COMPRENDRE ET MANIPULER LE SCHEDULER KUBERNETES

à quoi sert un Scheduler ?

Un scheduler (en FR : "planificateur") est **chargé de la planification des pods sur les nœuds**. Fondamentalement, cela fonctionne comme ceci :

1. Vous créez votre pod.
2. Le scheduler constate qu'il n'y a pas de nœud attribué au nouveau pod que vous avez créé.
3. Le scheduler assigne un nœud au pod.

> **Attention**
>
> Le scheduler n'est en aucun cas responsable de la gestion du pod, cette tâche est assignée à l'outil kubelet.

En effet, pour chaque pod découvert par le scheduler, il est responsable de la recherche du meilleur nœud sur lequel ce pod doit être exécuté. Il prend cette décision de placement selon les étapes suivantes :

- Il vérifie si un nœud candidat détient suffisamment de ressources disponibles pour répondre aux demandes de ressources spécifiques au pod.
- Il réunit alors une liste de nœuds contenant tous les nœuds appropriés. souvent, il y en aura plus d'un. Si la liste est vide, le pod n'est alors pas encore

ordonnancé et entre dans l'état `Pending`.

- Le scheduler attribue un score à chaque nœud ayant survécu au filtrage, en le basant sur ses propres règles de scoring comme par exemple : le nombre de ressources déjà consommées ou encore la présence ou non des images docker sur le nœud.
- Enfin, le scheduler assigne le pod au nœud avec le rang le plus élevé. Dans le cas où plusieurs nœuds possèdent des scores égaux, le scheduler sélectionne alors l'un d'entre eux aléatoirement.

> **Information**
>
> kube-scheduler est le planificateur par défaut pour Kubernetes et s'exécute dans le plan de contrôle. Vous retrouverez plus d'informations sur les règles de filtrage et de scoring dans la doc officielle.

Maintenant que les présentations sont faites, il est temps d'**étudier les différentes de manières d'affectation de pods à des nœuds**.

nodeName

Chaque pod possède un champ nommé `nodeName` qui prend comme valeur le nom du nœud qui accueillera le pod. Si ce champ est défini dans le spec du pod, il est alors prioritaire sur les différentes méthodes de sélection de nœuds que nous verrons plus tard.

Cependant, gardez à l'esprit que l'utilisation du **nodeName comporte quelques limitations** :

- Si le nœud spécifié dans le champ nodeName n'existe pas, alors le pod ne sera pas exécuté, et dans certains cas, il peut être automatiquement supprimé.
- Si le nœud nommé ne dispose pas des ressources nécessaires pour héberger le pod, celui-ci échouera.

Voici un exemple de fichier de configuration YAML de pod utilisant le champ nodeName :

```
apiVersion: v1
kind: pod
metadata:
  name: nginx
spec:
  containers:
  - name: nginx
    image: nginx
  nodeName: worker-1
```

Créons par la suite notre pod avec la commande suivante :

```
kubectl create -f pod.yaml
```

Le pod ci-dessus est sensé donc être exécuté sur le nœud `worker-1`, vérifions alors cela :

```
kubectl get pods -o wide
```

C'est bien le cas :

```
NAME    READY   STATUS    RESTARTS   AGE   IP          NODE     NOMINATED NODE   READI
nginx   1/1     Running   0          12s   10.44.0.1   worker-1   <none>           <no
```

> **Avertissement**
>
> Vous pouvez uniquement spécifier le nodeName qu'au moment de la création du pod. En effet Kubernetes ne vous permettra pas de modifier la propriété nodeName d'un pod tournant déjà dans le cluster.

nodeSelector

Le `nodeSelector` est une des formes de contrainte de sélection de nœud disponible dans Kubernetes. C'est un champ qu'on rajoute dans le spec des pods, et où on spécifie une paire de clé-valeur correspondant au label du nœud sur lequel on souhaite recevoir notre pod. Passons en revue un exemple d'utilisation de nodeSelector.

Avant toute chose, je dois vous dévoiler d'abord la **commande qui permet d'assigner un label à un nœud** :

```
kubectl label node <NODE_NAME> <key>=<value>
```

Dans cet exemple je vais affecter le label `ntype:html` pour mon nœud `worker-1`, soit :

```
kubectl label node worker-1 ntype=html
```

Vous pouvez vérifier que cela a fonctionné en exécutant la commande suivante :

```
kubectl get nodes worker-1 --show-labels
```

On retrouve bien notre label :

```
NAME     STATUS   ROLES    AGE   VERSION   LABELS
```

```
worker-1   Ready    <none>   82m   v1.14.0   beta.kubernetes.io/arch=amd64,beta.kuberr
```

L'étape suivante est d'ajouter le champ nodeSelector à la configuration YAML de votre pod :

```
apiVersion: v1
kind: pod
metadata:
  name: nginx
spec:
  containers:
  - name: nginx
    image: nginx
  nodeSelector:
    ntype: html
```

Exécutons notre pod avec la commande suivante :

```
kubectl create -f pod.yaml
```

Une fois la commande lancée, le pod est alors planifié sur le nœud auquel vous avez attaché le label. Constatons cela avec la commande suivante :

```
kubectl get pods -o wide
```

Résultat :

```
NAME    READY   STATUS    RESTARTS   AGE     IP          NODE     NOMINATED NODE   REA
nginx   1/1     Running   0          2m52s   10.44.0.1   worker-1   <node>           <
```

Néanmoins le **nodeSelector possède quelques limites**. Dans notre exemple ne nous n'avions utilisé qu'un seul label de selection pour atteindre notre objectif. Mais que ce passe-til si notre exigence est beaucoup plus complexe ?

Je m'explique, par exemple, nous pourrions souhaiter quelque chose comme lancer un pod sur plusieurs nœuds avec plusieurs labels, voire de lancer un pod sur tous les nœuds qui ne contiennent pas de tel ou tel label.

Pour résoudre ces problématiques complexes, nous utiliserons une autre fonctionnalité proposée par kubernetes qui est l'affinité ou anti-affinité de nœud, que nous aborderons dans la section suivante.

Vous pouvez **supprimer le label de votre nœud** en rajoutant le signe `-` (moins) à la fin de la commande `kubectl label` sans spécifier la valeur de la clé. Par exemple si je souhaite supprimer le label crée précédemment, je vais utiliser la commande suivante :

```
kubectl label node worker-1 ntype-
```

Node Affinity

La fonctionnalité d'affinité de nœud nous fournit des **fonctionnalités avancées pour limiter le placement de pods sur des nœuds** spécifiques.

Il existe actuellement deux types d'affinité de nœud, appelés :

- `requiredDuringSchedulingIgnoredDuringExecution`
- `preferredDuringSchedulingIgnoredDuringExecution`

J'avoue que le nom est un peu barbare, mais on peut distinguer ces types avec les états suivants :

During Scheduling	During Execution
required	ignored
preferred	ignored

Nous avons ainsi :

- `During Scheduling` : c'est l'état où un pod n'existe pas encore et qu'il est

créé pour la première fois. Cet état peut bénéficier de deux valeurs :

  - `required` : éxige que le pod soit placé sur un nœud respectant les règles d'affinité. S'il ne parvient pas à en trouver un, le pod ne sera alors pas planifié.
  - `preferred` : quand aucun nœud correspondant n'est trouvé. Le scheduler ignorera simplement les règles d'affinité du pod et le placera sur n'importe quel nœud disponible. C'est une façon de dire au scheduler : *"Fais de ton mieux pour placer le pod sur la correspondance d'affinité mais si tu ne peux vraiment pas en trouver un, place-le sur n'importe quel autre pod."*

- `During Execution` : c'est l'état lorsqu'un pod est déjà exécuté et qu'un changement a été apporté à l'environnement qui affecte l'affinité du nœud, tel qu'un changement dans le label du nœud. Comme vous pouvez le constater, les deux types d'affinité disponibles aujourd'hui sont à l'état `ingored` , ce qui signifie que les pods continueront à fonctionner et que toute modification de l'affinité des nœuds, n'aura aucun impact une fois qu'ils sont planifiés.

> **Information**
>
> Kubernetes prévoit de rajouter d'autres types de plan d'affinité de nœuds au moment je rédige cet article.

L'affinité de nœud est spécifiée dans le champ `nodeAffinity` du champ `affinity` dans la spécification du pod. Voici un exemple d'un pod utilisant une affinité de nœud :

```
apiVersion: v1
```

```
kind: pod
metadata:
  name: nginx
spec:
  affinity:
    nodeAffinity:
      requiredDuringSchedulingIgnoredDuringExecution:
        nodeSelectorTerms:
        - matchExpressions:
          - key: ntype
            operator: NotIn
            values:
            - virus
            - malware
  containers:
  - name: nginx
    image: nginx
```

Cette règle d'affinité de nœud indique que le pod ne peut pas être placé (grâce à l'opérateur `NotIn`) sur un nœud portant un label dont la clé est `ntype` et dont la valeur est `virus` ou `malware`.

Voici les différents opérateurs avec leur description pris en charge par les règles d'affinité de nœud :

- `In` : utiliser les nœuds avec les mêmes clés et valeurs à ceux qui sont définies dans le spec des pods.
- `NotIn` : ignorer les nœuds avec les mêmes clés et valeurs à ceux qui sont définies dans le spec des pods.
- `Exists` : utiliser les nœuds avec les mêmes clés à ceux qui sont définies dans le spec des pods.
- `DoesNotExist` : ignorer les nœuds avec les mêmes clés à ceux qui sont définies dans le spec des pods.
- `Gt` : utiliser les nœuds avec des valeurs numériques supérieur à ceux qui sont définies dans le spec du pod.

- `Lt` : utiliser les nœuds avec des valeurs numériques inférieur à ceux qui sont définies dans le spec du pod.

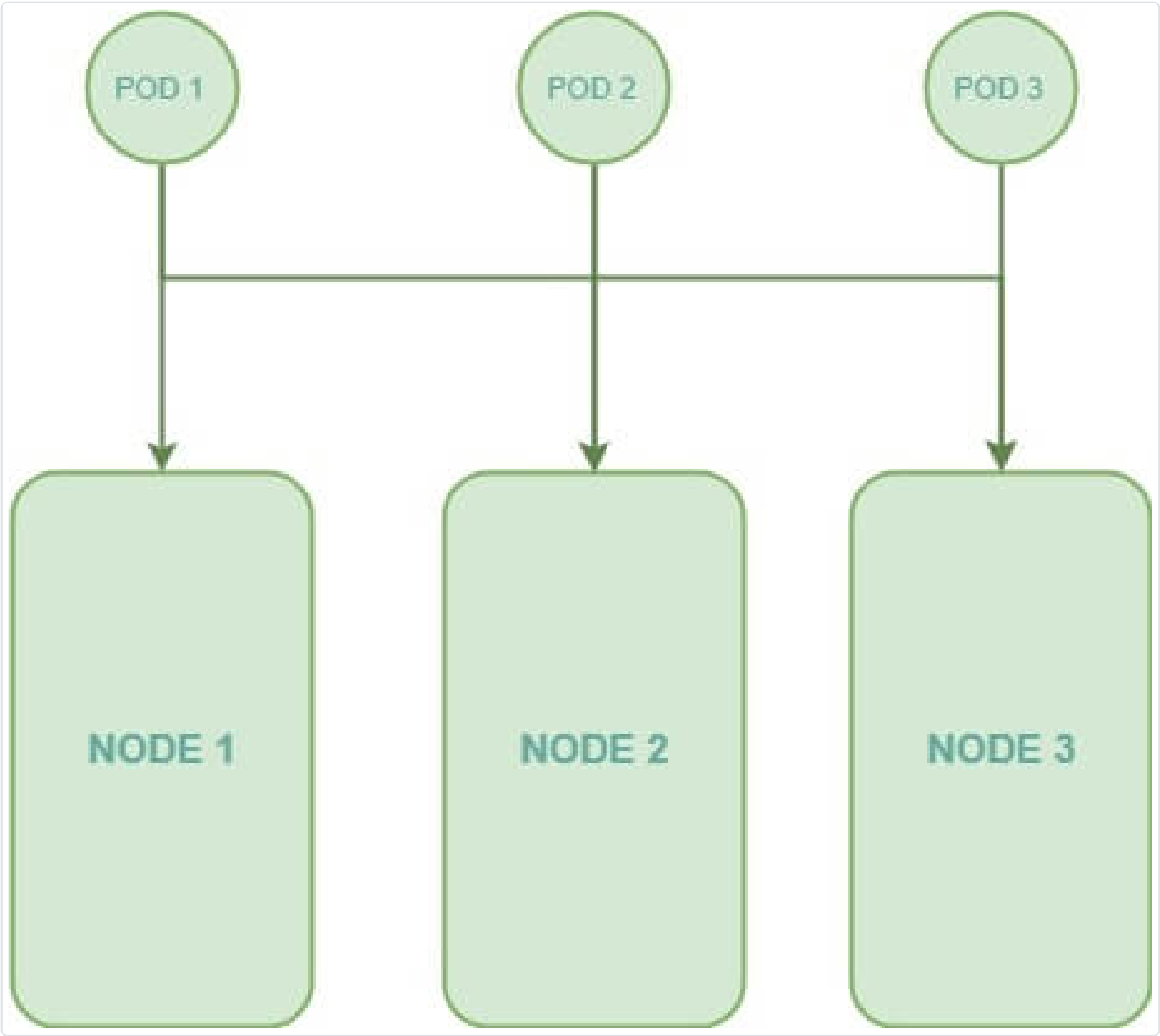
Les opérateurs `NotIn` et `DoesNotExist` provoquent ce que l'on appelle un **comportement anti-affinité de nœud** afin de repousser les pods des nœuds que vous avez spécifiés.

Taints and Tolerations

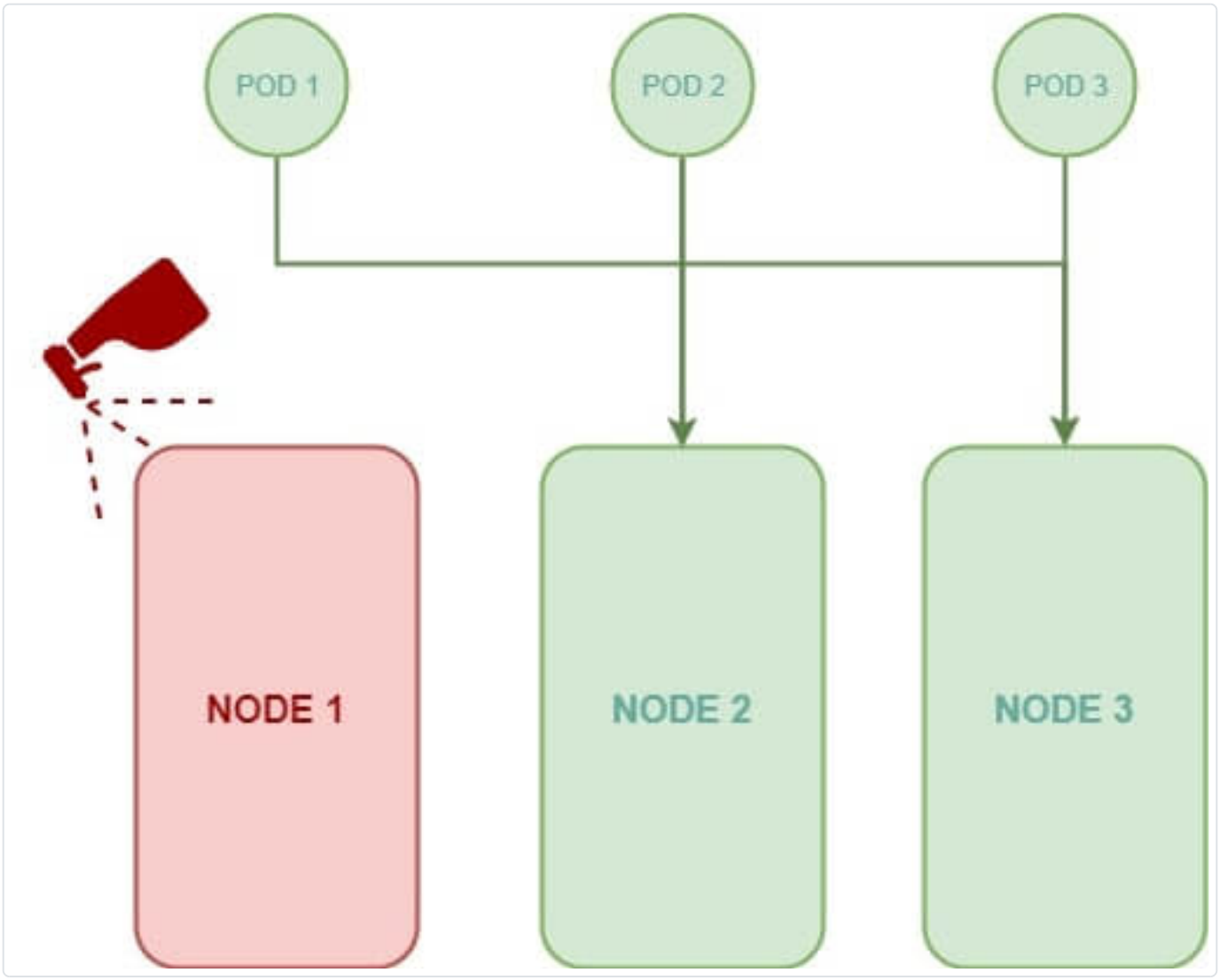
Les `Taints` (en FR : "rejets") et les `Tolerations` (en FR : "Tolérances") permettent de **définir des restrictions sur les pods pouvant être planifiés sur un nœud**. Ils fonctionnent ensemble afin de garantir que les pods ne soient pas planifiés sur des nœuds inappropriés.

Les Taints vous permettent de marquer un nœud afin que l'utilisateur puisse l'ignorer et empêcher son utilisation pour certains pods. Les Tolerations sont quant à eux appliquées aux pods et permettent aux pods d'organiser leur planification sur des nœuds "rejetés".

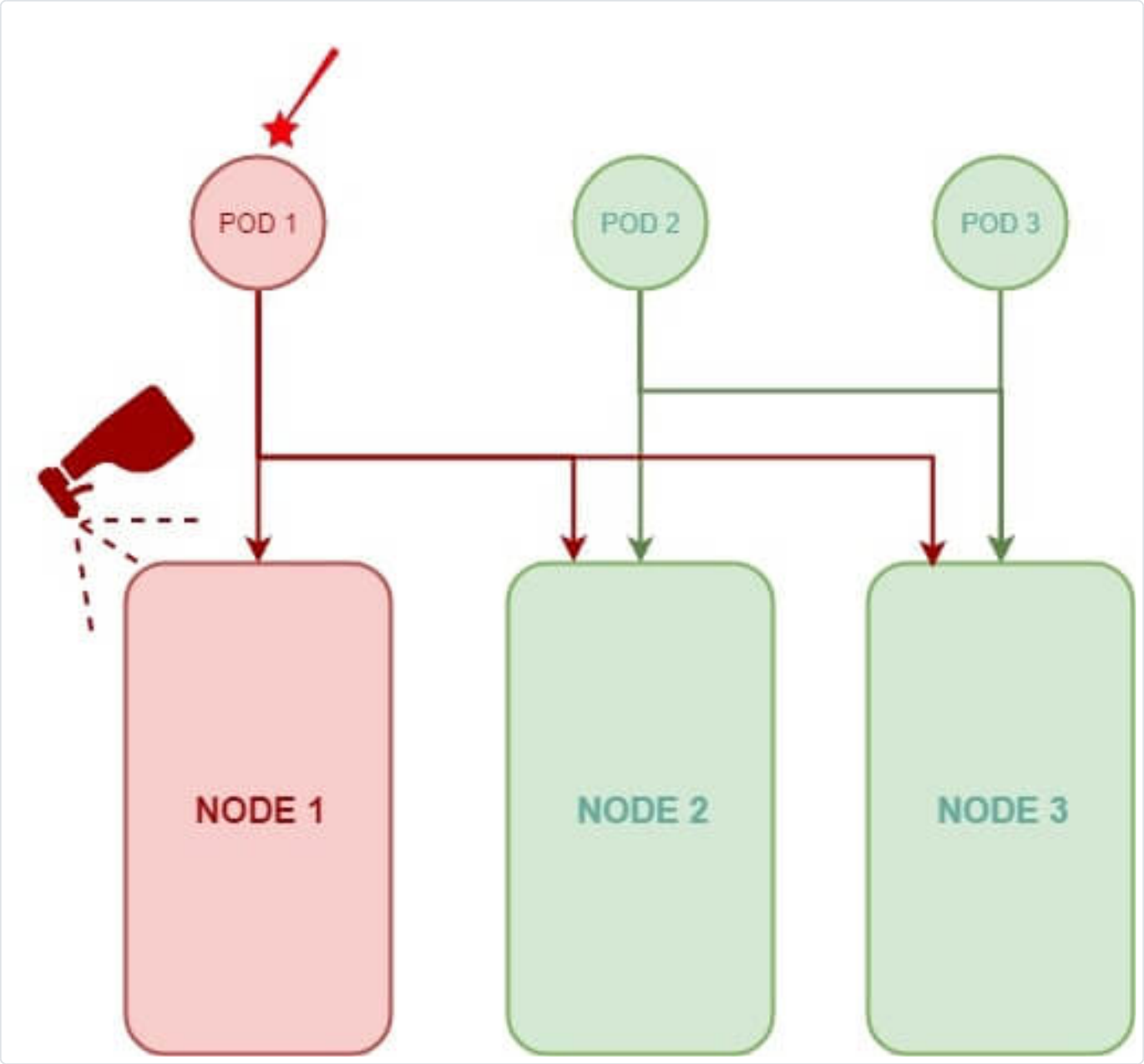
Dans le but de mieux **comprendre le concept les Taints et les Tolerations**, j'utiliserai désormais des schémas illustrant mes exemples. Prenons par exemple le cas, où nous avons trois pods à planifier, ainsi que trois nœuds différents dans notre cluster. Sans aucune utilisation de Taints ou de Tolerations, les pods peuvent dans ce cas être déployés sur n'importe quel nœud, ce qui nous donnera le schéma suivant :



Nous allons dorénavant, ajouter un rejet à notre nœud nommé `NODE 1` afin de le marquer comme nœud "indésirable", ce qui aura pour effet de repousser tous les pods du nœud `NODE 1`. Si on reprend le schéma précédent, nous obtiendrons le résultat suivant :



Enfin, je vais définir une tolération sur notre pod nommé `POD 1`, dans ce cas, seul le pod `POD 1` pourra être envoyé par le scheduler vers le nœud `NODE 1` mais aussi sur les autres nœuds.



Comme vous pouvez l'apercevoir dans le dernier schéma, les Taints et les Tolerences ne garantissent en aucun cas, qu'un pod sera toujours placé sur un nœud marqué avec un rejet, puisque aucun rejet ni restriction ne sont appliqués sur les autres nœuds, le pod **POD 1** peut très bien être placé sur les autres nœuds.

Souvenez-vous qu'une Tolerence n'indique pas au pod d'aller à un nœud particulier, mais elle indique plutôt au nœud de n'accepter que les pods avec une certaine tolérance. Si votre exigence est de restreindre un pod vers certains nœuds,

il est alors préférable d'utiliser les affinités de nœuds, d'ailleurs nous verrons plus tard l'intérêt de coupler les Taints avec les affinités.

Passons maintenant à la pratique. Pour créer un rejet nous utiliserons la commande suivante :

```
kubectl taint nodes <NODE NAME> key=value:<taint-effect>
```

Il faut donc définir un label dans le rejet de notre nœud et par la suite déterminer le `taint-effect` qui définit ce qu'il adviendrait des pods si ils ne toléraient pas le rejet. Actuellement il existe trois effets, soit :

- **`NoSchedule`** : le pod ne sera pas programmé sur le nœud marqué par un rejet.
- **`PreferNoSchedule`** : le système essaiera d'éviter de placer le pod sur le nœud marqué par un rejet, mais ne le garanti pas.
- **`NoExecute`** : si le pod est déjà en cours d'exécution sur le nœud marqué par un rejet, il est alors expulsé. Dans le cas contraire, il n'est pas planifié sur le nœud.

Dans cet exemple nous allons créer un rejet avec le label **`hold: virus`** pour notre nœud **`worker-1`** :

```
kubectl taint nodes worker-1 hold=virus:NoSchedule
```

Voici la commande qui permet de récupérer la liste des Taints disponibles dans notre cluster :

```
kubectl describe node | grep 'Name:\|Taints:'
```

Résultat :

```
Name:               master
Taints:             node-role.kubernetes.io/master:NoSchedule
Name:               worker-1
Taints:             hold=virus:NoSchedule
```

Ensuite nous allons créer un pod tolérant à notre nœud `worker-1` :

```
apiVersion: v1
kind: pod
metadata:
  name: nginx
spec:
  tolerations:
  - key: "hold"
    operator: "Equal"
    value: "virus"
  containers:
  - name: nginx
    image: nginx
```

Le champ `operator` peut prendre actuellement deux valeurs, soit la valeur `Exists` (pas besoin de spécifier le champ `value`), dans ce cas le pod sera tolérant à tous les nœuds ayant la même clé. Soit la valeur `Equal` que nous avions utilisée dans l'exemple précédent.

Une fois que vous avez créé votre pod, ce dernier pourra être lancé sur le nœud `worker-1`. Vérifions cela :

```
kubectl get pods -o wide
```

Résultat :

```
NAME    READY   STATUS              RESTARTS   AGE   IP       NODE       NOMINATED NOD
nginx   0/1     ContainerCreating   0          0s    <none>   worker-1   <none>
```

Vous pouvez retirer votre Taint en rajoutant le signe `-` à la fin de votre commande `kubectl taint`. Par exemple pour retirer un rejet de l'exemple précédent, nous

exécutons la commande suivante :

```
kubectl taint nodes worker-1 hold=virus:NoSchedule-
```

Résultat :

```
node/worker-1 untainted
```

Information

Par défaut, le scheduler ne planifie aucun pod sur le nœud master. Techniquement lorsque le cluster kubernetes est configuré pour la première fois, un rejet est défini sur le nœud master.

Ce rejet empêche alors les pods d'être planifiés sur ce nœud. Il est possible de supprimer ce rejet avec la commande suivante `kubectl taint nodes master node-role.kubernetes.io/master:NoSchedule-` .

Cependant, il est recommandé de ne pas déployer des pods dans le nœud maître .

Taints/Tolerations et Node Affinity

Les Taints et les Tolerences avec les nœuds d'affinités peuvent être utilisées ensemble pour dédier des nœuds à un pod spécifique. Vous devez d'abord utiliser les Taints et les Tolerences afin d'éviter que d'autres pods soient placés sur votre nœud spécifique, puis vous utilisez l'affinité de nœud pour empêcher que vos pods soit déployés sur les autres nœuds.

DaemonSet

L'objet `DaemonSet` garantit qu'une copie du pod est toujours présente dans tous les nœuds du cluster. Il exécute une copie de votre pod sur chaque nœud de votre cluster. Chaque fois qu'un nouveau nœud est ajouté au cluster, une réplique du pod est automatiquement ajoutée à ce nœud et lorsqu'un nœud est supprimé, le pod est automatiquement supprimé.

Dans cet exemple, nous allons créer et déployer un pod nginx sur tous les nœuds de notre cluster à l'aide de l'objet DaemonSet :

```
apiVersion: apps/v1
kind: DaemonSet
metadata:
  name: nginx-daemonset
spec:
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      name: nginx
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
        image: nginx
```

Une fois l'objet crée, nous apercevons sur le résultat ci-dessous que notre pod est

bien disponible sur tous nos nœuds :

```
kubectl get pods -o wide
```

Résultat :

```
NAME                    READY   STATUS    RESTARTS   AGE   IP              NODE
nginx-daemonset-xql6q   1/1     Running   0          7s    192.168.1.201   worker-1
nginx-daemonset-ztg78   1/1     Running   0          7s    192.168.0.56    worker-2
```

Conclusion

Nous avons pu comprendre et étudier l'intérêt d'un scheduler, nous avons aussi pu parcourir les différentes façons d'affectation de pods à des nœuds, leurs avantages et leurs inconvénients. Je vous partage ci-dessous un **aide-mémoire sur le scheduling dans Kubernetes** :

```
# Assigner un label à un nœud
kubectl label node <NODE NAME> <key>=<value>

# Supprimer un label d'un un nœud (rajouter le signe "-" à la fin)
kubectl label node <NODE NAME> <key>-

# Afficher les labels d'un nœud
kubectl get nodes <NODE NAME> --show-labels

# Créer une Taint
kubectl taint nodes <NODE NAME> key=value:<taint-effect>
    Valeurs possibles de <taint-effect> :
      - NoSchedule
      - PreferNoSchedule
      - NoExecute

# Récupérer la liste des Taints disponibles dans notre cluster
kubectl describe node [En option <NODE NAME>] | grep 'Name:\|Taints:'

# Supprimer une Taint à un nœud (rajouter le signe moins "-" à la fin)
kubectl taint nodes <NODE NAME> key=value:<taint-effect>-
```