Migwel

CC BY NC SA

Zeste de savoir

Expérimentation de Spring Boot et Podman sous WSL

3 novembre 2022

Table des matières

Introduction

Dernièrement, je travaillais sur une application Spring Boot qui nécessitait une communication entre plusieurs instances. Pour ce faire, j'ouvrais plusieurs terminaux et je lançais une commande Maven avec des arguments différents. Ce procédé m'a rapidement lassé et je me suis dis qu'il était temps d'utiliser une approche un peu plus moderne et je me suis essayé à Podman.

Dans ce billet, je vais partager dans ce billet. À la fois comme un pense-bête pour moi-même mais aussi en espérant que ça puisse servir à certains.

Ce billet est réellement une sorte d'approche rapide et naïve à Podman, ne le prenez pas comme une référence. Il est possible, voire probable, que je partage certaines méthodes qui ne sont pas du tout recommandées. En particulier, je partage ici une procédure que j'ai utilisé pour le développement d'un projet jouet, sur ma machine personnelle. Cette approche n'est pas adaptée pour une application en production par exemple.

1. Les conteneurs et Podman

La plupart d'entre vous sont sans doute familier au concept de conteneur mais pour ceux qui ne le seraient pas, voici comment un conteneur est défini par Wikipedia :

> un conteneur d'application est une architecture logicielle qui permet, sur un serveur informatique ou une grappe, d'isoler le fonctionnement d'un programme, donnant au responsable de son exécution, l'impression qu'il s'exécute dans un environnement dédié, ce qu'on appelle une virtualisation.

Ainsi, bien qu'utilisant WSL, en utilisant un système de conteneurisation, nous pouvons faire tourner des applications dans des conteneurs Debian ou Fedora et nos applications n'y verront

que du feu. L'avantage, c'est que tout ce que nous avons à définir, c'est le contenu de notre conteneur. Ensuite, nous pourrons le faire tourner sur n'importe quel système d'exploitation supportant Podman/Docker, sans aucun impact sur l'application et son environnement.

Docker est actuellement le moteur de conteneurisation le plus populaire mais de mon côté, j'ai décidé d'utiliser Podman, une alternative à Docker. La raison très simple pour laquelle j'ai opté pour Podman plutôt que Docker est que j'ai rencontré des soucis lorsque j'ai essayé d'utiliser Docker sous WSL et que j'avais envie de m'essayer à quelque chose de nouveau.

Mais dans les faits, podman se comporte et s'utilise de façon très similaire à Docker. Par exemple, Podman est capable de lire les Dockerfile pour construire nos images et la plupart des commandes sont très similaires à celles de Docker. Une des différences majeures est que Podman ne nécessite pas de démon pour faire tourner nos conteneurs.

2. Installation et configuration de podman

Mon ordinateur est sous Windows mais pour tout ce qui est développement, je préfère utiliser WSL et Debian. Je ne me souviens pas des détails mais il me semble qu'avec WSL version 1, il y avait des soucis avec Podman donc pour la suite de ce billet, sachez que j'utilise WSL version 2.

Pour l'installation de Podman, c'est relativement simple :

```
1 # apt-get install podman
```

Cette installation va créer des fichiers dans `/etc/containers/` que vous pouvez configurer comme vous le souhaitez. De mon côté, ce que j'ai fait, c'est copier les fichiers registries.conf et policy.json de Fedora.

En utilisant WSL, il y a des subtilités supplémentaires qui sont que si vous lancez des commandes podman, vous verrez des erreurs étranges du type:

```
1 unable to write pod event: "write unixgram
    @00017->/run/systemd/journal/socket: sendmsg: no such file or
    directory"
```

Pour régler ceci, j'ai suivi ces instructions et copié un fichier containers.conf

```
1 $ cp /usr/share/containers/containers.conf
    ~/.config/containers/containers.conf
```

Puis dans ce fichier fraichement créé, j'ai ajouté les lignes suivantes :

```
cgroup_manager = "cgroupfs"
events_logger = "file"
```

Maintenant que podman est installé, nous allons essayer d'exécuter quelques commandes pour voir ce que ça donne.

3. Programmons notre application avec Spring Boot

Je ne vais pas utiliser ici l'application sur laquelle je travaillais car elle est assez complexe et elle fera peut-être l'objet d'un billet dans le futur. Nous allons donc utiliser une application simplifiée mais qui nécessitera tout de même une communication entre instances.

Ce que nous allons donc faire est programmer une application web appelée Friends que nous pourrons appeler pour connaitre les statuts d'amitié entre les instances. Quand on lancera une instance, on lui donnera un numéro et deux instances seront amies si leur numéro a la même parité. Oui, je sais, c'est débile comme application mais c'est pour l'exemple.

```java
public record InstanceInformation(int id) {}
```

InstanceInformation.java

```java
@Controller
public class FriendshipController {

    private final InstanceInformation instanceInformation;
    private final HttpClient httpClient;

    public FriendshipController(InstanceInformation
        instanceInformation, HttpClient httpClient) {
        this.instanceInformation = instanceInformation;
        this.httpClient = httpClient;
    }

    @GetMapping("/friends")
    @ResponseBody
    public boolean friends(@RequestParam int otherInstancePort)
        throws IOException, InterruptedException {
        HttpRequest internalRequest = HttpRequest.newBuilder()
                .uri(URI.create("http://localhost:"+
                    otherInstancePort
                    +"/internalfriends?otherInstancePort="+
                    instanceInformation.id()))
                .build();
```

```
18          HttpResponse<String> httpInternalResponse =
                httpClient.send(internalRequest,
                HttpResponse.BodyHandlers.ofString());
19          return Boolean.parseBoolean(httpInternalResponse.body());
20      }
21
22      @GetMapping("/internalfriends")
23      @ResponseBody
24      public boolean internalFriends(@RequestParam int
            otherInstancePort) {
25          return instanceInformation.id() % 2 == otherInstancePort %
                2;
26      }
27  }
```

FriendshipController.java

```
1   @SpringBootApplication
2   public class FriendsApplication {
3
4       public static void main(String[] args) {
5           SpringApplication.run(FriendsApplication.class, args);
6       }
7
8       @Bean
9       public InstanceInformation
            instanceInformation(@Value("${instanceId}") String
            instanceIdStr) {
10          if (instanceIdStr == null) {
11              throw new
                    RuntimeException("Wrong instance ID provided");
12          }
13          int instanceId = Integer.parseInt(instanceIdStr);
14          return new InstanceInformation(instanceId);
15      }
16
17      @Bean
18      public HttpClient httpClient() {
19          return HttpClient.newBuilder()
20                  .version(HttpClient.Version.HTTP_2)
21                  .build();
22      }
23  }
```

FriendsApplication.java

Nous avons 3 classes :

— InstanceInformation qui contient simplement le numéro qu'on attribue à cette instance.

- FriendshipController qui est notre endpoint. Il contient deux méthodes :
  - `friends` que l'on appellera pour savoir si une autre instance (définie par le port sur lequel celle-ci écoute) est amie
  - `internalfriends` que notre application utilisera en interne pour communiquer avec l'autre instance et lui demander s'ils sont amis
- FriendsApplication qui est nécessaire pour le bon fonctionnement de Spring Boot et où l'on définit quelques Beans que notre application utilise.

Maintenant nous pouvons tester tout ceci pour vérifier que ça fonctionne.

```
java -jar target/Friends-0.0.1-SNAPSHOT.jar --server.port=8080
    --instanceId=4
java -jar target/Friends-0.0.1-SNAPSHOT.jar --server.port=8081
    --instanceId=6
```

Nous lançons deux instances. L'une avec le numéro 4 utilisant le port 8080 et l'autre avec le numéro 6 utilisant le port 8081. Nous pouvons maintenant leur demander si elles sont amies.

```
$ curl -X GET http://localhost:8080/friends?otherInstancePort=8081
true
```

TADA, ça fonctionne exactement comme nous le souhaitons. La prochaine étape est donc de faire tourner notre application dans un conteneur Podman.

4. Construisons une image de notre application avec podman build

Notre application maintenant codée et compilée, nous allons pouvoir créer une image que nous pourrons ensuite faire tourner dans un conteneur. Pour ce faire, nous allons ajouter un fichier Dockerfile à notre project.

```
FROM openjdk:17-alpine
COPY target/Friends-0.0.1-SNAPSHOT.jar Friends.jar
ENTRYPOINT ["java","-jar","/Friends.jar"]
```

Comme nous l'avons mentionné lors de la rapide présentation de Podman, nous pouvons voir que Podman est effectivement compatible avec les concepts propres à Docker tel que le Dockerfile. Dans ce fichier, nous ne faisons rien de bien compliqué :

- `FROM openjdk:17-jdk-alpine` précise quelle image va servir de base à notre image. Comme nous souhaitons faire tourner une application codée en Java 17, nous allons utiliser openjdk:17-alpine (il existe d'autres images qui auraient aussi fait l'affaire, `amazoncorretto:17-alpine-jdk` par exemple)

— `COPY target/Friends-1.0.0-SNAPSHOT.jar Friends.jar` précise que nous allons copier notre exécutable dans l'image que nous créons
— Et enfin, `ENTRYPOINT ["java","-jar","Friends.jar"]` précise la commande à exécuter lorsque notre image sera lancée dans un conteneur.

Pour construire notre image, nous allons utiliser Podman de la façon suivante :

```
$ podman build -t friends .
```

Nous pouvons vérifier que l'image a bien été créée convenablement en utilisant `podman image` :

```
$ podman image list
REPOSITORY                    TAG         IMAGE ID      CREATED
        SIZE
localhost/friends             latest      db02afa9fa3a  5 seconds
    ago  344 MB
```

5. Lançons plusieurs instances de nos applications dans des conteneurs

Nous avons maintenant Podman installé sur notre système, notre application est codée et nous avons construit une image de celle-ci, tout ce qu'il nous reste à faire, c'est de lancer celle-ci dans un conteneur.

Pour lancer une image dans un conteneur avec Podman, nous pouvons exécuter la commande suivante :

```
$ podman run -d -p 8080:8080 localhost/friends --server.port=8080
    --instanceId=6
```

Jetons un œil aux arguments utilisés :

— `-d` va faire tourner le conteneur de façon détachée, c'est-à-dire en tâche de fond
— `-p` va nous permettre de faire un *mapping* entre le(s) port(s) utilisé(s) par le conteneur et ceux de notre machine. Ici, notre application va utiliser le port 8080 et on va le mapper au port 8080 de notre machine

Nous pouvons vérifier que notre instance est bien active est exécutant la commande `podman ps` :

```
$ podman ps
CONTAINER ID  IMAGE              COMMAND               CREATED
       STATUS            PORTS                   NAMES
315ec94ef51e  localhost/friends  --server.port 808...  2 minutes
    ago  Up 2 minutes ago  0.0.0.0:8080->8080/tcp
    nostalgic_hypatia
```

Ensuite, si nous voulons voir les logs du conteneur, nous pouvons utiliser soit son ID, soit nom (NAMES) :

```
$ podman logs nostalgic_hypatia




    .   ____          _            __ _ _
/\\ / ___'_ __ _ _(_)_ __  __ _ \ \ \ \
( ( )\___ | '_ | '_| | '_ \/ _` | \ \ \ \
\\/  ___)| |_)| | | | | || (_| |  ) ) ) )
'  |____| .__|_| |_|_| |_\__, | / / / /
=========|_|==============|___/=/_/_/_/
:: Spring Boot ::                (v2.7.5)
⌋
    2022-11-02 21:52:06.378  INFO 1 --- [           main] dev.migwel.friends.Fr
⌋
    2022-11-02 21:52:06.381  INFO 1 --- [           main] dev.migwel.friends.Fr
⌋
    2022-11-02 21:52:07.365  INFO 1 --- [           main] o.s.b.w.embedded.tomc
⌋
    2022-11-02 21:52:07.376  INFO 1 --- [           main] o.apache.catalina.cor
⌋
    2022-11-02 21:52:07.377  INFO 1 --- [           main] org.apache.catalina.c
⌋
    2022-11-02 21:52:07.457  INFO 1 --- [           main] o.a.c.c.C.[Tomcat].[
⌋
    2022-11-02 21:52:07.458  INFO 1 --- [           main] w.s.c.ServletWebServe
⌋
    2022-11-02 21:52:07.987  INFO 1 --- [           main] o.s.b.w.embedded.tomc
    '
2022-11-02 21:52:07.997  INFO 1 --- [           main]
    dev.migwel.friends.FriendsApplication    : Started
    FriendsApplication in 2.026 seconds (JVM running for 2.447)
```

Nous pouvons maintenant lancer une autre instance qui utilisera le port 8081.

```
$ podman run -d -p 8081:8081 localhost/friends --server.port=8081
    --instanceId=8
```

Mais si nous essayons de faire un appel vers une de ces instances, nous allons voir l'erreur suivante :

```
miguel@LAPTOP-Q5GPM3K9:~$ curl -X GET
    http://localhost:8080/friends?otherInstancePort=8081
{"timestamp":"2022-11-02T21:59:23.379+00:00","status":500,"error":
    "Internal Server Error","path":"/friends"}
```

Et en inspectant les logs de notre premier conteneur, nous pouvons voir que cette erreur vient du fait qu'il ne parvient pas à faire appel au deuxième conteneur. Quand podman lance ces conteneurs, ils sont isolés au niveau du réseau donc lorsque notre premier conteneur fait un appel vers http://localhost:8081, cet appel échoue car du point de vue de ce conteneur, rien n'écoute sur le port 8081.

Mais qu'à cela ne tienne, podman nous offre une possibilité pour pallier ce problème: les pods.

6. Les pods à la rescousse

Un pod est un ensemble dans lequel nous pouvons faire tourner plusieurs conteneurs. Et l'avantage de cette approche est que tous les conteneurs présents dans le même pod partagent le même namespace, ce qui va nous permettre des faire des appels réseaux en utilisant localhost.

Donc allons-y, pour créer un pod, rien de plus simple, nous pouvons exécuter la commande suivante :

```
$ podman pod create -p 8080-8081:8080-8081
```

```
$ podman pod ps
POD ID        NAME                STATUS   CREATED         INFRA
    ID      # OF CONTAINERS
adddca3a85d1  awesome_sutherland  Created  49 seconds ago
    21f9a101d477  1
```

De façon similaire à la création d'un conteneur, nous précisons le mapping de ports entre le pod et notre machine. Et dans les commandes qui viennent, nous pourrons utiliser soit l'ID du pod soit son nom.

Nous pouvons maintenant créer de nouveaux conteneurs comme précédemment, mais en leur spécifiant le pod dans lequel ils doivent tourner.

```
$ podman run --pod awesome_sutherland  -d localhost/friends
    --server.port=8080 --instanceId=6
946d1820e1cebbbab29cae179125e0ff0af6765e9bd3a9de63fa310a8812799b
$ podman run --pod awesome_sutherland  -d localhost/friends
    --server.port=8081 --instanceId=8
01717c7f272a3f54c9e420272813e99260cf65cf2343c664e41f529920bcde42
```

Et maintenant, l'heure de vérité, rééssayons de faire appel à notre application :

```
$ curl -X GET http://localhost:8080/friends?otherInstancePort=8081

                                                                true
```

Ca fonctionne! En lançant nos conteneurs dans le même pod, plus de problèmes de réseaux et on reçoit bien la réponse `true` qu'on attendait.

Conclusion

Nous voilà à la fin de notre périple. J'espère que ce billet vous aura appris quelque chose ou vous aura été utile. De mon côté, la prochaine étape sera de creuser l'utilisation de Podman dans des situations plus complexes ainsi que comment l'utiliser en production.

Le code source utilisé dans ce billet est disponible sur Github .