Chapitre 3
Premiers pas avec Docker

1. Hello World, Docker

1.1 Démarrage d'un conteneur simple

Le sujet de l'installation de Docker étant désormais traité, il est temps de commencer enfin à explorer l'outil lui-même. Comme toute technologie informatique qui se respecte, Docker dispose d'un exemple "Hello World", soit dans notre cas un conteneur qui ne fait rien à part afficher un message de bienvenue. L'intérêt de ce genre d'approche est qu'il permet de valider que l'installation du produit s'est bien passée, de vérifier que l'ensemble de la chaîne logicielle fonctionne et de mettre le pied à l'étrier pour l'utilisateur débutant.

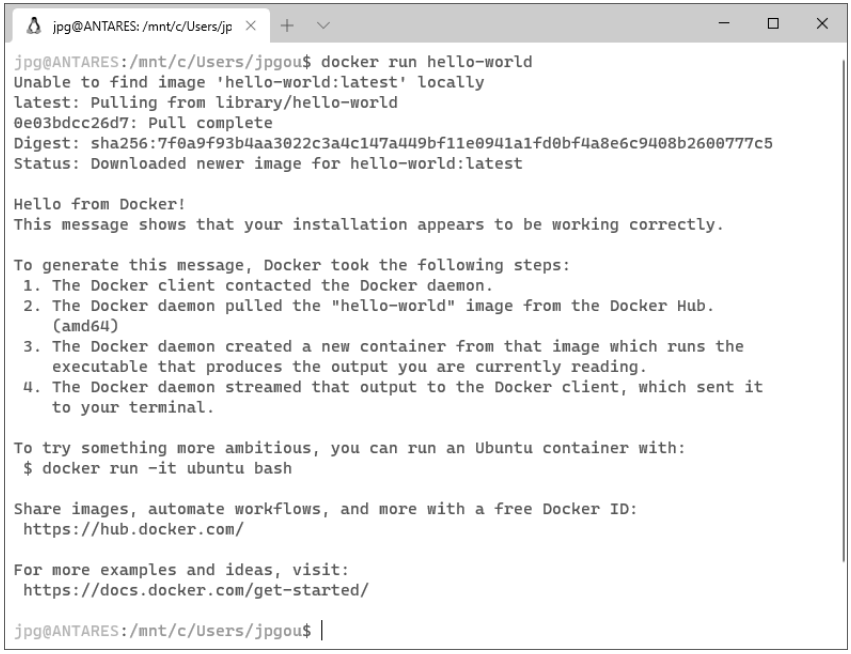
Démarrer un conteneur exemple minimaliste

```
docker run hello-world
```

Remarque

Dorénavant et pour la suite du livre, nous considérons que la gestion des droits est traitée. Ainsi, s'il est nécessaire de préfixer les commandes par le mot-clé sudo sous Linux, nous ne le montrerons plus dans les exemples. L'obtention d'un message comportant le texte "permission denied" ou une expression approchante doit donner le réflexe de rajouter le mot-clé si nécessaire ou de traiter le problème de sécurité. Dans le même souci de simplification, les exemples sont montrés sur Linux.

Sauf problème d'installation de Docker ou d'accès à Internet, l'affichage devrait être le suivant :



1.2 Détails des opérations effectuées

Bien que l'exécution de la commande ci-dessus soit très rapide, il s'est passé un certain nombre d'opérations pour arriver à ce résultat. C'est d'ailleurs la raison d'être de Docker que de rendre disponible en une seule ligne de commande un enchaînement d'opérations dont nous allons voir en les détaillant ci-dessous qu'elles sont très variées.

1.2.1 Récupération d'une image

Tout d'abord, Docker a besoin de lire l'image nommée `hello-world` qui lui a été passée en paramètre de la commande `run`.

Comme nous l'avions rapidement abordé dans l'introduction, Docker dispose d'un dépôt d'images auxquelles il peut accéder par Internet. Ce dépôt, accessible par l'URL https://registry.hub.docker.com, regroupe toutes les images "officielles" fournies par Docker. Comme il s'agit du registre d'images Docker utilisé par défaut, il est souvent désigné simplement comme "le registre", mais nous verrons un peu plus loin qu'il est possible de créer d'autres registres.

Remarque

Le registre par défaut est évidemment public, de façon que la commande ci-dessus soit la plus simple possible et n'ait pas à être alourdie par des informations d'authentification. D'autres registres peuvent être privés et réservés à telle ou telle organisation, voire hybrider ce mode d'accessibilité avec certaines ressources publiques et d'autres protégées.

Constatant que l'image n'était pas déjà présente sur la machine hôte, Docker l'a téléchargée. Il est toutefois possible de récupérer l'image au préalable.

Télécharger une image sans la lancer

```
docker pull hello-world
```

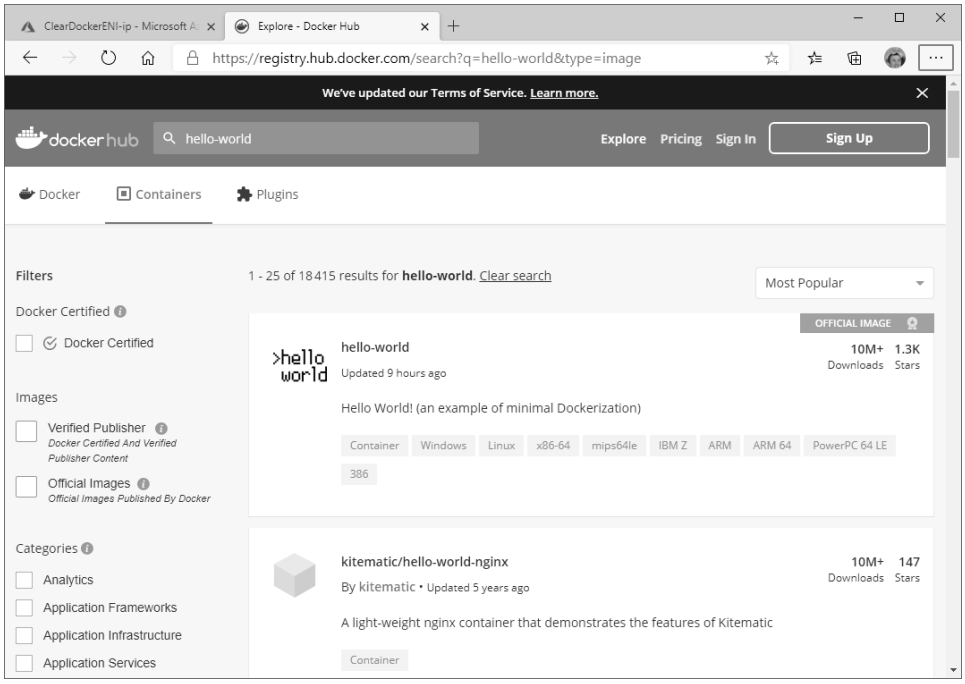
Dans le cas de l'image `hello-world`, qui pèse moins d'1 ko, la différence de temps est infime, mais pour des images plus volumineuses, il peut être particulièrement utile de séparer le téléchargement et le lancement. Dans tous les cas, l'image ne sera chargée qu'une seule fois. En effet, Docker gère un cache des images sur la machine, et le prochain démarrage d'une instance de conteneur sur la même image n'aboutira donc pas nécessairement au rechargement de celle-ci, sauf bien sûr si une instruction est fournie pour que Docker aille vérifier que la nouvelle version n'a pas changé.

Remarque

Une bonne pratique est de gérer des versions immuables des images Docker, c'est-à-dire que si une image a été publiée avec une version donnée et que des modifications ont été réalisées, il faut que la version augmente, de façon à signifier aux consommateurs cette modification. Dans le cas contraire, en ce qui concerne Docker, le client considérera qu'il possède déjà la version dans son cache et n'ira pas vérifier si la version sur le registre est différente, à moins qu'on lui demande explicitement de la recharger.

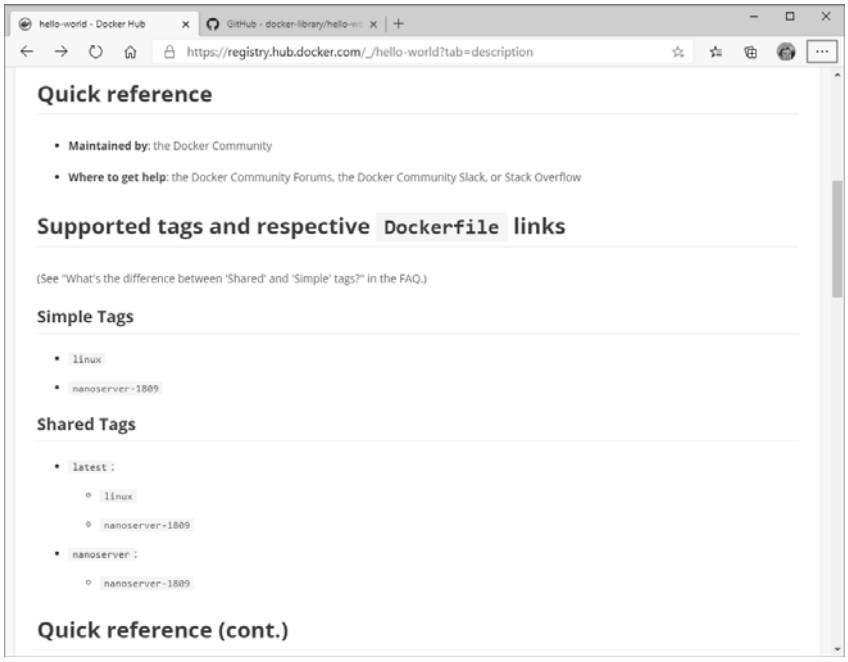
1.2.2 Identité de l'image

Une recherche sur le registre Docker permet de localiser la définition de l'image :



Dès l'affichage lié à la recherche, on constate que les images peuvent supporter plusieurs architectures informatiques et qu'`hello-world` en supporte justement un grand nombre. Il se trouve que cette image est disponible sur des plateformes Linux et Windows, mais aussi sur des systèmes MIPS 64, ARM, PowerPC, et même pour des mainframes IBM Z. Dans la ligne de commande lancée plus haut, la plateforme n'a pas eu besoin d'être précisée, car le démon Docker est capable, de lui-même, de négocier la bonne image avec le registre, et une erreur est remontée si l'image ne supporte pas la plateforme installée.

Un clic sur la première entrée de la liste ci-dessus, qui correspond à l'image cherchée, montre plus de détails sur cette image. Le premier onglet d'information, nommé **Description**, expose en particulier ce qui s'appelle les tags, ou étiquettes en français.



Bien qu'il y ait nécessairement un lien entre les différentes images et les différentes plateformes, une image étiquetée Linux étant prévue pour une plateforme Linux, il s'agit bien de deux notions séparées. Par exemple, il est habituel que des images pensées pour la plateforme Linux soient pour certaines basées sur une version de Linux très complète comme une Ubuntu et pour certaines sur une image réduite comme une Alpine. La plateforme est alors toujours Linux, mais les étiquettes permettront de faire la différence entre les deux images.

Lorsque l'étiquette pointe sur une seule image et que seul le choix de la plateforme sous-jacente reste à réaliser par le moteur Docker lors du téléchargement, on parle d'étiquette simple (**Simple Tags** dans l'interface en anglais), mais pour complexifier la compréhension – bien que cela simplifie au final les manipulations –, il existe des étiquettes partagées (**Shared Tags**) dont l'usage fait que le moteur Docker choisit non seulement la plateforme d'exécution, mais aussi négocie l'image même qui va être déployée. Dans l'écran plus haut, on voit que l'étiquette `latest` est dite partagée, car son usage fera que le moteur Docker choisit entre deux images, à savoir l'image prévue pour Linux et l'image prévue pour Windows Nano Server. Bien sûr, il se trouve que la première fonctionnera sur des plateformes Linux et la seconde sur des plateformes Windows, mais les deux notions restent bien différentes.

Comme expliqué plus haut, on peut avoir des images Linux très différentes en nombre de fonctionnalités (et donc en taille) et il en va de même pour Windows où, au lieu d'un Nano Server, une image pourrait se baser sur une version Windows Server complète, avec une image bien plus lourde en conséquence.

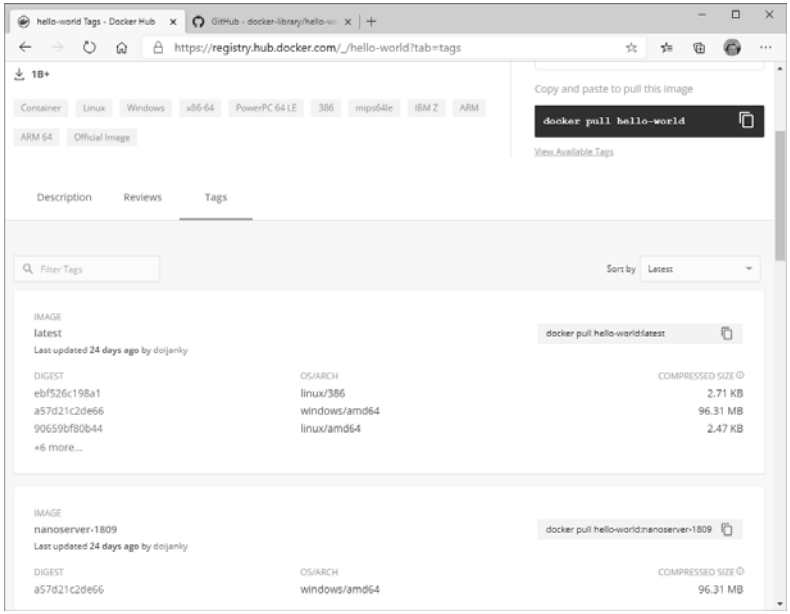
Dans notre cas, la commande ayant été lancée sans préciser d'étiquette, c'est l'étiquette `latest` qui est appelée par défaut (comportement standard de Docker dans son échange avec les registres). Cette étiquette donne le choix entre une image `nanoserver-1809`, qui supporte la plateforme Windows, et une image `linux`, qui supporte toutes les autres plateformes.

Remarque

Les images étant versionnées, comme expliqué plus haut, nous reviendrons sur cette étiquette `latest` bien particulière, qui permet également de choisir une version de référence (et non la plus récente, comme la traduction française de `latest` pourrait le faire croire).

1.2.3 Taille des images

En accédant à l'onglet **Tags** de l'interface, on obtient une description plus précise des étiquettes, qui devrait permettre d'aider à la compréhension de cette notion qui n'est pas évidente en première utilisation :



On voit tout d'abord qu'il existe bel et bien deux images simples, à savoir `linux` et `nanoserver-1809`, avec des tailles extrêmement différentes. La seconde, comme on peut le voir en bas de la capture d'écran, ne supporte qu'une seule plateforme, à savoir `windows/amd64`, sur laquelle elle pèse 96 Mo environ. La première, plus bas dans la liste et non visible sur la capture ci-dessus, supporte huit plateformes matérielles, et pèse quelques ko sur chaque, avec des variations très légères.

Remarque

Cette énorme différence de poids s'explique par deux raisons. La première est que Docker est une technologie pensée pour Linux et à laquelle Windows s'adapte sans avoir les caractéristiques prévues spécifiquement pour dans son architecture. Une image Windows, au lieu de s'appuyer simplement sur le système sous-jacent, doit embarquer un mini-système Windows pour être réellement autonome. Dans ces conditions, passer sous la barre des 100 Mo est déjà un exploit et rapproche ces images des poids standards d'images Docker pour Linux avec des serveurs complexes, ou embarquant des systèmes plus complets comme une image Ubuntu. Il y a toutefois une seconde part d'explication à la différence de taille : pour cette image particulière, les concepteurs sont partis de zéro et n'incluent dans l'image qu'un exécutable compilé nativement dans chacun des systèmes ciblés, ce qui fait naturellement que le poids est réduit à l'extrême, seul le binaire exécutable étant alors nécessaire. Nous verrons dans la section suivante comment ceci est mis en œuvre.

L'interface plus haut est également l'occasion de revenir sur la notion d'étiquette partagée et de montrer justement l'étiquette `latest`, tout en haut de la liste, et qui couvre quant à elle neuf plateformes, à savoir les huit de l'image `linux` et la plateforme supplémentaire de l'image `nanoserver-1809`. On comprend ainsi mieux l'effet "chapeau" de ces étiquettes partagées, qui permettent à l'utilisateur de ne pas se poser de questions et de demander à Docker de se débrouiller pour récupérer une image qui fonctionne sur son système et sa plateforme. Dans le cas de `latest`, le principe est aussi de laisser Docker s'occuper de choisir la bonne version, le cas échéant. Il se trouve que pour l'image `hello-world`, il n'y a qu'une version, donc ceci n'est pas visible. Mais pour des images de serveurs logiciels, nous reviendrons plus loin sur l'association entre `latest` et la dernière version stable de l'application, qui est une convention dans l'écosystème Docker.

On remarque aussi que les tailles sont exactement les mêmes entre les différentes plateformes des étiquettes simples et de l'étiquette partagée, les étiquettes n'étant en fait que des pointeurs sur les images au sens binaire, qui possèdent un contenu et donc un poids associé.

Chapitre 4
Automatisation et publication d'une application

1. Objectifs du chapitre et prérequis

Le chapitre précédent a constitué une introduction à l'utilisation de Kubernetes avec notamment le recours au dashboard pour le déploiement d'une application.

Ce chapitre va reprendre cet exercice. La différence se fera sur le mode opératoire puisque les opérations seront réalisées en ligne de commande.

2. Gestion par kubectl d'une application

2.1 Suppression d'un déploiement

Durant le précédent chapitre, l'application MailHog a été déployée à l'aide du dashboard. Même si vous ne l'avez pas fait, vous pouvez néanmoins suivre les instructions qui suivent.

Premier point, récupérer la liste des déploiements présents dans le serveur. Pour cela, il faut lancer la commande `kubectl` suivie du mot-clé `get` avec le type d'objet `deployment`. Ci-dessous la commande correspondante :

```
$ kubectl get deployment
```

Ci-dessous le résultat renvoyé :

```
NAME      READY   UP-TO-DATE   AVAILABLE   AGE
mailhog   1/1     1            1           8d
```

La suppression du déploiement se fait à l'aide de la commande `kubectl` suivie des éléments suivants :

– Le mot-clé `delete`.
– Le type d'objet à supprimer (`deployment`).
– Le nom de l'objet à supprimer (`mailHog`).

Ci-dessous la commande à lancer :

```
$ kubectl delete deployment mailhog
```

Ci-dessous le résultat renvoyé par cette commande :

```
deployment.extensions "mailhog" deleted
```

La consultation de la liste des pods renverra alors le contenu suivant :

```
NAME                       READY   STATUS        RESTARTS   AGE
mailhog-69bd8f74cb-kl9p5   1/1     Terminating   2          8d
```

Au bout de quelques instants, la commande devrait renvoyer le message suivant :

```
No resources found.
```

2.2 Création d'un déploiement

La commande `kubectl`, outre les opérations de consultation et de suppression, prend en charge la création d'objets. Dans le cas d'un déploiement, la commande doit être lancée avec les options suivantes :

– Le mot-clé `create`.
– Le type d'objet à créer : `deployment` (ou le raccourci `deploy`).
– Le nom du déploiement (`mailhog`).
– Le nom de l'image à déployer avec l'option `--image`.

Ci-dessous la commande correspondant à ces indications :

```
$ kubectl create deployment mailhog --image=mailhog/mailhog
```

Ci-dessous le résultat de cette commande :

```
deployment.apps/mailhog created
```

2.3 État du déploiement

Une fois créé, le déploiement est consultable à l'aide de la commande `kubectl` suivie des options suivantes :

– L'option `get`.

– Le type d'objet : `deployment` (ou `deploy`);

– Le nom de l'objet : `mailhog`.

La commande à lancer :

```
$ kubectl get deployment
```

Le résultat renvoyé :

```
NAME      READY   UP-TO-DATE   AVAILABLE   AGE
mailhog   1/1     1            1           3m
```

L'ajout de l'option `-o wide` permettra de renvoyer des champs supplémentaires. Ci-dessous les informations supplémentaires renvoyées :

– `CONTAINERS` : liste des containers.

– `IMAGES` : liste des images utilisées.

– `SELECTOR` : le label de sélection des pods.

En plus de l'option `get`, la commande `kubectl` propose l'option `describe`. Cette option permet de récupérer tout un ensemble d'informations sur les caractéristiques d'un objet Kubernetes. L'instruction `describe` prend en charge les mêmes options que `get`, à savoir un type d'objet ainsi qu'un nom d'objet.

Ci-dessous la commande à utiliser pour consulter l'état du déploiement de MailHog :

```
kubectl describe deployment mailhog
```

Ci-dessous le résultat renvoyé par cette commande :

```
Name:                   mailhog
Namespace:              default
CreationTimestamp:      Sun, 31 Mar 2019 14:54:03 +0200
Labels:                 app=mailhog
Annotations:            deployment.kubernetes.io/revision: 1
Selector:               app=mailhog
Replicas:               1 desired | 1 updated | 1 total |
1 available | 0 unavailable
StrategyType:           RollingUpdate
MinReadySeconds:        0
RollingUpdateStrategy:  25% max unavailable, 25% max surge
Pod Template:
   Labels:  app=mailhog
   Containers:
     mailhog:
       Image:        mailhog/mailhog
       Port:         <none>
       Host Port:    <none>
       Environment:  <none>
       Mounts:       <none>
   Volumes:        <none>
Conditions:
   Type           Status  Reason
   ----           ------  ------
   Available      True    MinimumReplicasAvailable
   Progressing    True    NewReplicaSetAvailable
OldReplicaSets:  <none>
NewReplicaSet:   mailhog-5dbc9c86c4 (1/1 replicas created)
Events:
  Type    ... Message
  ----    ... -------
  Normal  ... Scaled up replica set mailhog-5789447cbf to 1
```

Cette commande renvoie de nombreuses informations parmi lesquelles :

– le nom et l'espace de noms du déploiement,
– la date de création,
– les labels et annotations,
– le mécanisme de sélecteur,
– le nombre de réplicats de l'application (le nombre de pods),
– le mécanisme de mise à jour (ici RollingUpdate),
– le template permettant la création des pods associés,
– l'état du déploiement (conditions),
– le réplicat actuel ainsi que les anciens réplicats,
– les événements autour de la création de ce déploiement (ici le passage à un réplicat sur le déploiement de MailHog).

2.4 Mécanisme des réplicats

2.4.1 Consultation des réplicats

Parmi les informations remontées par la commande `kubectl describe` se trouve le ReplicaSet actuel ainsi que – en cas de mise à jour – la liste des anciens ReplicaSet.

Cet objet va prendre en charge la création d'un nombre donné de réplicats pour une application donnée. C'est lui qui va indiquer au cluster Kubernetes qu'il faut redémarrer un pod en cas de suppression d'un pod.

Il va également être associé à l'ensemble des caractéristiques d'une application à un instant donné, comme par exemple :

– les caractéristiques d'une image (numéro de version, emplacement, nom),
– la réservation d'une quantité de mémoire et CPU,
– certaines caractéristiques du déploiement (variables d'environnement).

En cas de mise à jour de l'objet Deployment, un nouvel objet ReplicaSet sera créé automatiquement et le ReplicaSet précédent sera ajouté à la liste des anciens objets ReplicaSet.

Remarque

Le mécanisme de retour arrière de Kubernetes s'appuie sur ces objets.

Tout comme pour les déploiements, il est possible de consulter les réplicats à l'aide de la commande `kubectl` et de l'option `get` suivie du type à consulter (`replicaset` ou son raccourci `rs`). Ci-dessous la commande correspondante :

```
$ kubectl get replicaset
```

Ci-dessous le résultat de cette commande :

```
NAME                  DESIRED   CURRENT   READY   AGE
mailhog-5dbc9c86c4    1         1         1       5h33m
```

Le nom du réplicat est constitué d'une partie fixe (reprenant le déploiement duquel il descend) suivie d'une partie aléatoire.

2.4.2 Description des réplicats

Tout comme pour un objet Deployment, il est possible de décrire un objet ReplicaSet. Ci-dessous la commande à lancer dans le cas du réplicat de MailHog précédemment remonté :

```
$ kubectl describe rs mailhog-5dbc9c86c4
```

Ci-dessous la sortie renvoyée par cette commande :

```
Name:           mailhog-5dbc9c86c4
Namespace:      default
Selector:       app=mailhog,pod-template-hash=5dbc9c86c4
Labels:         app=mailhog
                pod-template-hash=5dbc9c86c4
Annotations:    deployment.kubernetes.io/desired-replicas: 1
                deployment.kubernetes.io/max-replicas: 2
                deployment.kubernetes.io/revision: 3
                deployment.kubernetes.io/revision-history: 1
Controlled By:  Deployment/mailhog
Replicas:       1 current / 1 desired
```

```
Pods Status:    1 Running / 0 Waiting / 0 Succeeded / 0 Failed
Pod Template:
   Labels:  app=mailhog
            pod-template-hash=5dbc9c86c4
   Containers:
     mailhog:
     Image:        mailhog/mailhog
     Port:         <none>
     Host Port:    <none>
     Environment:  <none>
     Mounts:       <none>
   Volumes:        <none>
Events:
   Type    Reason            Age   From                   Message
   ----    ------            ----  ----                   -------
   Normal  SuccessfulDelete  62m   replicaset-controller  Deleted
pod: mailhog-5dbc9c86c4-22lzr
   Normal  SuccessfulCreate  59m   replicaset-controller  Created
pod: mailhog-5dbc9c86c4-8sjcs
   Normal  SuccessfulDelete  59m   replicaset-controller  Deleted
pod: mailhog-5dbc9c86c4-8sjcs
   Normal  SuccessfulDelete  58m   replicaset-controller  Deleted
pod: mailhog-5dbc9c86c4-49n9v
   Normal  SuccessfulCreate  57m   replicaset-controller  Created
pod: mailhog-5dbc9c86c4-vdgj6
```

Les informations sont sensiblement les mêmes que pour un objet Deployment au niveau de l'en-tête. En revanche, la partie `Events` est plus verbeuse, avec toutes les opérations qui ont été réalisées sur les pods du déploiement de l'application MailHog.

Le champ `Events` peut être une source d'informations intéressante à consulter en cas d'anomalies lors d'un déploiement.