



Introduction aux microservices, vert.x et
Elasticsearch



idriss.neumann@comwork.io



Comwork.io SASU

128 rue de la Boétie 75008 Paris SIRET : 83875798700014

Comwork



www.comwork.io





Au programme

❖ Présentation d'Elasticsearch

- Un moteur passe-partout et multi-fonctions
- Caractéristiques
- Utilisation dans le cadre des moteurs de recherche uprodit.com

❖ Vert.x et les microservices

- Définitions
- Benchmark et performances avec NodeJS
- Créer, lancer et tester des noeuds vert.x

❖ Injection de dépendance

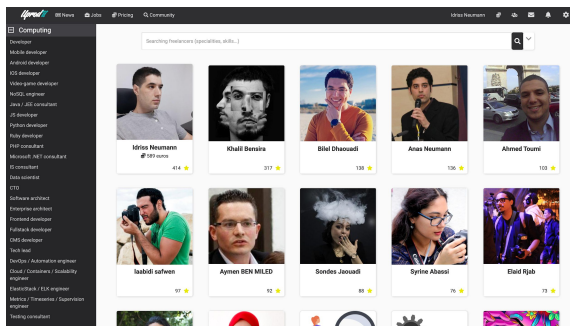
- Définition de CDI, JSR 299 / 346 vs JSR 330
- Les principales implémentations
- Implémentation avec Vert.x et guice

❖ Démonstration

❖ Conclusion / questions

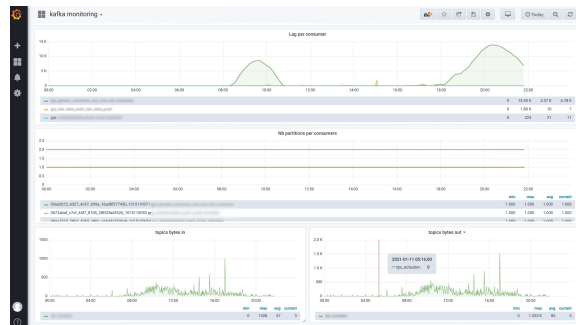
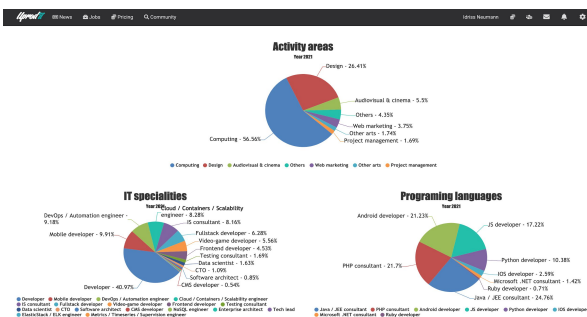
Présentation d'Elasticsearch

Un moteur passe-partout et multi-fonctions

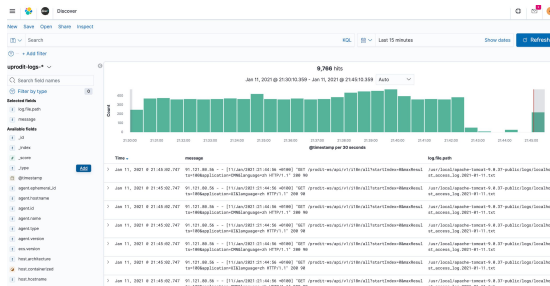


Moteur de recherche fulltext et bdd orientée document pour des applications web ou lourdes

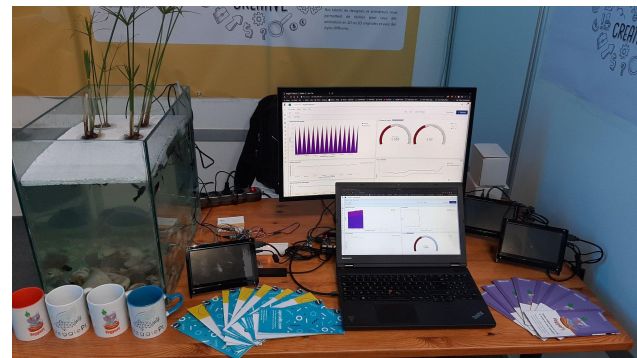
Statistiques métiers



Analyse de métriques remontée par des plugins collectés dans Elasticsearch et monitorée via grafana



Recherche et découverte de logs dans Kibana, remontés via FileBeat dans Elasticsearch



Utilisation comme une bdd timeseries pour des données remontées par des capteurs IoT

Utilisation de Kibana pour présenter les données des différents capteurs

Règles d'alerting sur Telegram et Slack en cas d'incident relevés par ces capteurs

Présentation d'Elasticsearch

Caractéristiques

A l'instar d'Apache SolR, il s'agissait d'un moteur d'indexation et de recherche basé sur Apache Lucene.

Aujourd'hui il est considérée comme une véritable base de données NoSQL orientée document pour les raisons suivantes :

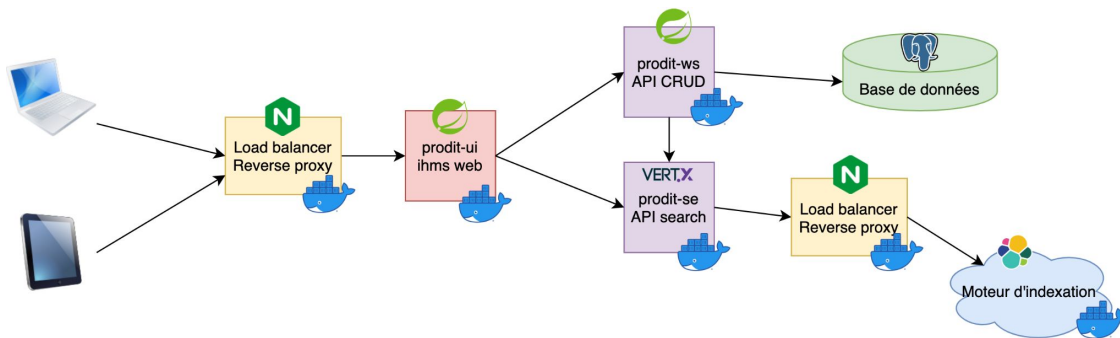
- ❖ Mapping naturel des documents JSON sans spécifier de schéma (à l'inverse d'Apache SolR)
- ❖ Scalabilité (possibilité de travailler avec plusieurs noeuds afin de mieux répartir la charge)

Autres caractéristiques :







- ❖ API RESTful comme pour Apache SolR
- ❖ Accès direct via des sockets TCP IP par l'intermédiaire d'une API Java simple d'utilisation
- ❖ Écosystème développé autour avec des outils tels que des ETL (Logstash), agents de collectes (beats) et visualisateur de données (Kibana)

Présentation d'Elasticsearch

Utilisation dans le cadre des moteurs de recherche uprodit.com

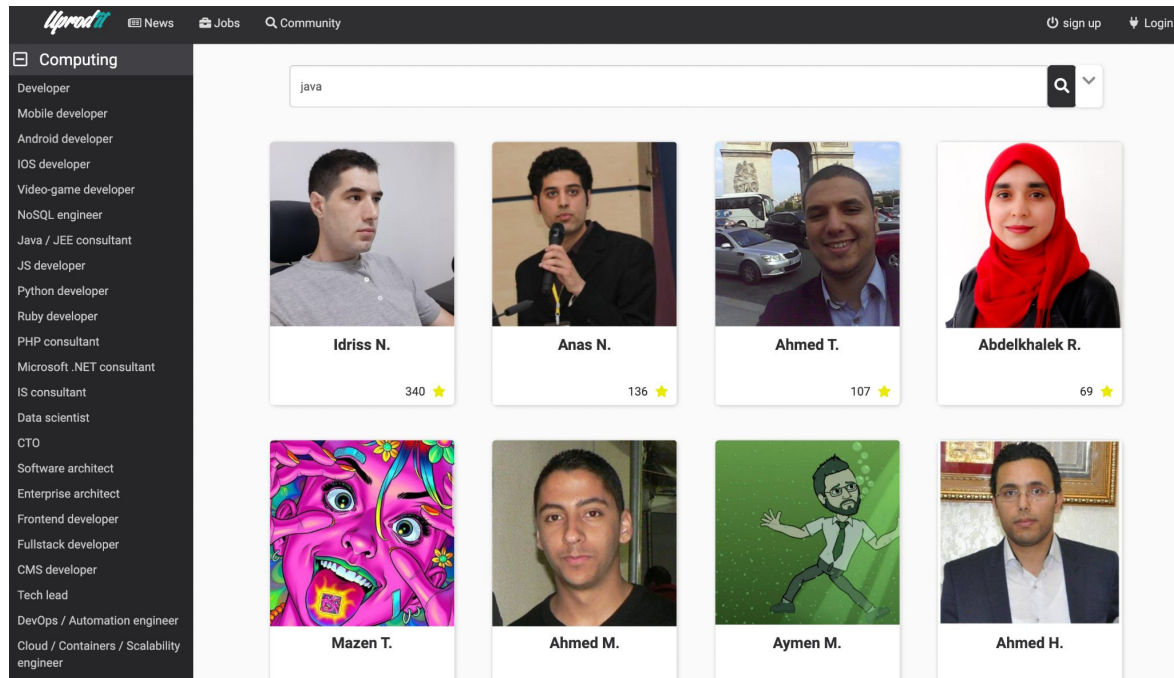


Légende

-  PostgreSQL - Système de gestion de base de données relationnelles
-  Elasticsearch - moteur d'indexation / base de données noSQL document
-  Vert.x - Framework Java pour microservices asynchrones
-  Spring - Framework Java pour applications webs synchrones
-  Nginx - reverse proxy et load balancer (répartition de charge + chiffrement TLS)
-  Docker - conteneurisation des composants (portabilité et scalabilité)

Challenge utilisation des API de uprodit.com

Faites la plus belle application de recherche de profils en utilisant l'API de uprodit.com



Envoyez votre github et lien de l'application hébergée sur challenge@comwork.io

Vert.x et microservices

Définition de l'architecture en microservices

En informatique, les microservices est un style d'architecture logicielle à partir duquel un ensemble d'applications est décomposé en petits modules, souvent avec une seule responsabilité. Les processus indépendants communiquent les uns avec les autres en utilisant des technologies d'architectures réparties, telles que des API RESTful, Grpc ou GraphQL ou encore des workers/consumers asynchrones. Ces services sont petits, hautement indépendants et se concentrent sur la résolution d'une petite tâches découpées.

Les avantages sont les suivants :

- ❖ lors d'un besoin critique en une ressource, seul le microservice lié sera augmenté, contrairement à la totalité de l'application dans une architecture n-tiers classique
- ❖ le plus souvent, on ne fait que déployé de nouveaux microservices sans interruptions des microservices déjà en production

Les inconvénients sont les suivants :

- ❖ Le périmètre des applications déployées grossit très vite
- ❖ Demande plus de compétences pour maintenir une telle architecture lié à la taille de son périmètre

Vert.x et microservices

Définition du modèle d'acteur

Modélisation informatique considérant des acteurs comme seules primitives permettant de faire de la programmation concurrente et asynchrone. C'est un modèle dans lequel il n'y a pas d'état partagé (stateless), les acteurs sont isolés et l'information ne peut circuler que sous forme de messages.

Un acteur est une entité capable de calculer, qui, en réponse à un message reçu, peut parallèlement :

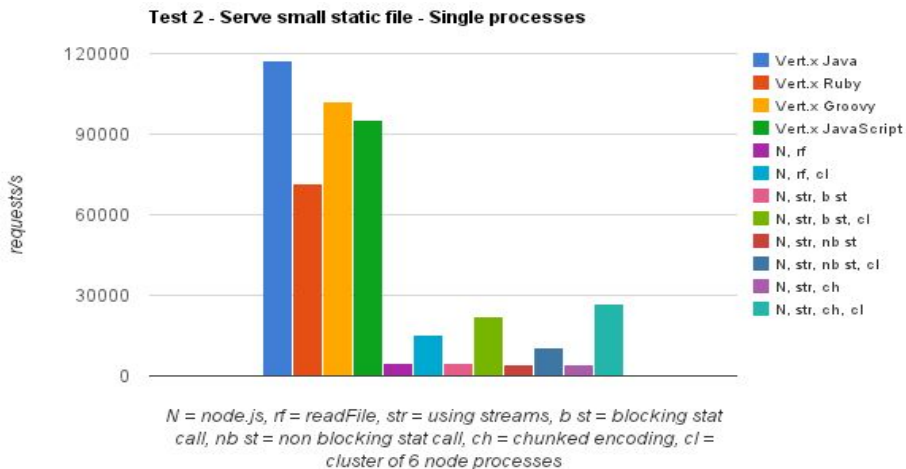
- ❖ envoyer des messages à d'autres acteurs ;
- ❖ créer de nouveaux acteurs ;
- ❖ spécifier le comportement à avoir lors de la prochaine réception de messages.

Les destinataires des messages sont identifiés à l'aide d'adresses. Un acteur doit connaître l'adresse de l'acteur à qui il veut envoyer un message. Les adresses des acteurs créés sont connues de l'acteur parent. Les adresses peuvent être échangées par message.

Les verticles (ou noeuds vert.x) sont des acteurs qui s'échangent des messages en suivant ce modèle au niveau de l'évent bus.

Vert.x et microservices

Benchmark performances avec Node.JS



Source : <http://www.cubrid.org/blog/dev-platform/inside-vertx-comparison-with-nodejs/>

Autre benchmark : <https://www.techempower.com/benchmarks/#section=data-r8&hw=i7&test=plaintext>

Benchmark plus récent à Devovx en 2019: <https://youtu.be/2lzvsyoooTk>

Vert.x et microservices

Benchmark performances avec Node.JS

En 2016, nous avons quand même voulu vérifier un minimum par nous même.

Sources disponibles ici : https://gitlab.comwork.io/comwork_training/microservices-vertx-elasticsearch

Résultat :

```
$ ./run.sh
Node test
<html><body><h1>Great BBL !</h1></body></html>
samedi 13 février 2016, 15:35:25 (UTC+0100)
samedi 13 février 2016, 15:55:04 (UTC+0100)
Vertx test
<html><body><h1>Great BBL !</h1></body></html>
samedi 13 février 2016, 15:55:04 (UTC+0100)
samedi 13 février 2016, 16:14:18 (UTC+0100)
```

Conclusion : Vert.x toujours plus rapide après plusieurs essais mais différence pas flagrante (moins d'une minute de différence pour 200000 requêtes HTTP chacun). Cela permet quand même de prouver que Vert.x tient le challenge sur les perfs.

Vert.x et microservices

Créer un projet vert.x maven

Nous nous baserons sur les versions suivantes pour cette formation:

- ❖ JDK 1.8
- ❖ Vert.x 4.2.2

```
<dependency>
  <groupId>io.vertx</groupId>
  <artifactId>vertx-core</artifactId>
  <version>${vertx.version}</version>
</dependency>
<dependency>
  <groupId>io.vertx</groupId>
  <artifactId>vertx-web</artifactId>
  <version>${vertx.version}</version>
</dependency>
```

```
<dependency>
  <groupId>junit</groupId>
  <artifactId>junit</artifactId>
  <version>${junit.version}</version>
  <scope>test</scope>
</dependency>
<dependency>
  <groupId>io.vertx</groupId>
  <artifactId>vertx-unit</artifactId>
  <version>${vertx.version}</version>
  <scope>test</scope>
</dependency>
```

Vert.x et microservices

Créer un verticle

Voici le code pour créer un verticle permettant d'afficher un "Trop cool cette formation" lorsque l'on fait la requête : `http://localhost:9980`

```
public class HelloWorld extends AbstractVerticle {  
    @Override  
    public void start() throws Exception {  
        vertx.createHttpServer()  
            .requestHandler(req -> req.response()  
                .putHeader("content-type", "text/html")  
                .end("<html><body><h1>Trop cool cette formation!</h1></body></html>"))  
            .listen(9980);  
    }  
}
```

Vert.x et microservices

Créer des routes

```
@Override
public void start() throws Exception {
    final Router router = Router.router.vertx();

    router.get("/youhou/")
        .handler(req -> req.response()
            .putHeader("content-type", "text/html")
            .end("<html><body>Youhou</body></html>"));
}
```

```
router.get("/youpi/")
    .handler(req -> req.response()
        .putHeader("content-type", "text/html")
        .end("<html><body>youpi</body></html>"));

vertx.createHttpServer()
    .requestHandler(router::handle)
    .listen(8080);
}
```

Vert.x et microservices

Déployer plusieurs verticles au sein d'une application vert.x

```
@Override
public void start() throws Exception {
    System.out.println("Starting the main verticle");
    vertx.deployVerticle("com.bblvertx.Verticle1");
    vertx.deployVerticle("com.bblvertx.Verticle2");
}
```

Vert.x et microservices

Échanges entre verticles via l'évent bus

```
// Envoi
EventBus eb = vertx.eventBus();
eb.send("verticle adresse", "Salut, tu va bien ?");

// Réception
Handler<Message<String>> aHandler = message -> {
    System.out.println("Message reçus : " + message.body());
    message.reply("Salut, bien et toi ?");
};

eb.registerHandler("verticle adresse", myHandler);
```

Vert.x et microservices

Lancer un verticle

1. Avec le serveur vert.x basé sur Netty téléchargeable ici : <https://vertx.io/download/>

```
vertx -version  
vertx run MyVerticle.java
```

2. Avec votre IDE :

```
Main class : io.vertx.core.Launcher  
Argument : run com.bblvertx.MyVerticle
```

3. Avec un fat jar buildé via maven (cf : exemple d'application avec ElasticSearch)

Injection de dépendance avec Google Guice

Installation avec maven

Github: <https://github.com/google/guice>

La dépendance Maven :

```
<dependency>
  <groupId>com.google.inject</groupId>
  <artifactId>guice</artifactId>
  <version>${guice.version}</version>
</dependency>
```

Injection de dépendance avec Google Guice

Modules permettant de déclarer des singletons

Les modules sont équivalents aux contextes d'injections de dépendance au sens Spring. Ils permettent de déclarer des singletons (équivalents des beans au sens Spring) :

```
import javax.inject.Singleton;
import com.google.inject.AbstractModule;

public class MyModule extends AbstractModule {
    private Vertx vertx;

    public MyModule(Vertx vertx) {
        this.vertx = vertx;
    }

    @Override
    protected void configure() {
        // bind(interface du service)
        // .to(implémentation du service)
        // .in(scope du service)

        // Exemples
        bind(Vertx.class).toInstance(vertx);
        bind(IPropertyReader.class).to(PropertyReaderImpl.class).in(Singleton.class);
    }
}
```

Injection de dépendance avec Google Guice

Initialisation du module et injection des dépendances dans le verticle

```
public class MyVerticle extends AbstractVerticle {  
    private MyModule myModule;  
  
    @Inject  
    private IPropertyReader reader;  
  
    @Override  
    public void start() throws Exception {  
        myModule = new MyModule(vertx);  
        Injector injector = Guice.createInjector(myModule);  
        injector.injectMembers(this);  
  
        // ...  
  
        vertx.createHttpServer().requestHandler(router::handle).listen(reader.getInt( APP_CONFIG_FILE,  
KEY_PORT));  
    }  
}
```

Démonstration

Démonstration composant avec vert.x, guice et Elasticsearch

Cette démo va démontrer un composant d'intégration avec Elasticsearch développé en vert.x qui :

- ❖ Permet de récupérer les données de deux datasources primaires PostgreSQL et Cassandra et les indexer dans Elasticsearch en asynchrone
- ❖ Permet d'exposer des endpoints Restful de recherche multicritères dans Elasticsearch

Le code de la démo est disponible ici: https://gitlab.comwork.io/comwork_training/microservices-vertx-elasticsearch

Conclusion / questions

Merci, avez-vous des questions ?

