Quarkus - Hibernate Search guide

You have a Hibernate ORM-based application? You want to provide a full-featured full-text search to your users? You're at the right place.

With this guide, you'll learn how to synchronize your entities to an Elasticsearch cluster in a heart beat with Hibernate Search. We will also explore how you can can query your Elasticsearch cluster using the Hibernate Search API.

> This technology is considered preview.
>
> In *preview*, backward compatibility and presence in the ecosystem is not guaranteed. Specific improvements might require to change configuration or APIs and plans to become *stable* are under way. Feedback is welcome on our mailing list or as issues in our GitHub issue tracker.
>
> For a full list of possible extension statuses, check our FAQ entry.

> This extension is based on a beta version of Hibernate Search. While APIs are quite stable and the code is of production quality and thoroughly tested, some features are still missing, performance might not be optimal and some APIs or configuration properties might change as the extension matures.

Prerequisites

To complete this guide, you need:

- less than 20 minutes
- an IDE
- JDK 1.8+ installed with `JAVA_HOME` configured appropriately
- Apache Maven 3.5.3+
- Docker
- GraalVM installed if you want to run in native mode

Architecture

The application described in this guide allows to manage a (simple) library: you manage authors and their books.

The entities are stored in a PostgreSQL database and indexed in an Elasticsearch cluster.

Solution

We recommend that you follow the instructions in the next sections and create the application step by step. However, you can go right to the completed example.

Clone the Git repository: `git clone https://github.com/quarkusio/quarkus-quickstarts.git`, or download an archive.

The solution is located in the `hibernate-search-elasticsearch-quickstart` directory.

> ℹ️ The provided solution contains a few additional elements such as tests and testing infrastructure.

Creating the Maven project

First, we need a new project. Create a new project with the following command:

```
mvn io.quarkus:quarkus-maven-plugin:1.3.0.Alpha1:create \
    -DprojectGroupId=org.acme \
    -DprojectArtifactId=hibernate-search-elasticsearch-quickstart \

-DclassName="org.acme.hibernate.search.elasticsearch.LibraryResourc
e" \
    -Dpath="/library" \
    -Dextensions="hibernate-orm-panache, hibernate-search-
elasticsearch, resteasy-jsonb, jdbc-postgresql"
cd hibernate-search-elasticsearch-quickstart
```

This command generates a Maven structure importing the following extensions: * Hibernate ORM with Panache, * the PostgreSQL JDBC driver, * Hibernate Search + Elasticsearch, * RESTEasy and JSON-B.

> ❗ For now, let's delete the two generated tests `LibraryResourceTest` and `NativeLibraryResourceIT` present in `src/test/java`. If you are interested in how you can test this application, just refer to the solution in the quickstarts Git repository: it contains a lot of tests and the required testing infrastructure.

Creating the bare entities

First, let's create our Hibernate ORM entities `Book` and `Author` in the `model` subpackage.

```java
package org.acme.hibernate.search.elasticsearch.model;

import java.util.List;
import java.util.Objects;

import javax.persistence.CascadeType;
import javax.persistence.Entity;
import javax.persistence.FetchType;
import javax.persistence.OneToMany;

import io.quarkus.hibernate.orm.panache.PanacheEntity;

@Entity
public class Author extends PanacheEntity { ①

    public String firstName;

    public String lastName;

    @OneToMany(mappedBy = "author", cascade = CascadeType.ALL,
orphanRemoval = true, fetch = FetchType.EAGER) ②
    public List<Book> books;

    @Override
    public boolean equals(Object o) {
        if (this == o) {
            return true;
        }
        if (!(o instanceof Author)) {
            return false;
        }

        Author other = (Author) o;

        return Objects.equals(id, other.id);
    }

    @Override
    public int hashCode() {
        return 31;
    }
}
```

① We are using Hibernate ORM with Panache, it is not mandatory.

② We are loading these elements eagerly so that they are present in the JSON output. In a real world application, you should probably use a DTO approach.

```java
package org.acme.hibernate.search.elasticsearch.model;

import java.util.Objects;

import javax.json.bind.annotation.JsonbTransient;
import javax.persistence.Entity;
import javax.persistence.ManyToOne;

import io.quarkus.hibernate.orm.panache.PanacheEntity;

@Entity
public class Book extends PanacheEntity {

    public String title;

    @ManyToOne
    @JsonbTransient ①
    public Author author;

    @Override
    public boolean equals(Object o) {
        if (this == o) {
            return true;
        }
        if (!(o instanceof Book)) {
            return false;
        }

        Book other = (Book) o;

        return Objects.equals(id, other.id);
    }

    @Override
    public int hashCode() {
        return 31;
    }
}
```

① We mark this property with @JsonbTransient to avoid infinite loops when serializing with JSON-B.

Initializing the REST service

While everything is not yet set up for our REST service, we can initialize it with the standard CRUD operations we will need.

Just copy this content in the `LibraryResource` file created by the Maven `create-project`

command:

```java
package org.acme.hibernate.search.elasticsearch;

import javax.inject.Inject;
import javax.persistence.EntityManager;
import javax.transaction.Transactional;
import javax.ws.rs.Consumes;
import javax.ws.rs.DELETE;
import javax.ws.rs.POST;
import javax.ws.rs.PUT;
import javax.ws.rs.Path;
import javax.ws.rs.Produces;
import javax.ws.rs.core.MediaType;

import org.acme.hibernate.search.elasticsearch.model.Author;
import org.acme.hibernate.search.elasticsearch.model.Book;
import org.jboss.resteasy.annotations.jaxrs.FormParam;
import org.jboss.resteasy.annotations.jaxrs.PathParam;

@Path("/library")
@Produces(MediaType.APPLICATION_JSON)
@Consumes(MediaType.APPLICATION_JSON)
public class LibraryResource {

    @Inject
    EntityManager em;

    @PUT
    @Path("book")
    @Transactional
    @Consumes(MediaType.APPLICATION_FORM_URLENCODED)
    public void addBook(@FormParam String title, @FormParam Long
authorId) {
        Author author = Author.findById(authorId);
        if (author == null) {
            return;
        }

        Book book = new Book();
        book.title = title;
        book.author = author;
        book.persist();

        author.books.add(book);
        author.persist();
    }

    @DELETE
```

```java
    @Path("book/{id}")
    @Transactional
    public void deleteBook(@PathParam Long id) {
        Book book = Book.findById(id);
        if (book != null) {
            book.author.books.remove(book);
            book.delete();
        }
    }

    @PUT
    @Path("author")
    @Transactional
    @Consumes(MediaType.APPLICATION_FORM_URLENCODED)
    public void addAuthor(@FormParam String firstName, @FormParam
String lastName) {
        Author author = new Author();
        author.firstName = firstName;
        author.lastName = lastName;
        author.persist();
    }

    @POST
    @Path("author/{id}")
    @Transactional
    @Consumes(MediaType.APPLICATION_FORM_URLENCODED)
    public void updateAuthor(@PathParam Long id, @FormParam String
firstName, @FormParam String lastName) {
        Author author = Author.findById(id);
        if (author == null) {
            return;
        }
        author.firstName = firstName;
        author.lastName = lastName;
        author.persist();
    }

    @DELETE
    @Path("author/{id}")
    @Transactional
    public void deleteAuthor(@PathParam Long id) {
        Author author = Author.findById(id);
        if (author != null) {
            author.delete();
        }
    }
}
```

Nothing out of the ordinary here: it is just good old Hibernate ORM with Panache operations in a standard JAX-RS service.

In fact, the interesting part is that we will need to add very few elements to make our full text search application working.

Using Hibernate Search annotations

Let's go back to our entities.

Enabling full text search capabilities for them is as simple as adding a few annotations.

Let's edit the `Author` entity again to include this content:

```java
package org.acme.hibernate.search.elasticsearch.model;

import java.util.Objects;

import javax.json.bind.annotation.JsonbTransient;
import javax.persistence.Entity;
import javax.persistence.ManyToOne;

import org.hibernate.search.mapper.pojo.mapping.definition.annotation.FullTextField;
import org.hibernate.search.mapper.pojo.mapping.definition.annotation.Indexed;

import io.quarkus.hibernate.orm.panache.PanacheEntity;

@Entity
@Indexed ①
public class Book extends PanacheEntity {

    @FullTextField(analyzer = "english") ②
    public String title;

    @ManyToOne
    @JsonbTransient
    public Author author;

    // Preexisting equals()/hashCode() methods
}
```

① First, let's use the `@Indexed` annotation to register our `Book` entity as part of the full text index.

② The `@FullTextField` annotation declares a field in the index specifically tailored for full text search. In particular, we have to define an analyzer to split and analyze the tokens (~ words) -

more on this later.

Now that our books are indexed, we can do the same for the authors.

Open the `Author` class and include the content below.

Things are quite similar here: we use the `@Indexed`, `@FullTextField` and `@KeywordField` annotations.

There are a few differences/additions though. Let's check them out.

```java
package org.acme.hibernate.search.elasticsearch.model;

import java.util.List;
import java.util.Objects;

import javax.persistence.CascadeType;
import javax.persistence.Entity;
import javax.persistence.FetchType;
import javax.persistence.OneToMany;

import org.hibernate.search.engine.backend.types.Sortable;
import org.hibernate.search.mapper.pojo.mapping.definition.annotation.FullTextField;
import org.hibernate.search.mapper.pojo.mapping.definition.annotation.Indexed;
import org.hibernate.search.mapper.pojo.mapping.definition.annotation.IndexedEmbedded;
import org.hibernate.search.mapper.pojo.mapping.definition.annotation.KeywordField;

import io.quarkus.hibernate.orm.panache.PanacheEntity;

@Entity
@Indexed
public class Author extends PanacheEntity {

    @FullTextField(analyzer = "name") ①
    @KeywordField(name = "firstName_sort", sortable = Sortable.YES, normalizer = "sort") ②
    public String firstName;

    @FullTextField(analyzer = "name")
    @KeywordField(name = "lastName_sort", sortable = Sortable.YES, normalizer = "sort")
    public String lastName;

    @OneToMany(mappedBy = "author", cascade = CascadeType.ALL, orphanRemoval = true, fetch = FetchType.EAGER)
    @IndexedEmbedded ③
    public List<Book> books;

    // Preexisting equals()/hashCode() methods
}
```

① We use a `@FullTextField` similar to what we did for Book but you'll notice that the analyzer is different - more on this later.

② As you can see, we can define several fields for the same property. Here, we define a `@KeywordField` with a specific name. The main difference is that a keyword field is not tokenized (the string is kept as one single token) but can be normalized (i.e. filtered) - more on this later. This field is marked as sortable as our intention is to use it for sorting our authors.

③ The purpose of `@IndexedEmbedded` is to include the Book fields into the `Author` index. In this case, we just use the default configuration: all the fields of the associated Book entities are included in the index (i.e. the `title` field). The nice thing with `@IndexedEmbedded` is that it is able to automatically reindex an `Author` if one of its `Book`s has been updated thanks to the bidirectional relation. `@IndexedEmbedded` also supports nested documents (using the `storage = NESTED` attribute) but we don't need it here. You can also specify the fields you want to include in your parent index using the `includePaths` attribute if you don't want them all.

Analyzers and normalizers

Introduction

Analysis is a big part of full text search: it defines how text will be processed when indexing or building search queries.

The role of analyzers is to split the text into tokens (~ words) and filter them (making it all lowercase and removing accents for instance).

Normalizers are a special type of analyzers that keeps the input as a single token. It is especially useful for sorting or indexing keywords.

There are a lot of bundled analyzers but you can also develop your own for your own specific purposes.

You can learn more about the Elasticsearch analysis framework in the Analysis section of the Elasticsearch documentation.

Defining the analyzers used

When we added the Hibernate Search annotations to our entities, we defined the analyzers and normalizers used. Typically:

```
@FullTextField(analyzer = "english")
```

```
@FullTextField(analyzer = "name")
```

```
@KeywordField(name = "lastName_sort", sortable = Sortable.YES,
normalizer = "sort")
```

We use:

- an analyzer called `name` for person names,
- an analyzer called `english` for book titles,
- a normalizer called `sort` for our sort fields

but we haven't set them up yet.

Let's see how you can do it with Hibernate Search.

Setting up the analyzers

It is an easy task, we just need to create an implementation of `ElasticsearchAnalysisConfigurer` (and configure Quarkus to use it, more on that later).

To fulfill our requirements, let's create the following implementation:

```
package org.acme.hibernate.search.elasticsearch.config;

import org.hibernate.search.backend.elasticsearch.analysis.ElasticsearchAnalysisConfigurationContext;
import org.hibernate.search.backend.elasticsearch.analysis.ElasticsearchAnalysisConfigurer;

public class AnalysisConfigurer implements ElasticsearchAnalysisConfigurer {

    @Override
    public void configure(ElasticsearchAnalysisConfigurationContext context) {
        context.analyzer("name").custom() ①
                .tokenizer("standard")
                .tokenFilters("asciifolding", "lowercase");

        context.analyzer("english").custom() ②
                .tokenizer("standard")
                .tokenFilters("asciifolding", "lowercase", "porter_stem");

        context.normalizer("sort").custom() ③
                .tokenFilters("asciifolding", "lowercase");
    }
}
```

① This is a simple analyzer separating the words on spaces, removing any non-ASCII characters by its ASCII counterpart (and thus removing accents) and putting everything in lowercase. It is used in

our examples for the author's names.

② We are a bit more aggressive with this one and we include some stemming: we will be able to search for `mystery` and get a result even if the indexed input contains `mysteries`. It is definitely too aggressive for person names but it is perfect for the book titles.

③ Here is the normalizer used for sorting. Very similar to our first analyzer, except we don't tokenize the words as we want one and only one token.

Adding full text capabilities to our REST service

In our existing `LibraryResource`, we just need to inject the following methods (and add a few `import`s):

```java
    @Transactional ①
    void onStart(@Observes StartupEvent ev) throws
 InterruptedException { ②
        // only reindex if we imported some content
        if (Book.count() > 0) {
            Search.session(em)
                    .massIndexer()
                    .startAndWait();
        }
    }

    @GET
    @Path("author/search") ③
    @Transactional
    public List<Author> searchAuthors(@QueryParam String pattern,
 ④
            @QueryParam Optional<Integer> size) {
        return Search.session(em) ⑤
                .search(Author.class) ⑥
                .predicate(f ->
                    pattern == null || pattern.trim().isEmpty() ?
                            f.matchAll() : ⑦
                            f.simpleQueryString()
                                    .fields("firstName", "lastName",
 "books.title").matching(pattern) ⑧
                    )
                .sort(f -> f.field("lastName_sort").then().field(
 "firstName_sort")) ⑨
                .fetchHits(size.orElse(20)); ⑩
    }
```

① Important point: we need a transactional context for these methods.

② As we will import data into the PostgreSQL database using an SQL script, we need to reindex the data at startup. For this, we use Hibernate Search's mass indexer, which allows to index a lot of data efficiently (you can fine tune it for better performances). All the upcoming updates coming through Hibernate ORM operations will be synchronized automatically to the full text index. If you don't import data manually in the database, you don't need that: the mass indexer should then only be used when you change your indexing configuration (adding a new field, changing an analyzer's configuration…) and you want the new configuration to be applied to your existing entities.

③ This is where the magic begins: just adding the annotations to our entities makes them available for full text search: we can now query the index using the Hibernate Search DSL.

④ Use the `org.jboss.resteasy.annotations.jaxrs.QueryParam` annotation type to avoid repeating the parameter name.

⑤ First, we get an Hibernate Search session from the injected entity manager.

⑥ We indicate that we are searching for `Author`s.

⑦ We create a predicate: if the pattern is empty, we use a `matchAll()` predicate.

⑧ If we have a valid pattern, we create a `simpleQueryString()` predicate on the `firstName`, `lastName` and `books.title` fields matching our pattern.

⑨ We define the sort order of our results. Here we sort by last name, then by first name. Note that we use the specific fields we created for sorting.

⑩ Fetch the `size` top hits, `20` by default. Obviously, paging is also supported.

> Hibernate Search supports a significant part of the Elasticsearch predicates (match, range, nested, phrase, spatial…). Feel free to explore the DSL by using autocompletion.

Configuring the application

As usual, we can configure everything in the Quarkus configuration file, `application.properties`.

Edit `src/main/resources/application.properties` and inject the following configuration:

```
quarkus.ssl.native=false ①

quarkus.datasource.url=jdbc:postgresql:quarkus_test ②
quarkus.datasource.driver=org.postgresql.Driver
quarkus.datasource.username=quarkus_test
quarkus.datasource.password=quarkus_test

quarkus.hibernate-orm.database.generation=drop-and-create ③
quarkus.hibernate-orm.sql-load-script=import.sql ④

quarkus.hibernate-search.elasticsearch.version=7 ⑤
quarkus.hibernate-
search.elasticsearch.analysis.configurer=org.acme.hibernate.search.
elasticsearch.config.AnalysisConfigurer ⑥
quarkus.hibernate-search.elasticsearch.index-
defaults.lifecycle.strategy=drop-and-create ⑦
quarkus.hibernate-search.elasticsearch.index-
defaults.lifecycle.required-status=yellow ⑧
quarkus.hibernate-search.automatic-
indexing.synchronization.strategy=searchable ⑨
```

① We won't use SSL so we disable it to have a more compact native executable.

② Let's create a PostgreSQL datasource.

③ We will drop and recreate the schema every time we start the application.

④ We load some initial data.

⑤ We need to tell Hibernate Search about the version of Elasticsearch we will use. It is important because there are significant differences between Elasticsearch mapping syntax depending on the version. Since the mapping is created at build time to reduce startup time, Hibernate Search cannot connect to the cluster to automatically detect the version.

⑥ We point to the custom `AnalysisConfigurer` which defines the configuration of our analyzers and normalizers.

⑦ Obviously, this is not for production: we drop and recreate the index every time we start the application.

⑧ We consider the `yellow` status is sufficient to connect to the Elasticsearch cluster. This is for testing purposes with the Elasticsearch Docker container. It should not be used in production.

⑨ This means that we wait for the entities to be searchable before considering a write complete. While, on a production setup, the `committed` default might be a suitable value, using `searchable` is especially important when testing as you need the entities to be searchable immediately.

> For more information about the Hibernate Search extension configuration please refer to the Configuration Reference.

Creating a frontend

Now let's add a simple web page to interact with our `LibraryResource`. Quarkus automatically serves static resources located under the `META-INF/resources` directory. In the `src/main/resources/META-INF/resources` directory, overwrite the existing `index.html` file with the content from this index.html file.

Automatic import script

For the purpose of this demonstration, let's import an initial dataset.

Let's create a `src/main/resources/import.sql` file with the following content:

```sql
INSERT INTO author(id, firstname, lastname) VALUES (nextval('hibernate_sequence'), 'John', 'Irving')
INSERT INTO author(id, firstname, lastname) VALUES (nextval('hibernate_sequence'), 'Paul', 'Auster')

INSERT INTO book(id, title, author_id) VALUES (nextval('hibernate_sequence'), 'The World According to Garp', 1);
INSERT INTO book(id, title, author_id) VALUES (nextval('hibernate_sequence'), 'The Hotel New Hampshire', 1);
INSERT INTO book(id, title, author_id) VALUES (nextval('hibernate_sequence'), 'The Cider House Rules', 1);
INSERT INTO book(id, title, author_id) VALUES (nextval('hibernate_sequence'), 'A Prayer for Owen Meany', 1);
INSERT INTO book(id, title, author_id) VALUES (nextval('hibernate_sequence'), 'Last Night in Twisted River', 1);
INSERT INTO book(id, title, author_id) VALUES (nextval('hibernate_sequence'), 'In One Person', 1);
INSERT INTO book(id, title, author_id) VALUES (nextval('hibernate_sequence'), 'Avenue of Mysteries', 1);
INSERT INTO book(id, title, author_id) VALUES (nextval('hibernate_sequence'), 'The New York Trilogy', 2);
INSERT INTO book(id, title, author_id) VALUES (nextval('hibernate_sequence'), 'Mr. Vertigo', 2);
INSERT INTO book(id, title, author_id) VALUES (nextval('hibernate_sequence'), 'The Brooklyn Follies', 2);
INSERT INTO book(id, title, author_id) VALUES (nextval('hibernate_sequence'), 'Invisible', 2);
INSERT INTO book(id, title, author_id) VALUES (nextval('hibernate_sequence'), 'Sunset Park', 2);
INSERT INTO book(id, title, author_id) VALUES (nextval('hibernate_sequence'), '4 3 2 1', 2);
```

Preparing the infrastructure

We need a PostgreSQL instance and an Elasticsearch cluster.

Let's use Docker to start one of each:

```
docker run -it --rm=true --name elasticsearch_quarkus_test -p
9200:9200 -p 9300:9300 -e "discovery.type=single-node"
docker.elastic.co/elasticsearch/elasticsearch:7.5.0
```

```
docker run --ulimit memlock=-1:-1 -it --rm=true --memory
-swappiness=0 --name postgresql_quarkus_test -e
POSTGRES_USER=quarkus_test -e POSTGRES_PASSWORD=quarkus_test -e
POSTGRES_DB=quarkus_test -p 5432:5432 postgres:11.3
```

Time to play with your application

You can now interact with your REST service:

- start Quarkus with `./mvnw compile quarkus:dev`

- open a browser to `http://localhost:8080/`

- search for authors or book titles (we initialized some data for you)

- create new authors and books and search for them too

As you can see, all your updates are automatically synchronized to the Elasticsearch cluster.

Building a native executable

You can build a native executable with the usual command `./mvnw package -Pnative`.

> ℹ️ As usual with native executable compilation, this operation consumes a lot of memory.
>
> It might be safer to stop the two containers while you are building the native executable and start them again once you are done.

Running it is as simple as executing `./target/hibernate-search-elasticsearch-quickstart-1.0-SNAPSHOT-runner`.

You can then point your browser to `http://localhost:8080/` and use your application.

> The startup is a bit slower than usual: it is mostly due to us dropping and recreating the database schema and the Elasticsearch mapping every time at startup. We also inject some data and execute the mass indexer.
>
> In a real life application, it is obviously something you won't do at startup.

Further reading

If you are interested into learning more about Hibernate Search 6, the Hibernate team has some documentation in the works.

It is still work in progress but covers all main concepts and features, so it can guide you in your exploration.

FAQ

Why Hibernate Search 6 (and not a fully supported version)?

To optimize the Hibernate Search bootstrap for Quarkus, the Hibernate team had to reorganize things a bit (collect the metadata offline, start the Elasticsearch client later…).

This couldn't be done in the 5.x code base so we decided to go with the in-progress Hibernate Search 6.

Can I really use it?

While Hibernate Search 6 is still at Beta stage, the code is of production quality and can be relied on.

What we don't guarantee is that there might be API changes along the way to the final release of Hibernate Search 6 and you might have to adapt your code.

If it is not a major issue for you, then sure you can use it.

Why Elasticsearch only?

Hibernate Search supports both a Lucene backend and an Elasticsearch backend.

In the context of Quarkus and to build microservices, we thought the latter would make more sense. Thus we focused our efforts on it.

We don't have plans to support the Lucene backend in Quarkus for now.

Hibernate Search Configuration Reference

🗄 Configuration property fixed at build time - ⚙ Configuration property overridable at runtime

Configuration property	Type	Default
🗄 `quarkus.hibernate-search.background-failure-handler` The class or the name of the bean that should be notified of any failure occurring in a background process (mainly index operations). Must implement `org.hibernate.search.engine.reporting.FailureHandler`.	class name	
⚙ `quarkus.hibernate-search.query.loading.cache-lookup.strategy` The strategy to use when loading entities during the execution of a search query.	`skip`, `persistence-context`, `persistence-context-then-second-level-cache`	`skip`
⚙ `quarkus.hibernate-search.query.loading.fetch-size` The fetch size to use when loading entities during the execution of a search query.	int	`100`
⚙ `quarkus.hibernate-search.automatic-indexing.synchronization.strategy` The synchronization strategy to use when indexing automatically. Defines the status for which you wait before considering the operation completed by Hibernate Search. Use `queued` or `committed` in production environments. `searchable` is useful in integration tests.	`queued`, `committed`, `searchable`	`committed`
⚙ `quarkus.hibernate-search.automatic-indexing.enable-dirty-check` Whether to check if dirty properties are relevant to indexing before actually reindexing an entity. When enabled, re-indexing of an entity is skipped if the only changes are on properties that are not used when indexing.	boolean	`true`
Default backend	Type	Default

🗄 `quarkus.hibernate-search.elasticsearch.version` The version of Elasticsearch used in the cluster. As the schema is generated without a connection to the server, this item is mandatory. It doesn't have to be the exact version (it can be 7 or 7.1 for instance) but it has to be sufficiently precise to choose a model dialect (the one used to generate the schema) compatible with the protocol dialect (the one used to communicate with Elasticsearch). There's no rule of thumb here as it depends on the schema incompatibilities introduced by Elasticsearch versions. In any case, if there is a problem, you will have an error when Hibernate Search tries to connect to the cluster.	ElasticsearchVersion	
🗄 `quarkus.hibernate-search.elasticsearch.analysis.configurer` The class or the name of the bean used to configure full text analysis (e.g. analyzers, normalizers).	class name	
⚙ `quarkus.hibernate-search.elasticsearch.hosts` The list of hosts of the Elasticsearch servers.	list of string	`localhost:9200`
⚙ `quarkus.hibernate-search.elasticsearch.protocol` The protocol to use when contacting Elasticsearch servers. Set to "https" to enable SSL/TLS.	`http`, `https`	`http`
⚙ `quarkus.hibernate-search.elasticsearch.username` The username used for authentication.	string	
⚙ `quarkus.hibernate-search.elasticsearch.password` The password used for authentication.	string	
⚙ `quarkus.hibernate-search.elasticsearch.connection-timeout` The connection timeout.	Duration ❓	`3S`
⚙ `quarkus.hibernate-search.elasticsearch.max-connections` The maximum number of connections to all the Elasticsearch servers.	int	`20`

⚙ `quarkus.hibernate-search.elasticsearch.max-connections-per-route` The maximum number of connections per Elasticsearch server.	int	`10`
⚙ `quarkus.hibernate-search.elasticsearch.discovery.enabled` Defines if automatic discovery is enabled.	boolean	`false`
⚙ `quarkus.hibernate-search.elasticsearch.discovery.refresh-interval` Refresh interval of the node list.	Duration ❓	`10S`
⚙ `quarkus.hibernate-search.elasticsearch.index-defaults.lifecycle.strategy` The strategy used for index lifecycle.	`none`, `validate`, `update`, `create`, `drop-and-create`, `drop-and-create-and-drop`	`create`
⚙ `quarkus.hibernate-search.elasticsearch.index-defaults.lifecycle.required-status` The minimal cluster status required.	`green`, `yellow`, `red`	`green`
⚙ `quarkus.hibernate-search.elasticsearch.index-defaults.lifecycle.required-status-wait-timeout` How long we should wait for the status before failing the bootstrap.	Duration ❓	`10S`

⚙ quarkus.hibernate-search.elasticsearch.indexes."index-name".lifecycle.strategy The strategy used for index lifecycle.	none, validate, update, create, drop-and-create, drop-and-create-and-drop	create
⚙ quarkus.hibernate-search.elasticsearch.indexes."index-name".lifecycle.required-status The minimal cluster status required.	green, yellow, red	green
⚙ quarkus.hibernate-search.elasticsearch.indexes."index-name".lifecycle.required-status-wait-timeout How long we should wait for the status before failing the bootstrap.	Duration ❓	10S
Additional backends	**Type**	**Default**
🗄 quarkus.hibernate-search.elasticsearch.default-backend Only useful when defining `backends additional backends`: the name of the default backend, i.e. the backend that will be assigned to `@Indexed` entities that do not specify a backend explicitly through `@Indexed(backend = …)`.	string	
🗄 quarkus.hibernate-search.elasticsearch.backends."backend-name".version The version of Elasticsearch used in the cluster. As the schema is generated without a connection to the server, this item is mandatory. It doesn't have to be the exact version (it can be 7 or 7.1 for instance) but it has to be sufficiently precise to choose a model dialect (the one used to generate the schema) compatible with the protocol dialect (the one used to communicate with Elasticsearch). There's no rule of thumb here as it depends on the schema incompatibilities introduced by Elasticsearch versions. In any case, if there is a problem, you will have an error when Hibernate Search tries to connect to the cluster.	ElasticsearchVersion	

▤ `quarkus.hibernate-search.elasticsearch.backends."backend-name".analysis.configurer` The class or the name of the bean used to configure full text analysis (e.g. analyzers, normalizers).	class name		
⚙ `quarkus.hibernate-search.elasticsearch.backends."backend-name".hosts` The list of hosts of the Elasticsearch servers.	list of string	`localhost:9200`	
⚙ `quarkus.hibernate-search.elasticsearch.backends."backend-name".protocol` The protocol to use when contacting Elasticsearch servers. Set to "https" to enable SSL/TLS.	`http`, `https`	`http`	
⚙ `quarkus.hibernate-search.elasticsearch.backends."backend-name".username` The username used for authentication.	string		
⚙ `quarkus.hibernate-search.elasticsearch.backends."backend-name".password` The password used for authentication.	string		
⚙ `quarkus.hibernate-search.elasticsearch.backends."backend-name".connection-timeout` The connection timeout.	Duration ❓	`3S`	
⚙ `quarkus.hibernate-search.elasticsearch.backends."backend-name".max-connections` The maximum number of connections to all the Elasticsearch servers.	int	`20`	
⚙ `quarkus.hibernate-search.elasticsearch.backends."backend-name".max-connections-per-route` The maximum number of connections per Elasticsearch server.	int	`10`	

⚙ `quarkus.hibernate-search.elasticsearch.backends."backend-name".discovery.enabled` Defines if automatic discovery is enabled.		boolean	`false`
⚙ `quarkus.hibernate-search.elasticsearch.backends."backend-name".discovery.refresh-interval` Refresh interval of the node list.		Duration ❓	`10S`
⚙ `quarkus.hibernate-search.elasticsearch.backends."backend-name".index-defaults.lifecycle.strategy` The strategy used for index lifecycle.		`none`, `validate`, `update`, `create`, `drop-and-create`, `drop-and-create-and-drop`	`create`
⚙ `quarkus.hibernate-search.elasticsearch.backends."backend-name".index-defaults.lifecycle.required-status` The minimal cluster status required.		`green`, `yellow`, `red`	`green`
⚙ `quarkus.hibernate-search.elasticsearch.backends."backend-name".index-defaults.lifecycle.required-status-wait-timeout` How long we should wait for the status before failing the bootstrap.		Duration ❓	`10S`

⚙ `quarkus.hibernate-search.elasticsearch.backends."backend-name".indexes."index-name".lifecycle.strategy` The strategy used for index lifecycle.	`none`, `validate`, `update`, `create`, `drop-and-create`, `drop-and-create-and-drop`	`create`
⚙ `quarkus.hibernate-search.elasticsearch.backends."backend-name".indexes."index-name".lifecycle.required-status` The minimal cluster status required.	`green`, `yellow`, `red`	`green`
⚙ `quarkus.hibernate-search.elasticsearch.backends."backend-name".indexes."index-name".lifecycle.required-status-wait-timeout` How long we should wait for the status before failing the bootstrap.	Duration ❓	`10S`

> ℹ️ *About the Duration format*
>
> The format for durations uses the standard `java.time.Duration` format. You can learn more about it in the Duration#parse() javadoc.
>
> You can also provide duration values starting with a number. In this case, if the value consists only of a number, the converter treats the value as seconds. Otherwise, PT is implicitly prepended to the value to obtain a standard `java.time.Duration` format.