|  |
| --- |
|  |
| Rapport - Sujet B3 : Requêtes et visualisation de données issues de Dbpedia |
|  |
|  |
| **Peter Serge** |
| **3/6/2015** |

Formation MAS-ICT CAS : WEB SEMANTIQUE

Julien Tscherrig, Omar Abou Khaled et Maria Sokhn

|  |
| --- |
| Rapport de réalisation d’une application de mise en relation de données issues de dbpedia. |

|  |
| --- |
|  |

Table des matières

[1. Introduction 3](#_Toc413418896)

[1.1 Rappel sur les objectifs 3](#_Toc413418897)

[2. Analyse 4](#_Toc413418898)

[2.1 Liste des artistes (Query 1.1) 4](#_Toc413418899)

[2.2 Détail sur un artiste (Query 1.2) 5](#_Toc413418900)

[2.3 Recherche d’artistes ou bands liés (Query 1.3) 6](#_Toc413418901)

[2.4 Recherche des genres (Query 1.4) 7](#_Toc413418902)

[3. Architecture technique 8](#_Toc413418903)

[3.1 Composants 8](#_Toc413418904)

[3.2 Dbpedia et sparql 8](#_Toc413418905)

[3.3 Jena 8](#_Toc413418906)

[3.4 Spring 9](#_Toc413418907)

[3.5 Junit & Spring test 9](#_Toc413418908)

[3.6 AngularJS 9](#_Toc413418909)

[3.7 Maven 9](#_Toc413418910)

[4. Mise en route 10](#_Toc413418911)

[4.1 Installation de Java 10](#_Toc413418912)

[4.2 Installation de Maven 10](#_Toc413418913)

[4.3 Installation de Eclipse 10](#_Toc413418914)

[4.4 Installation de Tomcat 11](#_Toc413418915)

[5. Réalisation 14](#_Toc413418916)

[5.1 Backend 14](#_Toc413418917)

[5.2 Maven Web Application 14](#_Toc413418918)

[5.3 Organisation des packages 16](#_Toc413418919)

[5.4 Diagramme de classe simplifié 17](#_Toc413418920)

[5.5 Beans 18](#_Toc413418921)

[5.6 Services 19](#_Toc413418922)

[5.7 Controller REST 22](#_Toc413418923)

[5.8 Tests 23](#_Toc413418924)

[5.9 Frontend 25](#_Toc413418925)

[5.10 Bootstrapping et Templating 25](#_Toc413418926)

[5.11 Construction de la vue 27](#_Toc413418927)

[5.12 Vues de détails 29](#_Toc413418928)

[5.13 Code JavaScript 30](#_Toc413418929)

[6. Exécution de l’application 32](#_Toc413418930)

[6.1 Directement dans Eclipse 32](#_Toc413418931)

[6.2 En mode ligne de commande avec Maven 33](#_Toc413418932)

[6.3 Sur le site Openshift 33](#_Toc413418933)

[7. Conclusion 34](#_Toc413418934)

[8. Références 35](#_Toc413418935)

# Introduction

Ce document est un rapport pour la réalisation d’un mini projet ; dans le cadre de la formation MAS-ICT CAS : WEB SEMANTIQUE Julien Tscherrig, Omar Abou Khaled et Maria Sokhn.

Mon application est composée de deux parties principales, le backend, application web java permettant d’opérer comme fournisseur de données et cachant la complexité des requêtes sparql ; et un frontend écrit en JavaScript avec l’aide du Framework structurant AngularJS. Ces deux principales composantes seront détaillées plus tard dans ce document.

## Rappel sur les objectifs

Créer une application Web en Java permettant de visualiser des groupes musicaux, membres et albums.

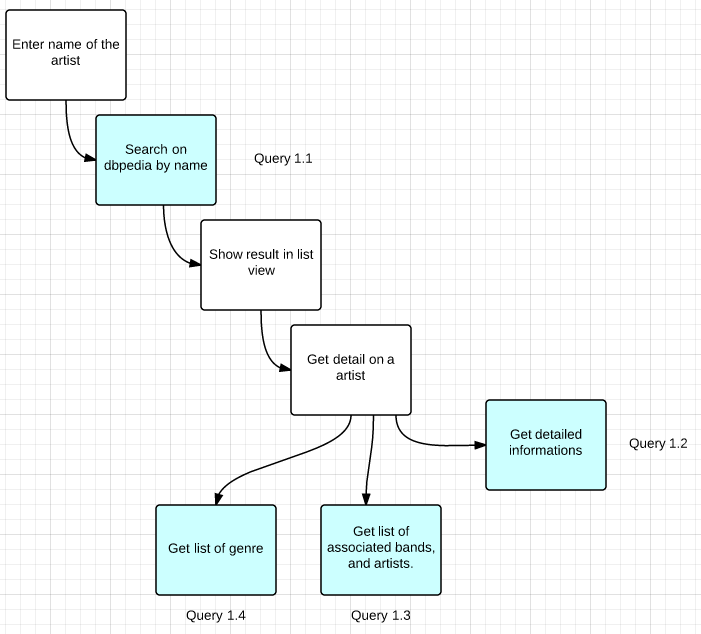
En partant d’un nom d’artiste, il faudra faire la liste des groupes musicaux liés, albums réalisés ainsi que des informations liées comme le style.

Il serait intéressant de pouvoir mettre en relation deux artistes par rapport aux informations trouvées (groupes, albums, style musical, …).

# Analyse

L’application obtiendra ces données via des requêtes sparql, faites directement sur les sources de données de dbpedia.org.

Le workflow principal correspond à ceci :



L’application commence par une recherche sur la source de données <http://dbpedia.org/ontology/MusicalArtist> et affichera une liste d’artiste (Query 1.1). On pourra ensuite afficher des détails supplémentaires dans un écran dédié, afficher la liste de genres (Query 1.4) associés ainsi que les artistes (Query 1.2) et groupes liés (Query 1.3).

## Liste des artistes (Query 1.1)

Nous voulons dans un premier temps obtenir la liste des artistes par leur nom (ou fragment).

prefix prop: <http://dbpedia.org/property/>

prefix owl: <http://dbpedia.org/ontology/>

prefix res: <http://dbpedia.org/resource/>

prefix rdfs: [http://www.w3.org/2000/01/rdf-schema#](http://www.w3.org/2000/01/rdf-schema)

select distinct ?artist ?name

where {

?artist a owl:MusicalArtist .

?artist rdfs:label ?name .

FILTER (regex(?name, 'Mark Knopfler', 'i')

&& langMatches(lang(?name), 'en'))

} LIMIT 100

A noter que notre source de données est : <http://dbpedia.org/ontology/MusicalArtist> qui est une sous-classe de <http://dbpedia.org/ontology/Artist>



Dans cette requête nous filtrons les résultats qui sont en anglais en utilisant la fonction FILTER

FILTER (langMatches(lang(?name), 'en'))

La sélection des données se fait aussi en utilisant FILTER avec l’option i pour ne pas tenir compte de la case:

FILTER (regex(?name, 'Mark Knopfler', 'i'))

## Détail sur un artiste (Query 1.2)

Pour ce faire il faut utiliser l’URI de la ressource dans la sélection des données comme ceci et la propriété sameAs

?artist owl:sameAs? <http://dbpedia.org/resource/Mark\_Knopfler> .

Attention aussi aux champs ***null***, qui ici sont traités avec la fonction OPTIONAL afin de sortir les données partielles de la requête.

OPTIONAL { ?artist owl:birthPlace ?birthPlace }

Ce qui fait une requête comme ceci:

prefix prop: <http://dbpedia.org/property/>

prefix owl: <http://dbpedia.org/ontology/>

prefix res: <http://dbpedia.org/resource/>

prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#>

select \*

where {

?artist owl:sameAs? <http://dbpedia.org/resource/Mark\_Knopfler> .

?artist rdfs:label ?name .

OPTIONAL { ?artist owl:birthDate ?birthDate }

OPTIONAL { ?artist owl:birthPlace ?birthPlace }

OPTIONAL { ?artist owl:thumbnail ?image}

OPTIONAL { ?artist prop:shortDescription ?shortDescription }

OPTIONAL { ?artist owl:abstract ?abstract }

OPTIONAL { ?artist prop:website ?website }

OPTIONAL { ?artist prop:yearsActive ?yearsActive }

FILTER (langMatches(lang(?name), 'en')

&& langMatches(lang(?abstract), 'en'))

} LIMIT 1

## Recherche d’artistes ou bands liés (Query 1.3)

Là aussi la propriété sameAs permet de faire la sélection au niveau de la ressource. On procèdera de la même manière pour les propriétés fournissant une liste de valeurs come owl :associatedBand, owl :associatedArtist.

prefix prop: <http://dbpedia.org/property/>

prefix owl: <http://dbpedia.org/ontology/>

prefix res: <http://dbpedia.org/resource/>

prefix rdfs: [http://www.w3.org/2000/01/rdf-schema#](http://www.w3.org/2000/01/rdf-schema)

select distinct \* where { ?artist owl:sameAs? http://dbpedia.org/resource/Mark\_Knopfler> .

?artist rdfs:label ?name .

?artist owl:associatedBand ?associatedBand.

?associatedBand rdfs:label ?assName .

FILTER (langMatches(lang(?name), 'en')

&& langMatches(lang(?assName), 'en'))

} LIMIT 100

## Recherche des genres (Query 1.4)

Voici un exemple de requête, similaire au point précédent en utilisant la propriété owl:genre.

prefix prop: <http://dbpedia.org/property/>

prefix owl: <http://dbpedia.org/ontology/>

prefix res: <http://dbpedia.org/resource/>

prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#>

select \*

where {

?artist owl:sameAs? <http://dbpedia.org/resource/Mark\_Knopfler> .

?artist rdfs:label ?name .

OPTIONAL { ?artist owl:genre ?genre}

OPTIONAL { ?genre rdfs:label ?genrelabel}

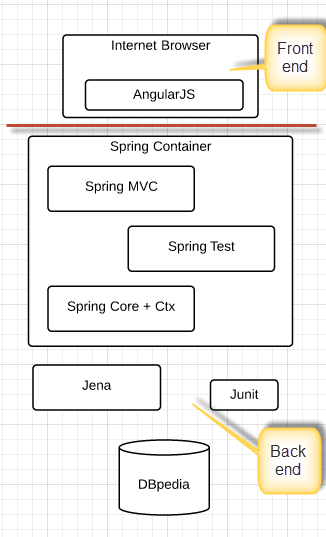
FILTER (langMatches(lang(?name), 'en')

&& langMatches(lang(?genrelabel), 'en') )

} LIMIT 10

# Architecture technique

Nous utiliserons ici Spring pour créer une application Web permettant de mettre en relation les diverses couches. Spring MVC nous permet de construire une couche service et de l’exposer pour qu’elle puisse être utilisée du côté front.



## Composants

Voici une brève présentation des éléments composants notre architecture logicielle.

### Dbpedia et sparql

Dbpedia fera office ici de source de données RDF. Nous utilisons des requêtes sparql pour requêter les données fournies dans <http://dbpedia.org>.

L’outil disponible sur internet <http://dbpedia.org/sparql> permet de mettre au point ces requêtes avant d’en faire l’intégration dans du code Java.

### Jena

Jena est une librairie permettant d’exploiter des données issues de sources de données sémantique.

Cette librairie fait pas mal de chose comme par exemple exécuter des commandes sparql à l’image de ce qui se fait sur des bases de données traditionnelles.

Nous trouvons toutes les informations sur le site : <https://jena.apache.org>.

Nous utiliserons Jena pour exécuter des requêtes sparql et récupérer les données résultantes dans ResultSet java.

### Spring

Spring est un Framework très populaire dans le monde des applications d’entreprise Java JEE. Le but du Framework est de simplifier au maximum la mise en oeuvre pour éviter de perdre trop de temps sur des aspects liés à la configuration ou au code dit technique.

Nous utiliserons Spring MVC pour nous permettre d’exposer des services REST et de fournir une vue Web à notre application.

Le site de Spring <http://projects.spring.io/spring-framework/> fournit toute sorte d’information sur ce Framework.

### Junit & Spring test

Junit est utiliser pour réaliser des tests en plus de Spring test, qui nous permet de charger les services, contexte Spring et configuration.

### AngularJS

AngularJS est le Framework utilisé pour réaliser la partie front ou présentation de l’application. Nous utilisons aussi Twitter Bootstrap afin de disposer de toute sorte de composant graphique dans le rendu de l’application.

### Maven

Tous les composants sont définit dans le fichier pom.xml de l’application et permet d’être obtenu via ce Framework dynamiquement.

# Mise en route

Les outils nécessaires pour effectuer ces développements sont :

|  |  |  |
| --- | --- | --- |
| Composants | Version | Location |
| Java JDK | 1.7+ | <http://www.oracle.com/technetwork/java/javase/downloads/index.html> |
| Maven | 3.0 et plus | <http://maven.apache.org/download.cgi> |
| Tomcat | 8 | <http://tomcat.apache.org/download-80.cgi> |
| Eclipse | Luna | <https://www.eclipse.org/downloads/> |

## Installation de Java

Java JDK est disponible ici :

<http://www.oracle.com/technetwork/java/javase/downloads/index.html>

Une fois le logiciel téléchargé faite l’installation. Ceci ne pose pas de difficulté.

A noter que nous avons besoin du JDK et que le JRE ne suffit pas.

## Installation de Maven

Maven est disponible ici :

<http://maven.apache.org/download.cgi>

Une fois le package [apache-maven-3.\*.\*-bin.zip](http://mirror.switch.ch/mirror/apache/dist/maven/maven-3/3.2.5/binaries/apache-maven-3.2.5-bin.zip) downloadé, décompressez dans un répertoire local (par exemple c:\apps).

Vous devez ensuite :

1. Créer une variable d’environnement M2\_HOME correspondant au répertoire d’installation de Maven.
2. Créer une variable M2 correspondant a %M2\_HOME%\bin
3. Ajouter M2 au PATH.

Il faut aussi créer un fichier settings, contenant les chemins locaux de repositories, si existant ; des informations d’utilisation de proxy ou autre.

Ce fichier doit être placé ici. <HOME>/.m2/settings.xml.

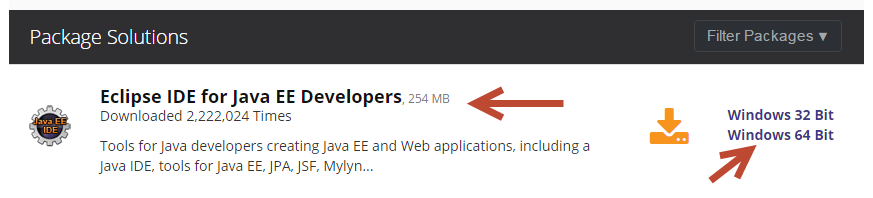
Plus d’info sont disponible ici : <http://maven.apache.org/settings.html>.

## Installation de Eclipse

Eclipse Luna est disponible ici :

<https://www.eclipse.org/downloads>

Il faut ensuite faire attention de prendre la version EE :



Une fois le package téléchargé, décompresser et exécuter le fichier eclipse.exe.

## Installation de Tomcat

Tomcat est disponible ici :

<http://tomcat.apache.org/download-80.cgi>

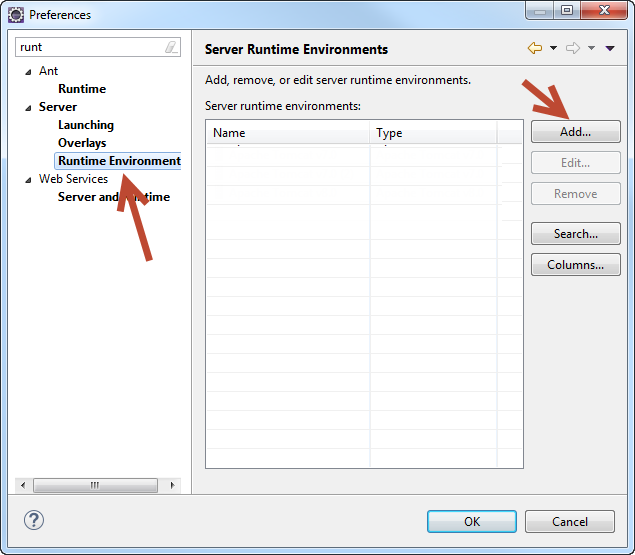
Une fois le package téléchargé, il faut le décompresser en local (par exemple sur c:\apps\tomcat8).

Nous utiliserons Eclipse pour les opérations de start/stop ainsi que le déploiement de l’application.

Pour ce faire il faut créer un serveur Tomcat8 dans Eclipse comme ceci :

**Ajouter un runtime Tomcat** :

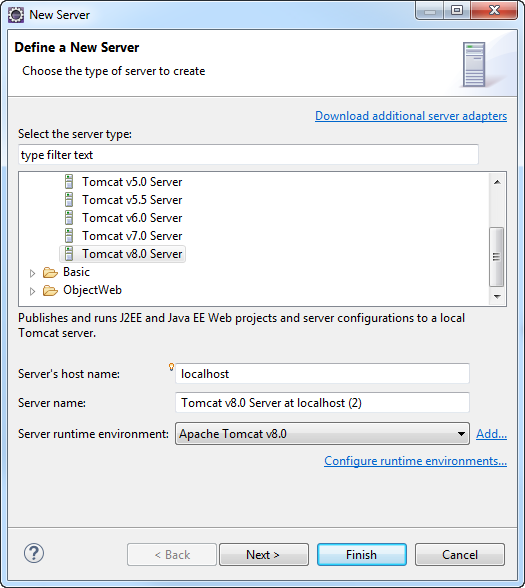
* Dans l’onglet Préférences de Eclipse, rechercher : runtime ; clicker sur Add…



* Définissez le répertoire Tomcat de travail, ainsi qu’un JRE, qui doit être préalablement définit dans Eclipse, ou le runtime par défaut.

**Ajouter un serveur Tomcat :**

* Dans l’onglet « Servers », visible dans la perspective JEE, créez un nouveau serveur basé sur le runtime Tomcat 8 :



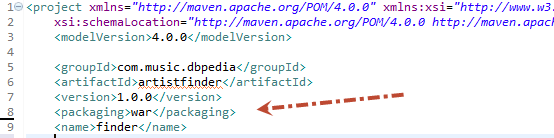
# Réalisation

## Backend

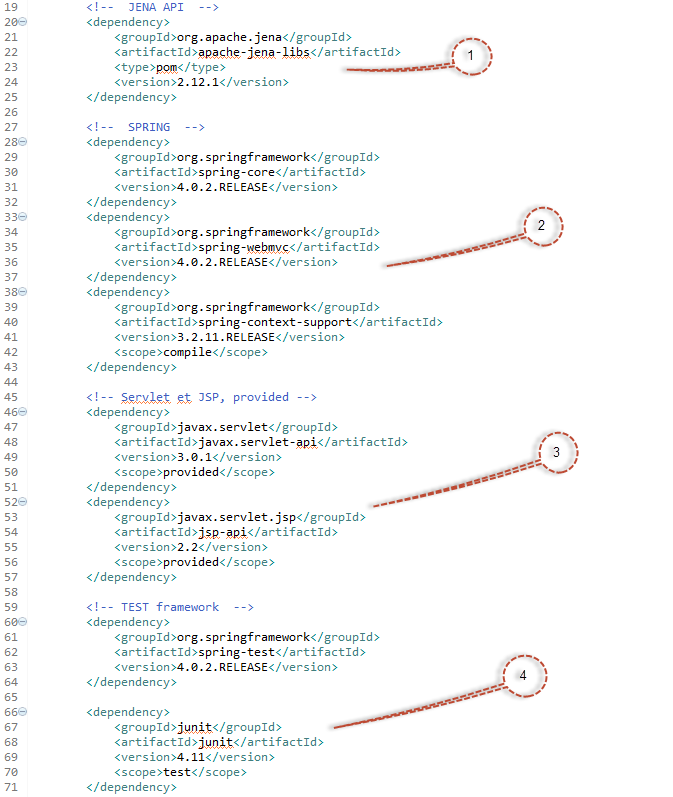
### Maven Web Application

Nous commençons le projet par la création d’une application web Maven (basée sur l’archétype maven-archetype-webapp ; de toute façons le contenu sera adapté à notre besoin.

Pour commencer nous nommons notre application dont le packaging sera war



Voici les dépendances nécessaires :



1. Apache Jena (de type pom), nous permet d’obtenir d’un coup les librairies nécessaires à la librairie,
2. Les librairies Spring, en version 4.0.2. Spring Mvc nous permettra de créer les services REST,
3. Les classes servlets utile à l’IDE Eclispe en provided,
4. Les librairies de test Junit et Spring Test.

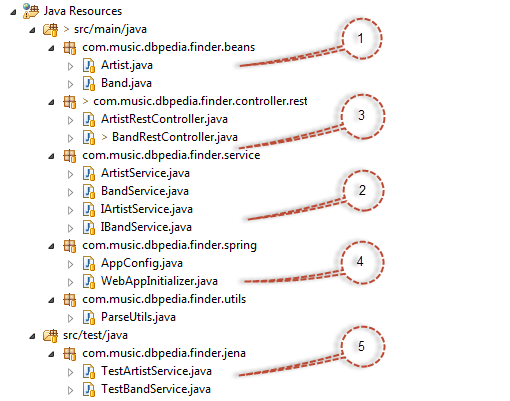
Nous définissons aussi quelques plugins Maven dont le plugin Tomcat afin de facilité l’exécution du projet en dehors d’Eclipse.



1. Compilation plugin en java 1.7
2. Plugin de déploiement,
3. Plugin Tomcat7, permettant d’exécuter l’application via un ***mvn tomcat7:run***,
4. Gestion des ressources,
5. Plugin de gestion de dépendance
6. Définition des fichiers ressources à packager.

### Organisation des packages

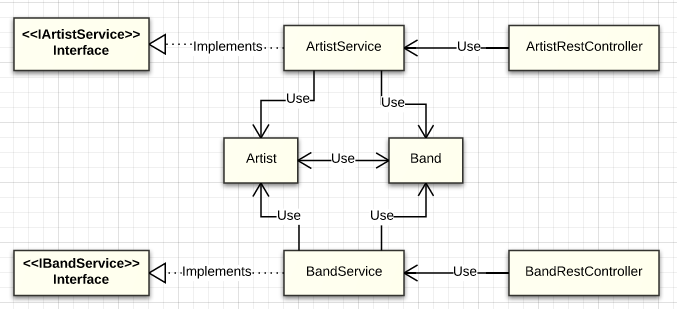
Notre application est très simple et ne comporte pas beaucoup de composant :



1. Beans contient deux structures utilisées pour stocker les informations relatives au deux principales vues,
2. Service contient les classes qui exécutent les requêtes sparsql,
3. Les contrôleurs REST sont les classes qui font l’exposition des services REST, interface entre le backend et le frontend,
4. Les classes application et configuration Spring (on oublie les fichiers de contexte pour faire la configuration par annotation),
5. Les classes de tests.

### Diagramme de classe simplifié

Voici une vue simplifiée de l’application backend :

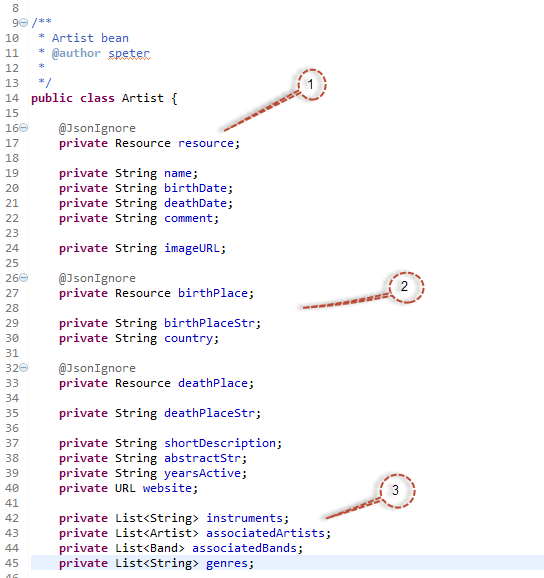


Voici les différents types de classe de manière plus détaillée.

### Beans

Les Beans sont des classes simples permettant de stocker les données issues de manière plus organisée.

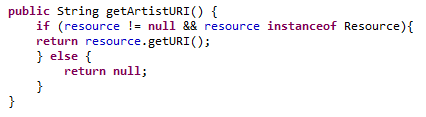
En voici un exemple :



On peut noter les points suivants :

1. Champs de type Resource, permettant de stocker une ressource au sens sémantique du terme, @JSonIgnore empêche cet objet d’être passé au frontend.
2. Champs de tout type (String, Date, URL).
3. Listes de valeurs (de tout type).

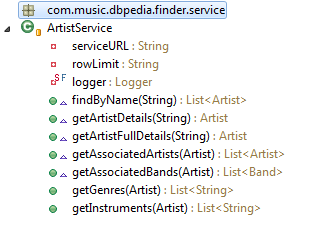
Nous fournirons à la partie front les URI correspondant aux ressources comme ceci :



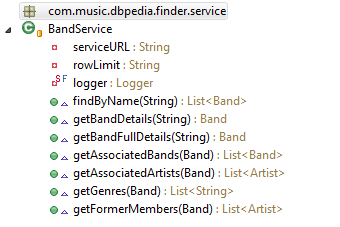
### Services

Les services implémentent les interfaces correspondantes. C’est cette couche qui exécute les commandes sparql.

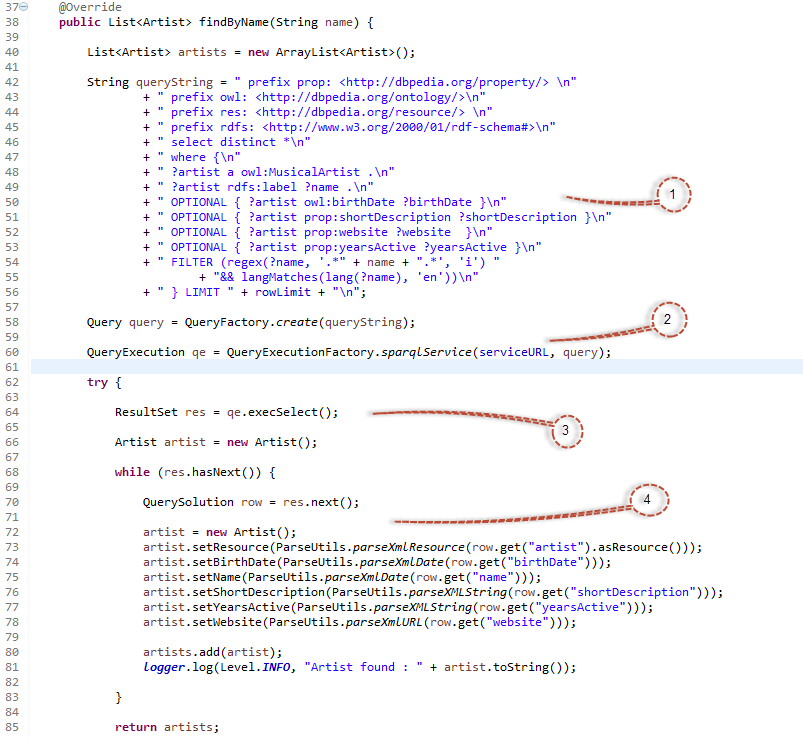
Par exemple pour un artiste, voici les méthodes implémentées qui permette de fournir les informations à la vue :



Et pour un groupe



Les commandes sont exécutées comme ceci :



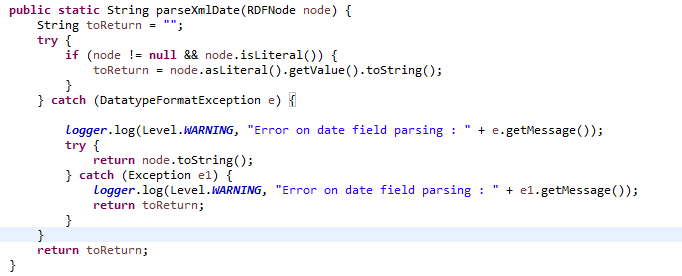
1. Mise en forme d’une requête (sous la forme d’une chaine de caractère) permettant ici de rechercher un artiste,
2. Connexion au service de données : <http://dbpedia.org/sparql>,
3. Exécution de la requête,
4. Récupération des données sous forme d’un ResultSet.

Pour des raisons de performance j’ai séparé la méthode getArtistDetails en deux pour éviter de multiplier les requêtes successives sparql. Il serait judicieux de le faire encore plus, afin de rendre l’obtention des données plus fluide.

Voici comment les requêtes d’obtention des détails sont faites, par exemple pour un artiste :



1. La construction de la requête,
2. Attention au filtrage sur les langues, afin d’éviter les doublons,
3. Exécution de la requête,
4. Obtention des données issues du resultset,
5. J’ai fait des fonctions utilitaires de filtrage, car les données reçues ne sont pas toujours correctement typée et parfois même complétement fausse :

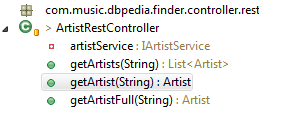


(ceci est une version un peu basique qui mériterait d’être complétée.)

### Controller REST

Les contrôleurs fournissent la donnée sous forme de service REST au frontend AngularJS.

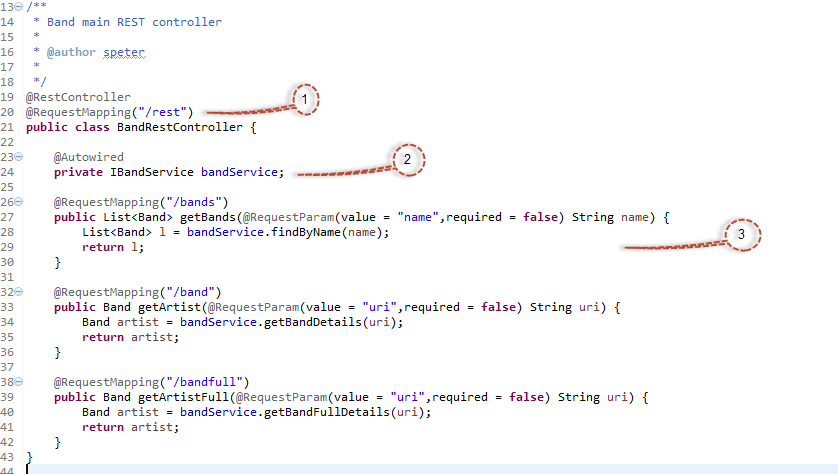
Nous avons ceci pour un artiste :



Et pour un band :



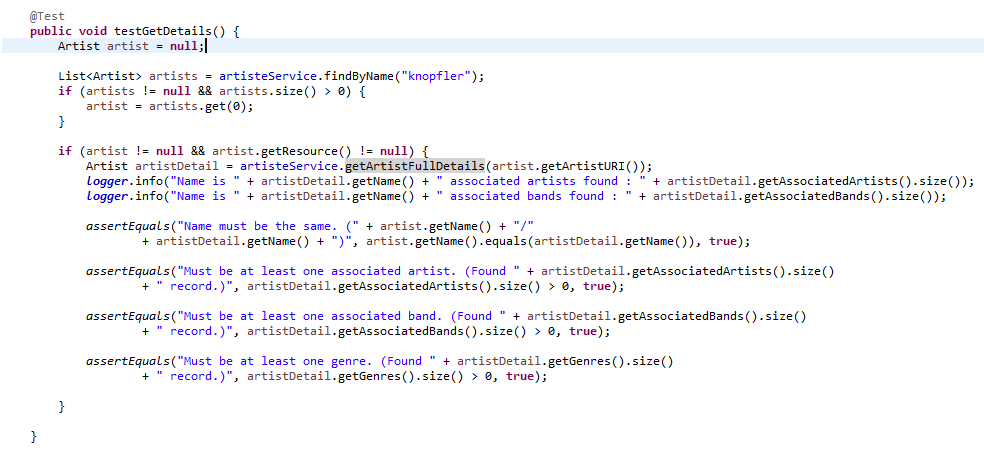
Voici en détails le contenu pour BandRestController :



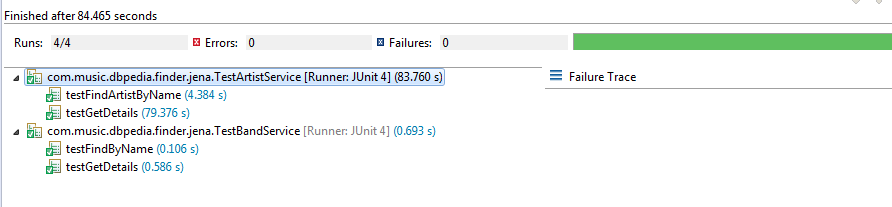
1. Définition par annotation du controller REST ; les services seront disponibles par l’URL relative au contexte de l’application par /rest.
2. BandService est injecté par la mécanique Spring (@Autowired),
3. Les services REST sont ensuite définit :
   1. /bands pour obtenir la liste des bands ; utilisation d’un paramètre Get pour le champ name.

### Tests

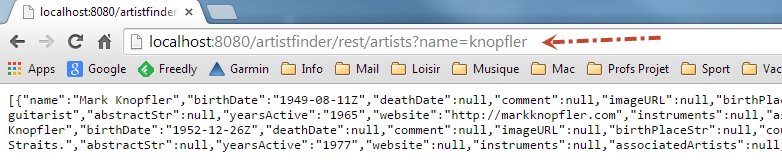
Nous sommes au bout de la création des services REST. Nous pouvons créer des classes de test pour la couche Service comme ceci :



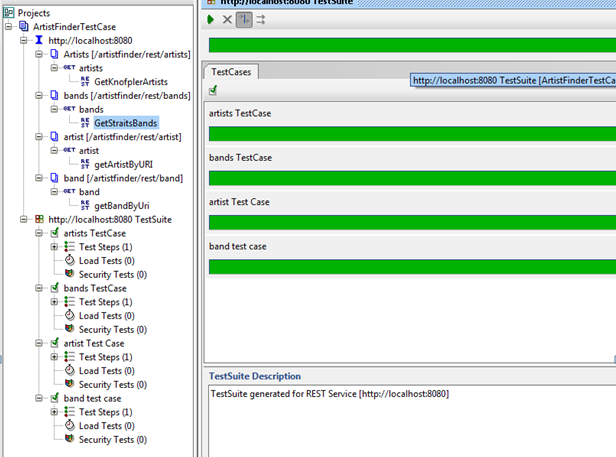
Et lancer les tests :



Nous pouvons aussi contrôler que les services REST fonctionnent correctement



Ou en utilisant SoapUI qui permet d’automatiser des tests sur les web services. Voici un exemple d’utilisation :



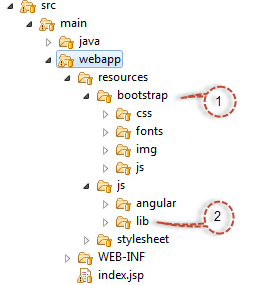
## Frontend

Le frontend est une application JavaScript faite en utilisant le Framework AngularJS et les composants graphiques Twitter Bootstrap.

Mais avant tout il faut télécharger et importer les divers fichiers librairies JS et CSS dans l’application.

1. Bootstrap : <http://getbootstrap.com/getting-started/#download>
2. AngularJS : <https://angularjs.org/>

Il faut ensuite les rendres disponibles pour la partie webapp de l’application :



1. Librairies boostrap,
2. Librairies angular.

### Bootstrapping et Templating

Nous allons créer une page d’index qui contiendra le code HTML de l’application.

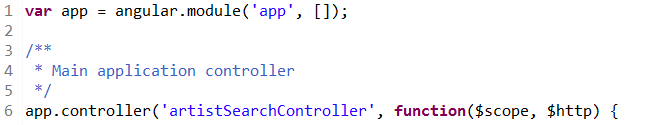


1. Signal de chargement du Framework AngularJS (Bootstrapping), le DOM, les directives, les binding sont chargés,
2. Feuilles de style Twitter Bootstrap, ainsi que celle de l’application,
3. Les scripts jquery.js, angular.js et boostrap.js,
4. Le fichier contenant les sources de notre application (du côté JavaScript).

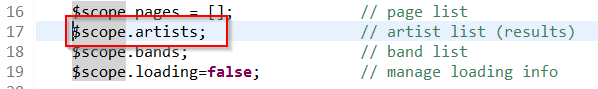
AngularJS est ainsi un MVC disposé dans le Browser. Afin de finaliser le chargement de AngularJS nous créons un contrôleur, charger dans la page HTML de cette façon :

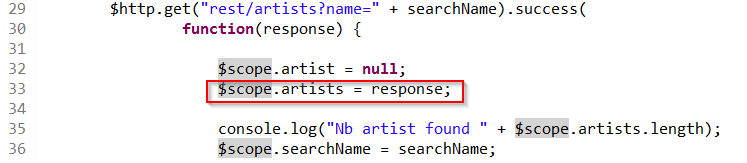


Code mettant ainsi en correspondance le code JavaScript suivant :

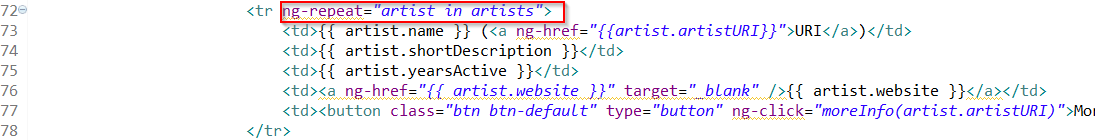


Nous remarquons l’utilisation de la variable $scope, permettant de lier les données depuis le contrôleur jusqu’à la vue de manière bidirectionnelle, par exemple la variable $scope.artists (tableaux de valeurs) sera initialisée dans le contrôler AngularJS comme ceci :





Et mise à disposition dans la vue de cette façon :



En utilisant la directive ng-repeat permettant de boucler sur les valeurs du tableau.

Remarquons aussi cette instruction {{…}} qui permet de reprendre et d’afficher les valeurs des variables.



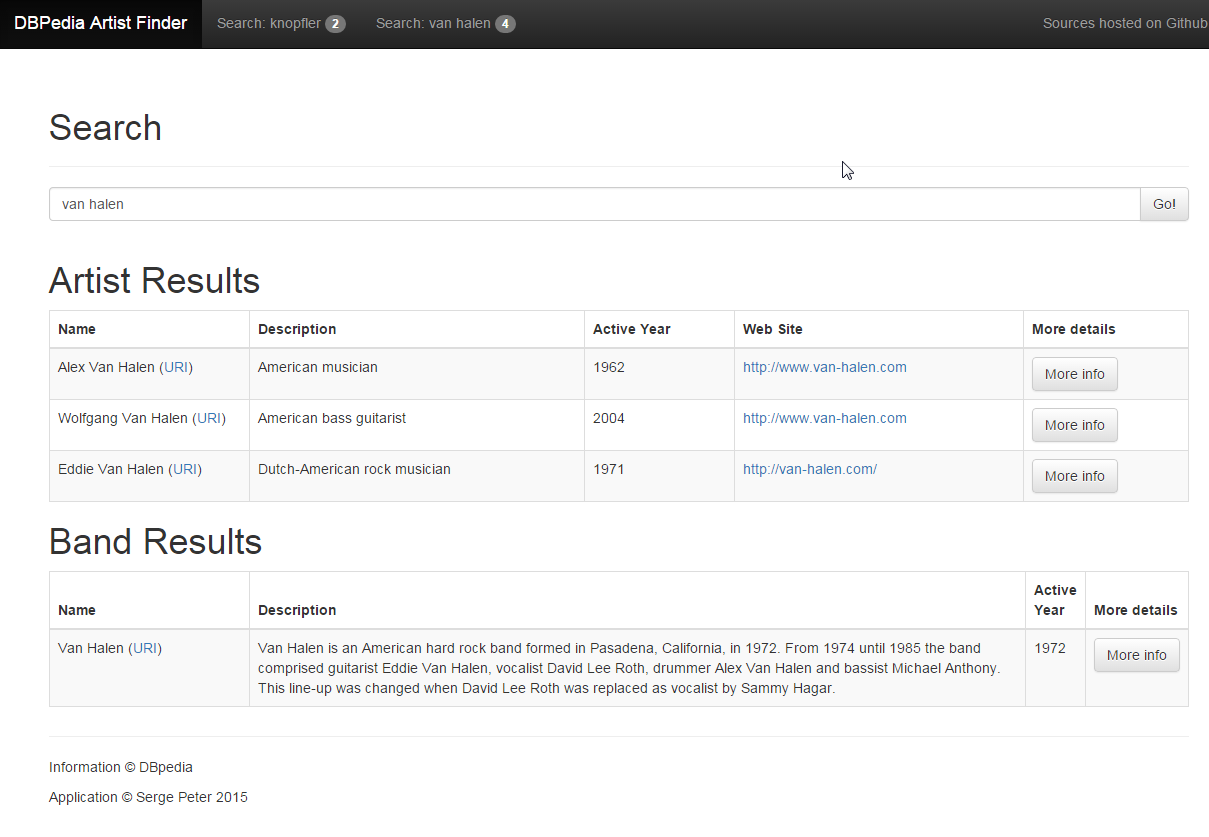
### Construction de la vue

L’application permet de faire des recherches sur des noms d’artiste ou de groupe. En réponse donc à la requête de recherche, deux listes apparaissent :

1. Artiste
2. Groupes

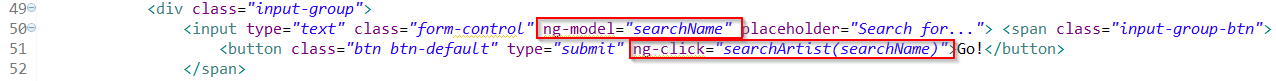
Pour autant que des occurrences existent pour l’un et l’autre.

En voici un exemple :

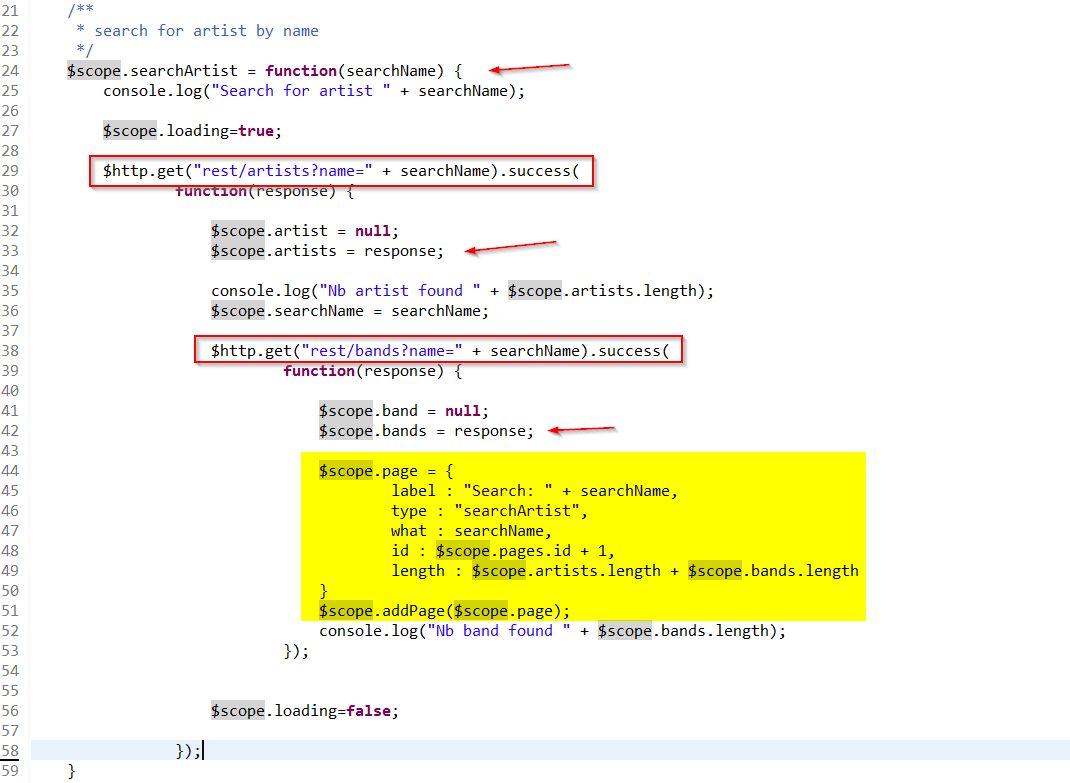


Voici comment cela fonctionne :

1. Le champs recherche aliment la variable : searchName du model $scope,



1. La fonction searchArtist(searchName) est exécutée dans le contrôleur :



La fonction searchArtist prend en argument la chaine de charactère searchName et fera deux choses :

1. Recherche des artistes correspondants en utilisant le service REST ***rest/artists?name***…
2. Rechercher des groupes dons le nom correspond en utilisant le service RESR ***rest/bands?name***…

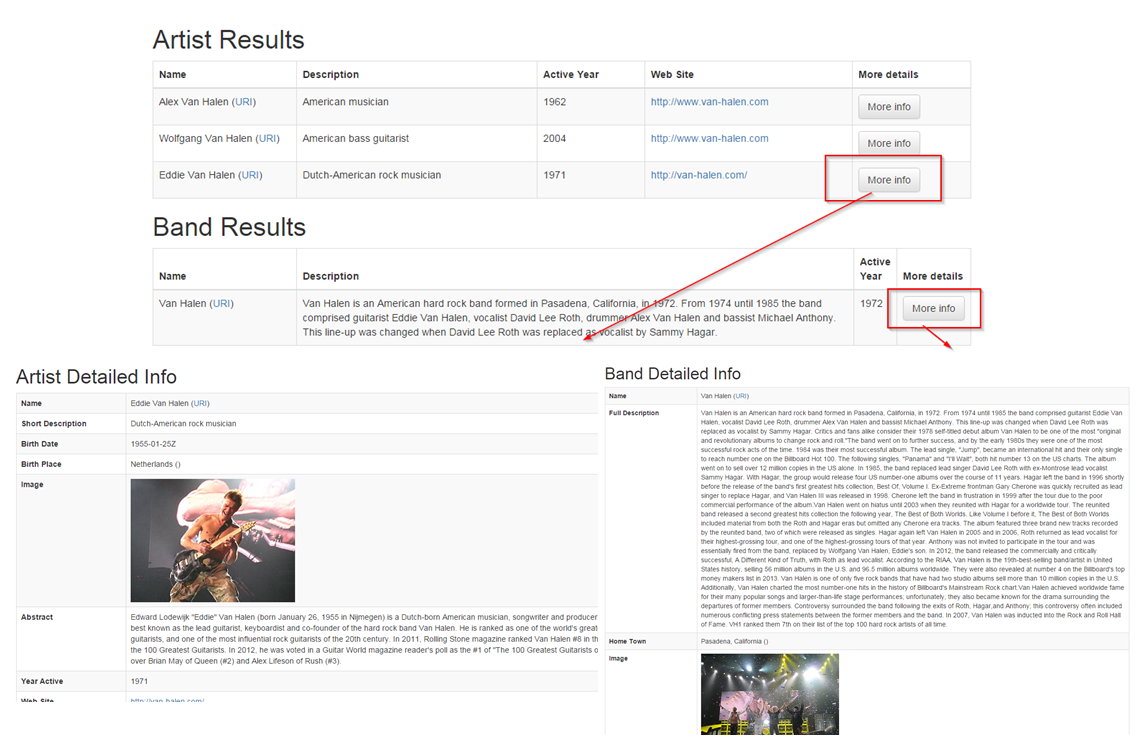
Les variables du $scope.artists et $scope.bands sont alimentées en fonction des données en retours.

Une variable $scope.page sera initialisée et ajoutée au tables $scope.pages, ceci afin de permettre l’affichage d’onglets en haute de la page :



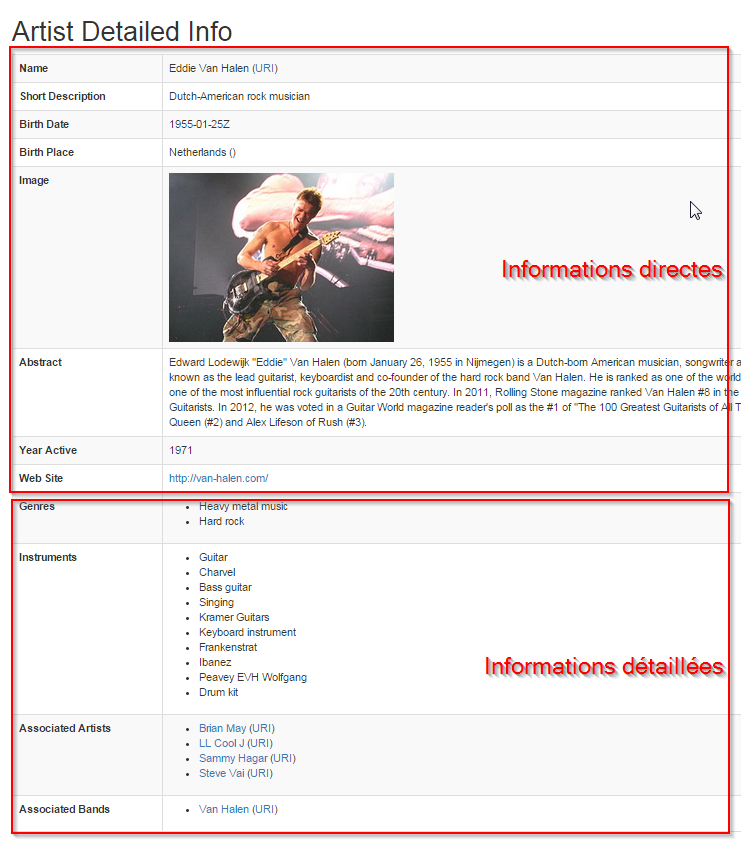
(Ceci est fait de manière assez basique mais est utile si on veut revenir sur une recherche antérieure).

### Vues de détails



J’ai construits ensuite deux vues détaillées pour Artist et Band. Afin d’accélérer le chargement des pages, j’ai décomposé les appels en deux (bien qu’une décomposition plus détaillée devraient être faite) :

1. Obtention des informations basiques
   1. service REST ***rest/artist?uri***=… et
   2. ***rest/band?uri***=…
2. Obtention des informations détaillées, nécessitant des requêtes supplémentaire du côté backend (et donc DBPedia) :
   1. service REST ***rest/artistfull?uri=***… et
   2. ***rest/bandfull?uri***=…



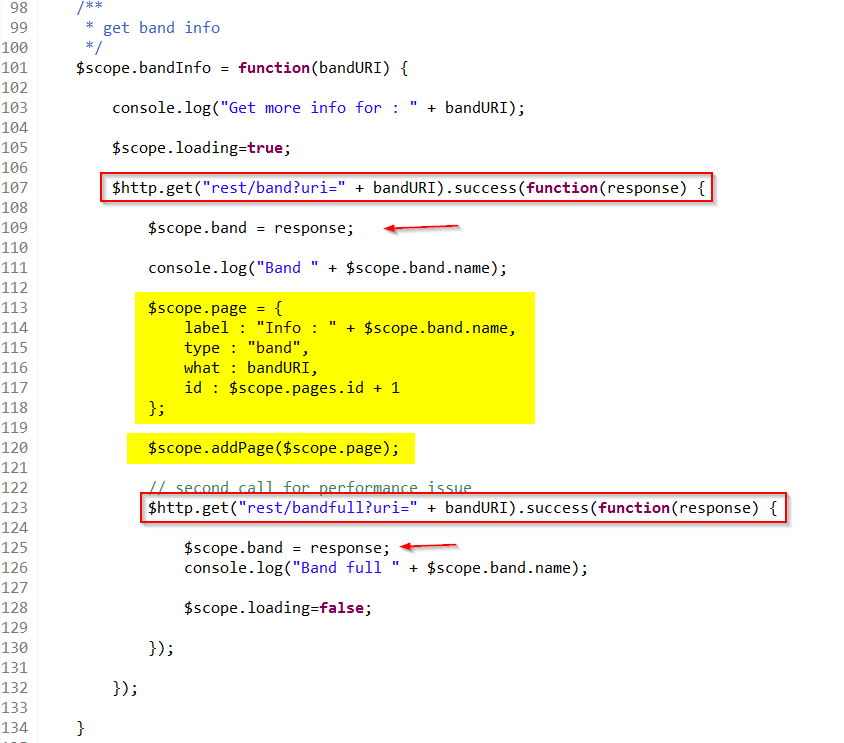
Note :

Si vous tester l’application, vous noterez que les performances de DBPedia sont relativement faibles, ce qui se comprend aisément car la base de données est distante et très vaste. Une indication à l’écran vous permet de voir que les requêtes sont en cours d’exécution ou terminée.



### Code JavaScript

Le chargement des détails Artist et Band se fond de la même manière. En voici un exemple pour Band :



Ici aussi il y a deux appels aux services rest ; band et bandfull. Ceci pour des nécessité de fluidité dans le chargement de la page.

La variable $scope.band est ainsi enrichi, et mise à disposition pour la vue.

La variable $scope.page est aussi mise à jour afin de facilité la navigation dans l’application.

Note :

Ceci est fait de manière un peu basique et nécessiterait un découpage plus précis des informations mise à jour et peut-être plus d’appel au backend fluidifier le chargement de la page.

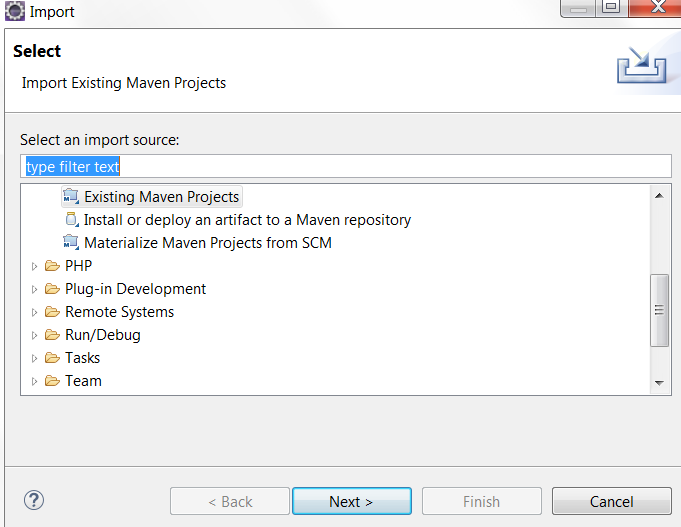
# Exécution de l’application

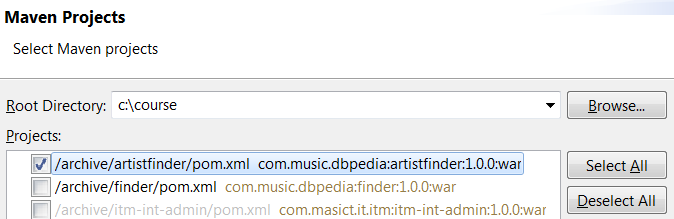
Il y a plusieurs possibilités pour exécuter cette application. Attention de faire des recherches d’artistes anglophone car je suis connecter à la source de données DBpedia par défaut.

## Directement dans Eclipse

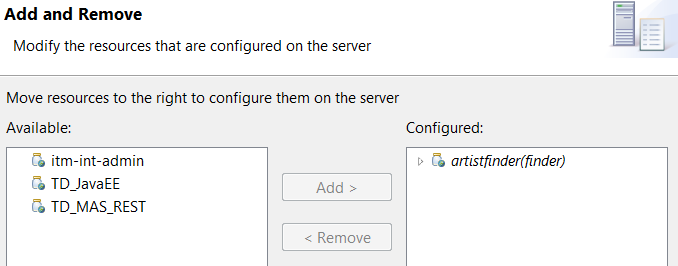
Une fois toutes les opérations du chapitre « Mise en route », chapitre 4 faite, il reste à faire ceci :

1. Mettre les sources sur le disque local,
2. Importer le projet en tant que projet Maven,





1. Installer l’application sur le serveur Tomcat intégré et installé dans Eclipse,



1. Ouvrir l’application sous : [http://localhost:8080/artistfinder (8080](http://localhost:8080/artistfinder%20(8080) étant le port définit pour l’instance de Tomcat).

## En mode ligne de commande avec Maven

Là aussi il faut faire l’installation des logiciels prérequis, notamment Maven. Une fois les sources chargées sur le disque local, il faut exécuter depuis le répertoire du projet la commande

mvn clean tomcat7:run

## Sur le site Openshift

J’ai installé l’application Artist Finder sur le site Openshift de Redhat.

<http://artistfinder-spe.rhcloud.com/>

Note : il se peut qu’Openshift mette en veille l’application si elle n’est pas utilisée. Il faut donc attendre un peu et recharger l’application si elle ne s’ouvre pas directement.

# Conclusion

# Références

Dbpedia :

* <http://dbpedia.org>

Sparql

* <https://jena.apache.org/tutorials/sparql.html>

Jena :

* <http://www.ibm.com/developerworks/xml/library/j-sparql/>
* <https://jena.apache.org/tutorials/index.html>

Livre sur AngularJS

* AngularJS de Brad Green, Shyam Seshadri, publié par O'Reilly en avril 2013; ISBN:978-1-4493-4485-6
* Mastering Web Application Development with AngularJS de Pawel Kozlowski et Peter Bacon Darwin, publié par Packt Publishing en Août 2013; ISBN 978-1-78216-182-0
* https://docs.angularjs.org/tutorial

Spring

* <http://projects.spring.io/spring-data>
* <http://www.programming-free.com/2014/07/spring-data-rest-with-angularjs-crud.html>
* <http://docs.spring.io/spring/docs/current/spring-framework-reference/html>