTags '<myAweSomeTag>'
    postTags '</myAweSomeTag>'
}
```

5.7. Sorting

To sort the search results, either a field name or a SortBuilder must be passed.

Returned sort values

The sort values are not part of the search results themselves but are part of `result.sort`.

`sort` contains all search values calculated by the ElasticSearch server as a list mapped to the id of the respective domain objects

Example

```
assert [1:[23, 42], 2: [24, 40]] == result.sort
```

5.7.1. Geo Distance Sorting

To sort for geo distances, a SortBuilder must be passed to `search()`

Example

```
def sortBuilder = SortBuilders.geoDistanceSort("location")
    .point(48.141, 11.57)
    .unit(DistanceUnit.KILOMETERS)
    .order(SortOrder.ASC)

def result = elasticSearchService.search(
    [indices: Building, types: Building, sort: sortBuilder],
    null as Closure,
    {
        geo_distance(
            'distance': '5km',
            'location': [lat: 48.141, lon: 11.57]
        )
    })
```

The calculated distances are not part of the search results themselves but are part of `result.sort`. `sort` contains all search values calculated by the ElasticSearch server as a list mapped to the id of the respective domain objects

```
assert [1:[2.34567], 2: [2.4402342]] == result.sort
```

Chapter 6. Admin

The plugin implements a few convenience methods for a few admin-oriented actions.

6.1. Refresh

Explicitly refresh one or more index, making all operations performed since the last refresh available for search. It will also flush the current IndexRequestQueue if there are pending index or delete requests from the application side.

The refresh method is not asynchronous, meaning that it will wait for all operations to complete before resuming the execution of your application.

```
elasticSearchService.index(domain)
// Some code...
// ...

elasticSearchService.index(domain2)
// Some code...
// ...

elasticSearchService.index(domain3)
// Some code...
// ...

elasticSearchAdminService.refresh() // Ensure that the 3 previous index
                                    // requests have been made searchable by ES
```

6.2. Delete Index

Delete an index, all its mapping and its content from the ElasticSearch instance. Be careful when using this command because it cannot be undone.

The generated mapping from the grails plugin is also deleted.

The method can be limited to one or more specific indices or applied to all indices at once (called with no parameter).

```
elasticSearchAdminService.deleteIndex()
```

Chapter 7. Low Level API

If you need to use the Elastic Search client directly, you can use the `elasticSearchHelper` bean that is injected in any services/controllers to get the current instance.

Simply encapsulate your code within a `withElasticSearch` bloc, and you will get a `org.elasticsearch.client.Client` implementation to play with.

```
class MySearchService {
    static transactional = true

    def elasticSearchHelper

    def myMethod(indexName, settings) {
        elasticSearchHelper.withElasticSearch { client ->
            // Do some stuff with the ElasticSearch client
            client.admin()
                .indices()
                .prepareCreate(indexName)
                .setSettings(settings)
                .execute()
                .actionGet()
        }
    }
}
```

Please refers to the Elastic Search API for more information on the methods and properties available on the client.

Chapter 8. Example

8.1. Twitter

8.1.1. The Domains

```
class Tweet {
  static searchable = {
    message boost:2.0
  }

  static belongsTo = [
          user:User
  ]

  static hasMany = [
          tags:Tag
  ]

  static constraints = {
    tags nullable:true, cascade:'save, update'
  }

  String message = ''
  Date dateCreated = new Date()
}
```

```
class User {
  static searchable = {
    except = 'password'
    lastname boost:20
    firstname boost:15, index:'not_analyzed'
    listOfThings index:'no'
    someThings index:'no'
    tweets component:true
  }

  static constraints = {
    tweets cascade:'all'
  }
  static hasMany = [
          tweets:Tweet
  ]
  static mappedBy = [
          tweets:'user'
  ]

  String lastname
  String firstname
  String password
  String activity = 'Evildoer'
  String someThings = 'something'
  ArrayList<String> listOfThings = ['this', 'that', 'andthis']
}
```

```
class Tag {
  static searchable = {
    except=['boostValue']
  }

  String name
  Integer boostValue = 1
}
```

8.1.2. The Controller

- **A action triggering indexation**

`ElasticSearchController` (`testCaseService` is just dealing with GORM instructions):

```
class ElasticSearchController {
  def elasticSearchService
  def testCaseService

  def postTweet = {
    if(!params.user?.id) {
      flash.notice = "No user selected."
      redirect(action: 'index')
      return
    }
    User u = User.get(params.user.id)
    if (!u) {
      flash.notice = "User not found"
      redirect(action: 'index')
      return
    }
    // Create tweet
    testCaseService.addTweet(params.tweet?.message, u, params.tags)

    flash.notice = "Tweet posted"
    redirect(action: 'index')
  }
}
```

With this code (considering that there are already `User` in the database), new Tweets will be indexed automatically, and corresponding `User` indexed documents will be updated since we have set the `tweets` association as component.

- **Searching for Tweets**

```
def searchForUserTweets = {
    def tweets = Tweet.search("${params.message.search}").searchResults
    def tweetsMsg = 'Messages : '
    tweets.each {
      tweetsMsg += "<br />Tweet from ${it.user?.firstname} ${it.user?.lastname} :
${it.message} "
      tweetsMsg += "(tags : ${it.tags?.collect{t -> t.name}})"
    }
    flash.notice = tweetsMsg
    redirect(action: 'index')
}
```

- **Searching for anything**

```
def searchAll = {
    def res = elasticSearchService.search("${params.query}").searchResults
    def resMsg = '<strong>Global search result(s):</strong><br />'
    res.each {
      switch(it){
        case Tag:
          resMsg += "<strong>Tag</strong> ${it.name}<br />"
          break
        case Tweet:
          resMsg += "<strong>Tweet</strong> \"${it.message}\" from
${it.user.firstname} ${it.user.lastname}<br />"
          break
        case User:
          resMsg += "<strong>User</strong> ${it.firstname} ${it.lastname}<br />"
          break
        default:
          resMsg += "<strong>Other</strong> ${it}<br />"
          break
      }

    }
    flash.notice = resMsg
    redirect(action:'index')
 }
```

8.2. Geo Distance Search

A search for buildings with a geo_distance filter, ordered by distance.

8.2.1. Domains

```
class GeoPoint {

    Double lat
    Double lon

    static searchable = {
        root false
    }
}
```

GeoPoint represents the geo coordinates for a building. The field names `lat` and `lon` are mandatory.

```
class Building {

    String name
    GeoPoint location

    static searchable = {
        location geoPoint: true, component: true
    }
}
```

The location of the building is mapped to an ElasticSearch geo_point.

8.2.2. Service Methods

Searching for all buildings sorted by distance with 5km radius around geo location (lat=41.141, lon=11.57)

```
def searchForBuildings() {
    Closure filter = {
        geo_distance(
            'distance': '5km',
            'location': [lat: 48.141, lon: 11.57]
        )
    }

    def sortBuilder = SortBuilders.geoDistanceSort("location").
        point(48.141, 11.57).
        unit(DistanceUnit.KILOMETERS).
        order(SortOrder.ASC)

    def result = elasticSearchService.search(
        [indices: Building, types: Building, sort: sortBuilder],
        null as Closure,
        filter)

    return [results: result.searchResults, distances: result.sort]
}
```

The calculated distances are not part of the search results themselves but are part of `result.sort`. `sort` contains all search values calculated by the ElasticSearch server as a list mapped to the id of the respective domain objects

Example

```
assert [1:[23, 42], 2: [24, 40]] == result.sort
```

8.3. Parent/Child mapping

A store with many departments

```
class Store {

    String name
    String description = "A description of a store"
    String owner = "Shopowner"

    static searchable = true

    static constraints = {
        name blank: false
        description nullable: true
        owner nullable: false
    }
}

class Department {

    String name
    Long numberOfProducts
    Store store

    static constraints = {
        numberOfProducts nullable: true
    }

    static searchable = {
        store parent: true, component:true
    }
}
```

Search for all departments which are childs of a store with the owner "Shopowner"

```
def result = elasticSearchService.search(
    QueryBuilders.hasParentQuery("store", QueryBuilders.matchQuery("owner",
"Shopowner")),
    null as Closure,
    [indices: Department, types: Department]
)
```