GÉRER DES ENVIRONNEMENTS ÉPHÉMÈRES SUR KUBERNETES

Introduction

C'est quoi un environnement éphémère ?

La révision de code est un outil puissant pour détecter les problèmes avant que le code ne soit fusionné sur la branche principale. Mais certains problèmes sont difficiles à découvrir en lisant simplement du code. Pensez aux modifications de conception, aux flux de travail d'intégration des utilisateurs ou aux améliorations des performances.

Le problème c'est que si vous souhaitez voir le code d'une merge request (MR) en action, vous devrez: cloner la branche sur votre machine locale, redéployer votre environnement local, probablement reconfigurer vos données, puis valider les modifications. C'est beaucoup de travail et cela pourrait être difficile à faire par certains membres de l'équipe. Heureusement, les environnements éphémères ou de preview (prévisualisation) sont là pour vous aider.

Un environnement éphémère est un environnement créé avec le code de vos MR. Il fournit un environnement réaliste pour voir vos modifications en direct avant de fusionner sur la branche principale. Un lien vers l'environnement de preview est ajouté à la MR, afin que tous les membres de votre équipe puissent voir vos modifications en direct en un seul clic. L'environnement de prévisualisation sera mis à jour à chaque validation, et finalement détruit lorsque votre MR sera fermée. Ces environnement aident à détecter les erreurs avant que le mauvais code ne soit

fusionné sur votre branche principale et casse votre environnement de production.

Quelle est la meilleure façon de créer vos environnements de preview ? Avez-vous besoin de beaucoup de scripts ? Non, vous pouvez tirer parti de Kubernetes pour **simplifier la gestion de vos environnements de preview**.

Si vous n'avez pas la base sur Kubernetes, je vous conseille de suivre mon cours sur kubernetes. Ceci dit nous allons quand même rappeler certains concepts et voir **pourquoi kubernetes est un choix idéal pour gérer facilement les environnements éphémères**.

Pourquoi déployer des envs de preview sur k8s ?

Kubernetes est une plate-forme portable, extensible et open source pour la gestion des charges de travail et des services conteneurisés, qui facilite à la fois la configuration déclarative et l'automatisation. Il possède un vaste écosystème en croissance rapide.

Je ne vais pas m'étendre ici sur les avantages de Kubernetes (il y a des tonnes d'articles de blog à ce sujet). Au lieu de cela, je me concentrerai sur les fonctionnalités clés fournies par Kubernetes qui en font une solution idéale pour **déployer vos environnements de prévisualisation**. Ces fonctionnalités sont:

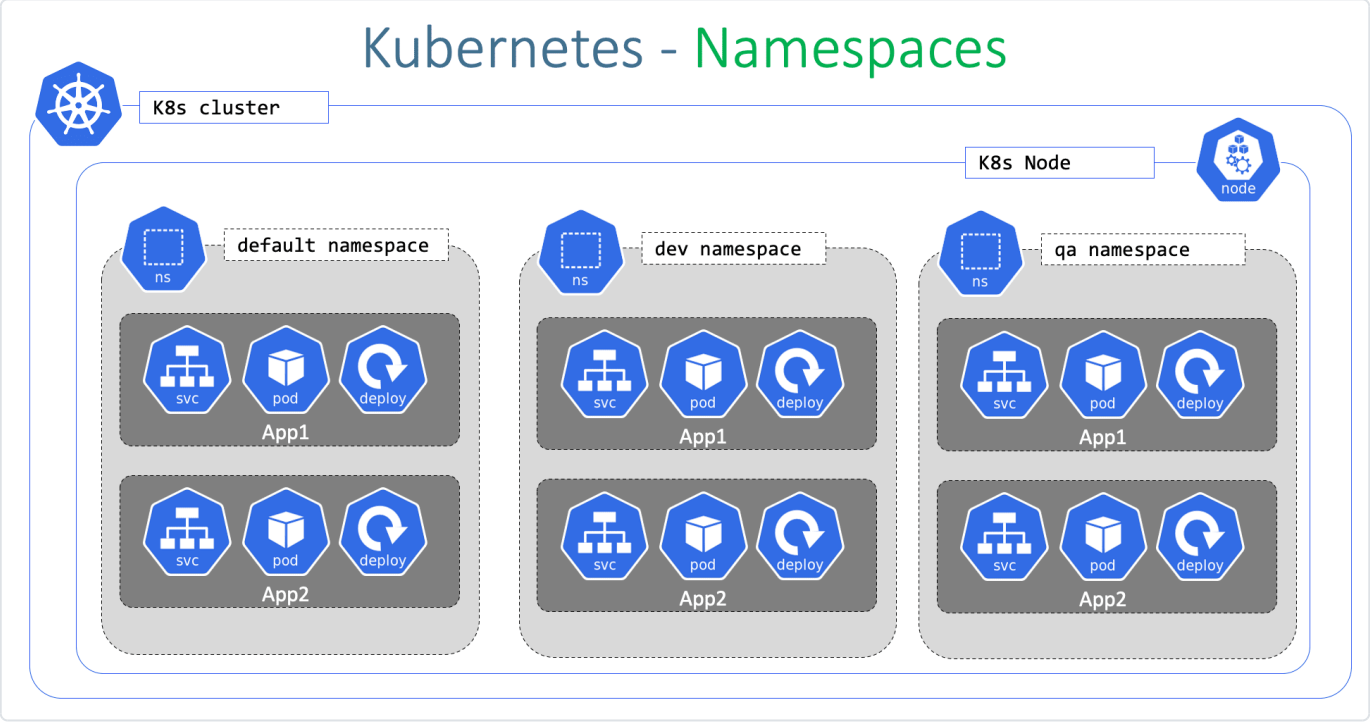
- Namespaces
- Temps de démarrage rapide des applications
- Ingress, certificats et wilcards
- Allocation optimisée des ressources
- Production-like integrations

Namespaces

Un namespace (espace de noms) est similaire à un cluster virtuel à l'intérieur de votre cluster Kubernetes. Vous pouvez avoir plusieurs namespaces dans un seul cluster Kubernetes, et ils sont tous logiquement isolés les uns des autres.

Les namespaces sont la limite parfaite pour les environnements de preview. Ils sont faciles à créer et à supprimer. Déployez chaque nouvel environnement de preview sur un namespace différent pour vous assurer qu'il n'y a pas de collisions de noms et d'incompatibilités entre vos différents environnements.

Lorsque votre MR est clôturée, il suffit uniquement de supprimer le namespace associé et tout ce qui a été créé par votre environnement de preview sera détruit en quelques secondes (oui oui c'est aussi simple et rapide que ça).



"Schéma sur le fonctionnement des namespaces"

Temps de démarrage rapide des applications

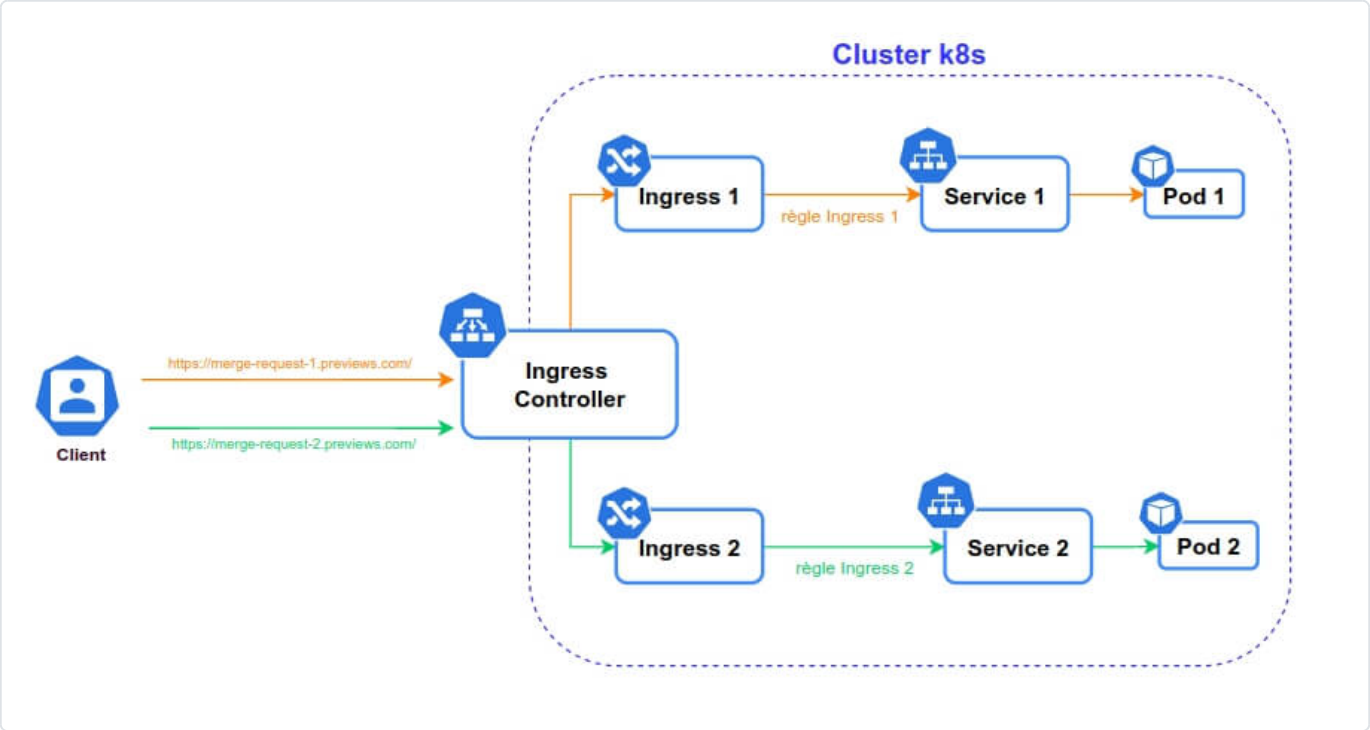
Chaque fois qu'il y a une MR, vous devez créer un namespace (c'est instantané), créer vos images Docker (très rapidement si vous optimisez le comportement de votre cache Docker) et déployer votre application votre cluster.

Kubernetes est conçu pour des déploiements rapides et à grande échelle. Vos conteneurs seront prêts en quelques secondes. C'est beaucoup plus rapide, moins cher et plus facile à configurer que de créer des machines virtuelles pour chaque environnement de prévisualisation.

Ingress, certificats et wilcards

Un Ingress Controller est un équilibreur de charge spécialisé pour les environnements Kubernetes. Il peut être utilisé pour partager un seul équilibreur de charge cloud entre toutes les applications exécutées dans le cluster. Il fournit une prise en charge L7 (équilibrage de charge de couche 7) pour acheminer le trafic vers votre application en fonction du domaine de requête entrant (nous expliquons plus en détail l'Ingress Controller plus tard dans cet article).

Les Ingress Controllers facilitent, accélèrent la configuration de l'accès à vos environnements de prévisualisation. Par exemple, vous pouvez créer une règle d'Ingress pour transférer le trafic du domaine `https://merge-request-1.previews.com` vers l'environnement preview déployé dans le namespace `merge-request-1`. Et vous pouvez créer une autre règle d'Ingress pour transférer le trafic du domaine `https://merge-request-2.previews.com` vers l'environnement de prévisualisation déployé dans l'espace de noms `merge-request-2`.

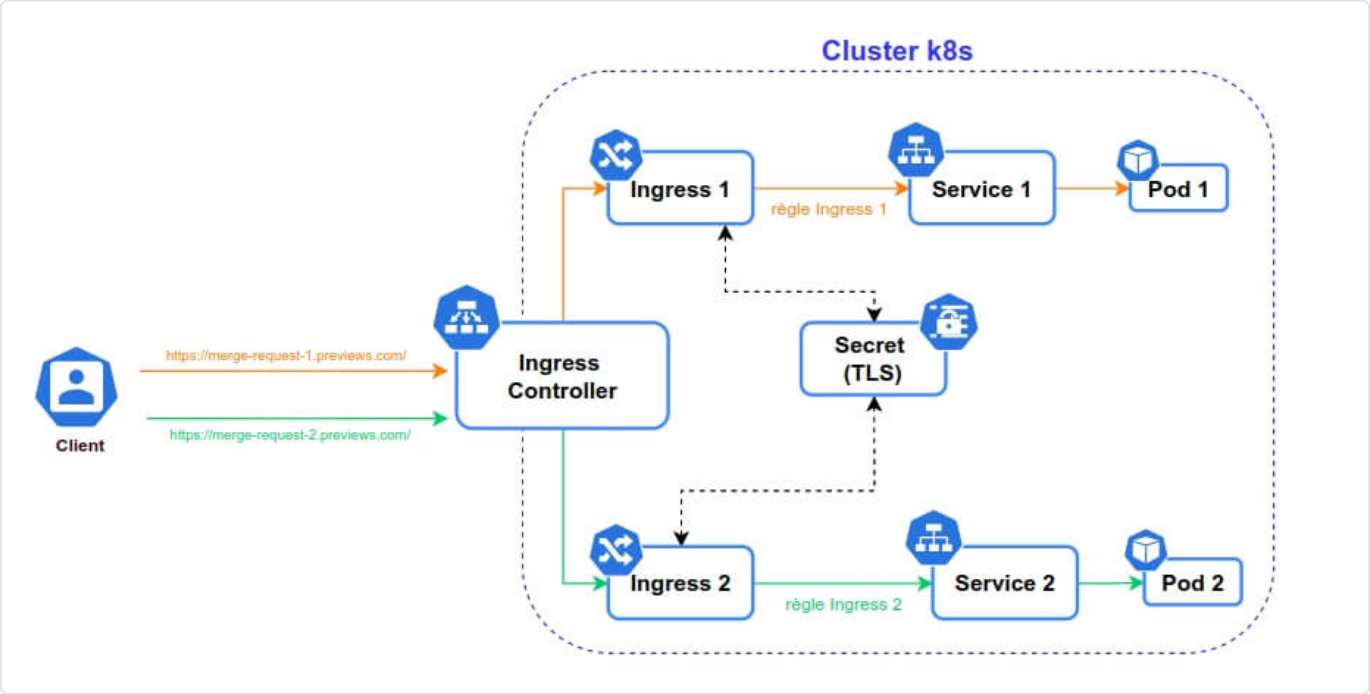


"Schéma sur le fonctionnement des Ingress Controllers"

Et ce n'est pas tout. Vous pouvez associer un certificat wildcard `*.previews.com` à ces règles d'Ingress (dans un secret k8s) et faire pointer le domaine générique `*.previews.com` vers l'adresse IP publique de votre Ingress Controllers, ainsi tous vos environnements de preview auront un domaine différent avec le support TLS.

Information

Un certificat SSL/TLS wildcard est un certificat unique avec un caractère générique `*` dans le champ du nom de domaine. Cela permet au certificat de sécuriser plusieurs noms de sous-domaine (hôtes) appartenant au même domaine de base. Par exemple, un certificat wildcard pour `*.mondomaine.com` , pourrait être utilisé pour `www.mondomaine.com` , `mail.mondomaine.com` , `store.mondomaine.com` , etc ...



"Schéma sur le fonctionnement des Ingress Controllers avec TLS"

Allocation optimisée des ressources

Les conteneurs exécutés sur le même nœud de cluster Kubernetes partagent les ressources CPU et mémoire du nœud. Cela peut être ajusté avec des requests et limits, et cela est très pratique pour les environnements de preview.

Les environnements de prévisualisation ne reçoivent généralement pas beaucoup de trafic. Il peut y avoir un pic pendant le démarrage de votre application, mais elles ne consommeront probablement presque pas de processeur et très peu de mémoire une fois qu'elles seront opérationnelles. Cela permet de déployer des dizaines de conteneurs sur le même nœud de cluster et de réduire votre facture cloud lors de l'utilisation d'environnements de preview.

Production-like integrations

Enfin, l'exécution de vos environnements de prévisualisation dans Kubernetes vous offre un environnement beaucoup plus réaliste. Vous testerez vos environnements

de preview avec la même sécurité, réseau, infrastructure cloud, etc... que vos environnements de production.

Vous pouvez même intégrer vos environnements de preview avec d'autres outils que vous utilisez en production, des politiques de réseau, des outils de surveillance comme Prometheus, des outils de visualisation comme Grafana et tout autre outil fonctionnant avec le riche écosystème de Kubernetes.

Prérequis

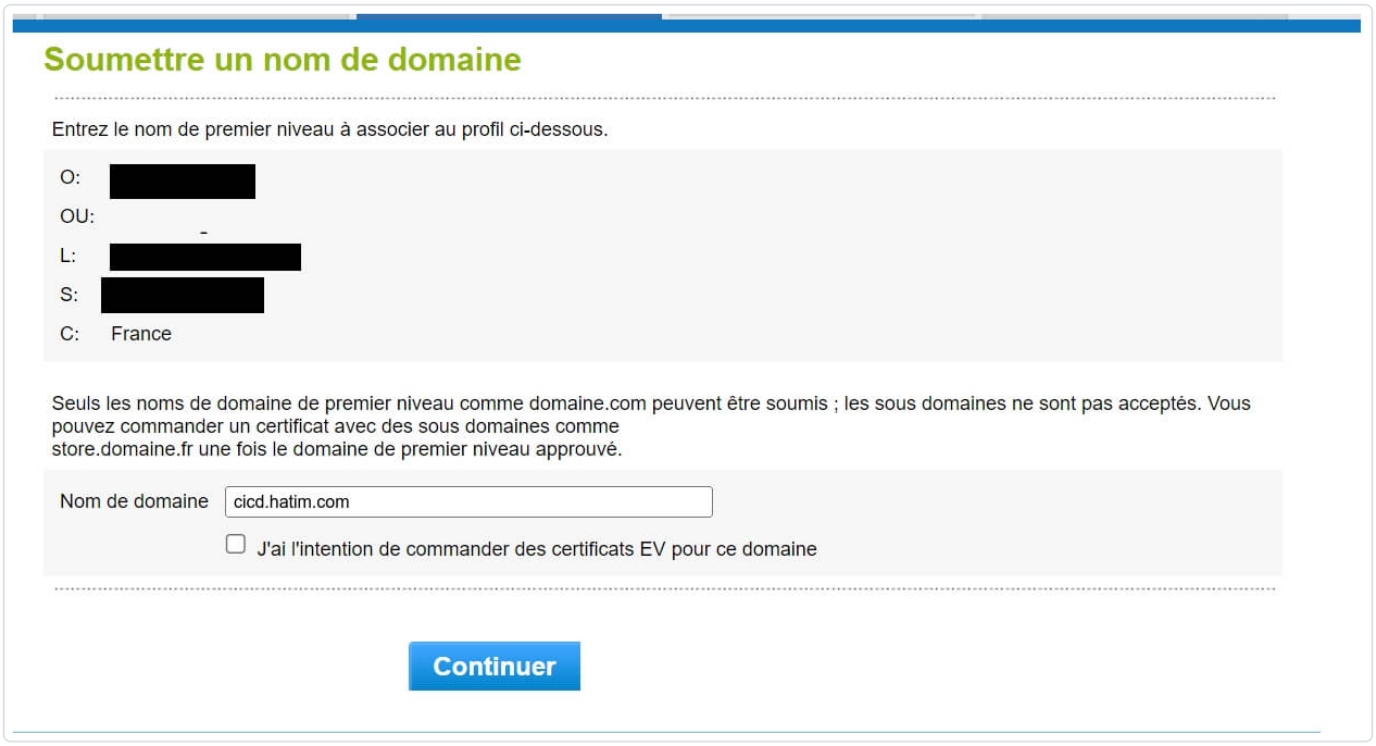
Nous venons de voir que Kubernetes possède les bons blocs pour gérer les environnements éphémères. Dans cet article, notre objectif principal sera de simuler le déploiement de deux applications différentes sur des environnements éphémères différents et d'être capable des les supprimer automatiquement très facilement. Les exigences seront donc les suivantes:

- **URL publique dynamique:** une url d'accès à l'application qui soit différente.
- **SSL/TLS:** comme les environnements de production contiennent des certificats SSL/TLS, notre environnement éphémère en aura également besoin.
- **Dynamique:** l'environnement doit être prêt en quelques secondes et peut être détruit à tout moment.

Pour y parvenir, pour la partie cluster kubernetes, je vais partir sur un cluster GKE (Google Kubernetes Engine) qui est déjà existant.

Comme décrit à l'introduction de cet article, pour pouvoir utiliser du TLS sur les URLs dynamiques (généré par exemple à chaque MR), il vous faut impérativement un certificat wildcard. De mon côté j'en ai déjà commandé un qui va pointer vers `*.cicd.hatim.com`. Dans cet exemple, j'ai réservé le certificat wildcard chez GlobalSign

mais vous pouvez bien sûr commander chez n'importe quel autre fournisseur.

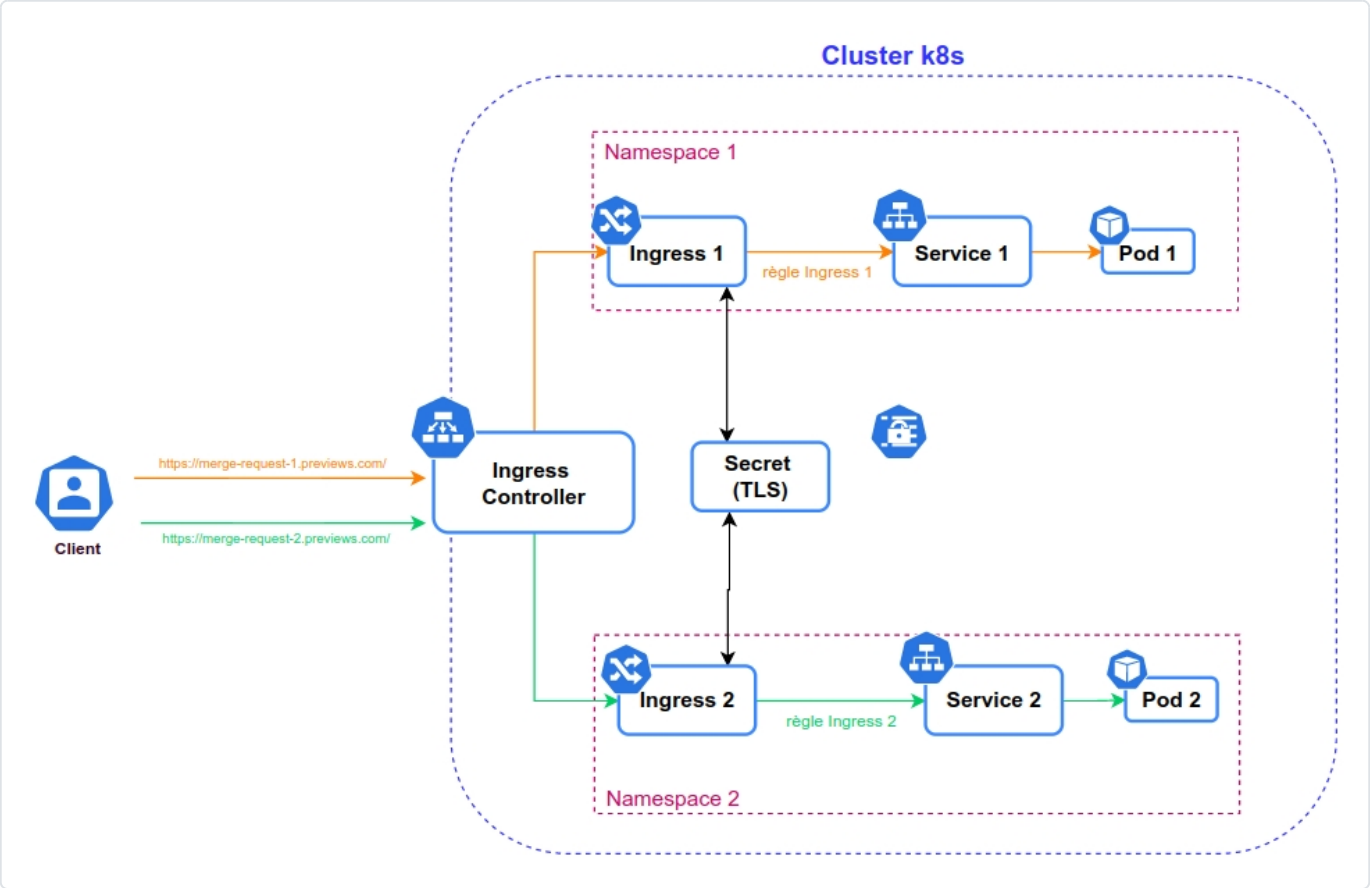


"Commande du certificat wildcard chez GlobalSign"

Création de notre environnement éphémère k8s

Création d'un secret k8s partagé

Une raison courante d'utiliser un secret est d'ajouter un certificat SSL/TLS à un cluster qui sera utilisé pour sécuriser nos applications éphémères exécutées sur Kubernetes. Ce qui nous donnera le schéma suivant :



Ce secret TLS doit contenir le contenu de deux fichiers, le premier est le fichier avec l'extension `*.crt` qui représente le certificat wildcard signé par l'autorité de certification et un autre fichier avec l'extension `*.key` qui représente la clé privée du certificat wildcard.

Le moyen le plus simple de créer un secret TLS dans Kubernetes consiste à utiliser la commande `kubectl` suivante:

```
kubectl create secret tls wildcard-cicd-hatim-com --cert=tls.crt --key=tls.key
```

Notre certificat est bien créé mais nous allons être confrontés à un véritable problème … En effet, les ressources k8s de type secrets résident uniquement dans un namespace. Ils ne peuvent être utilisés que par des ressources dans ce même namespace, mais pour notre besoin nous allons créer nos ressources sur différents namespaces, nous avons donc besoin d'un outil qui va nous permettre de **partager**

notre secret TLS sur tous les namespaces, et ça tombe bien car il existe un outil nommé Kubed qui répond parfaitement à notre besoin.

> **Information**
>
> Helm est un outil de déploiement Kubernetes permettant d'automatiser la création, le conditionnement, la configuration et le déploiement d'applications et de services sur des clusters Kubernetes .
>
> Vous pouvez installer le client helm en suivant les instructions helm.

Kubed peut être installé via Helm:

```
helm repo add appscode https://charts.appscode.com/stable/
helm repo update
helm search repo appscode/kubed
helm install kubed appscode/kubed
```

Afin de synchroniser notre secret `wildcard-cicd-hatim-com` entre tous les namespaces nous devons lui rajouter l'annotation `kubed.appscode.com/sync=""` comme suit:

```
kubectl annotate secrets wildcard-cicd-hatim-com kubed.appscode.com/sync=""
```

Ingress Controller Nginx

Avant de vous parler de l'Ingress Controller Nginx, je vais d'abord vous expliquer l'intérêt des Ingress et des Ingress Controllers.

C'est quoi un Ingress ?

Bien que les pods d'un cluster Kubernetes puissent facilement communiquer entre eux, ils ne sont pas accessibles par défaut aux réseaux et au trafic externe. Une ressource Ingress dans Kubernetes est donc un objet API qui montre comment le trafic provenant de l'extérieur du cluster doit atteindre les services internes du cluster Kubernetes qui envoient des requêtes à des groupes de pods.

L'Ingress lui-même n'a aucun pouvoir. Il s'agit d'une demande de configuration pour l'Ingress Controller qui permet à l'utilisateur de définir comment les clients externes sont acheminés vers les services internes d'un cluster. l'Ingress Controller entend cette demande et ajuste sa configuration pour faire ce que l'utilisateur demande.

Chaque Ingress Controller a sa propre syntaxe de configuration, et la beauté de la ressource Ingress est qu'elle agit comme une couche d'abstraction au-dessus de ces spécificités. Un utilisateur n'a pas besoin de savoir quel Ingress Controller se trouve dans un cluster. La même ressource Ingress aura le même résultat dans chaque cluster Kubernetes avec n'importe quel Ingress Controller. Autrement dit, les Ingress Kubernetes définissent la façon dont le cluster achemine le trafic de la couche 7 pour les requêtes HTTP/HTTPS.

Ils viennent également avec un tas de fonctionnalités supplémentaires. Par exemple, les entrées permettent de définir la configuration TLS utilisée pour la terminaison TLS (ce que nous ferons plus tard pour utiliser notre secret TLS avec le certificat wildcard dans nos ingress).

C'est quoi un Ingress controller ?

Un Ingress Controller agit comme un reverse proxy et un LoadBalancer, il implémente une entrée Kubernetes. l'Ingress Controller ajoute une couche

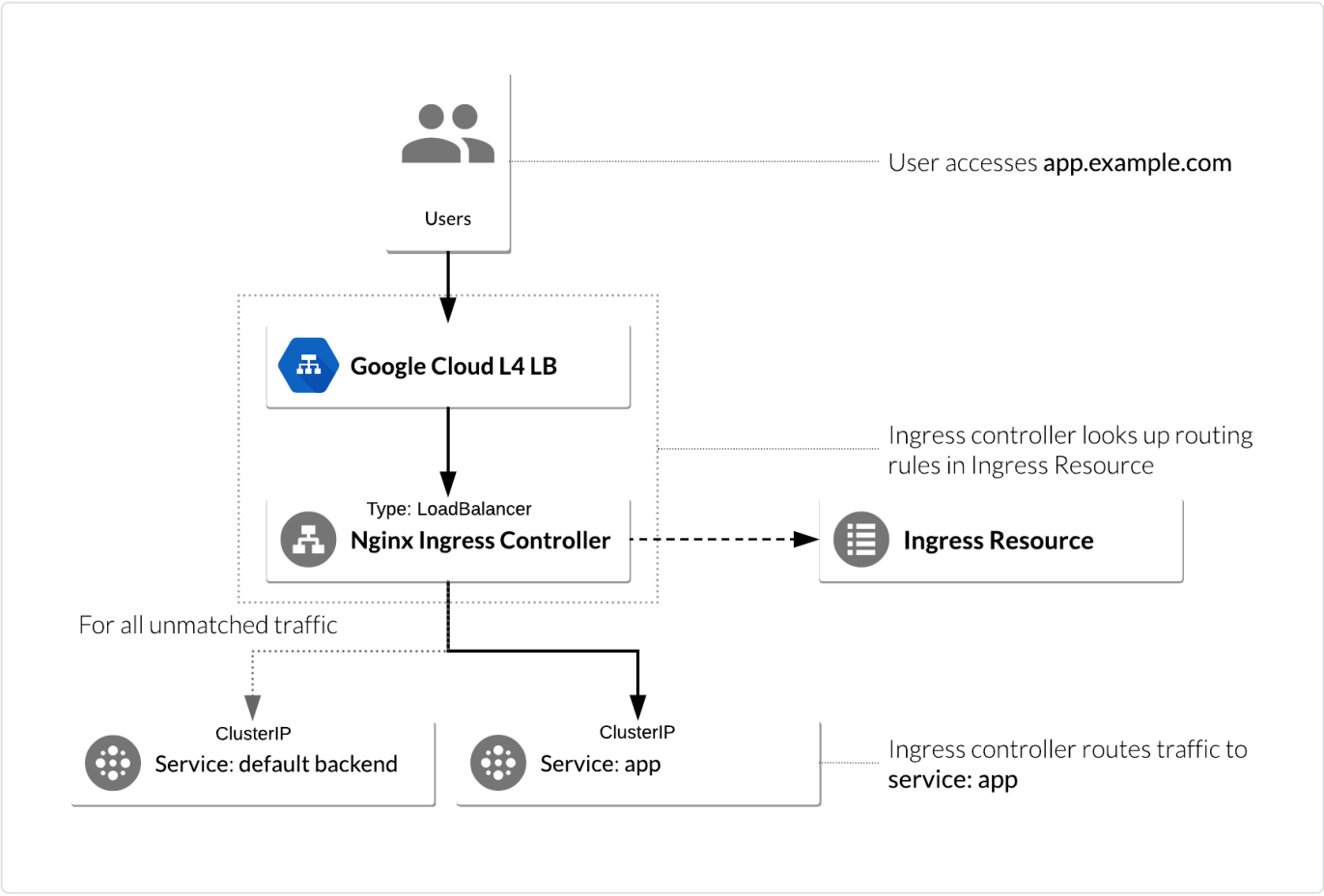
d'abstraction au routage du trafic, acceptant le trafic provenant de l'extérieur de la plate-forme Kubernetes et l'équilibrant en charge vers les pods exécutés à l'intérieur de la plate-forme. Il convertit les configurations des ressources Ingress en règles de routage que les reverse proxys peuvent reconnaître et mettre en œuvre.

Il agit en surveillant les ressources Ingress à l'intérieur d'un cluster, en les transformant en un état demandé par l'utilisateur. Par ailleurs, un cluster Kubernetes peut avoir plusieurs Ingress Controller. Lorsque l'utilisateur créent, met à jour ou supprime une entrée, l'Ingress Controller reçoit l'événement et lit la configuration à partir des spécifications et des annotations de l'Ingress. L'Ingress Controller convertit ensuite la configuration YAML ou JSON exprimée par l'utilisateur en quelque chose que le reverse proxy peut comprendre et intégrer au cluster.

Mise en œuvre

Pour gérer le routage dynamique des urls éphémères et avoir le contrôle sur tous nos Ingress sur tous les namespaces, nous allons utiliser un Ingress Controller Nginx

Voici un flux de base de la solution de l'Ingress Controller Nginx sur GKE (Google Kubernetes Engine):



Ici l'Ingress Controller Nginx est déployé en tant que service et exposé pour un accès externe. Cela se fait car le service de contrôleur NGINX utilise un service de `type: LoadBalancer`. Sur GKE, cela crée automatiquement un équilibreur de charge (couche 4) Google Cloud Network (TCP/IP) avec le service de contrôleur NGINX en tant que backend. Google Cloud crée également les règles de pare-feu appropriées au sein du réseau VPC du Service pour autoriser le trafic Web HTTP(S) vers l'adresse IP frontale de l'équilibreur de charge.

Passons maintenant à la pratique. Nous utiliserons Helm afin de faciliter le déploiement du Contrôleur Nginx:

```
helm repo add ingress-nginx https://kubernetes.github.io/ingress-nginx
helm repo update
```

Ensuite nous allons déployer le Deployment et le Service du contrôleur NGINX dans un namespace personnalisé en exécutant la commande suivante:

```
kubectl create namespace nginx-controller
helm install --namespace="nginx-controller" preview ingress-nginx/ingress-nginx
```

Par la suite, vérifiez que les ressources se sont déployées sur le cluster GKE dans le namespace `nginx-controller`:

```
kubectl get deployment preview-ingress-nginx-controller -n nginx-cor
kubectl get service preview-ingress-nginx-controller -n nginx-contro
```

Résultat :

```
NAME                                READY   UP-TO-DATE   AVAILABLE   AGE
preview-ingress-nginx-controller    1/1     1            1           3m27s

NAME                                TYPE           CLUSTER-IP       EXTERNAL-IP   PORT
preview-ingress-nginx-controller    LoadBalancer   10.XX.XX.XX      80:XXXXX/TCP,443:X
```

Information

Pour des raisons de sécurité, je cache certaines informations de mon cluster par des "X". Dans le résultat précédent j'ai par exemple caché l'IP de mon service par le symbole "X".

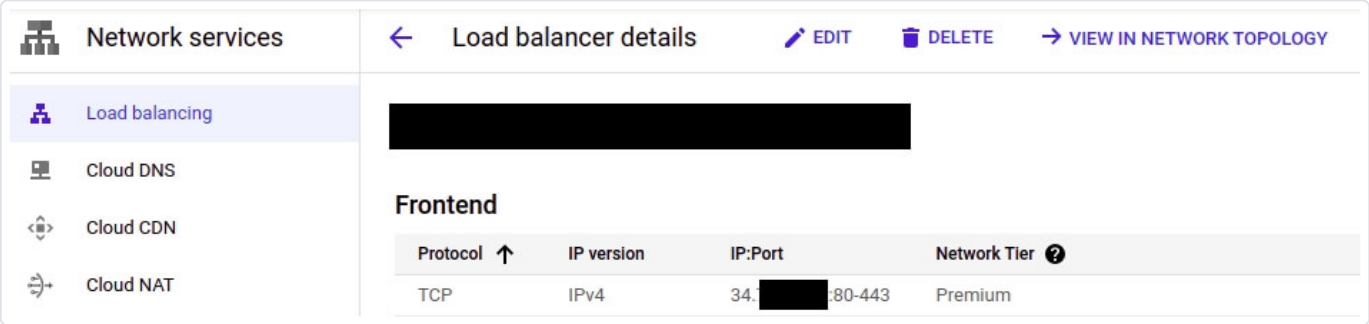
Attendez quelques instants pendant que le déploiement de l'équilibreur de charge Google Cloud L4 se fasse, puis confirmez que le Service de l'Ingress Controller Nginx a été déployé et qu'une adresse IP externe est associée au service:

```
kubectl get service preview-ingress-nginx-controller -n nginx-controller
```

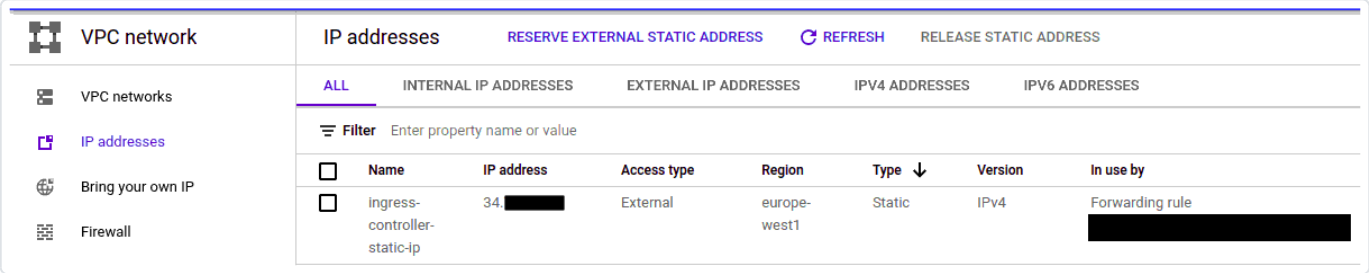
Résultat :

```
NAME                                TYPE           CLUSTER-IP      EXTERNAL-IP   PORT
preview-ingress-nginx-controller    LoadBalancer   10.XX.XX.XX     34.XX.XX.XX   80:XXXX
```

Dans cet exemple, le service vient de créer un équilibreur de charge Google Cloud L4 avec une adresse IP publique `34.XX.XX.XX` que vous pouvez trouver dans la console gcp:



Ainsi que l'IP Statique:



Auparavant, nous avions créé un certificat wild pouvant être utilisé pour les URL `*.cicd.hatim.com`. Il faut maintenant créer une entrée DNS : `*.cicd.hatim.com` vers l'ip publique utilisée par le service Nginx Controller, soit:

```
*.cicd.hatim.com => 34.XX.XX.XX
```

Création de l'environnement éphémère

L'idée générale derrière l'architecture que nous allons déployer est de parvenir à créer facilement des environnements éphémères. Pour ce faire, nous allons créer toutes nos ressources de notre application dans un namespace différent et ce

namespace sera notre environnement éphémère.

Namespaces

La première étape consiste à créer deux namespaces différents où nous déploierons la première application sur le premier namespace et une deuxième sur le second namespace:

```
kubectl create namespace test-preview-1
kubectl create namespace test-preview-2
```

Applications

L'étape suivante consiste à déployer l'application 1 qui sera une application simple avec la page d'accueil nginx et la deuxième application est basée sur l'image hello-app qui affiche uniquement les informations du conteneur sur sa page d'accueil. Voici les commandes de déploiement sur leurs namespaces respectifs:

Application 1:

```
kubectl create deployment nginx-app --image=nginx:1.7.9 -n test-preview-1
kubectl expose deployment nginx-app --port=80 --target-port=80 -n test-preview-1
```

Vérifiez que vos ressources ont bien été créées:

```
kubectl get deploy,service -n test-preview-1 -o wide
```

Résultat :

```
NAME                         READY   UP-TO-DATE   AVAILABLE   AGE     CONTAINERS   IMA
deployment.apps/nginx-app    1/1     1            1           3m58s   nginx        ngi

NAME                 TYPE        CLUSTER-IP     EXTERNAL-IP   PORT(S)   AGE     SELECT
service/nginx-app    ClusterIP   10.XX.XX.XX           80/TCP    2m47s   app=nginx-app
```

Application 2:

```
kubectl create deployment hello-app --image=gcr.io/google-samples/hello-app:1.0 -n te
kubectl expose deployment hello-app --port=8080 --target-port=8080 -n test-preview-2
```

Vérifiez que vos ressources ont bien été créées:

```
kubectl get deploy,service -n test-preview-2 -o wide
```

Résultat :

```
NAME                          READY   UP-TO-DATE   AVAILABLE   AGE
deployment.apps/hello-app     1/1     1            1           2m23s

NAME                  TYPE        CLUSTER-IP       EXTERNAL-IP   PORT(
service/hello-app     ClusterIP   10.XX.XX.XX           8080/TCP    2m1
```

Ingress

Pour que votre application soit accessible depuis une URL éphémère, vous devez créer un Ingress dans le namespace que votre application. Cette entrée contiendra les informations suivantes :

- L'url éphémère pour accéder à votre application.
- Le service de votre application en backend.
- Le certificat wildcard à utiliser que nous avions stocké précédemment dans notre secret.

Autre chose à savoir et comme expliqué plus haut, un objet Ingress est un ensemble de règles L7 pour acheminer le trafic entrant vers les services Kubernetes et contrôlé par l'Ingress Controller. Pour les Ingress que nous souhaitons créer, on veut qu'ils soient contrôlés par l'Ingress Controller Nginx que nous avons créé précédemment. Cela peut être défini avec l'annotation `kubernetes.io/ingress.class: nginx` dans la section des métadonnées dans

l'Ingress ou en utilisant l'`IngressClass Nginx` créé auparavant par le helm avec le paramètre `ingressClassName: nginx` dans les spécifications de la ressource Ingress.

Nous allons également forcer l'utilisation de notre certificat wildcard grâce au paramétrage de la partie `tls` dans les spécifications de la ressource Ingress.

Pour cela, créez un premier fichier Yaml qui contiendra vos informations d'entrée pour l'application 1:

```
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: ingress-preview-1
  namespace: test-preview-1
spec:
  ingressClassName: nginx
  rules:
    - host: test-preview-1.cicd.hatim.com
      http:
        paths:
          - pathType: Prefix
            backend:
              service:
                name: nginx-app
                port:
                  number: 80
            path: /
  tls:
    - hosts:
      - test-preview-1.cicd.hatim.com
      secretName: wildcard-cicd-hatim-com
```

Ensuite, créez-le:

```
kubectl create -f ingress-1.yml
```

Au moment de la rédaction de cet article, il y a un bug dans la ressource `ValidatingWebhookConfiguration`. Cette ressource vérifie la configuration de l'Ingress avant de le créer, mais actuellement pour chaque création d'Ingress, elle nous affiche le message suivant:

```
Error from server (InternalError): error when creating "ingress-1.yml": Internal erro
```

La solution que j'ai trouvée et de la sauvegarder et ensuite la supprimer pour permettre la création de l'ingress:

```
kubectl get ValidatingWebhookConfiguration preview-ingress-nginx-admission -o yaml >
kubectl delete ValidatingWebhookConfiguration preview-ingress-nginx-admission
```

Sinon vous pouvez aussi demander à Helm de déployer l'Ingress Controller Nginx sans activer la `ValidatingWebhookConfiguration`

```
helm install --namespace="test-hatim" preview ingress-nginx/ingress-nginx --set contr
```

On fait la même chose pour l'application 2:

```
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: ingress-preview-2
  namespace: test-preview-2
spec:
  ingressClassName: nginx
  rules:
    - host: test-preview-2.cicd.hatim.com
      http:
        paths:
          - pathType: Prefix
            backend:
              service:
                name: hello-app
                port:
                  number: 8080
            path: /
  tls:
    - hosts:
      - test-preview-2.cicd.hatim.com
      secretName: wildcard-cicd-hatim-com
```

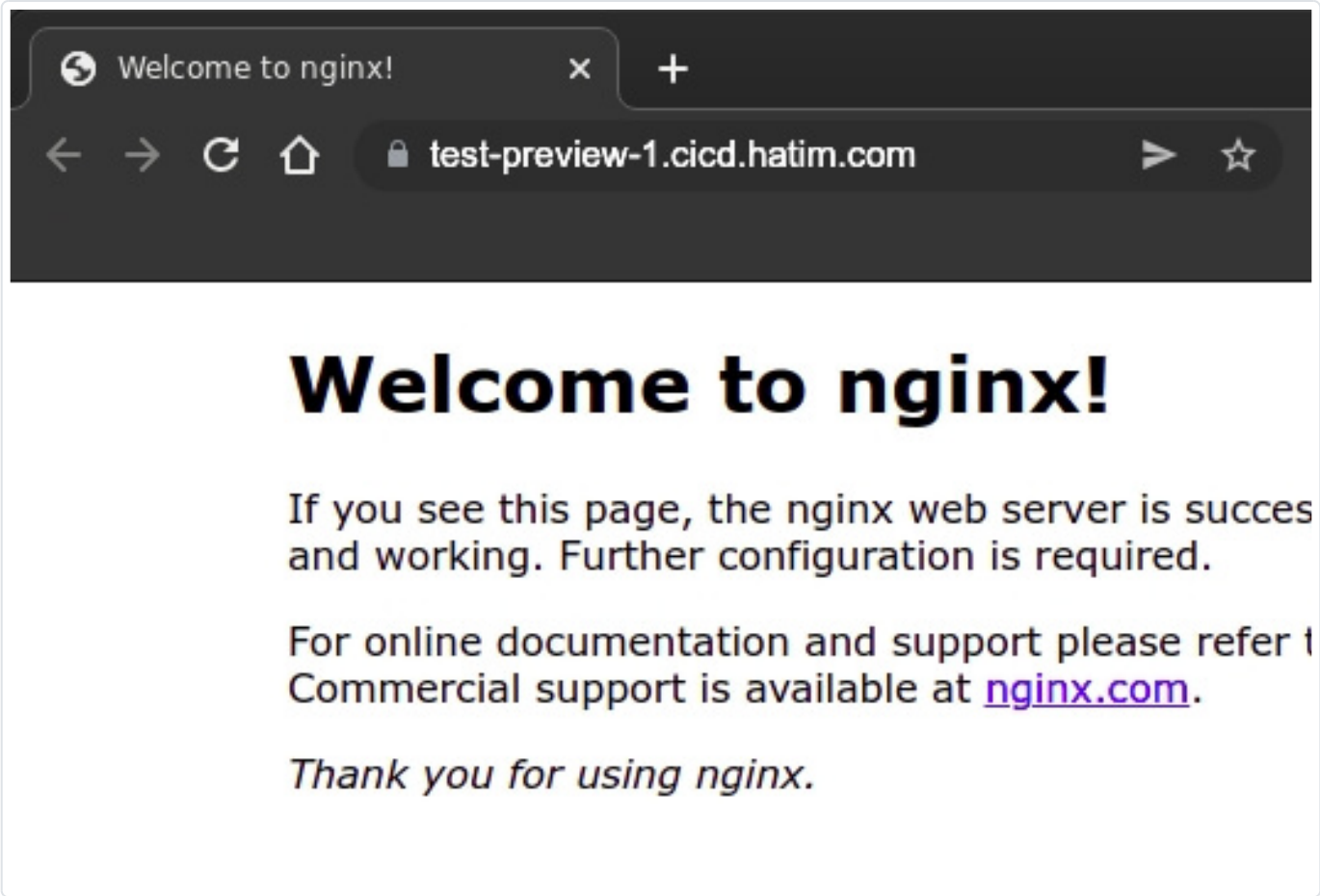
Ensuite, créez-le:

```
kubectl create -f ingress-2.yml
```

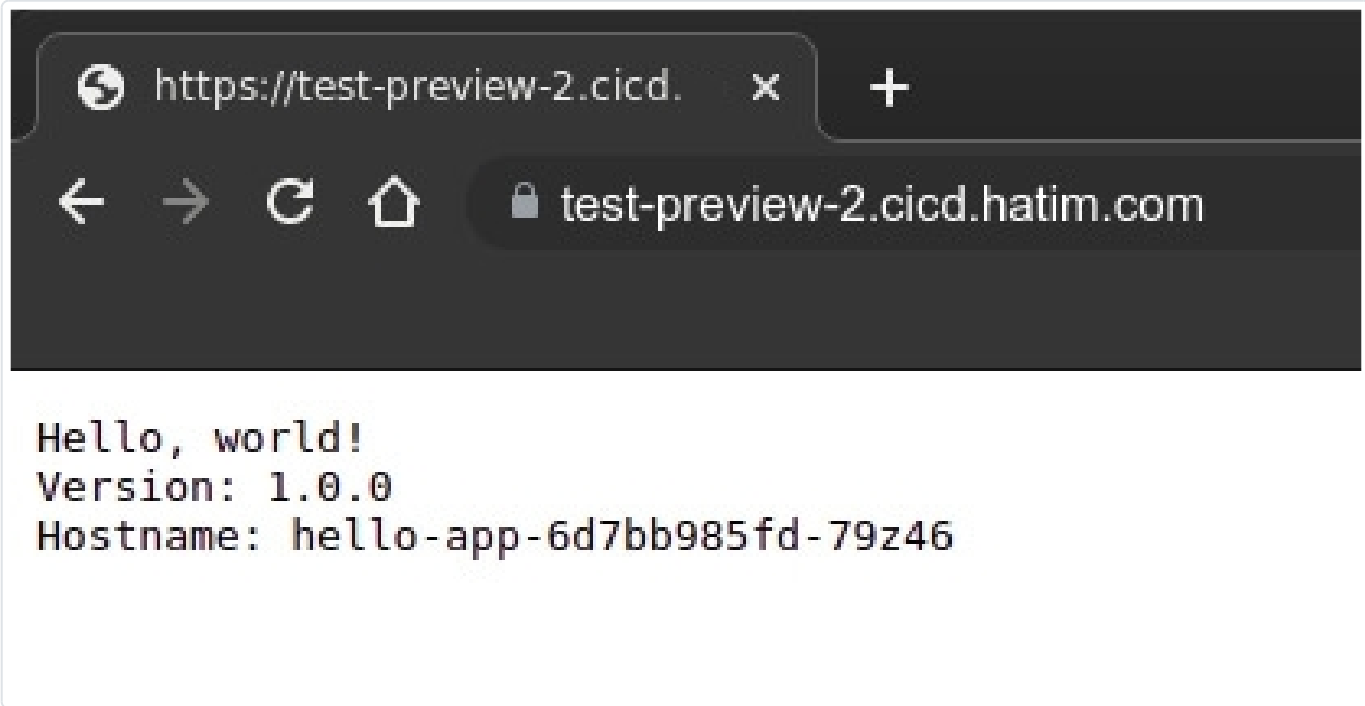
Nos deux applications sont désormais accessibles depuis l'url `test-preview-1.cicd.hatim.com` et `test-preview-2.cicd.hatim.com`

Tester

Depuis le navigateur accédez à votre application 1 en HTTPS:



Et l'application 2 :



Supprimer l'environnement éphémère

Pour supprimer vos environnements éphémères, supprimez simplement les 2 espaces de noms créés précédemment

```
kubectl delete namespace test-preview-1
kubectl delete namespace test-preview-2
```

Conclusion

Nous arrivons à terme de cet article qui a été très fun à réaliser. Vous remarquerez que je n'ai pas traité la partie CI/CD notamment sur Gitlab car je préfère d'abord faire un cours complet dessus.

Nous venons de voir en pratique qu'il est très facile et rapide de créer des environnements éphémères sur un cluster k8s ce qui nous permet d'utiliser des ressources ne nécessitant pas d'être engagées en permanence. C'est notamment très pratique pour tester la préparation de la production sur un nouvel

environnement Production-like avec le nouveau code déployé et qui n'a pas besoin d'être présent une fois les tests terminés.