# Résumé

Cet article décrit une stratégie permettant d'utiliser Elasticsearch comme support de stockage principal au travers d'un cas d'utilisation concret.

Vous pouvez retrouver toutes les sources du projet décrit dans cet article sur GitHub (https://github.com/biconou/SubsonicES).

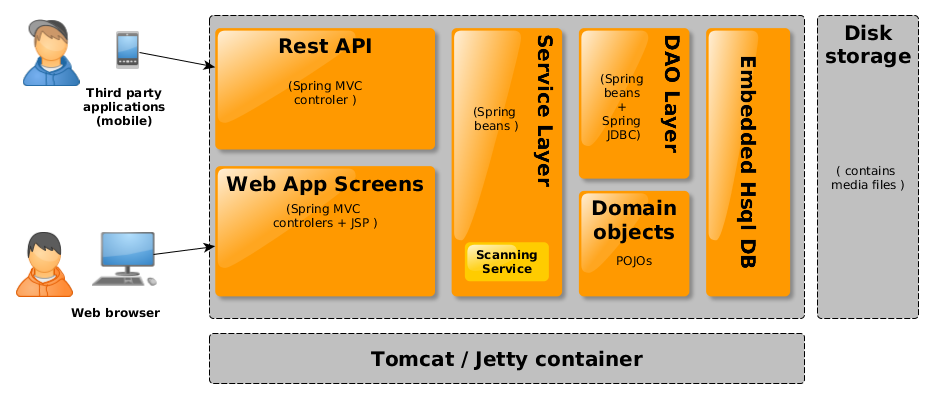
# Introduction

Subsonic (http://www.subsonic.org) est un serveur multimédia développé en Java. Il vous permet de construire une librairie musicale à partir de fichiers audio (et vidéo) stockés sur un disque dur et d'écouter la musique en streaming sur différents périphériques.

*Subsonic était open source jusqu'en mai 2016 lorsque son développeur a décidé de fermer les sources. Si le projet vous intéresse, je vous conseille plutôt de vous diriger vers Libresonic (https://github.com/Libresonic), un fork open source de Subsonic dont je suis un fervent contributeur.*

La librairie musicale est matérialisée classiquement dans une base de données relationnelle.

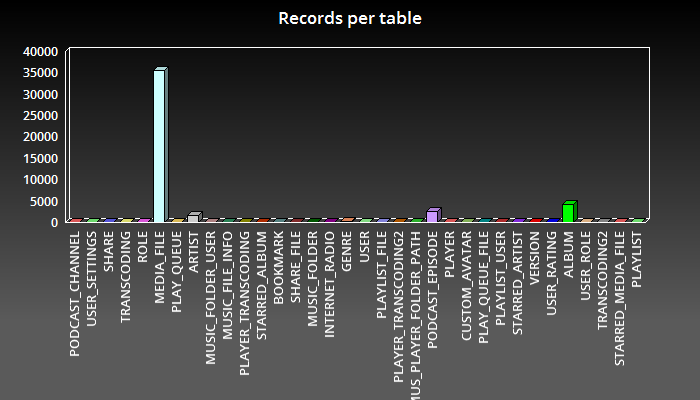
Voici une vue de l'architecture globale de Subsonic.

La base de données est une simple instance HSQLDB embarquée dans l'application web. 

Subsonic dispose d'un service spécial pour alimenter la base à partie des fichiers musicaux présents sur disque. c'est le « scanning service ». Il parcourt le système de fichier à la recherche de fichiers musicaux et met à jour les descriptions de documents musicaux dans la base de données.

Toutes les données significatives sont contenues dans quelques tables seulement (MEDIA\_FILE, ALBUM, ARTIST et GENRE). Les autres tables servent à des fin d'administration

Si on regarde la distribution du nombre d'enregistrements par table (ci-dessous), on se rend compte que pratiquement toute la donnée est contenue dans la table MEDIA\_FILE.



Il y a très peu de logique relationnel dans cette base de donnée.

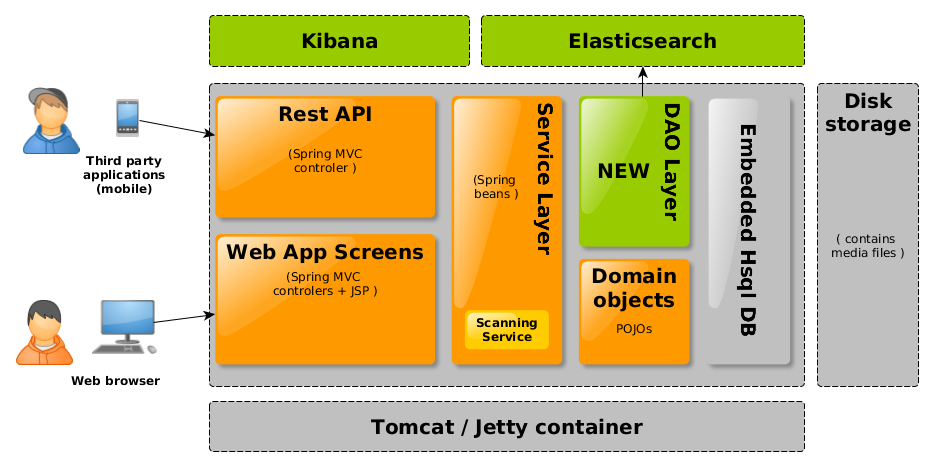
En outre, Subsonic utilise un index Lucene pour ses fonctions de recherche.

Voilà ce qu'on peut appeler un système orienté documents. Ce n'est pas surprenant puisqu'un fichier musical est effectivement un document.

Il m'a donc semblé intéressant d'essayer de remplacer le système de stockage relationnel par un système plus nativement orienté documents. Dans une telle application, la fonction de recherche est très importante.

Comme par ailleurs je m'intéressait de près à Lucene, j'ai voulu tenter l'expérience de remplacer la base HSQLDB par un service Elasticsearch.

L'objectif est donc d'arriver à l'architecture suivante :

La base de données embarquée est progressivement supprimée (elle sera peut être conservée pour les données d'administration).

Par ailleurs, on remarque que l'ajout de Kibana doit nous permettre d'avoir une nouvelle vue sur les données métiers.

# La stratégie

Comme vous l'avez certainement remarqué, Subsonic dispose d'une couche DAO. En théorie, il doit être possible de changer de gestionnaire de stockage en réécrivant simplement la couche DAO; tant que le contrat de la couche DAO reste inchangé, on doit pouvoir remplacer la base de données relationnelle par Elasticsearch sans avoir à toucher à la couche services.

Partant de là, la principale préoccupation est de trouver une structure d'index et une bonne méthode d'interrogation pour reproduire les comportements attendus par la couche DAO.

## Deux problèmes épineux

Est si simple de réécrire la couche DAO ? globalement oui mais j'ai tout de même été confronté à deux problèmes délicats.

### Le problème des identifiants

Subsonic utilise des entiers comme identifiants d'objets. Par ailleurs Elasticsearch génère des id de document qui ne sont pas numériques.

Mais il est impossible de changer le type des id sans casser le contrat général de l'API Rest Subsonic.

J'ai finalement opté pour un système à deux id. Un id est généré par Elasticsearch lors de l'indexation d'un document puis un second id numérique est créé en calculant le hascode du chemin d'accès au fichier audio sur disque.

### Le problème des transactions

Elasticsearch n'est pas un système de gestion de base de données. L'indexation d'un nouveau document n'est pas transactionnelle puisque qu'elle s'effectue de façon asynchrone. En effet, lorsqu'on indexe un nouveau document, ce dernier est visible au bout d'un certain laps de temps (très court bien sûr mais indéterminé).

Cela n'est pas spécialement problématique dans un cas d'utilisation tel que Subsonic ; en effet les documents sont ajouté par le scanning service et cela ne met pas en jeu de transaction à proprement parler.

Le problème vient du fait que, dans Subsonic, le scanner service est développé de telle manière que chaque document ajouté dans la base de données est immédiatement relu pour effectuer différents traitements. J'ai donc été amené à reconcevoir différemment le processus de scan.

## Design des index

Comme vu plus haut, les données « business » de Subsonic sont contenues dans très peu de tables. La plus importante est MEDIA\_FILE ; elle contient chaque document audio de la bibliothèque. A côté, les tables telles que ARTIST, ALBUM ou GENRE créent des relations entre les notions d'artiste, d'album de genre et les documents audio eux même.

Dans l'optique de stocker toutes les données dans Elastisearch, il n'est pas question de créer un index pour chacune de ces tables ; ce serait un contresens.

Au lieu de cela, on créer un index unique contenant tous les documents audio. La structure très souple des index Elasticsearch permet d'enregistrer les notions d'artiste, album, genre en tant qu'attribut de chaque document.

Ainsi, une collection de musiques est un ensemble de documents tout simplement.

Notons au passage que notre index Elasticsearch remplace à la fois la base de données relationnelle et l'index Lucene créé par Subsonic pour ses recherches.

Dans Subsonic, on peut déclarer plusieurs dossiers de musique. Chaque dossier est une partie de la librairie musicale, ce qui permet d'attribuer des droits distincts à différents utilisateurs.

J'ai décidé de créer un index distinct pour chaque dossier musical, ce qui est très pratique pour implémenter les droits d'accès des utilisateurs puisqu'Elasticsearch permet de choisir facilement sur quels indexes doit porter une requête de recherche.

# Réécriture de la couche DAO

La couche DAO est composée de beans métier et de beans Spring qui implémentent les opérations de lecture et écriture sur la base de données.

L'objectif est de réécrire les beans Spring de façon à attaquer un index Elasticsearch à la place de la base HSQLDB pour chaque opération élémentaire (CRUD). Pour ce faire, j'ai utilisé une combinaison de deux techniques : le client java Elasticsearch (transport client) et le query DSL basé sur json.

Les beans métier eux même ne sont pas modifiés sauf pour ce qui est de l'ajout d'un identifiant Elasticsearch.

## Une classe utilitaire DAO

La classe ElasticSearchDaoHelper se charge de créer les index et fournit des utilitaires pour interroger Elasticsearch.

Au démarrage de l'application, si aucun index n'existe, la classe ElasticSearchDaoHelper va créer explicitement un index pour chaque dossier musical définit dans Subsonic (comme l'illustre le code ci-dessous).

for (String indexName : indexNames) {

boolean indexExists = elasticSearchClient.admin().indices().prepareExists(indexName)

.execute().actionGet().isExists();

if (!indexExists) {

elasticSearchClient.admin().indices()

.prepareCreate(indexName)

.addMapping(MEDIA\_FILE\_INDEX\_TYPE,

"path", "type=string,index=not\_analyzed",

"parentPath", "type=string,index=not\_analyzed",

"mediaType", "type=string,index=not\_analyzed",

"folder", "type=string,index=not\_analyzed",

"format", "type=string,index=not\_analyzed",

"genre", "type=string,index=not\_analyzed",

"artist", "type=string,index=not\_analyzed",

"albumArtist", "type=string,index=not\_analyzed",

"albumName", "type=string,index=not\_analyzed",

"name", "type=string,index=not\_analyzed",

"coverArtPath", "type=string,index=not\_analyzed",

"created", "type=date",

"changed", "type=date",

"childrenLastUpdated", "type=date",

"lastPlayed", "type=date",

"lastScanned", "type=date",

"starredDate", "type=date")

.execute().actionGet();

}

}

Chaque champ déclaré dans le mapping correspond à une propriété de l'objet métier MediaFile. Vous pouvez noter que la plupart de ces champs sont « not\_analyzed » ; Ceci est important pour que ces champs puissent servir à des recherche selon des valeurs exactes (il faut se souvenir que dans ce contexte, Elasticsearch est utilisé comme une sorte de base de données).

Ensuite on trouve les utilitaires.

### Ajout de fichiers audio dans l'index

Des objets de type MediaFile sont ajoutés (ou mis à jour) dans Elasticsearch par le scanning service de Subsonic. Cela consiste à créer à chaque fois un nouveau document dans l'index.

C'est le travail de la méthode indexObject ci-dessous.

public void indexObject(SubsonicESDomainObject obj, String indexName) {

try {

// Convert the object to a json string representation.

String mediaFileAsJson = getMapper().writeValueAsString(obj);

IndexResponse indexResponse = getClient().prepareIndex(

indexName,

ElasticSearchDaoHelper.MEDIA\_FILE\_INDEX\_TYPE)

.setSource(mediaFileAsJson).setVersionType(VersionType.INTERNAL).get();

}

} catch (JsonProcessingException e) {

throw new RuntimeException("Error trying indexing object " + e);

}

}

Ici, le paramètre indexName est le nom de l'index correspondant au dossier musical dans lequel le document doit être ajouté.

L'objet métier de type SubsonicESDomainObject (classe mère de tous les objets du domaine) devant être ajouté à l'index est préalablement sérialisé en json à l'aide de la librairie jackson (getMapper fait référence à la classe com.fasterxml.jackson.databind.ObjectMapper)

### Recherche de documents dans l'index

La classe ElasticSearchDaoHelper contient un ensemble de méthodes utilitaires qui facilitent le requêtage sur Elasticsearch (je me suis un peu inspiré pour cela du framework Spring-jdbc).

Pour exécuter une requête, le client java Elasticsearch est utilisé mais chaque requête est spécifiée sous forme json et est placée dans un fichier de resource à part du code java. Chaque fichier de ressource contenant une requête est en fait un template FreeMarker dans lequel on place des variables correspondant aux paramètres de la requête.

Par exemple, voici le contenu du fichier searchMediaFileByPath.flt qui contient la requête de recherche par identifiant d'un document audio.

{

"constant\_score" : {

"filter" : {

"bool" : {

"must" : [

{"term" : {"path" : "${path}"}},

{"type" : { "value" : "MEDIA\_FILE" }}

]

}

}

}

}

Et voici la méthode java utiliraire permet d'exécuter une recherche d'ojet par clé primaire.

public <T extends SubsonicESDomainObject> T extractUnique(String queryName, Map<String, String> vars, Class<T> type) {

String jsonQuery;

try {

jsonQuery = getQuery(queryName,vars);

} catch (IOException | TemplateException e) {

throw new RuntimeException(e);

}

SearchRequestBuilder searchRequestBuilder = getClient().prepareSearch(indexNames())

.setQuery(jsonQuery).setVersion(true);

SearchResponse response = searchRequestBuilder.execute().actionGet();

long totalHits = response == null ? 0 : response.getHits().totalHits();

if (totalHits == 0) {

return null;

} else if (totalHits > 1) {

throw new RuntimeException("Document is not unique "+type.getName()+" "+vars);

} else {

return convertFromHit(response.getHits().getHits()[0],type);

}

}

La méthode convertFromHit se charge de matérialiser un objet métier à partir du champ \_source du document trouvé.

private <T extends SubsonicESDomainObject> T convertFromHit(SearchHit hit, Class<T> type) throws RuntimeException {

T object = null;

if (hit != null) {

String hitSource = hit.getSourceAsString();

try {

object = getMapper().readValue(hitSource,type);

object.setESId(hit.id());

object.setVersion((int)hit.getVersion());

} catch (IOException e) {

throw new RuntimeException("Error while reading MediaFile object from index. ", e);

}

}

return object;

}

Voici enfin un ensemble de méthodes dont le but est d’extraire un ensemble d'objets à partir d'une liste de documents retrouvés depuis une requête Elasticsearch.

/\*\*

\* Extract objects from indices of certain music folders using a named query.

\*

\* @param queryName The name of the query to use. It refers to the name of a .ftl file that

\* contains the query expressed in json.

\* @param vars A map of parameters and their value to bind to the query (can be null if the

\* query has no parameter.

\* @param from The offset index of the first object to extract from the query results.

\* @param size The number of objects to extract.

\* @param sortClause A sort clause to apply to the query.

\* @param musicFolders The list of music folders and thus indices to consider for the query.

\* @param type The exact type of object being extracted.

\* @param <T> The exact type of object being extracted.

\* @return A list of all extracted objects. may be empty. The size is equal to size or less.

\*/

public <T extends SubsonicESDomainObject> List<T> extractObjects(String queryName, Map<String, String> vars,

Integer from, Integer size,

Map<String, SortOrder> sortClause,

List<MusicFolder> musicFolders, Class<T> type) {

String jsonQuery;

try {

jsonQuery = getQuery(queryName, vars);

} catch (IOException | TemplateException e) {

throw new RuntimeException(e);

}

return extractObjects(jsonQuery, from, size, sortClause, musicFolders, type);

}

/\*\*

\* Extract objects from indices of certain music folders using a json query.

\*

\* @param jsonSearch A query expressed in json.

\* @param from The offset index of the first object to extract from the query results.

\* @param size The number of objects to extract.

\* @param sortClause A sort clause to apply to the query.

\* @param musicFolders The list of music folders and thus indices to consider for the query.

\* @param type The exact type of object being extracted.

\* @param <T> The exact type of object being extracted.

\* @return A list of all extracted objects. may be empty. The size is equal to size or less.

\*/

public <T extends SubsonicESDomainObject> List<T> extractObjects(String jsonSearch, Integer from, Integer size,

Map<String, SortOrder> sortClause,

List<MusicFolder> musicFolders, Class<T> type) {

SearchRequestBuilder searchRequestBuilder = getClient().prepareSearch(indexNames(musicFolders))

.setQuery(jsonSearch).setVersion(true);

return extractObjects(searchRequestBuilder, from, size, sortClause, type);

}

/\*\*

\* Extract objects from indices of certain music folders using a searchRequestBuilder.

\*

\* @param searchRequestBuilder A searchRequestBuilder that represents a query.

\* @param from The offset index of the first object to extract from the query results.

\* @param size The number of objects to extract.

\* @param sortClause A sort clause to apply to the query.

\* @param type The exact type of object being extracted.

\* @param <T> The exact type of object being extracted.

\* @return A list of all extracted objects. may be empty. The size is equal to size or less.

\*/

public <T extends SubsonicESDomainObject> List<T> extractObjects(SearchRequestBuilder searchRequestBuilder,

Integer from, Integer size,

Map<String, SortOrder> sortClause, Class<T> type) {

if (from != null) {

searchRequestBuilder.setFrom(from);

}

if (size != null) {

searchRequestBuilder.setSize(size);

}

if (sortClause != null) {

sortClause.keySet().forEach(sortField -> searchRequestBuilder.addSort(sortField, sortClause.get(sortField)));

}

SearchResponse response = searchRequestBuilder.execute().actionGet();

List<T> returnedSongs = Arrays.stream(response.getHits().getHits()).map(hit -> convertFromHit(hit, type)).collect(Collectors.toList());

return returnedSongs;

}

A chaque fois, le paramètre mediaFolders précise dans quels dossiers musicaux on cherche les documents ; on utilise donc ici des requête multi-indexes.

## Un exemple de bean DAO : MediaFileDao

La classe MediaFileDao est le principal bean DAO car l'objet MediaFile est l'objet pivot du domaine métier Subsonic. Voici à titre d'exemple deux méthodes de ce DAO qui utilisent l'utilitaire décrit au chapitre précédent :

### Recherche d'un fichier audio à partir de sont unique chemin d'accès sur disque

/\*\*

\* Retrieve a MediaFile identified by a path.

\*

\* @param path The path.

\* @return

\*/

@Override

public MediaFile getMediaFile(String path) {

Map<String,String> vars = new HashMap<>();

vars.put("path",path);

return elasticSearchDaoHelper.extractUnique("searchMediaFileByPath",vars,MediaFile.class);

}

La requête Elasticsearch correapondante («  searchMediaFileByPath ») se présente ainsi :

{

"constant\_score" : {

"filter" : {

"bool" : {

"must" : [

{"term" : {"path" : "${path}"}},

{"type" : { "value" : "MEDIA\_FILE" }}

]

}

}

}

}

### Recherche de toutes les chansons d'un même album

@Override

public List<MediaFile> getSongsForAlbum(String artist, String album) {

Map<String,String> vars = new HashMap<>();

vars.put("artist",artist);

vars.put("album",album);

return elasticSearchDaoHelper.extractObjects("getSongsForAlbum",vars,MediaFile.class);

}

La requête correspondante est :

{

"constant\_score" : {

"filter" : {

"bool" : {

"must" : [

{"term" : {"albumArtist" : "${artist}"}},

{"term" : {"albumName" : "${album}"}},

{"type" : { "value" : "MEDIA\_FILE" }}

],

"should" : [

{"term" : {"mediaType" : "MUSIC"}},

{"term" : {"mediaType" : "AUDIOBOOK"}},

{"term" : {"mediaType" : "PODCAST"}}

]

}

}

}

}

# Jouons avec les données

Nous avons maintenant tout ce qu'il faut pour construire un index de documents musicaux.Nous pouvons utiliser les formidables facultés de recherche d'Elasticsearch pour exploiter nos données.

Au début de l'article, j'ai mentionné le fait que Subsonic possède des tables permettant de répertorier les albums, les artistes et les genres musicaux. Maintenant ces information font partie intégrante des propriétés de nos documents musicaux indexés.

Seulement il ne s'agit pas de recherche des documents un par un mais nous devons aussi réaliser des recherches groupées.

L'API Subsonic doit pouvoir répondre à des questions telles que :

* Quels sont les différents genres musicaux de ma collection ? Combien y a-t-il de chansons et d'album pour chaque genre ?
* Qui sont les différents artistes et combien ont-ils d'albums dans ma collection ?

Pour répondre à ces questions, nous pouvons utiliser le framework d'aggrégation d'Elasticsearch.

Par exemple, la méthode MediaFileDao.getGenres retourne une liste ordonnée de tous les genres musicaux de la librairie. L'objet Genre contient par ailleurs le nombre de chansons de d'albums concernés.

public List<Genre> getGenres(boolean sortByAlbum) {

List<Genre> genres = new ArrayList();

SearchResponse genresResponse = elasticSearchDaoHelper.getClient().prepareSearch()

.setQuery(QueryBuilders.typeQuery("MEDIA\_FILE"))

.addAggregation(AggregationBuilders.terms("genre\_agg").field("genre")

.subAggregation(AggregationBuilders.terms("mediaType\_agg").field("mediaType"))).setSize(0).get();

StringTerms genreAgg = genresResponse.getAggregations().get("genre\_agg");

for (Terms.Bucket genreEntry : genreAgg.getBuckets()) {

Genre genre = new Genre(genreEntry.getKeyAsString());

StringTerms mediaTypeAgg = genreEntry.getAggregations().get("mediaType\_agg");

for (Terms.Bucket mediaTypeEntry : mediaTypeAgg.getBuckets()) {

if ("ALBUM".equals(mediaTypeEntry.getKeyAsString())) {

genre.setAlbumCount((int) mediaTypeEntry.getDocCount());

}

if ("MUSIC".equals(mediaTypeEntry.getKeyAsString())) {

genre.setSongCount((int) mediaTypeEntry.getDocCount());

}

}

genres.add(genre);

}

// Sort the list.

if (sortByAlbum) {

genres.sort((o1, o2) -> {

if (o1.getAlbumCount() > o2.getAlbumCount()) {

return -1;

}

if (o1.getAlbumCount() < o2.getAlbumCount()) {

return 1;

}

return 0;

});

} else {

genres.sort((o1, o2) -> {

if (o1.getSongCount() > o2.getSongCount()) {

return -1;

}

if (o1.getSongCount() < o2.getSongCount()) {

return 1;

}

return 0;

});

}

return genres;

}

De la même manière, nous utilisons les aggregations pour la méthode ArtistDao.getAlphabetialArtists qui construit la liste ordonnée de tous les artistes.

public List<Artist> getAlphabetialArtists(int offset, int count, List<MusicFolder> musicFolders) {

List<Artist> artists = new ArrayList<>();

SearchResponse genresResponse = elasticSearchDaoHelper.getClient().prepareSearch(elasticSearchDaoHelper.indexNames(musicFolders))

.setQuery(QueryBuilders.typeQuery("MEDIA\_FILE"))

.addAggregation(AggregationBuilders.terms("artist\_agg").field("artist").order(Terms.Order.term(true))

.subAggregation(AggregationBuilders.terms("mediaType\_agg").field("mediaType"))).setSize(0).get();

StringTerms artistAgg = genresResponse.getAggregations().get("artist\_agg");

for (Terms.Bucket entry : artistAgg.getBuckets()) {

String artistName = entry.getKeyAsString();

Artist artist = new Artist();

artists.add(artist);

artist.setName(artistName);

StringTerms mediaTypeAgg = entry.getAggregations().get("mediaType\_agg");

for (Terms.Bucket mediaTypeEntry : mediaTypeAgg.getBuckets()) {

String mediaTypeKey = mediaTypeEntry.getKeyAsString();

if ("ALBUM".equals(mediaTypeKey)) {

artist.setAlbumCount((int)mediaTypeEntry.getDocCount());

}

}

}

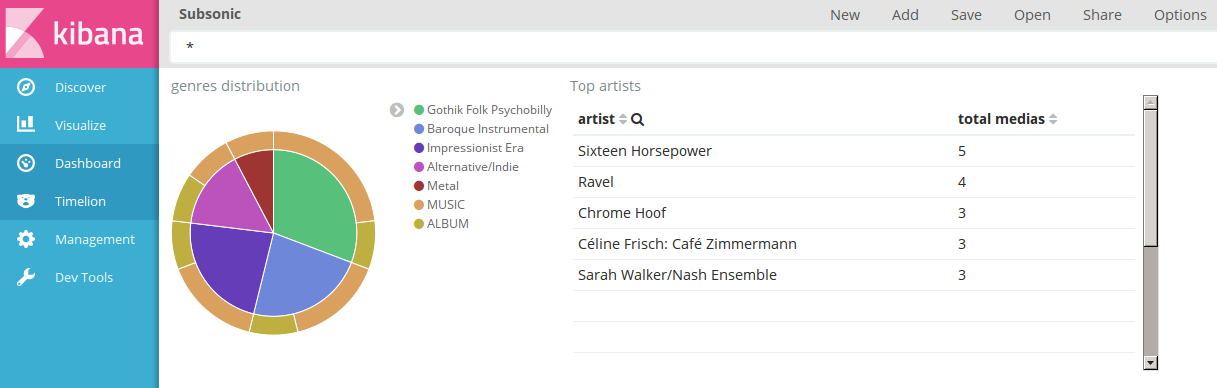
return artists;

}

# Aller plus loin avec Kibana

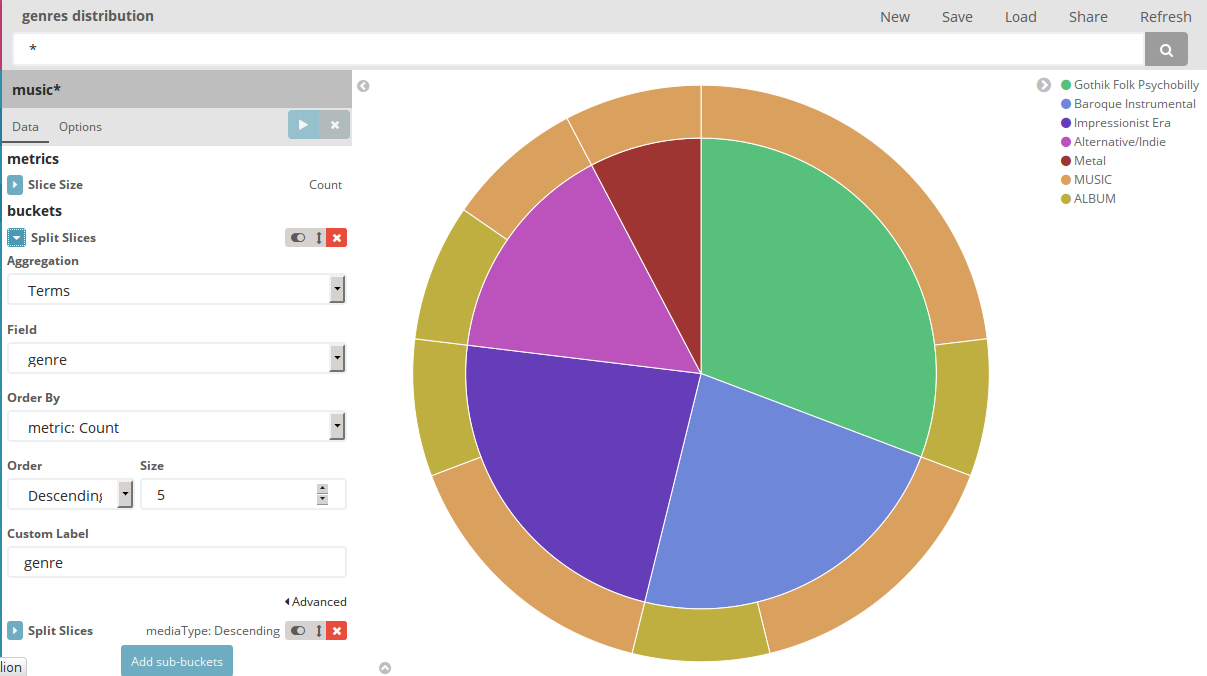
Un des apports indéniables d'Elasticsearch est l'utilisation de Kibana pour interroger facilement les données.

Par exemple, cela ne prend que quelques minutes pour créer un joli dashboard to visulaliser la distribution des genres musicaux et la liste des top artistes de la librairie.



Les visualisations sont créées avec la même logique d'agrégation que celle utilisée dans la couche DAO.

Ci-dessous voici l'écran de création de la visu des genres musicaux. On peut reconnaître ici les agrégations genre\_agg/mediaType\_agg que l'on a utilisées dans la méthode MediaFileDao.getGenres. Chaque ensemble (bucket) de l'agrégation est représentée par une part du camembert.



Il est également très agréable de naviguer parmi les documents indexés en appliquant quelques filtres à la souris.

