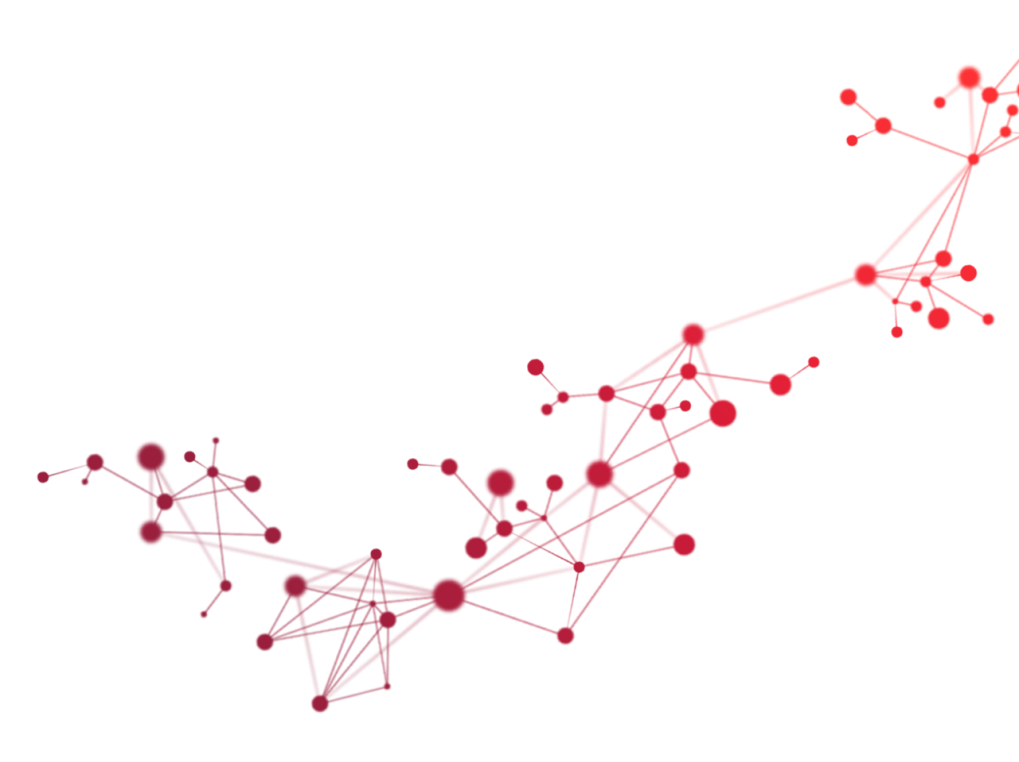
# Mise en place d’une POC[[1]](#footnote-1) sur ElasticSearch.



Stagiaire : Maxime ROSE.

Formation : Licence 3 informatique, Université de Lille 1

Entreprise : CGI

Titre : Développeur informatique.

Durée : ~5 mois (13/04 – 28/08)

Tuteur enseignant : F. Paszkowski.

Tuteur professionnel : A. Ramarojohn.

# Avant-Propos

Je voudrais tout d’abord remercier l’université de Lille, qui m’a permis de réaliser ce stage très enrichissant pour moi, ainsi que tout le personnel enseignant. Et plus particulièrement Franck Paszkowski, mon tuteur universitaire, qui m’a suivi tout le long de mon stage.

Je remercie également Adrien Ramarojohn, mon tuteur professionnel, pour le temps qu’il m’a consacré, ainsi que son aide et ses conseils. Ainsi que toutes les autres personnes de l’équipe, et CGI, qui m’ont permis d’associer le travail et la bonne humeur.

Table des matières

[Avant-Propos 2](#_Toc422692229)

[Présentation de l’entreprise 4](#_Toc422692230)

[Introduction 5](#_Toc422692231)

[Le sujet de stage 6](#_Toc422692232)

[Le stage en détail 7](#_Toc422692233)

[1. ElasticSearch 7](#_Toc422692234)

[a. Première approche 7](#_Toc422692235)

[*b.* *Proof Of Concept* 8](#_Toc422692236)

[*c.* *Mapping* 9](#_Toc422692237)

[d. Relation *Parent/Child* 10](#_Toc422692238)

[2. L'interface Web 12](#_Toc422692239)

[a. NgGrid 12](#_Toc422692240)

[b. Première requête REST 13](#_Toc422692241)

[c. NVD3 14](#_Toc422692242)

[*d.* *Typeahead* 17](#_Toc422692243)

[Conclusion 19](#_Toc422692244)

# Présentation de l’entreprise

CGI (Conseillers en Gestion et Informatique), implanté dans 40 pays répartis en Amérique, Europe et Asie-Pacifique, trouve son siège social à Montréal. En France et au Luxembourg, CGI possède 21 bureaux, où travaillent 10 000 professionnels parmi les 68 000 dans le monde. Leurs clients s’étendent sur de nombreux secteurs, comme les finances les télécommunications et médias, les énergies, les \*Transports Tertiaires et Industries, et le secteur public. (Voir Figure 1). 39 d’entre eux font partie du CAC 40. L’entreprise propose des services de conseils, d’intégration et d’outsourcing (sous-traitance).

CGI Nord est représenté par 410 professionnels, qui forment 6 business team.

Je fais partie de la business team des Nouvelles Technologies de l’Information et de l’Internet, et plus particulièrement de l’équipe IT-ToolBox. Cette équipe est composée de 8 embauchés, 2 alternants, et 5 stagiaires.

Figure 1 : Principaux clients de CGI Nord.



# Introduction

Étudiant en 3e année de licence Informatique à l’université de Lille 1, j’ai effectué mon stage chez CGI pour une durée de 5 mois.

Cette entreprise étant mondiale, ça m’a rassuré au niveau de la stabilité professionnelle, et d’une éventuelle évolution de carrière dans cette entreprise. Et c’est ce pour quoi j’ai postulé chez CGI.

J’étais à la recherche d’un stage de développement, web ou applicatif, pour pouvoir finaliser et validé ma licence informatique, et c’est ce que CGI m’a proposé.

J’ai attendu de ce stage d’apprendre de nouvelles technologies, des petites astuces pour mieux coder, et surtout confirmer mon gout du travail en équipe.

Dans ce rapport, je vais vous présenter les différentes missions qui m’ont été attribuées en détail, comme la POC [[2]](#footnote-2)et la montée en compétence sur ElasticSearch, ainsi que le développement de l’interface web.

.

# Le sujet de stage

Pour mieux comprendre mon sujet de stage, je vais vous expliquer le contexte.

Il existe une application nommée OpenFramework (OF) qui permet de collecter des flux dans le parc d’application d’une entreprise. Ce grand nombre de données sera stocké, puis analysé pour afficher, dans une interface web, des graphiques et des statistiques.

La version 3 de cette application semble avoir atteint ses limites au vu du nombre de données qui augmente de plus en plus. C’est pourquoi ma mission globale de stage était de participer à l’élaboration de la version 4 d’OF, qui sera doté de nouvelles technologies pour pallier à ce manque de performance.

Cette nouvelle version est découpée en 2 projets, un premier en Java et utilisant le Framework Spring Data, qui va collecter les flux venant d’un ESB[[3]](#footnote-3), et les insérer dans le moteur de recherche, ElasticSearch. Le second projet est une application web utilisant AngularJS, qui va chercher les données dans ElasticSearch, pour les afficher dans l’interface web

# Le stage en détail

## ElasticSearch

ElasticSearch est un moteur de recherche totalement basé sur le format JSON[[4]](#footnote-4), et qui stocke les données sous forme de documents. Si l’on compare ElasticSearch avec une base de données de type SQL, une base de données est un index, et une table est un type dans ElasticSearch.

GET http://localhost:9200/<nom\_de\_l\_index>/<nom\_du\_type>

Pour récupérer des documents, il faut utiliser l’API[[5]](#footnote-5) REST[[6]](#footnote-6), avec des méthodes de type GET, POST, PUT ou DELETE, par le protocole HTTP. Voici un exemple d’une requête qui permet de récupérer tous les documents d’un type.

J’utiliserai cette notation tout au long de ce rapport, avec en début, le type de méthode, puis l’adresse d’ElasticSearch pour atteindre un certain type. Il se peut qu’il y ait des paramètres en plus au format JSON en dessous.

### Première approche

PUT localhost :9200/index/type/nouvel\_id

{

"user" : "mrose",

" description" : "Étudiant en L3 Info"

}

J’ai commencé par faire des tests, en utilisant la documentation se trouvant sur le [Site Officiel d’ElasticSearch](http://www.elastic.co/guide/index.html)[[7]](#footnote-7), j’en ai appris beaucoup sur les recherches qui utilisent les requêtes REST, avec des paramètres spécifiques, au format JSON, pour ElasticSearch. Par exemple, pour insérer un nouveau document, on peut exécuter cette requête :

POST localhost :9200/index/type/**\_search**

{

"query" : {

"match\_all" : {}

}

}

Et voici un autre exemple, qui permet d’aller rechercher tous les documents d’un certain type :

{

"\_index" : "my\_index",

"\_type" : "my\_type",

"\_id" : "nouvel\_id",

"\_source" : {

"user" : "mrose",

"description" : "Étudiant en L3 info"

}

}

Et nous obtenons la réponse (simplifié) d’ElasticSearch, qui ressemble à ceci :

### *Proof Of Concept*

{"index" : { "\_index" : "monIndex", "\_type" : "monType", "\_id" : "1"}}

{"field1" : "value1", "filed2" ; "value2", "field3" : "value3"}

{"index" : { "\_index" : "monIndex", "\_type" : "monType", "\_id" : "2"}}

{"field1" : "value1", "filed2" ; "value2", "field3" : "value3"}

Avant de me lancer réellement dans le projet, je devais d’abord prouver par des tests qu’ElasticSearch est plus performant qu’une base de données. C’est pourquoi j’ai importé des données d’une base de données Oracle déjà existantes, avec des données qui pourraient correspondre aux futures données utiliser pour la nouvelle version d’OF. Cependant, le système d’export ne propose pas un type d’exportation au format JSON. Donc au départ, j’ai exporté 5-6 lignes de la base de données au format CSV, puis sur le site [Convert CSV](http://www.convertcsv.com/csv-to-json.htm)[[8]](#footnote-8), j’ai converti mon fichier CSV au format JSON. ElasticSearch est capable de recevoir un fichier JSON pour insérer un grand nombre de données en une seule fois. Mais je me suis rendu compte qu’il demande une certaine structure dans le fichier JSON pour qu’il puisse l’accepter, donc j’ai modifié à la main ce fichier pour pouvoir y insérer mes quelques lignes.

Même pour un petit nombre de données, c’était assez long, et laborieux, parce qu’il faut ajouter une ligne pour chaque donnée. C’est pourquoi j’ai créé un petit script en Python qui modifie automatiquement le fichier JSON, pour qu’il soit accepté par ElasticSearch. Ce script n’est que temporaire, parce que la partie collecte et insertion de données en Java n’était toujours pas finalisée. Donc en attendant j’utilisais ce script, pour justement pouvoir tester des requêtes sur un grand nombre de données. Et après de nombreux tests, j’ai su montrer qu’avec un très grand nombre de données (de l’ordre du million), ElasticSearch est plus rapide et performant qu’une base de données « classique » ; les requêtes s’exécutent plus vite.

### *Mapping*

Une fois ces premières requêtes maitrisées, je me suis intéressé au mapping, qui consiste à définir un type (définir la structure d’une table SQL) avant l’insertion de données. Si on ne spécifie pas de mapping, ElasticSearch le créé lui-même en fonction des données qu’il reçoit. Mais nous voulions que le mapping soit déjà prédéfini pour éviter les erreurs d’insertions.

J’ai donc demandé à mon tuteur quelle était la structure des données qui seront insérées plus tard, pour que je puisse créer le mapping. Étant donné que cette structure n’était pas encore définie précisément, il m’a donné une ébauche de ce que pourrait ressembler plus tard les données. Nous aurons pour le moment un type, « instances », avec un identifiant, un état global, une date de début, une date de fin, et d’autres champs pas encore utiles pour le moment. La requête pour insérer un mapping à un type dans ElasticSearch s’effectue de cette manière :

PUT localhost:9200/index/**\_mapping**/type

{

"properties" : {

"instanceId" : {

"type" : "string"

},

"etatGlobal" : {

"type" : " string"

},

....

}

}

En contrepartie, la partie du projet en Java qui collecte les flux de l’ESB fut exploitable pour faire des premiers tests. J’ai donc analysé un peu le code pour comprendre le fonctionnement, et pour pouvoir y ajouter mes mappings. Pour cela, j’ai ajouté 2 fichiers JSON avec les 2 mappings différents, pour d’abord tester que le mapping s’effectue correctement. J'ai donc ajouté l'annotation @Mapping sur la classe de référence à ElasticSearch.

@Mapping(mappingPath = "mapping-instance.json")

putMapping(String indexName, String typeName, Object mapping);

Cependant, cela n'avait pas l'aire de fonctionner, donc j'ai cherché s'il existait une autre solution pour ajouter un mapping avec Spring Data. J’ai trouvé une méthode putMapping dans la documentation de Spring Data pour ElasticSearch, avec plusieurs signatures. J’ai donc choisi en premier lieu celle-ci :

Mais le problème, c'est qu'il faut noter dans le code, tout le mapping, qui prend une centaine de lignes pour un seul mapping… Avec Adrien, nous avons donc approfondi le comportement de la méthode, via une analyse du code source. Nous avons utilisé cette signature de fonction :

putMapping(Class.clazz);

J'ai passé en argument la classe de référence (la même que celle où j'avais ajouté l'annotation @Mapping), et c'est justement grâce à cette méthode que l'annotation trouve son importance, et le mapping s'exécute correctement avant l'ajout de données.

Nous avons également un autre type "description", qui est en quelque sorte, le type d’une instance, par exemple, il peut y avoir une instance de type Médiation, ou une instance de type Échange.

Auparavant, ces deux tables dans la base de données étaient liées par une jointure. Je me suis donc penché sur la documentation de la relation Parent/Child d’ElasticSearch

### Relation *Parent/Child*

Avant tout, j’ai commencé par noter sur feuille les 2 mappings voulus, pour voir plus clairement qui est parent, et qui est enfant. Après réflexion, il faut que l’instance soit enfant de la description. Pour définir cette relation, il faut le préciser dès le mapping de cette façon :

PUT localhost:9200/index/**\_mapping**/type

{

"\_parent" : {

"type" : "description",

"identifier" : "descriptionCode"

},

"properties" : {

"instanceId" : {

"type" : "string"

},

"etatGlobal" : {

"type" : " string"

},

....

}

}

Dans le "type", on indique le type d'ElasticSearch qui sera lié avec ce type, et l'"identifier" sert à préciser sur quel champ, de l'autre type, nous souhaitons que la relation se fasse. Et un des avantages d'utiliser ElasticSearch est que l'on est pas obligé de doubler ce champ dans chaque type, la relation se fait automatiquement.

Le mapping prenant une forme convenable et acceptable pour de vrai tests, je me suis concentré sur la partie Java pour faire correspondre le mapping que j'avais réalisé, et les classes de références. Ce fut un travail assez laborieux, parce que pour supprimer une simple variable, il fallait modifier au moins 3 fichiers pour ne plus avoir d'erreurs. Après une petite journée de modification, j'ai pu commencer à tester si mon mapping était correct, et c'est à ce moment que je me suis confronté à un nouveau problème.

Étant donné qu'il existe une relation Parent/Child, lord de l'insertion d'une instance, l'exception qui est levée nous indique qu'il est obligatoire de spécifier le parent pour pouvoir insérer le document. J'ai analysé la liste des méthodes existantes dans l'API Spring Data, et j'ai trouvé une méthode qui permet de spécifier l'id du parent juste avant d'indexer le document :

ElasticsearchTemplate.**setParentId(String parentId)**;

Et étant donné que l'application Java ne gérait pas encore les descriptions, j'en ai créé une moi-même avec des données factices, pour pouvoir tester mon mapping, et ma relation Parent/Child. Et ce fut une réussite qui me permit de passer au développement de l'interface web.

## L'interface Web

Avant de commencer, mon tuteur m'a proposé de ne pas commencer à développer l'interface web directement, mais plutôt de créer de petits modules, pour apprendre et me familiariser avec AngularJS. En quelque sorte de créer un petit Framework, pour qu'une fois que la maquette de l'interface soit réalisée, il ne reste plus qu'à utiliser mes petits modules, déjà prêt à l'utilisation, et de les implémenter dans la page finale.

J'ai tout d'abord suivi le tutoriel sur le site de [w3school](http://www.w3schools.com/angular/default.asp)[[9]](#footnote-9), pour apprendre, et comprendre le fonctionnement d'AngularJS, avec les applications, modules, controllers, services, et directives. Et grâce aussi à l'aide d'Adrien qui maitrise déjà Angular.

### NgGrid

Var columnOptions = [

{"field" : "descCode", "displayName" : "Code de la Description"},

....

];

//On affilie cette variable au paramètre columnDefs du tableau

$scope.gridOptions.columnDefs = columnOptions;

Une fois une base créée, j'ai essayé d'utiliser le module ngGrid, qui est fourni avec AngularJS, et qui permet de réaliser un tableau plus ou moins complexe. Pour commencer, j'ai pris des données factices pour étudier le comportement de ngGrid, et il se trouve que le tableau se génère automatiquement, et inscrit des noms de colonne si l'on ne lui en a pas spécifié. Cependant ces noms de colonne ne sont pas très explicites pour le futur utilisateur, car ça correspond au nom de la variable correspondante, avec un espace entre deux mots (par exemple: "descCode" deviendra "Desc Code"). J'ai donc utilisé le paramètre "columnDefs", qui permet de gérer les colonnes. Par exemple, nous pouvons spécifier pour la variable "descCode" que nous voulons afficher "Code de la Description". Pour faire cela, il suffit de faire :

### Première requête REST

Après cela, Adrien m'a demandé de créer une requête REST, qui utilise des agrégations, pour pouvoir grouper par descriptions, et faire des statistiques sur les instances reliées à ces descriptions. La logique de cette requête doit se faire de cette manière :

* Récupérer toutes les descriptions.
* Les grouper par rapport à leur identifiant.
* Récupérer toutes les instances liées à chaque description :
  + Utiliser l'agrégation "stats" sur le champ de durée.
  + Grouper sur le champ "flowStatus" pour simplement les compter.

POST localhost:9200/index/descriptions/**\_search**

**{**

**"query" : {**

**"match\_all" : {}**

**},**

**"aggs" : {**

**"group\_description" : {**

**"terms" : {**

**"field" : "descriptionCode"**

**}**

**},**

**"aggs" : {**

**"instances" : {**

**"children" : {**

**"type" : "instances"**

**}**

**},**

**"aggs" : {**

**"stats\_duration" : {**

**"stats" : {**

**"field" : "duration"**

**}**

**},**

**"group\_flowStatus" : {**

**"terms" : {**

**"field" : "flowStatus"**

**}**

**}**

**}**

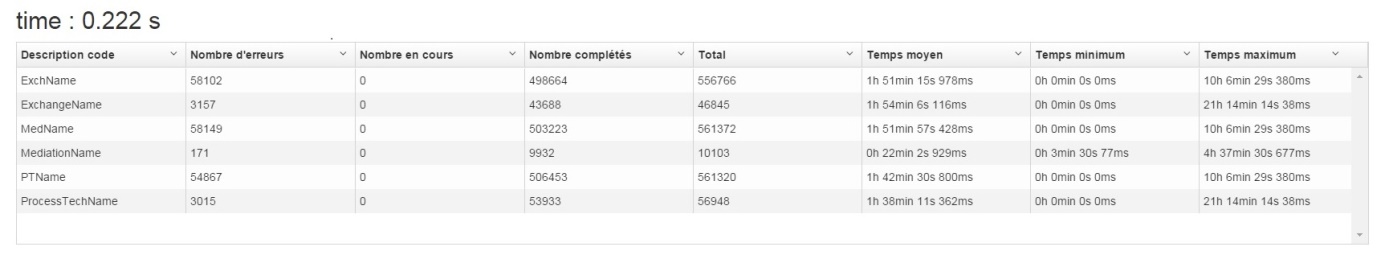
**}**

**}**

**}**

La requête aura cette forme :

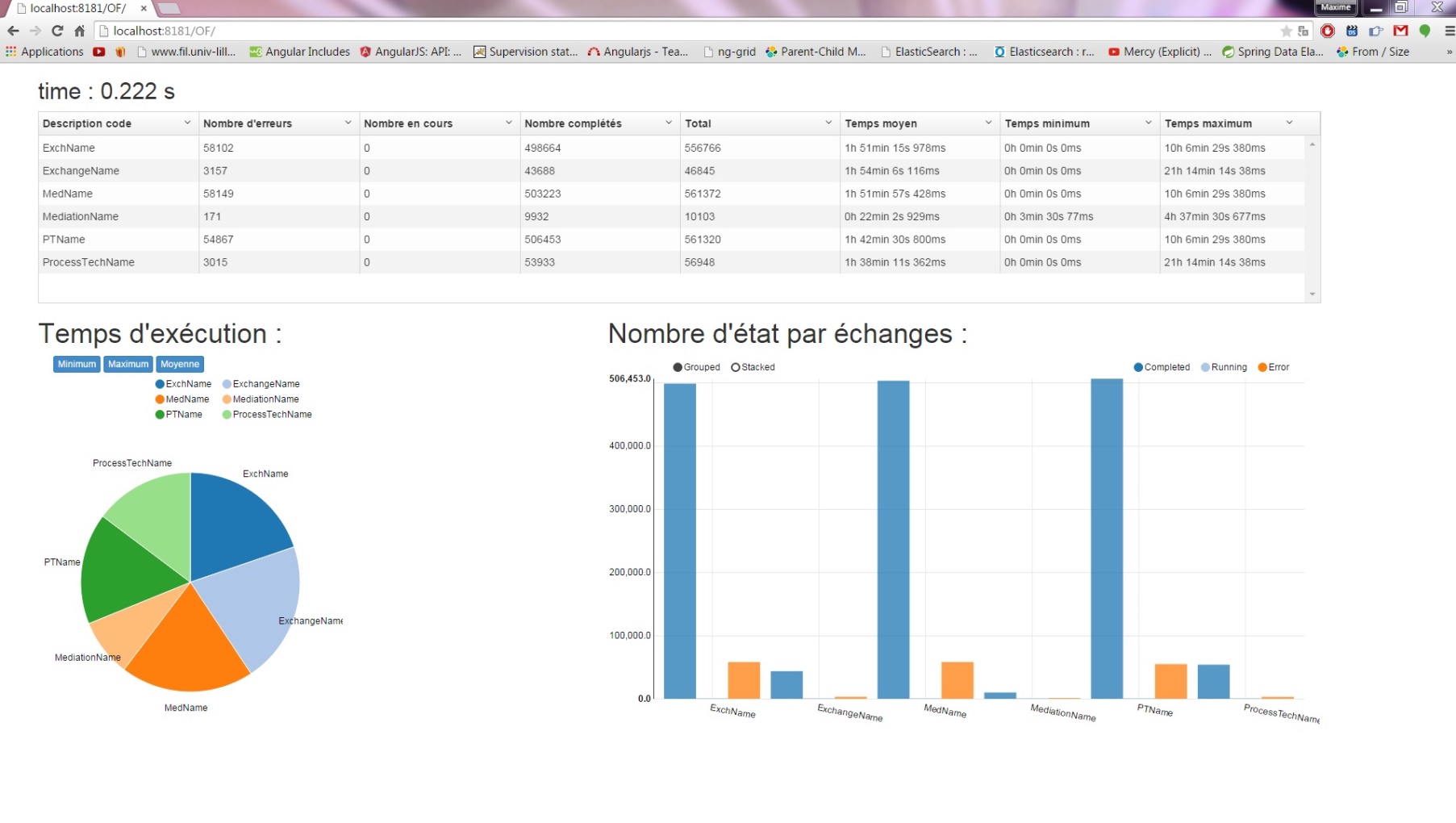
Comme on peut le voir, ElasticSearch permet de faire plusieurs agrégations à la suite, ou en cascade.

Il suffisait de structurer les données reçues, pour pouvoir les afficher dans le tableau. Ce qui nous donne le tableau suivant :

### NVD3

Une fois ce tableau réalisé, je devais utiliser un module externe, nvd3 qui permet de réaliser de très beaux graphiques dynamiques. Voici une [liste d'exemple](http://nvd3.org/examples/index.html)[[10]](#footnote-10).

Je me suis servi du Pie Chart, et Multi-bar Chart. Pour le graphe en camembert (pie chart), je n'ai pas rencontré de gros soucis, ça s'adapte très facilement à toutes sortes de données. Par contre, pour le multi-bar, ce fut plus complexe, parce que de base, il ne reçoit que des entiers ou des dates pour les valeurs des axes, cependant, dans notre cas, nous voulons un simple texte. Pour résoudre ce souci, j'ai dû modifier le code qui initialise le graphique, pour ajouter un filtre sur les axes, pour qu'il accepte le format de chaine de caractères. Après cela, les données insérées nous ont renvoyé le bon graphique attendu. Ce qui conclut la fin d'une première page qui analyse des données, les affiches, et forme des graphiques.

 Il m'a été demandé de faire une nouvelle page, pour faire une nouvelle fonctionnalité. Un page qui va chercher toutes les instances existantes, pour les afficher. Avec également des filtres, qui correspondent aux différentes colonnes du tableau, et un filtre qui permet de filtrer les filtres, pour pouvoir ajouter ou supprimer des filtres, ce qui évitera de surcharger la page.

J'ai commencé par créer les filtres, sans qu'ils ne soient exploitables tout de suite, mais pour que la structure de la page soit déjà réalisée. Ensuite, j'ai créé la requête qui renvoie toutes les instances.

POST localhost:9200/index/instances/\_search

{

"query" : {

"match\_all" : {}

}

}

Cependant, cette requête nous renvoie les 10 premiers éléments. C'est pourquoi j'ai dû ajouter le paramètre "size", pour lui donner comme valeur 1000, pour qu'il récupère toutes les données. Mais pour le moment, je n'avais pas énormément de données, donc j'en ai inséré environ 1 million, pour voir si tout fonctionne bien, et ça fonctionne, mais le temps d'exécution est assez long (de l'ordre d'une vingtaine de secondes). Comparé à l'ancienne version, c'est plus qu'acceptable, puisque maintenant, les données peuvent être chargées en temps réel. Pour optimiser davantage cette fonctionnalité, ElasticSearch propose comme pour le paramètre "size", un paramètre "from", qui permet de charger les données à partir à cette valeur. Par exemple, si nous avons 10 000 données, et que nous effectuons cette requête :

POST localhost:9200/index/instances/\_search

{

"size" : 100,

"from" : 100,

"query" : {

"match\_all" : {}

}

}

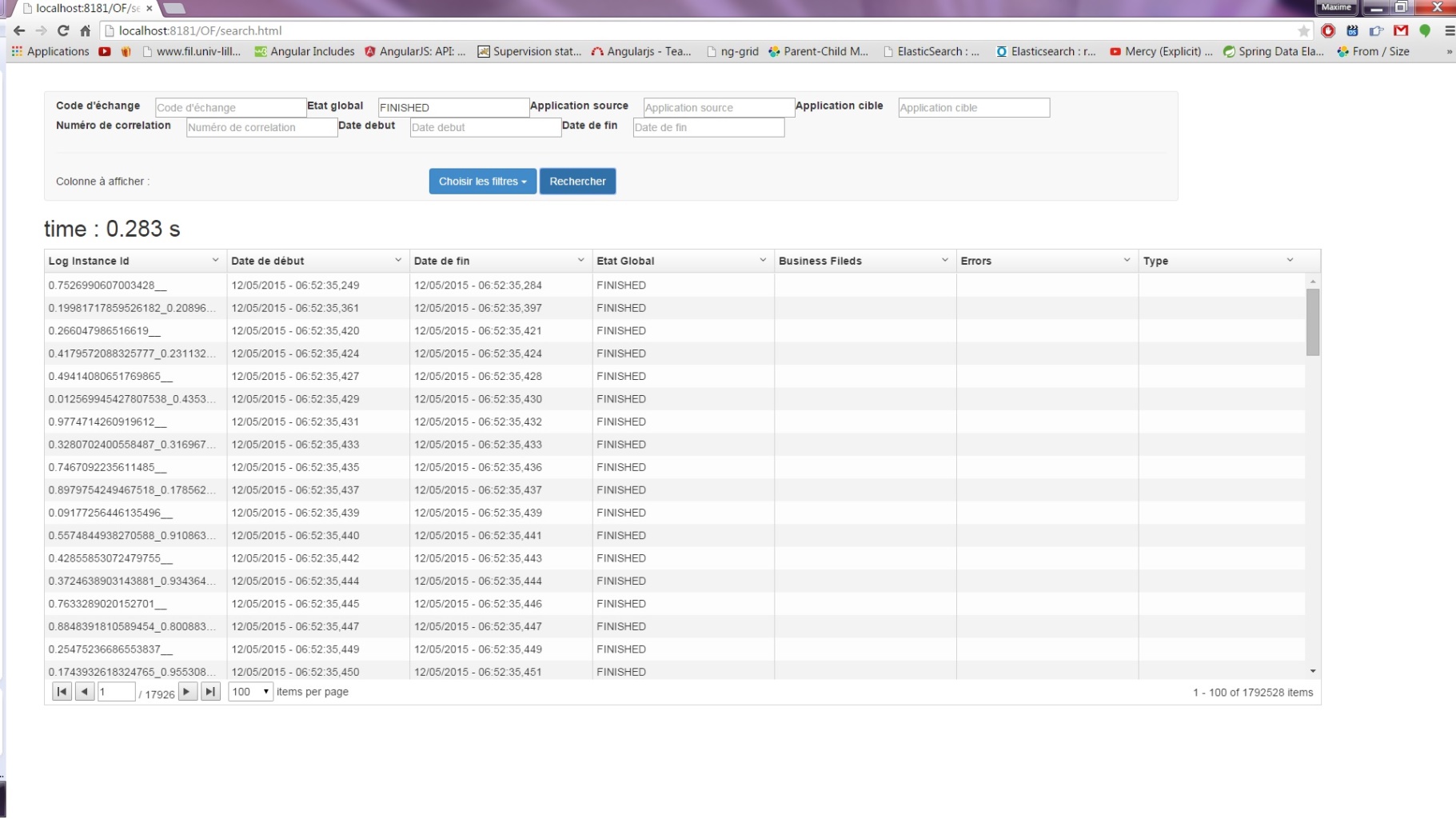
Nous recevrons:

* 100 données, grâce au paramètre "size"
* À partir de la 100e donnée, grâce au paramètre "from"

Cette requête nous permet donc de gérer notre pagination pour le tableau. Un petit problème est apparu à ce moment, comme ElasticSearch nous renvoie 100 éléments, et que le tableau nous affiche 100 par 100 items, il créer donc une seule page. Pour remédier à ce problème, je me suis servi d'un champ dans la réponse d'ElasticSearch, qui contient le total d'éléments. Cette valeur m'a permis de dire à mon tableau le nombre total d'items, pour pouvoir créer le nombre total de pages. J'ai ensuite créé la fonction qui permet d'appeler la requête adéquate en fonction de la page courante, et si l'on choisit la page suivante ou précédente.

Le résultat de cette fonctionnalité est donc bien optimisé, nous obtenons un temps de moins d'une seconde pour charger la page et le tableau.

Nous pouvons l'observer sur cette image :



### *Typeahead[[11]](#footnote-11)*

Le typeahead est un module d'UI-Bootstrap, qui est lui-même un module natif d'AngularJS, et qui permet de faire de l'auto-complétion. Je m'en suis servi pour réaliser une nouvelle fonctionnalité. Un champ de recherche, pour rechercher toutes les descriptions, et avoir en auto-complétion la liste des descriptions. Ce module fonctionne très bien, et n'est pas si difficile à mettre en place au premier abord. Le gros souci que j'ai rencontré, c'est que nous voulions que lorsque l'on clique sur le champ de texte, la liste des descriptions apparaisse avant d'avoir tapé une lettre. Mais après une longue analyse du code source de ce module, je n'ai pas trouvé cette option, donc j'ai décidé de copier-coller le code source dans un nouveau fichier, et d'y ajouter ma propre fonction qui détecte le clique pour ouvrir la liste.

Au niveau de la requête REST, j'ai combiné la "term query" et la "regexp query", parce qu'il m'était demandé aussi de pouvoir utiliser une notation de type regex pour faire des recherches. Par exemple, dire MedName-1.\* pour avoir toutes les médiations qui sont numérotées par un nombre commençant par 1. J'ai utilisé le filtre booléen OR dans ma requête pour qu'il prenne soit le terme exact, soit le regexp. Ce qui ressemble à ceci :

POST localjost:9200/index/type/\_search

{

"query" : {

"filtered" : {

"filter" : {

"or" : [

{ "terms" : {

"descriptionCode" : "MedName-1"

}},

{ "regexp" : {

"Med.\*|Ex.\*|2.\*"

}}

]

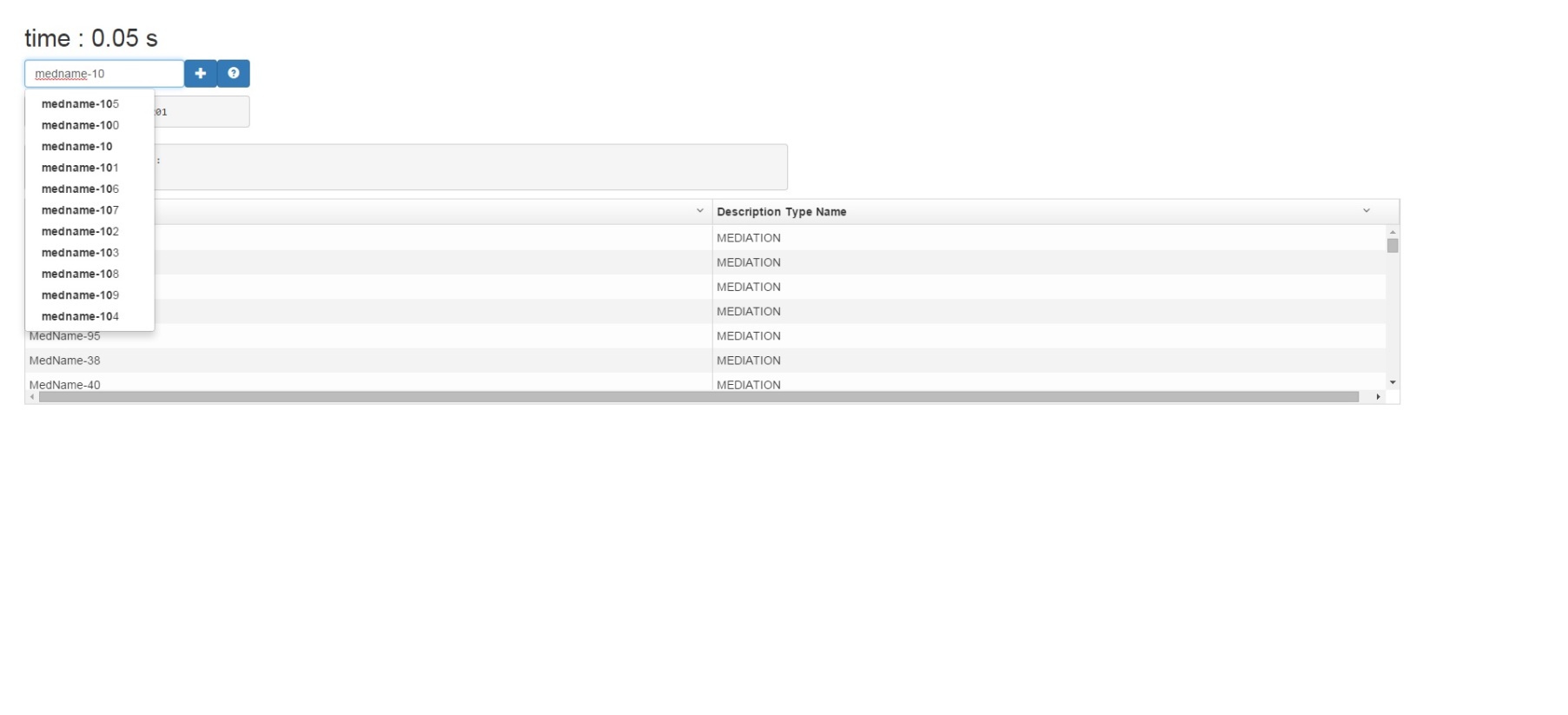
}

}

}

}

Cependant, je suis tombé sur un souci. Le champ descriptionCode n'est pas analysé par ElasticSearch, ce qui veut dire qu'il lui faut la valeur exacte pour le trouver. Mais pour la première page, j'ai besoin que ce champ ne soit pas analysé pour pouvoir faire le groupement par descriptionCode.

 J'ai trouvé dans la documentation d'ElasticSearch qu'il était possible de créer notre propre analyseur, cependant je n'ai pas encore pu tester si cette solution pourra résoudre ce problème, je suis actuellement en train de travailler dessus. Voici à quoi ressemble pour le moment la page :

Mon stage se terminant en fin aout, j'ai encore beaucoup de choses à apprendre d'ElasticSearch et Angular JS.

# Conclusion

Ma mission correspondait tout à fait à ce que je recherchais, travaillé sur de nouvelles technologies, orienté web.

À ce jour, mon stage ne prend fin qu'en fin aout. J'ai encore beaucoup de choses à réaliser, néanmoins, l'avancement de mes travaux satisfait pleinement mon tuteur.

Je sortirais de ce stage grandi, grâce aux nombreuses compétences que j’ai acquises, aussi bien techniques que dans la gestion de projet ou le travail d'équipe. C’est important pour moi, pour ma poursuite d’étude, et mon plan de carrière.

CGI me conforte dans le choix de travailler dans l'informatique, et dans une SSII.

1. POC : *Proof Of Concept* [↑](#footnote-ref-1)
2. POC : *Proof* Of *Concept* [↑](#footnote-ref-2)
3. ESB : Entreprise Service Bus. [↑](#footnote-ref-3)
4. JSON : *JavaScript Object Notation.* [↑](#footnote-ref-4)
5. API: *Application Programming Interface.* [↑](#footnote-ref-5)
6. REST: *Representational State Transfer.* [↑](#footnote-ref-6)
7. Site Officiel ES : http://www.elastic.co/guide/index.html [↑](#footnote-ref-7)
8. Convert CSV : http://www.convertcsv.com/csv-to-json.htm [↑](#footnote-ref-8)
9. W3school : http://www.w3schools.com/angular/default.asp [↑](#footnote-ref-9)
10. Exemples : http://nvd3.org/examples/index.html [↑](#footnote-ref-10)
11. *Typeahead* : "tapez en avance". [↑](#footnote-ref-11)