BBL Microservices avec Vert.X, CDI, ElasticSearch

Spring et Node.JS n'ont qu'à bien se tenir !

Par Idriss Neumann, le 18/02/2016

Sommaire

- Vert.x et microservices
  - Définitions
  - Benchmark performances avec Node.JS
  - Créer, lancer et tester des noeuds vert.x (verticles)
- Injection de dépendance avec HK2
  - Définition de CDI, JSR 299 / 346 vs JSR 330
  - Les principales implémentations
  - Implémentation avec Vert.x et HK2
- Exemple d'application avec ElasticSearch
  - Rappels sur ElasticSearch
  - Démonstration
- Conclusion / Questions

Vert.x et microservices

Définition vert.x

Framework Java sous Licence Apache conçu pour faciliter la création d'applications asynchrones et événementielles, tout en fournissant des performances (vitesse, mémoire, threads) excellentes. Vert.x est donc une bonne solution pour l'écriture de microservices à l'instar de Node.JS ou Akka.
Voici les principales caractéristiques que propose Vert.x :

- **Polyglotte** : on peut écrire dans plusieurs langages, dont Java, JavaScript, CoffeeScript, Ruby, Python, Groovy, Scala... On peut mélanger plusieurs de ces langages au sein d'une même application. C'est Vert.x qui se charge de tout ;
- **Simple** : Vert.x permet d'écrire des applications non bloquantes sans nécessiter des compétences avancées ;
- **Scalable** : Vert.x utilise un système de messages entre les verticles, ce qui lui permet d'être plus efficace dans la gestion des CPU ;
- **Concurrent** : étant très proche d'un modèle d'acteurs et/ou microservice, l'API permet de se concentrer sur le métier, sans se préoccuper des contraintes et problèmes des architectures multithread.

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, très souvent des API REST. Ces services sont petits, hautement indépendants et se concentrent sur la résolution d'une petite tâche.

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 :

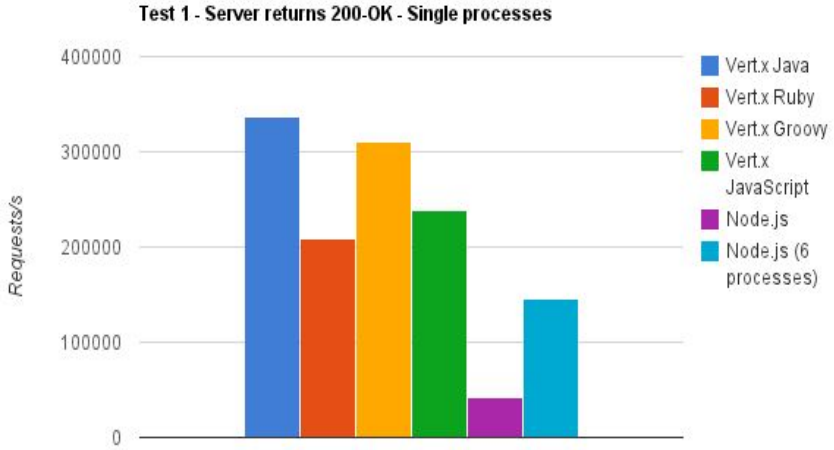
- envoyer des messages à d'autres acteurs ;
- créer de nouveaux acteurs ;
- spécifier le comportement à avoir lors de la prochaine réception de messages.

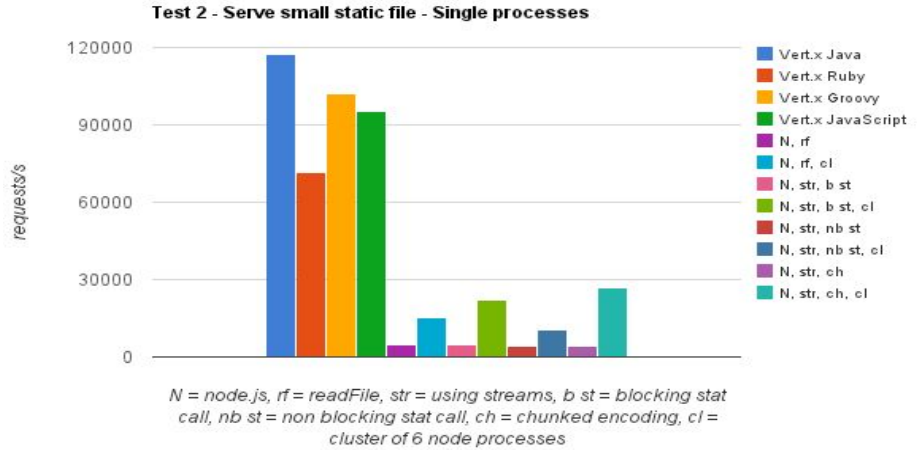
Les destinataires des messages sont identifiés à l'aide d'adresses. Un acteur doit connaître l'adresse de l'acteur à qui il veut envoyer un message. Les adresses des acteurs créés sont connues de l'acteur parent. Les adresses peuvent être échangées par message.

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

Vert.x et microservices

Benchmark performances avec Node.JS

Nous avons quand même voulu vérifier un minimum par nous même.

Sources disponibles ici : https://github.com/idrissneumann/vertx-vs-node

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 ce BBL :

- JDK 1.8
- Vert.x 3.0.0

```
<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 ce BBL" 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 ce BBL !</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::accept)
        .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");
}
```

Echanges entre verticles via l'event 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://bintray.com/vertx/downloads/distribution/view

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

2) Avec votre IDE :

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

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

Vert.x et microservices

Tester un verticle

```
import org.junit.runner.RunWith;
import io.vertx.ext.unit.junit.VertxUnitRunner;

@RunWith(VertxUnitRunner.class)
public class HelloWorldTest {
  @Before
  public void before(final TestContext context) {
    vertx = Vertx.vertx();
    vertx
    .deployVerticle(MyVerticle.class.getName(),
     context.asyncAssertSuccess());
  }
  @After
  public void after(final TestContext context) {
    vertx.close(context.asyncAssertSuccess());
  }
```

```
  private static final int PORT = 8080;
  private static final String HOST = "localhost";
  private static final String PATH = "/youhou/";

  private final static String EXPECTED_MESSAGE =
"<html><body>youhou</body></html>" ;

  @Test
  public void testHelloWorld(final TestContext context) {
    final Async async = context.async();

    vertx.createHttpClient()
        .getNow(PORT, HOST, PATH, response -> {
            response.handler(body -> {
                context.assertTrue(body.toString()
                        .contains(EXPECTED_MESSAGE));
                async.complete();
            });
        });
  }
}
```

Injection de dépendance avec HK2

Définition CDI, JSR 249/346 vs JSR 330

CDI (Context and Dependency Injection) est une spécification Java destinée à standardiser l'injection de dépendances et de contextes d'objets.
Intégrée à la spécification JEE 6, sa version 1.0 est sortie en décembre 2009 et a été suivie des versions 1.1 (mai 2013) et 1.2 (avril 2014).
Son élaboration est le résultat des JSR 299 et 346.

Les implémentations de cette normes sont:

- Weld : implémentation de JBOSS qui est l'implémentation de référence
- OpenWebBeans : implémentation d'Apache
- Caucho CanDI

Les annotations du package javax.inject, utilisées dans CDI sont quant à elles normalisées par la JSR 330 et on les retrouves dans d'autres implémentations telles que :

- Guice : implémentation de Google
- HK2 : implémentation de Glassfish
- Dagger : http://square.github.io/dagger/
- Spring : eh oui, depuis Spring 3.0

Principales annotations :

- `@Inject` équivalent de `@Autowired` avec Spring permettant d'injecter une dépendance (ou bean)
- `@Named` équivalent de `@Component` avec Spring permettant de créer un bean nommé
- `@Singleton` pour définir le scope du bean comme étant un singleton

Injection de dépendance avec HK2

Implémentation avec Vert.x et HK2

La dépendance Maven :

```
<dependency>
    <groupId>com.englishtown.vertx</groupId>
    <artifactId>vertx-hk2</artifactId>
    <version>${vertx-hk2.version}</version>
</dependency>
```

A notez que englishtown fournis de nombreuses extensions à Vert.x dont celle-ci qui facilite énormément la mise en place d'un conteneur d'injection dépendance, mais beaucoup d'autres comme une intégration simplifiée à ElasticSearch, une surcouche Jersey pour faciliter la mise en place de contrats d'interfaces Restful...

Injection de dépendance avec HK2

Implémentation avec Vert.x et HK2

Le binder permettant de déclarer des beans :

```
import javax.inject.Singleton;
import org.glassfish.hk2.utilities.binding.AbstractBinder;
import com.bblvertx.singleton.PropertyReader;

public class MyBinder extends AbstractBinder {
    @Override
    protected void configure() {
        // bind(implémentation du service)
        // .to(contrat de service)
        // .in(scope du service)

        // Exemple
        bind(PropertyReader.class).to(PropertyReader.class).in(Singleton.class);
    }
}
```

Injection de dépendance avec HK2

Implémentation avec Vert.x et HK2

Faire un fichier de configuration pour indiquer le binder :

```
{
    "hk2_binder": "com.bblvertx.MyBinder"
}
```

Au lancement du verticle, passer comme argument :

```
run java-hk2:com.bblvertx.MyVerticle -conf src/main/resources/config.json
```

Il ne vous reste plus qu'à faire des `@Inject` dans votre verticle ou dans vos singletons.

Application avec ElasticSearch

Rappels ElasticSearch

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'intérmédiaire d'une API Java simple d'utilisation
- Ecosystème dévoloppé autour avec des outils tels que des ETL (Logstash) et visualisateur de données (Kibana, Marvel, Sense)

Application avec ElasticSearch

Démonstration

Disponible sur Github : https://github.com/idrissneumann/vertx-cdi-elasticsearch-example

Suivre les indications sur le wiki pour la faire fonctionner sur un environnement local.

Questions

Merci, avez vous des questions ?

Sources

Les sources ayant servi à faire ce BBL :

- Tutoriel Vert.x 3 de Thierry Leriche Dessirier sur Developpez.com : http://thierry-leriche-dessirier.developpez.com/tutoriels/java/vertx/creer-lancer-tester-verticle/
- Vert.x / HK2 (CDI) sur Github : https://github.com/englishtown/vertx-hk2
- Application blanche composant d'intégration avec Elastic Search par Idriss Neumann sur github : https://github.com/idrissneumann/vertx-cdi-elasticsearch-example
- Site officiel de Vert.x : http://vertx.io/
- Benchmark cubird : http://www.cubrid.org/blog/dev-platform/inside-vertx-comparison-with-nodejs/
- Benchmark techempower : https://www.techempower.com/benchmarks/#section=data-r8&hw=i7&test=plaintext