Tuto Docker - Comprendre Docker (Partie1)

Présentation de Docker

Docker est un produit développé par la société du même nom. Initialement développé par un ingénieur français, **Solomon Hykes,** le produit a été dévoilé en mars 2013. Depuis cette date, **Docker** est devenu le *software* à la mode ! Nous allons voir à quoi il sert et comment vous pouvez vous en servir au quotidien.

Qu'est-ce que Docker ?

Docker permet de créer des environnements (appelées containers ou conteneurs) de manière à isoler des applications. **Docker** repose sur le *kernel* **Linux** et sur une fonctionnalité : les conteneurs, que vous connaissez peut-être déjà sous le nom de **LXC**. L'idée est de lancer du code (ou d'exécuter une tâche, si vous préférez) dans un environnement isolé. Je m'arrêterai ici pour ce qui est des détails des containers Linux, pour plutôt me concentrer sur les fonctionnalités mises en avant par Docker. Enfin, un troisième composant est requis, **cgroups** qui va avoir pour objectif de gérer les ressources (utilisation de la RAM, CPU entre autres).

Est-ce Docker une VM ?

Oui… et non ! **Docker** et *les containers Linux* ne se comportent pas de la même manière qu'une **VM**. Une machine virtuelle isole tout un système (son OS), et dispose de ses propres ressources.

Dans le cas de **Docker**, le *kernel* va partager les ressources du système hôte et interagir avec le(s) container(s). Techniquement, **Docker n'est pas une VM**, mais en terme d'utilisation, **Docker** peut être apparenté à une **VM**.

Lancer un environnement, et isoler les composants de ce container avec les composants de mon hôte, voilà ce que **Docker** sait faire ! Il le fait d'ailleurs très bien, et reste une alternative bien plus performante que les **VMs** (à utilisation équivalente).

Sur VM, chaque machine virtuelle a son propre système d'exploitation, qui prend des minutes pour démarrer. Alors que sur Docker, les conteneurs sont isolés, et se partagent les

ressources de la machine hôte (OS, librairies, …). Un conteneur ne met que quelques secondes pour démarrer.

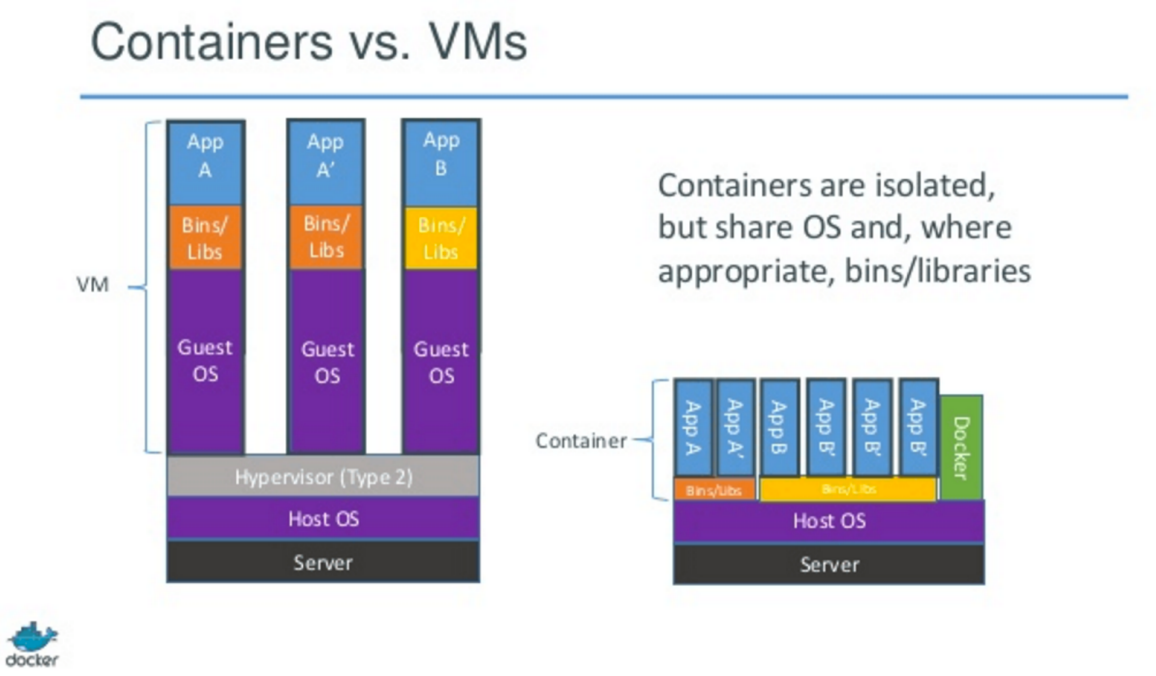

Schéma réalisé par Docker

Docker, cas d'utilisation

Prenons un cas d'utilisation simple et concret.

Une entreprise lyonnaise que j'appellerai à tout hasard **Wanadev**, souhaite développer un projet*Symfony* comme ils savent le faire. Leur équipe est composée de deux personnes.

- Manu est un vieux de la vieille. Lui, Debian 7, c'est parfait. De plus, il est ISO avec les serveurs de production.
- Baptiste lui est un vrai hippie. Il dispose de la dernière distribution exotique. Lui, **PHP**, c'est la toute dernière version ou rien. Cependant, il peut générer du code qui ne peut pas s'exécuter correctement avec une version plus vieille, comme celle de Manu.

Jusqu'à aujourd'hui, on disait tous, « faisons des tests unitaires ». **Oui**, cela répondait à une bonne partie de nos problèmes (à la condition de faire de bons tests unitaires complets).

En effet, ça pose problème que deux développeurs ne travaillent pas sur les mêmes environnements... Du coup, grâce à Docker, on peut faire en sorte que Manu et Baptiste travaillent sur les mêmes versions Linux sans craindre des problèmes de compatibilité entre leurs codes respectifs ?
Exactement ! Dans ce cas, le plus simple est de mettre en place un **Dockerfile,** document "chef d'orchestre", qui permettra à Manu et à Baptiste de monter une image similaire. En étant malin, ce **Dockerfile** sera calqué sur les éléments présents en production. De ce fait,

Manu et Baptiste en plus de travailler sur un environnement identique, seront sur un environnement similaire à celui de la production !

Docker, déployer vos applications facilement

Dans mon exemple précédent, j'ai mis en avant l'utilisation de **Docker** pour déclarer un unique environnement de développement. Je pense que tous les utilisateurs de **Docker** doivent commencer par ça avant de voir plus grand.

Plus haut, je vous parlais d'une version particulière de **PHP**. Il est évident que vous ne risquez pas d'oublier d'installer **PHP** sur votre serveur de production, mais peut-être que la configuration de votre **PHP-FPM** sera légèrement différente. Peut-être avez-vous besoin d'une librairie supplémentaire comme **Imagick** et de son extension **PHP**. Peut-être avez-vous besoin d'un **Postfix** pour envoyer des mails. Autant de détails que vous pourriez oublier lorsque vous monterez votre serveur de production.

Pour limiter ces erreurs, ne serait-il pas plus simple de déployer le ou les *container(s)* permettant de faire tourner votre application ? De cette manière, plutôt que de *pull* un code source et d'effectuer quelques scripts de déploiement, vous pourriez déployer un ensemble cohérent *packagé* correspondant à une application.

On parle souvent d'application dans le monde du web, une "application" **Symfony**, ne serait pas grand-chose sans un frontal comme **Nginx ou Apache2,** un engine comme **FPM** ou **HHVM**, et une base de donnée.

Si je peux lancer plusieurs Dockers sur ma machine, ça veut dire que je peux multiplier les configurations comme je le souhaite ?
C'est tout à fait ça ! De cette manière, vous pouvez construire des environnements autonomes et isolés les uns les autres. Le bon vieux projet **SPIP** en 5.3 pourrait côtoyer sur le même serveur le dernier projet Symfony avec PHP 5.5.

Conclusion

Grâce à Docker, multipliez les environnements sur votre machine, sans limiter les performances de votre ordinateur. Les ressources sont partagées avec la machine hôte ! Chaque environnement peut être configuré simplement grâce à son Dockerfile, présent à sa racine.

Webographie

https://www.wanadev.fr/23-tuto-docker-comprendre-docker-partie1/