1. Créer un conteneur pour la base de données MySQL

docker run -d - -name mysql-db -v db-vol:/var/lib/mysql -p 3307:3306 mysql:5.7.44

 L’option de stockage la plus adéquate est celle des volumes nommés, car les volumes Docker sont conçus pour être indépendants du cycle de vie des conteneurs. Cela signifie que même si le conteneur est arrêté ou supprimé, les données stockées dans le volume persistent. De plus, ils garantissent une protection contre toute manipulation externe à Docker et permettent une distinction du volume créé par son nom sans nécessiter une connaissance préalable du système de fichiers de Docker.

1. Créer un réseau Docker (tout en lui précisant un driver convenable). Puis connecter le conteneur de la base de données mysql avec ce réseau.

docker network create - -driver bridge - -subnet 10.10.10.0/24 reseau

****

Le driver convenable pour ce projet est ‘bridge’ car on ne veut connecter que les conteneurs qu’on va créer dans un même démon docker.

****

1. Créer un fichier Dockerfile pour le projet Backend (essayez d’appliquer le maximum de bonnes pratiques).

# syntax=docker/dockerfile:1

FROM openjdk:19-jdk-alpine3.16 as builder

WORKDIR /app

COPY . /app/

RUN ./mvnw clean package

FROM openjdk:19-jdk-alpine3.16

ENV NODE\_ENV production

WORKDIR /app

RUN addgroup -S group && adduser -S user -G group

USER user

COPY --chown=user:group --from=builder /app/target/spring-boot-data-jpa-0.0.1-SNAPSHOT.jar .

EXPOSE 8080

CMD ["java","-jar","spring-boot-data-jpa-0.0.1-SNAPSHOT.jar"]

Explication du dockerfile :

* + - * **FROM openjdk:19-jdk-alpine3.16 as builder:** définit l'image de base pour la première étape de construction «OpenJDK 19 »basée sur Alpine Linux nommée "builder" pour référence dans l’étape de production.
      * **WORKDIR /app :** Définit le répertoire de travail à l'intérieur du conteneur (/app).
      * **COPY . /app/ :** Copie le contenu du répertoire local où se trouve le Dockerfile dans le répertoire /app du conteneur.
      * **RUN ./mvnw clean package**: Exécute la commande Maven pour nettoyer le projet et construire le package.
      * **FROM openjdk:19-jdk-alpine3.16 :** définit l'image de base pour l’étape de production «OpenJDK 19 ».
      * **ENV NODE\_ENV production :** Définit la variable d'environnement NODE\_ENV à la valeur "production".
      * **RUN addgroup -S group && adduser -S user -G group:** Ajoute un groupe « group » et un utilisateur « user » au système sans mot de passe( l’intérêt de l’option -S).
      * **USER user :** Passe à l'utilisateur non-root « user » nouvellement créé.
      * **COPY --chown=user:group --from=builder /app/target/spring-boot-data-jpa-0.0.1-SNAPSHOT.jar . :** Copie le fichier JAR généré à partir de l'étape précédente (étape builder) dans le répertoire courant du conteneur en définissant le propriétaire et le groupe du fichier copié à user et group.
      * **EXPOSE 8080 :** Indique que le conteneur écoute sur le port 8080.
      * **CMD ["java","-jar","spring-boot-data-jpa-0.0.1-SNAPSHOT.jar"]:** Spécifie la commande par défaut à exécuter lorsque le conteneur démarre. Dans ce cas, il exécute le fichier JAR avec la commande java -jar.

1. Je vais présenter ici les bonnes pratiques que j’ai utilisé dans les projets Frontend ou/et Backend :
   1. Utilisez des builds en plusieurs étapes :

Les constructions multi-étapes permettent de réduire la taille de l’image finale en créant une séparation nette entre l’image de la construction et celle de la production et assurent que cette dernière ne contient que les fichiers nécessaires à l'exécution de l'application. Ils permettent également de construire de manière plus efficace en exécutant des étapes de construction en parallèle.

Pour le Backend :

* L’étape de construction est basée sur une image jdk 19. On a défini le répertoire de travail (/app) et y copié les fichiers nécessaires pour compiler l’application en installant l’environnement Maven du projet et créant l’exécutable .jar.
* L’étape de production copie sous un utilisateur non admin «user» puis exécute que l’exécutable .jar pour déployer l’application.

Pour le frontend :

* L’étape de construction est basée sur une image Node. On a copié les fichiers nécessaires pour l’installation des dépendances, et le code de l’application pour la construction.
* Dans l’étape de production, on a créé une image Nginx pour créer un serveur nginx et on y a déployé notre application en copiant les éléments de construction dans /app/dist/angular-16-crud vers le serveur Nginx à l'emplacement /usr/share/nginx/html. La commande avec le CMD sert à démarrer le serveur au moment de l’instanciation du conteneur.
  1. [Exclure les fichiers inutiles avec .dockerignore](https://docs.docker.com/develop/develop-images/guidelines/#exclude-with-dockerignore).
  2. [Créer](https://docs.docker.com/develop/develop-images/guidelines/#create-ephemeral-containers) des conteneurs éphémères en créant des images qui construisent des conteneurs pouvant être stoppés, détruits et remplacés avec le minimum de configuration. Cela est réalisé avec :
     1. Variables d'Environnement (ENV) : Permettent de rendre la configuration de l'application flexible au moment de l'exécution sans reconstruire l'image.
     2. Répertoire de Travail (WORKDIR) : Spécifie le répertoire de travail pour les instructions suivantes dans le Dockerfile.
     3. Script avec CMD : Externalise les actions de configuration, de démarrage ou d'autres tâches au moment du lancement du conteneur sans reconstruire l'image.
  3. [Découpler les applications](https://docs.docker.com/develop/develop-images/guidelines/#decouple-applications)

Chaque conteneur devrait se concentrer sur une seule préoccupation. La découpe des applications en plusieurs conteneurs facilite la mise à l'échelle horizontale et la réutilisation des conteneurs et Si les conteneurs dépendent les uns des autres, on peut utiliser des réseaux de Docker pour assurer leur communication.

Dans mon cas, on a créé trois images, chacune pour une spécification particulière ; une pour le frontend, une pour le backend et une pour la base de données.

* 1. Organiser les arguments sur plusieurs alphanumériquement avec l’ajout d’un espace avant ‘\’ pour faciliter la maintenance en évitant la duplication de packages et rendant la liste beaucoup plus facile à mettre à jour. (Je n’ai pas rencontré des commandes avec plusieurs arguments.)
  2. Structurer les instructions de manière à maximiser l'utilisation du cache de construction.
  3. Utiliser la forme CMD ["executable", "param1", "param2"] pour les instructions CMD.
  4. Utiliser des ports communs et traditionnels pour les appliactions dans les instructions EXPOSE.
  5. Utiliser VOLUME pour toute combinaison de parties mutables ou modifiables par l'utilisateur de l’image (stockage de base de données, stockage de configuration, ou des fichiers et dossiers créés par le conteneur). (Je n’ai pas besoin d’utiliser un volume VOLUME, car je copie simplement les fichiers nécessaires à l’application et j’utilise carrément MySQL pour stocker les données de l’application).
  6. Utiliser des chemins absolus pour WORKDIR. De plus, il est recommandé utiliser WORKDIR au lieu de multiplier les instructions du type RUN cd … && #autres-instructions.
  7. Utiliser des balises d’image de base Docker explicites et déterministes
     1. Utiliser une image officielle (la récente si possible).
     2. Préciser la version utilisée de l’image.
     3. Utiliser des images de petite taille (une image alpine).
  8. Installer uniquement les dépendances stables dans l’image Docker (de production)

Dans le projet frontend, CLI Angular est nécessaire pour exécuter ‘npm run build’ mais il se trouve dans les devdependencies, donc je n’ai pas respecté cette pratique.

Dans le projet backend, j’ai été censée créer un environnement Maven adéquat pour l’application en se basant sur Maven Wrapper et donc j’ai apporté dans l’étape de construction tous les fichiers associés. Mais dans l’étape de production, seul l’exécutable .jar est chargé dans l’image. J’ai pu suivre une autre démarche qui consiste de au lieu de déléguer la compilation du code à Docker, on peut tout simplement compiler l’application localement et se contenter de l’étape de production dans le dockerfile. Mais cette méthode contient quelques inconvénients.

* 1. Optimiser l’outillage de production (ENV NODE\_ENV production).
  2. Ne pas exécuter les conteneurs en tant que root.
  3. Quitter en toute sécurité les applications web Docker Node.js(Eviter ‘CMD "npm" "start"’ ou ‘CMD “node” “server.js”’ et utiliser l’outil dumb-init pour gérer les signaux dans le conteneur).

Nginx est généralement capable de gérer correctement les signaux, même sans dumb-init).

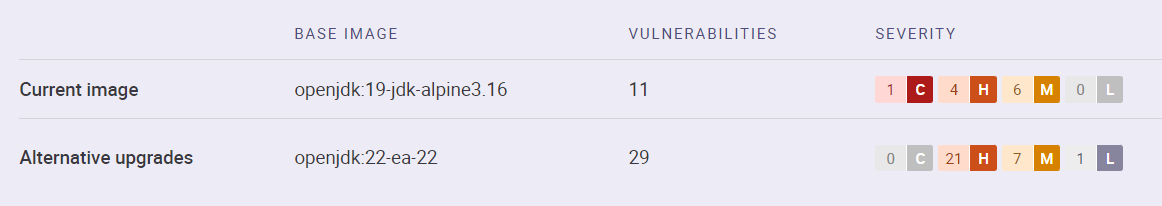
Cette bonne pratique est spécifique à node donc elle est absente dans le dockerfile de l’application du backend.

1. Taper les commandes docker nécessaires pour :
   1. Créer une image docker en local à partir de ce fichier Dockerfile.

>docker build -t mesnaoui/img-backend:1.0.0 .

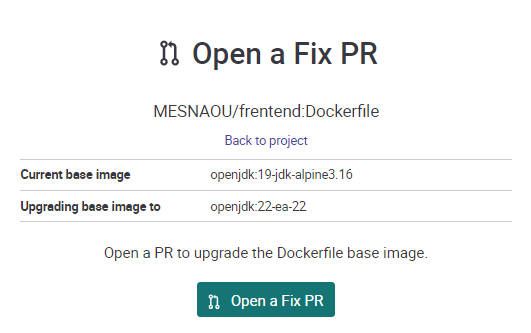
* 1. Scanner l’image des vulnérabilités qu’elle peut contenir. En cas d’existence de ces vulnérabilités, effectuer les actions nécessaires pour les éviter.

Pour scanner mon image, je vais utiliser Snyk :



D’après l’image ci-dessus, mon dockerfile contient 1 vulnérabilité critique, 4 haute et 6 moyennes).

Snyc propose de mettre à niveau l’image du jdk dans le deux étapes:



# syntax=docker/dockerfile:1

FROM openjdk:22-ea-22 as builder

WORKDIR /app

COPY . /app/

RUN ./mvnw clean package

FROM openjdk:22-ea-22

ENV NODE\_ENV production

WORKDIR /app

RUN addgroup -S group && adduser -S user -G group

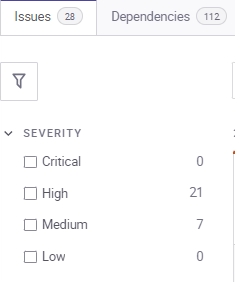
USER user

COPY --chown=user:group --from=builder /app/target/spring-boot-data-jpa-0.0.1-SNAPSHOT.jar .

EXPOSE 8080

CMD ["java","-jar","spring-boot-data-jpa-0.0.1-SNAPSHOT.jar"]

Ce qui a réduit les vulnérabilités :



* 1. Publier cette image dans Docker Hub.

>docker push mesnaoui/img-backend:1.0.0

* 1. Instancier cette image en créant un conteneur s’exécutant en local et partageant le même réseau avec le conteneur de la base de données.



L’intérêt de chaque option est :

* -d : executer le conteneur en arrière-plan.
* --name : attriburer le nom spécifié (backend) au conteneur créé.
* -p : mapper le port 8080 de la machine hôte au port 8080 du conteneur créé.
* --network : connecter le conteneur au réseau ‘reseau’.
* Le conteneur est basé sur l’image mesnaoui/img-backend:1.0.0
  1. Inspecter ce conteneur du backend

>docker inspect a135

‘a135’ est l’id du conteneur.

Cette commande permet d’afficher les informations détaillées sur le conteneur.

* 1. Afficher les logs liés à ce conteneur du backend.

>docker logs a135

Cette commande permet de consulter les journaux générés par l'exécution du conteneur.

1. Refaire le travail demandé dans les questions 3), 4) et 5) pour le projet Frontend.
   1. Créer un fichier Dockerfile pour le projet Frontend.

# syntax=docker/dockerfile:1

FROM node:21-alpine3.17 as builder

WORKDIR /app

COPY package\*.json ./

RUN npm install

COPY . .

RUN npm run build --prod

FROM nginx:alpine3.18-slim

ENV NODE\_ENV production

COPY --from=builder /app/\* /usr/share/nginx/html/

EXPOSE 80

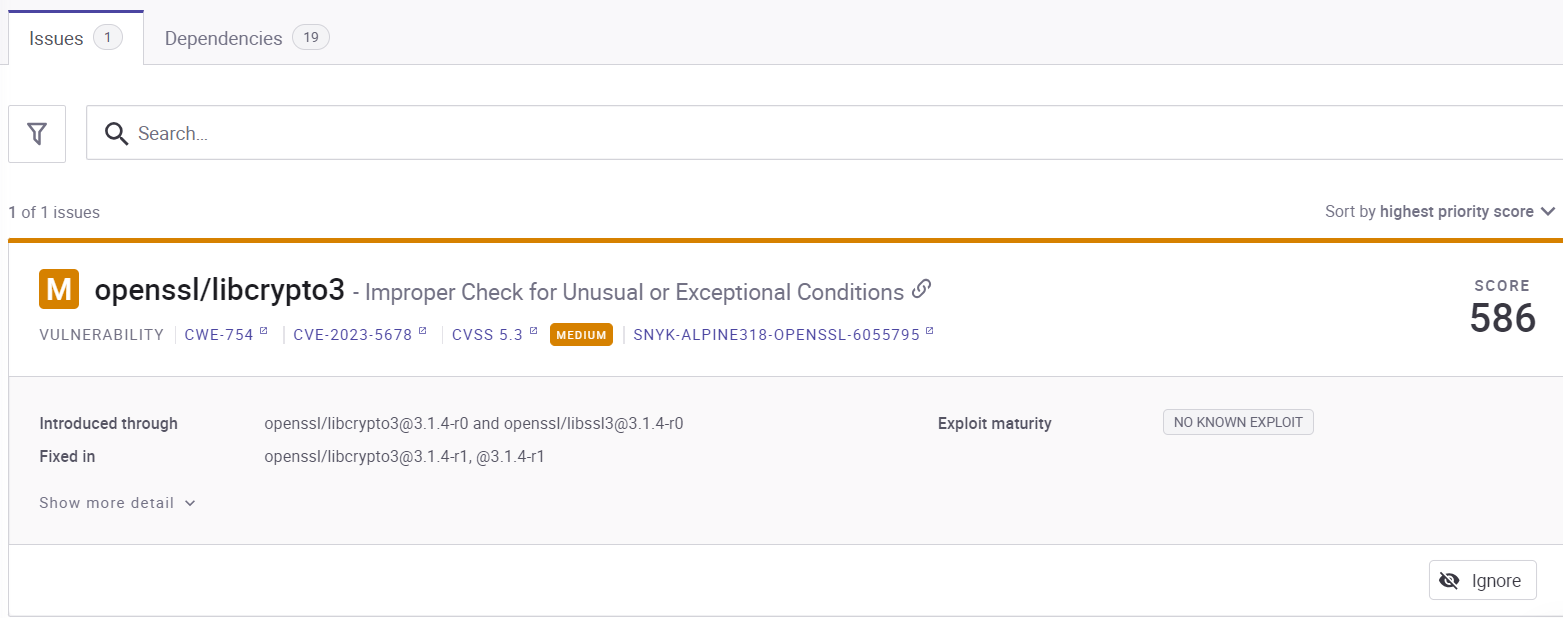
CMD ["nginx", "-g", "daemon off;"]

* + - * **FROM node:21-alpine3.17 as builder** : Cette ligne spécifie l'image de base à utiliser pour l’étape de construction (builder). Elle utilise l'image officielle de Node.js version 21 basée sur Alpine Linux version 3.17 et l'identifie comme "builder" pour référence ultérieure.
      * **WORKDIR /app :** Définit le répertoire de travail à /app, ce qui signifie que les commandes suivantes seront exécutées à partir de ce répertoire.
      * **COPY package\*.json ./ :** Copie les fichiers package.json et package-lock.json dans le répertoire de travail (/app).
      * **RUN npm install :** Exécute la commande npm install pour installer les dépendances définies dans les fichiers package.json et package-lock.json.
      * **COPY . . :** Copie les fichiers locales dans le répertoire de travail (/app).
      * **RUN npm run build --prod :** Exécute la commande npm run build --prod pour construire l'application
      * **FROM nginx:alpine3.18-slim :** Définit l’étape de construction basée sur l'image officielle de Nginx version 1.18 basée sur Alpine Linux version 3.18.
      * **ENV NODE\_ENV production :** Définit la variable d'environnement NODE\_ENV à "production".
      * **COPY --from=builder /app/\* /usr/share/nginx/html/ :** Copie les fichiers générés lors de la phase de construction à partir de l'étape "builder" dans le répertoire de travail de l'image Nginx (/usr/share/nginx/html/).
      * **EXPOSE 80 :** Indique que le conteneur écoute sur le port 80.
      * **CMD ["nginx", "-g", "daemon off;"] :** Définit la commande par défaut à exécuter lorsque le conteneur est démarré. Dans ce cas, il exécute Nginx en mode démon.
  1. Créer une image docker en local à partir de ce fichier Dockerfile.



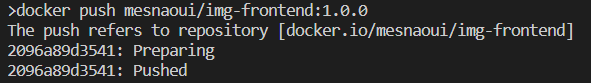
* 1. Scanner l’image des vulnérabilités qu’elle peut contenir. En cas d’existence de ces vulnérabilités, effectuer les actions nécessaires pour les éviter.

Pour scanner mon image, je vais utiliser Snyk :



On constate qu’il y a une seule vulnérabilité. C’est déjà bien.

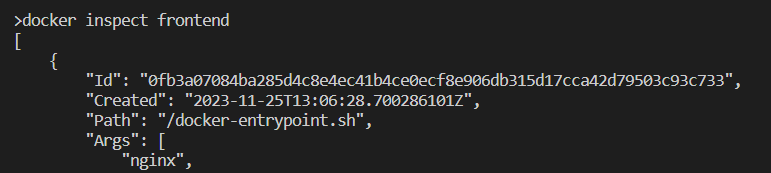
* 1. Publier cette image dans Docker Hub.



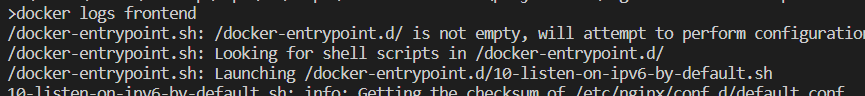
* 1. Instancier cette image en créant un conteneur s’exécutant en local et partageant le même réseau avec le conteneur de la base de données.



* 1. Inspecter ce conteneur du backend



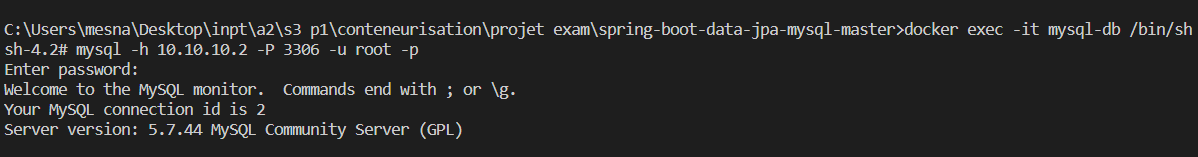
* 1. Afficher les logs liés à ce conteneur du backend.



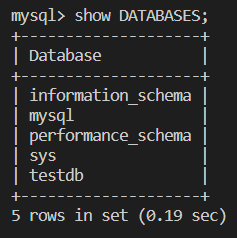
1. Avant de lancer l’application, il faut accéder au conteneur «mysql-db» et créer la base de données testdb :

docker exec -it mysql-db /bin/sh

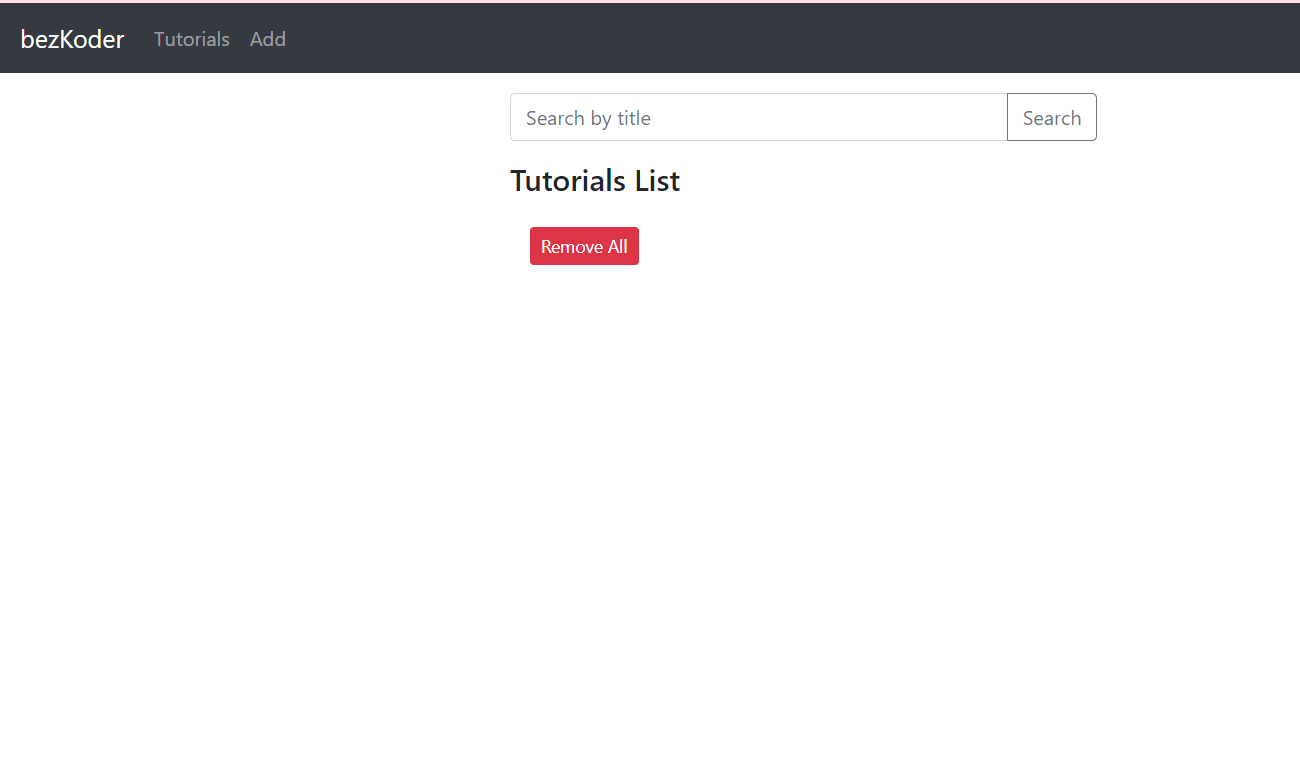
mysql -h 10.10.10.2 -P 3306 -u root -p

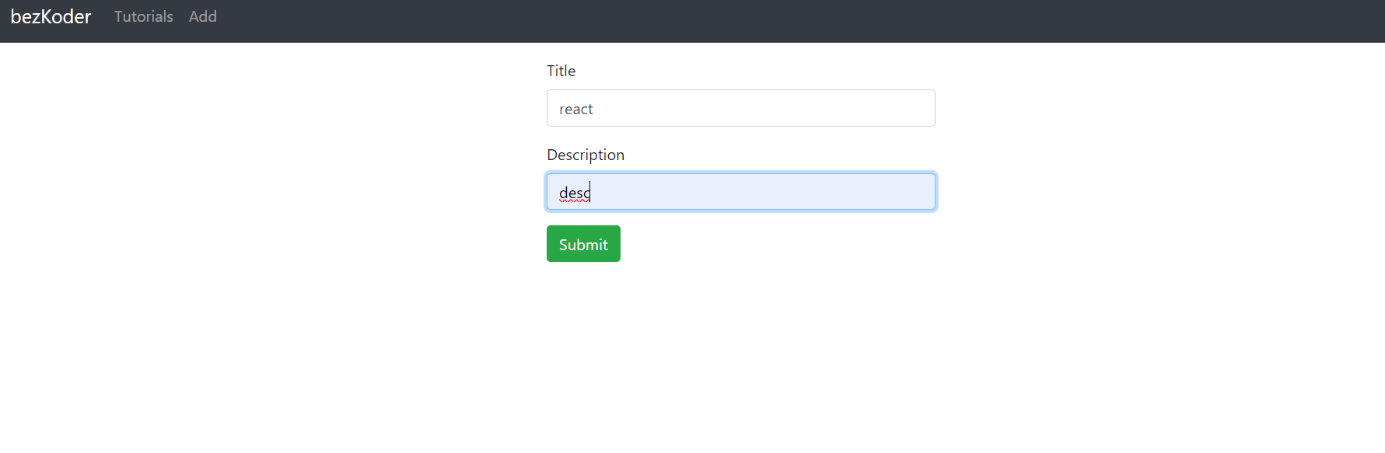


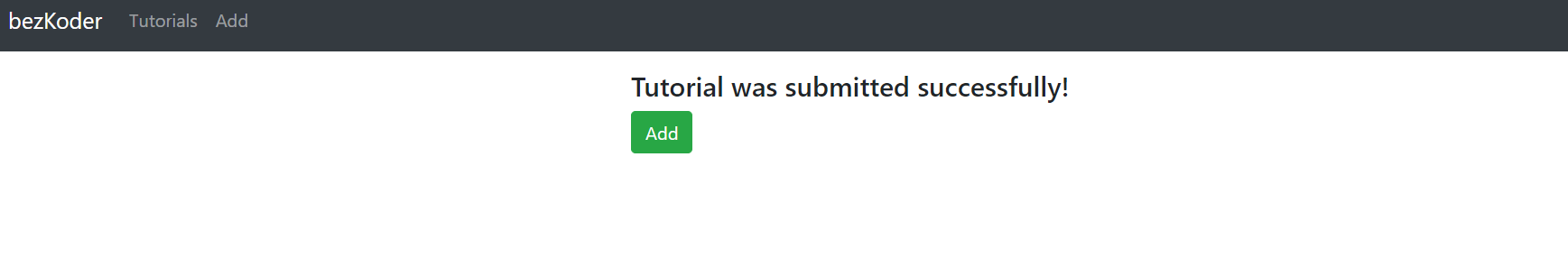
10.10.10.2 : est adresse ip de ‘mysql-db’ dans le réseau ‘reseau’.

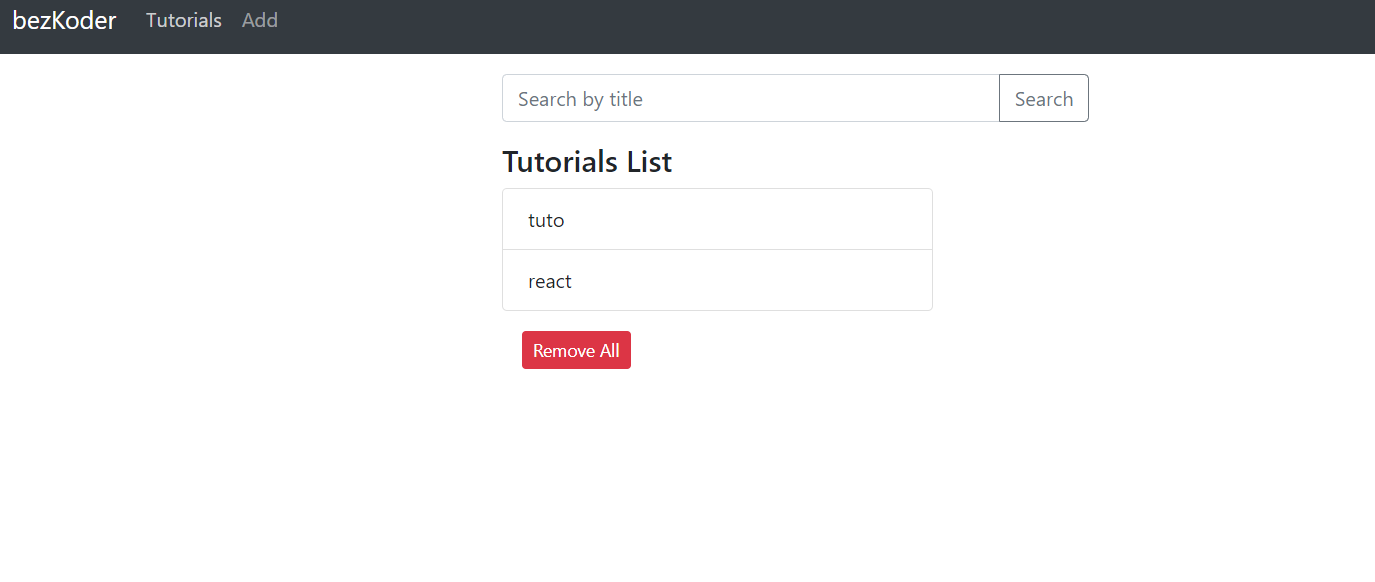


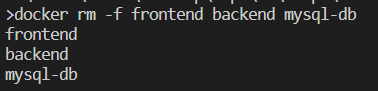
La démo :











L’option ‘-f’ : sert à supprimer des conteneurs même s’ils sont en cours d’exécution.

1. Redéployer l’application (frontend, backend et base de données) en utilisant docker-compose.
   1. Créer le fichier docker-compose.

version: '3.8'

services:

  mysqldb-service:

    image: mysql:5.7.44

    container\_name: mysql-db

    ports:

      - 3307:3306

    networks:

      - reseau

    volumes:

      - mysql-db-vol:/var/lib/db

environment:

MYSQL\_ROOT\_PASSWORD: 123456

MYSQL\_DATABASE: testdb

  backend:

    image: mesnaoui/img-backend:1.0.0

    container\_name: backend

    ports:

      - 8080:8080

    networks:

      - reseau

    depends\_on:

      - mysqldb-service

  frontend:

    image: mesnaoui/img-frontend:1.0.0

    container\_name: frontend

    ports:

      - 8081:80

    networks:

      - reseau

    depends\_on:

      - backend

networks:

  reseau:

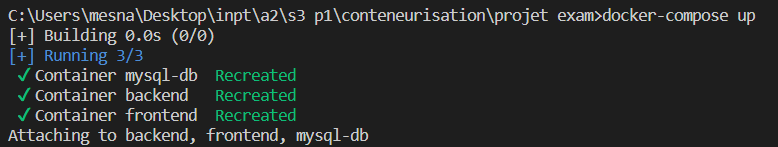
    driver: bridge

volumes:

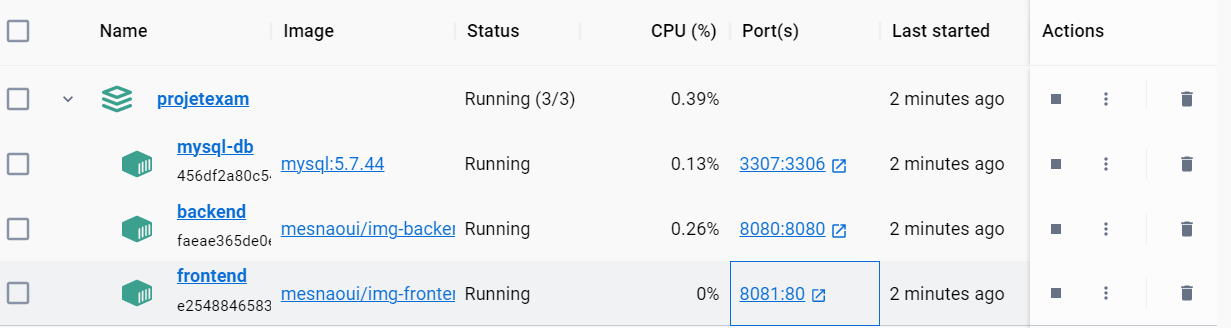
  mysql-db-vol:

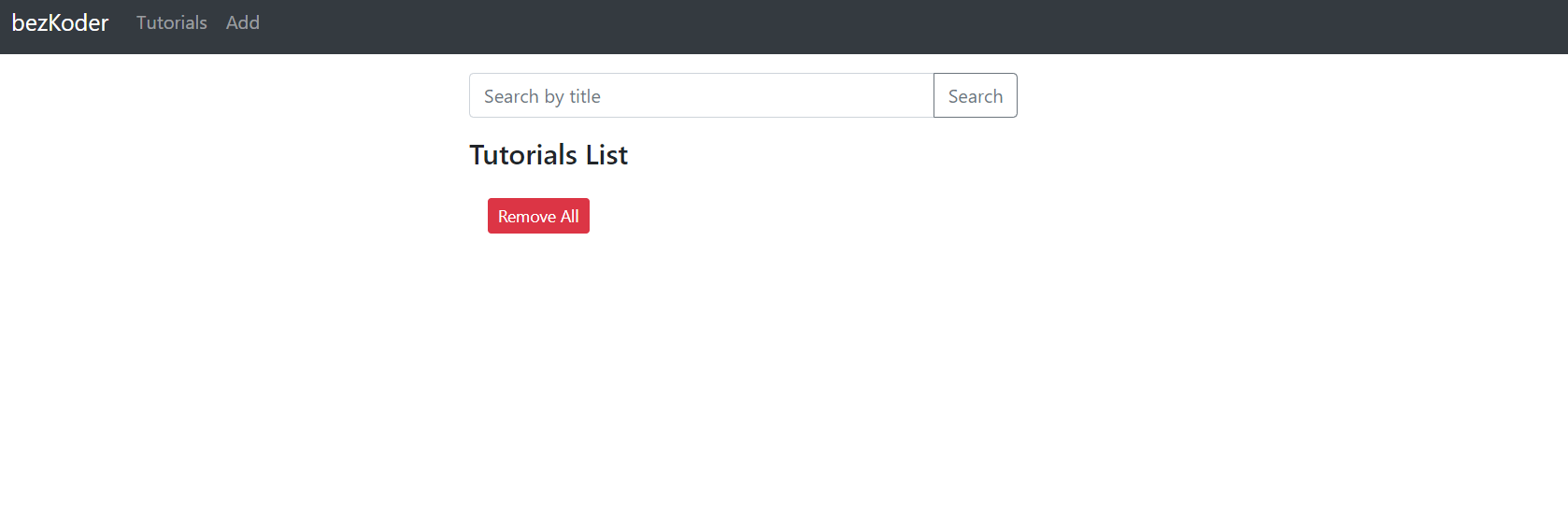
    driver: local

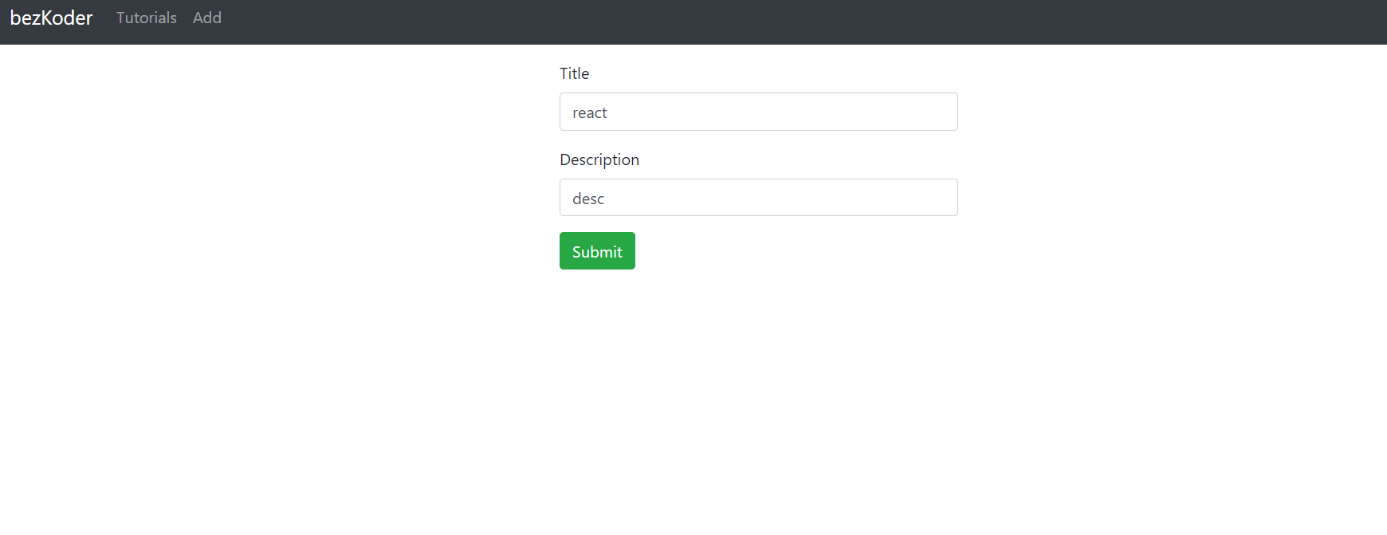
* **version: '3.8':** Spécifie la version de la syntaxe Docker Compose à utiliser. La version 3.8 est utilisée.
* **services::** Indique la section des services, où chaque service est défini.
* **mysqldb-service**:: Définit le premier service appelé "mysqldb-service".
* **image: mysql:5.7.44:** Utilise l'image Docker MySQL version 5.7.44 pour le conteneur de ce service.
* **container\_name: mysql-db**: Donne un nom au conteneur MySQL, dans ce cas "mysql-db".
* **ports: - 3307:3306**: Mappe le port 3306 du conteneur MySQL sur le port 3307 de l'hôte.
* **networks: - reseau**: Associe le service au réseau appelé "reseau".
* **volumes: - mysql-db-vol:/var/lib/db**: Monte un volume nommé "mysql-db-vol" sur le chemin /var/lib/db dans le conteneur MySQL.
* **environment: MYSQL\_ROOT\_PASSWORD: 123456 MYSQL\_DATABASE: testdb**: Définit des variables d'environnement pour configurer le mot de passe root de MySQL (MYSQL\_ROOT\_PASSWORD) et la base de données à créer (MYSQL\_DATABASE) sans devoir faire cela manuellement comme dans les travaux précédents.
* **backend::** Définit le deuxième service appelé "backend".
* **image: mesnaoui/img-backend:1.0.0:** Utilise l'image Docker "mesnaoui/img-backend" avec la version 1.0.0.
* **container\_name: backend:** Donne un nom au conteneur du service backend.
* **ports: - 8080:8080:** Mappe le port 8080 du conteneur backend sur le port 8080 de l'hôte.
* **networks: - reseau:** Associe le service au réseau appelé "reseau".
* **depends\_on: - mysqldb-service:** Indique que le service backend dépend du service mysqldb-service, assurant ainsi que le service MySQL est lancé avant le backend.
* **frontend::** Définit le troisième service appelé "frontend".
* **image: mesnaoui/img-frontend:1.0.0:** Utilise l'image Docker "mesnaoui/img-frontend" avec la version 1.0.0.
* **container\_name: frontend:** Donne un nom au conteneur du service frontend.
* **ports: - 8081:80:** Mappe le port 80 du conteneur frontend sur le port 8081 de l'hôte.
* **networks: - reseau:** Associe le service au réseau appelé "reseau".
* **depends\_on: - backend:** Indique que le service frontend dépend du service backend, assurant ainsi que le backend est lancé avant le frontend.
* **networks::** Définit la section des réseaux.
* **reseau::** Définit un réseau nommé "reseau".
* **driver: bridge:** Spécifie le type de pilote de réseau à utiliser, dans ce cas, le pilote de pont (bridge).
* **volumes::** Définit la section des volumes.
* **mysql-db-vol::** Définit un volume nommé "mysql-db-vol".
  + - * **driver: local:** Spécifie le pilote de volume à utiliser, dans ce cas, le pilote local.
  1. Taper la commande docker permettant de l’exécuter.

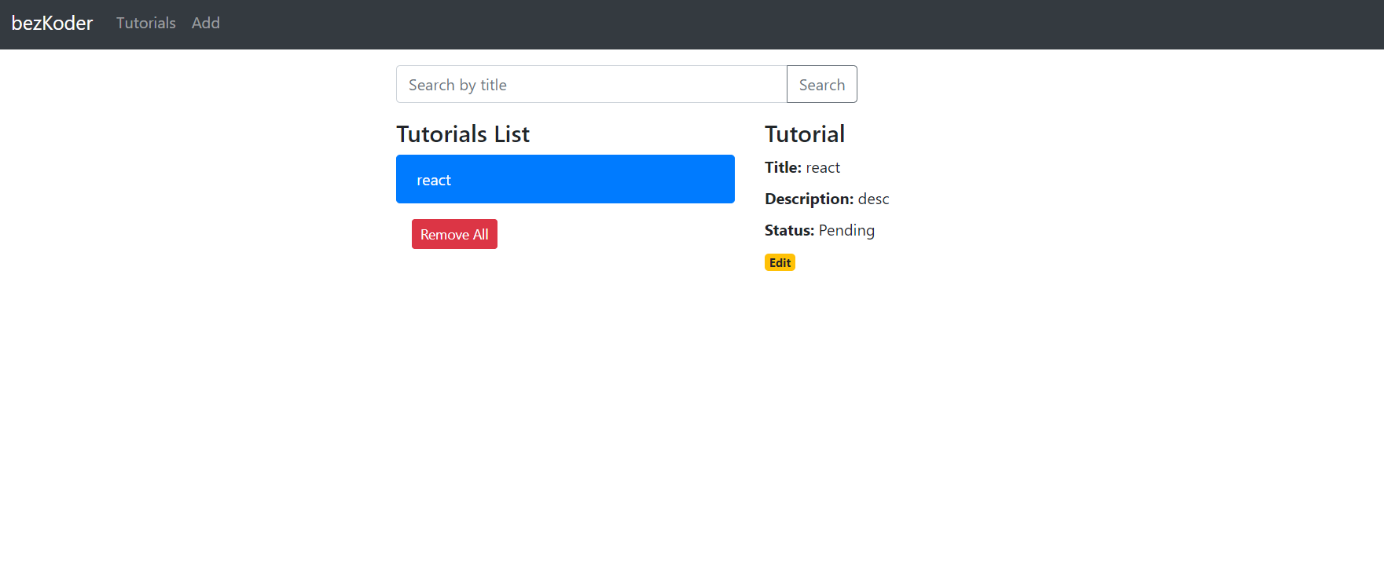


* 1. S’assurer que l’application fonctionne correctement (prises d’écran du navigateur)





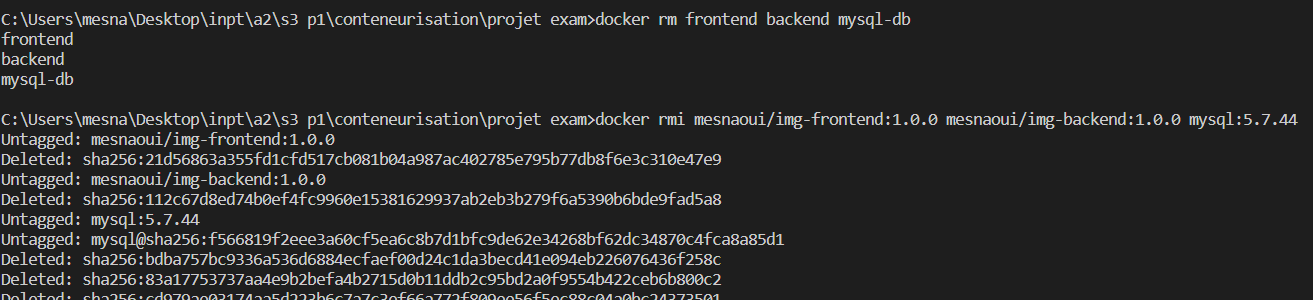




1. Supprimer les conteneurs qui s’exécutent ainsi que les images se trouvant dans le docker host.

docker rm -f frontend backend mysql-db

docker rmi mesnaoui/img-frontend :1.0.0 mesnaoui/img-backend :1.0.0



1. Créer un registre privé d’images docker :

Pour créer un registre privé d'images Docker en local, j’ai utilisé l’image registry :

1. Créer un conteneur Registry Docker :



1. Construire les Images à pousser dans ce registre :

docker build -t mesnaoui/img-frontend:1.0.0 .

docker build -t mesnaoui/img-backend:1.0.0 .

docker pull mysql:5.7.44

1. Taguer l'image pour le registre local construit :

docker tag mesnaoui/img-frontend:1.0.0 localhost:5000/mesnaoui/img-frontend:1.0.0

docker tag mesnaoui/img-backend:1.0.0 localhost:5000/mesnaoui/img-backend:1.0.0

docker tag mysql:5.7.44 localhost:5000/mysql:5.7.44

1. Pousser l'image vers le Registre Local :

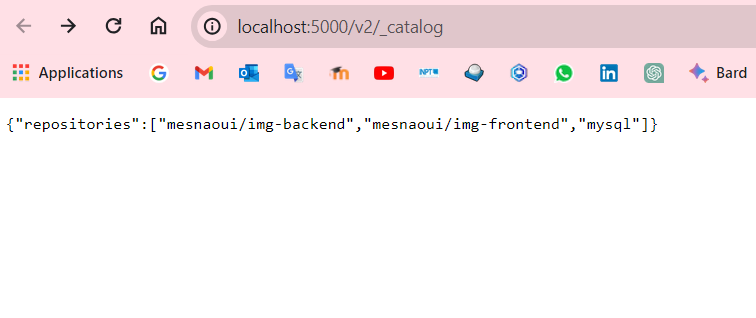


docker push localhost:5000/mesnaoui/img-frontend:1.0.0

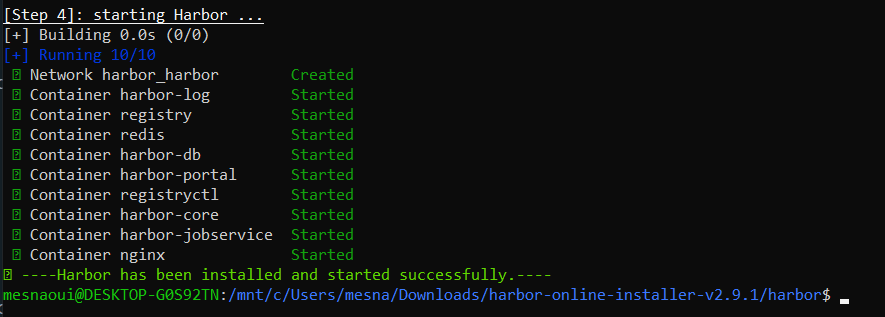
docker push localhost:5000/mesnaoui/img-backend:1.0.0

docker push localhost:5000/mysql:5.7.44

à ce nveau, on a créé le registre privé et y stocké et images :

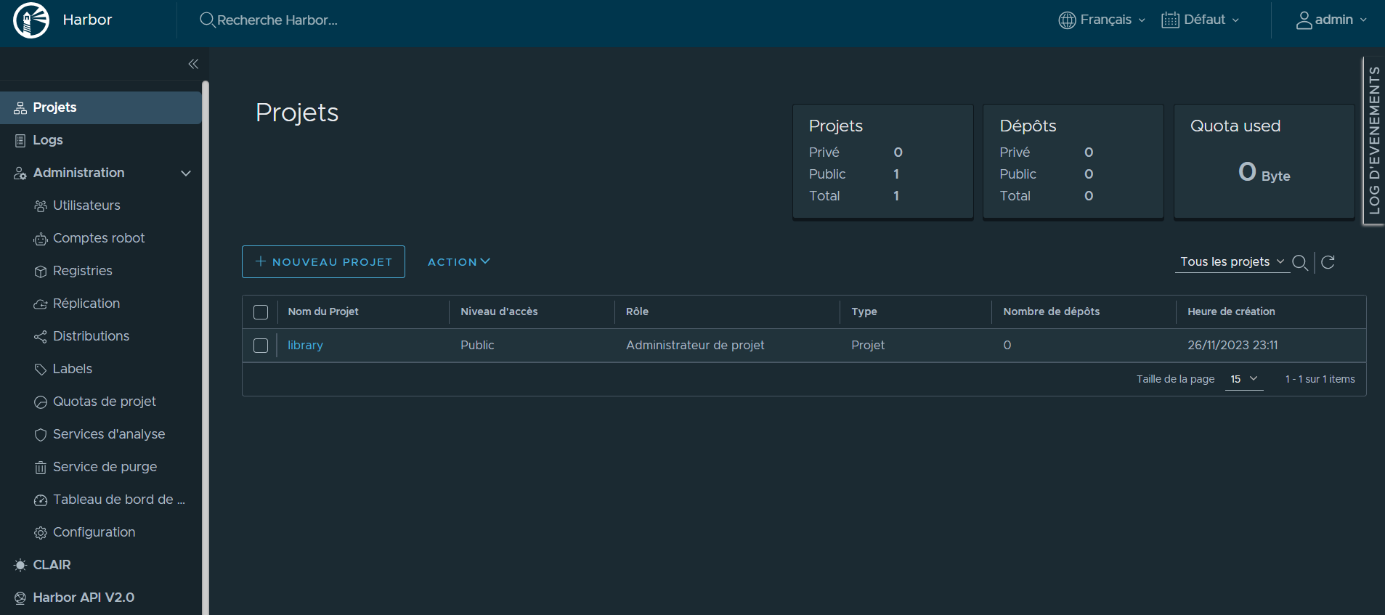


Ou on peut utiliser Harbor, une plateforme open source de registre d'images qui offre la gestion, la sécurité et la distribution des conteneurs Docker. Pour faire, il faut configurer et installer Harbor.

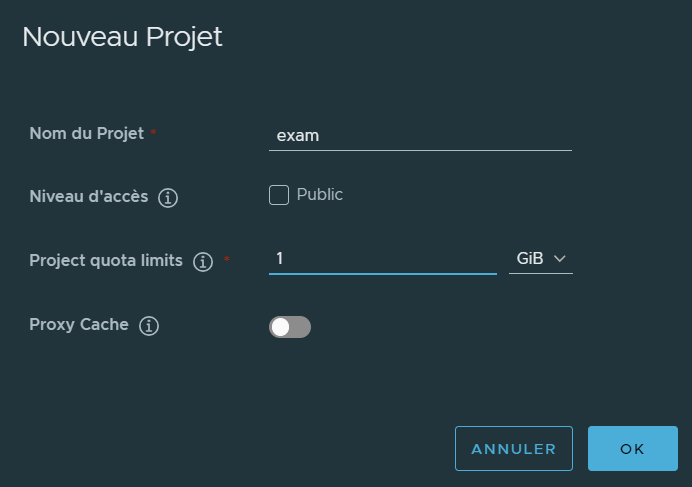


Une fois installé, on peut accèder à son interface graphique. Toute information de configuration nécessaire est dans le fichier harbor.yml :



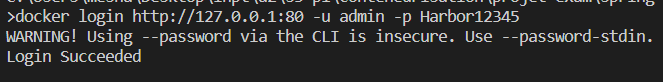


On crée un project ‘exam’ par l’interface graphique de Harbor :



Pour y déposer des images, il faut se connercter à harbor :

docker login <http://127.0.0.1:80> -u ‘USERNAME’ -p ‘PASSWORD’



Puis, taguer les images avec l'URL du référentiel Harbor :

docker tag mesnaoui/img-frontend:1.0.0 127.0.0.1:80/exam/mesnaoui/img-frontend:1.0.0

docker tag mesnaoui/img-backend:1.0.0 127.0.0.1:80/exam/mesnaoui/img-backend:1.0.0

docker tag mysql:5.7.44 127.0.0.1:80/exam/mysql:5.7.44

Puis, pousser les images vers le regitre privé dans Harbor :

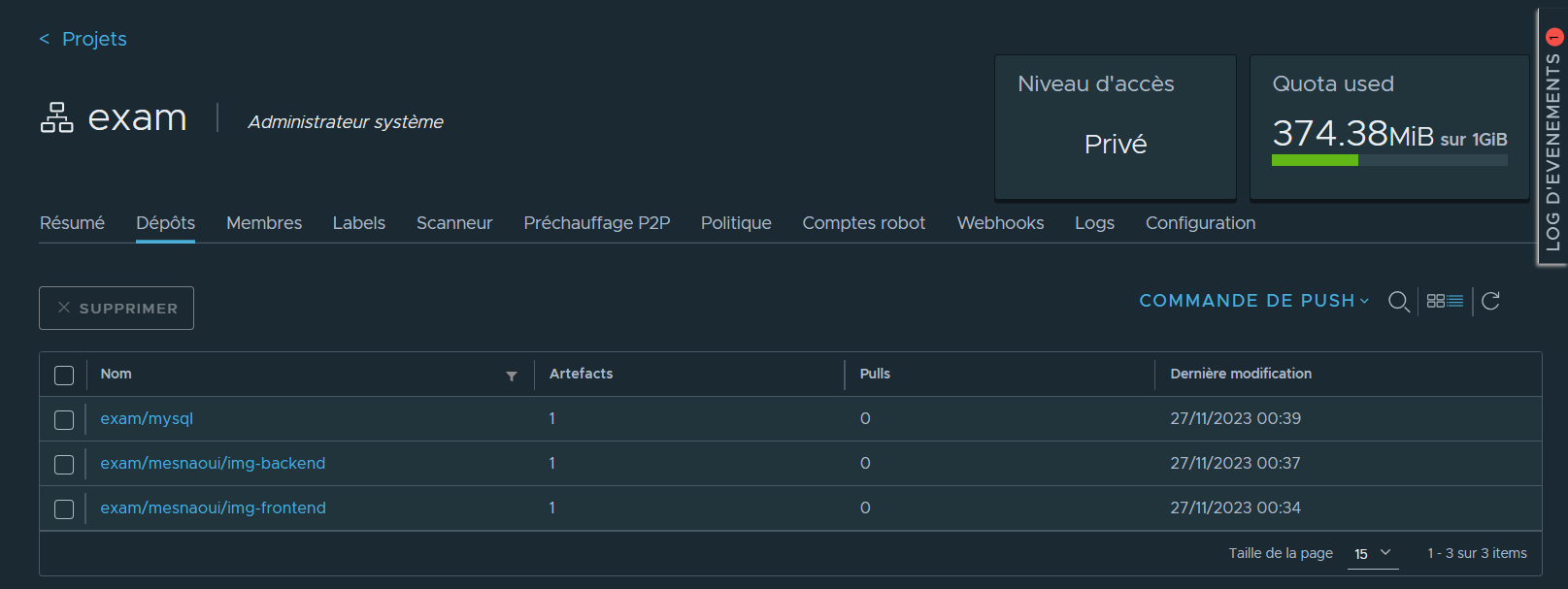
docker push 127.0.0.1:80/exam/mesnaoui/img-frontend:1.0.0

docker push 127.0.0.1:80/exam/mesnaoui/img-backend:1.0.0

docker push 127.0.0.1:80/exam/mysql:5.7.44



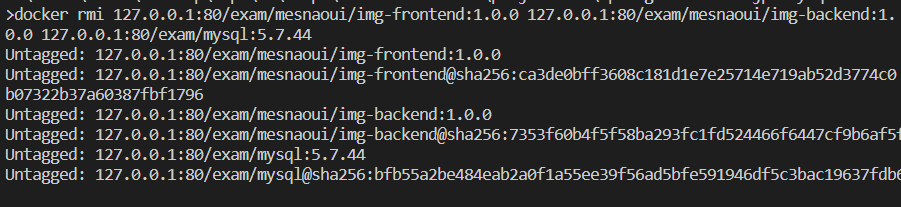
Maintenant, on peut visualiser l’ensemble des images dans l’interface de Harbor :



1. Redéployer l’application (frontend, backend et base de données) en utilisant docker-compose tout en récupérant l’image des conteneurs depuis ce nouveau registre privé.

Pour s’assurer du bon fonctionnement du répertoire privé Harbor, supprimons les images localement :

docker rmi 127.0.0.1:80/exam/mesnaoui/img-frontend:1.0.0 127.0.0.1:80/exam/mesnaoui/img-backend:1.0.0 127.0.0.1:80/exam/mysql:5.7.44



Mettons à jour les images dans notre docker-compose.yml :

version: '3.8'

services:

  mysql-service:

    image: 127.0.0.1:80/exam/mysql:5.7.44

    container\_name: mysql-db

    ports:

      - 3307:3306

    networks:

      - reseau

    volumes:

      - mysql-db-vol:/var/lib/mysql

    environment:

      MYSQL\_ROOT\_PASSWORD: 123456

      MYSQL\_DATABASE: testdb

  backend:

    image: 127.0.0.1:80/exam/mesnaoui/img-backend:1.0.0

    container\_name: backend

    ports:

      - 8080:8080

    networks:

      - reseau

    depends\_on:

      - mysql-service

  frontend:

    image: 127.0.0.1:80/exam/mesnaoui/img-frontend:1.0.0

    container\_name: frontend

    ports:

      - 8081:80

    networks:

      - reseau

    depends\_on:

      - backend

networks:

  reseau:

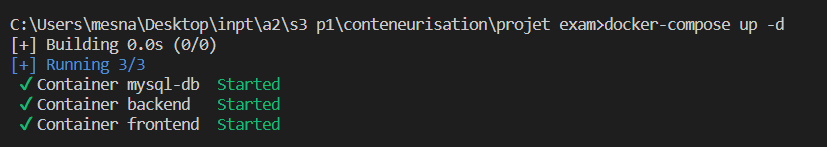
    driver: bridge

volumes:

  mysql-db-vol:

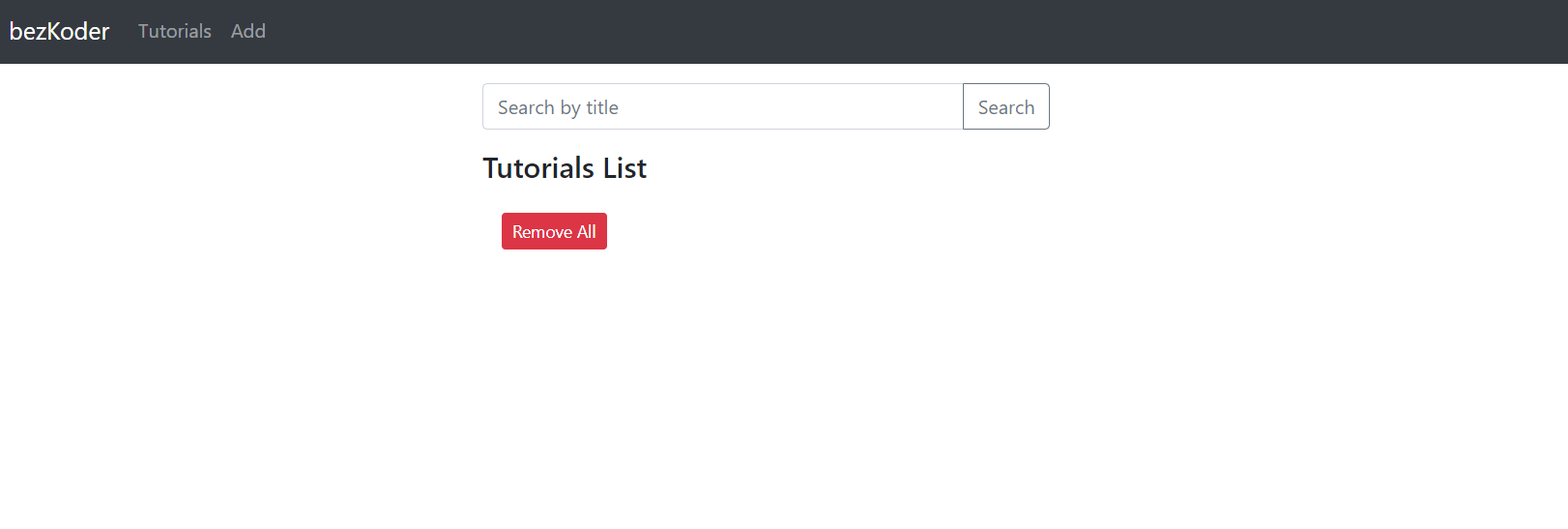
    driver: local

Lançons docker-compose:

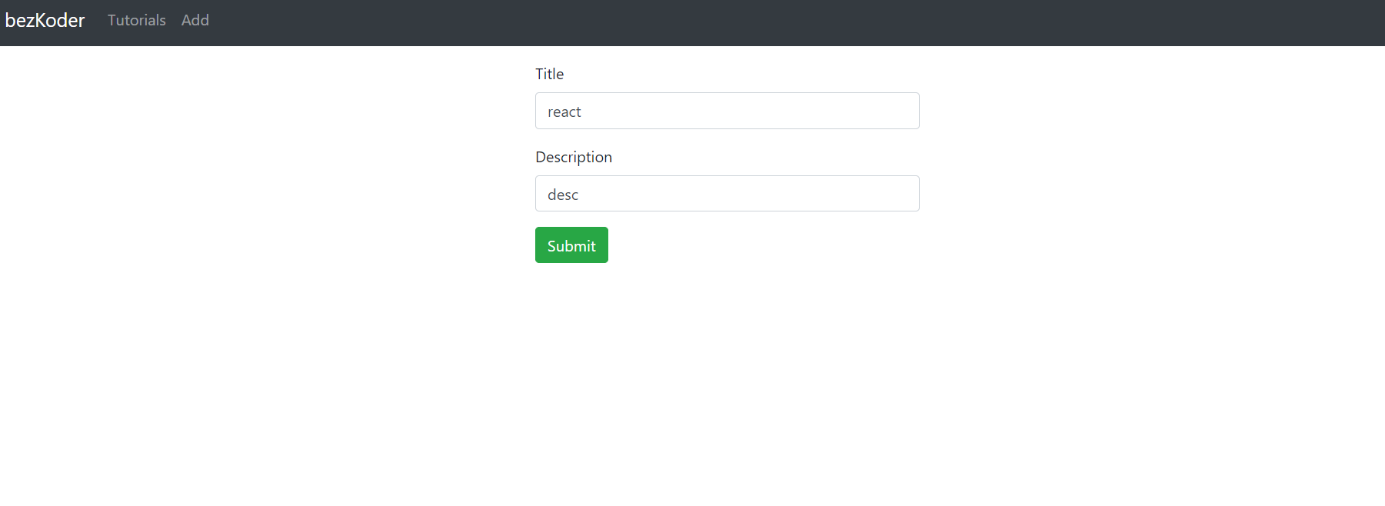


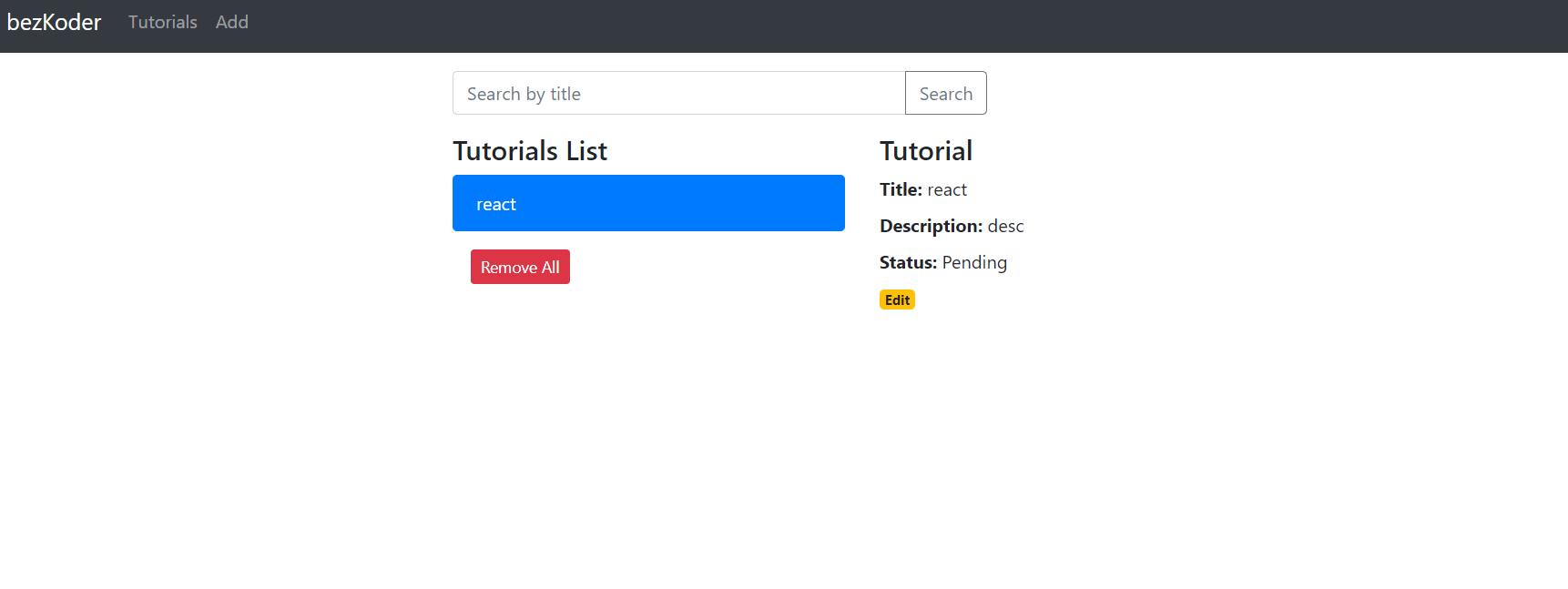
L’option -d sert à executer le processus en arrière plan.

Maintenant notre application est bien conteneurisée :



Testons maintenant les connexions entre le frontend, le backend et la base données :





**Partie : 2**

1. Créer les Kubernetes manifests (fichiers YAML) nécessaires pour déployer cette application dans un cluster Kubernetes (sous un namespace nommé "exam").

* Création d’un namespace ‘exam’ à l’aide de **project-ns.yaml** :

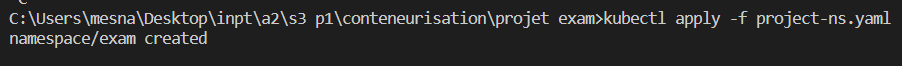
apiVersion: v1

kind: Namespace

metadata:

  name: exam

kubctl apply -f project-ns.yaml

****

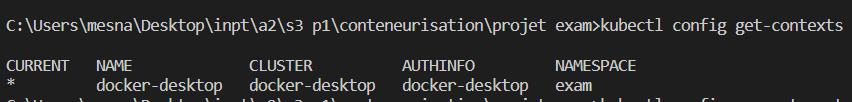
Après pour ne pas avoir à spécifier le namespace pour tous les fichiers YAML, on va rendre ce namespace le namespace par défaut :

* On se renseigne du contexe courant :



* On change son namespace par défaut :





* Mise à jour du traitement de connexion du backend avec la base de données :

Avant de commencer la création des manifestes pour le déploiement, on doit faire des petites modifications sur le code source du fichier application.properties. On va utiliser des variables d'environnement pour la connexion avec la base de données, cette modification va nous permettre d'utiliser les objets « Configmap » et « Secret » pour le déploiement de notre application.

spring.datasource.url=jdbc:mysql://${DB\_HOST}:3306/${DB\_NAME}?useSSL=false

spring.datasource.username=${MYSQL\_USERNAME}

spring.datasource.password=${MYSQL\_PASSWORD}

spring.jpa.properties.hibernate.dialect=org.hibernate.dialect.MySQLDialect

spring.jpa.hibernate.ddl-auto=update

En plus de cela on fera des modifications de code pour permettre la connexion entre le frontend et le backend.

* Au niveau de la classe controlleur du backend :

@CrossOrigin(origins = "http://project-angular.com")

Pour admettre les requêtes parvenant du frontend.

* Au niveau de la classe services du frontend :

const baseUrl = 'http://backend.com/api/tutorials';

Pour envoyer les requêtes vers le backend.

On a créé ainsi les images mesnaoui/img-backend :1.0.2 et mesnaoui/img-frontend :1.0.3 qui tiennent en compte ces changements.

* Création des manifestes pour la base de données (contenant : Secret, ConfigMap, Service et StatefulSet) :
* **dasabase.yaml**

apiVersion: v1

kind: Secret

metadata:

  name: mysql-secret

data:

  password: MTIzNDU2

  username: cm9vdA==

---

apiVersion: v1

kind: ConfigMap

metadata:

  name: db-conf

data:

  host: mysql

  dbname: testdb

---

apiVersion: v1

kind: Service

metadata:

  name: mysql

spec:

  selector:

    app: mysql-db

  ports:

    - port: 3306

      protocol: TCP

      targetPort: 3306

---

apiVersion: apps/v1

kind: StatefulSet

metadata:

  name: mysql-statefulset

  labels:

    app: mysql-db

spec:

  replicas: 2

  serviceName: mysql

  selector:

    matchLabels:

      app: mysql-db

  template:

    metadata:

      labels:

        app: mysql-db

    spec:

      containers:

        - name: mysql

          image: mysql:5.7.44

          ports:

            - containerPort: 3306

              name: db-port

          env:

            - name: MYSQL\_ROOT\_PASSWORD

              valueFrom:

                secretKeyRef:

                  name: mysql-secret

                  key: password

            - name: MYSQL\_DATABASE

              valueFrom:

                configMapKeyRef:

                  name: db-conf

                  key: dbname

          volumeMounts:

            - name: mysql-vol

              mountPath: /data/db

  volumeClaimTemplates:

    - metadata:

        name: mysql-vol

      spec:

        accessModes: ["ReadWriteOnce"]

        resources:

          requests:

            storage: 1Gi

ici on a créé un objet Secret nommé "mysql-secret" contenant les données d’authentification à la base de données sous forme de clé valeur encodées par «base64 ».

* Création des manifestes pour le backend (contenant : Service et Deployment) :
* **backend.yaml**

apiVersion: v1

kind: Service

metadata:

  name: backend-service

spec:

  selector:

    app: backend

  ports:

    - port: 8080

      protocol: TCP

      targetPort: 8080

---

apiVersion: apps/v1

kind: Deployment

metadata:

  name: backend-deployment

  labels:

    app: backend

spec:

  replicas: 2

  selector:

    matchLabels:

      app: backend

  template:

    metadata:

      labels:

        app: backend

    spec:

      containers:

        - name: backend

          image: mesnaoui/img-backend:1.0.2

          ports:

            - containerPort: 8080

              name: backend-port

          env:

            - name: DB\_HOST

              valueFrom:

                configMapKeyRef:

                  name: db-conf

                  key: host

            - name: DB\_NAME

              valueFrom:

                configMapKeyRef:

                  name: db-conf

                  key: dbname

            - name: MYSQL\_USERNAME

              valueFrom:

                secretKeyRef:

                  name: mysql-secret

                  key: username

            - name: MYSQL\_PASSWORD

              valueFrom:

                secretKeyRef:

                  name: mysql-secret

                  key: password

* Création des manifestes pour le frontend (contenant : Service et Deployment) :
* **frontend.yaml**

apiVersion: v1

kind: Service

metadata:

  name: frontend-service

spec:

  ports:

    - port: 8081

      targetPort: 80

      protocol: TCP

  selector:

    app: frontend

---

apiVersion: apps/v1

kind: Deployment

metadata:

  name: frontend-deployment

  labels:

    app: frontend

spec:

  replicas: 2

  selector:

    matchLabels:

      app: frontend

  template:

    metadata:

      labels:

        app: frontend

    spec:

      containers:

        - name: frontend

          image: mesnaoui/img-frontend:1.0.3

          ports:

            - containerPort: 80

              name: frontend-port

1. Définir des quotas pour la consommation des ressources Mémoire et CPU de chaque Pod exécuté dans le namespace « exam ».

Dans Kubernetes, les quotas de ressources peuvent être définis au niveau du namespace pour limiter la consommation de ressources mémoire et CPU de chaque Pod.

Pour définir des quotas de mémoire et de CPU dans le namespace "exam", on va utiliser un objet ResourceQuota dans ce namespace.

apiVersion: v1

kind: ResourceQuota

metadata:

  name: quota-memoire-cpu

  namespace: exam

spec:

  hard:

    requests.memory: 1Gi

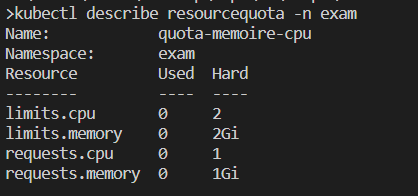
    limits.memory: 2Gi

    requests.cpu: 1

    limits.cpu: 2

* **apiVersion: v1**: Spécifie la version de l'API Kubernetes utilisée, dans ce cas, la version 1 at aussi montre qu’il s’agit d’un objet .
* **kind: ResourceQuota**: Indique le type de l’obet Kubernetes à créer, qui est un quota de ressources dans ce cas.
* **name: quota-memoire-cpu**: nom à la ressource de quota "quota-memoire-cpu".
* **namespace: exam**: Spécifie le namespace dans lequel cette ressource de quota doit être créée, ici "exam".
* **hard:**: Indique les limites pour les différentes ressources spécifiées.
* **requests.memory: 1Gi**: définit la limite minimale de mémoire que chaque Pod peut demander (1Go).
* **limits.memory: 2Gi**: Fixe une limite maximale à 2 Go de mémoire.
* **requests.cpu: 1**: définit la limite minimale de CPU que chaque Pod peut demander (une unité de cpu).
* **limits.cpu: 2**: Fixe une limite maximale à 2 unités de CPU.

Cette commande montre que cette configuration est bien appliquée :



1. Contrôler l’accès aux ressources existantes dans le namespace « exam » par un rôle RBAC.

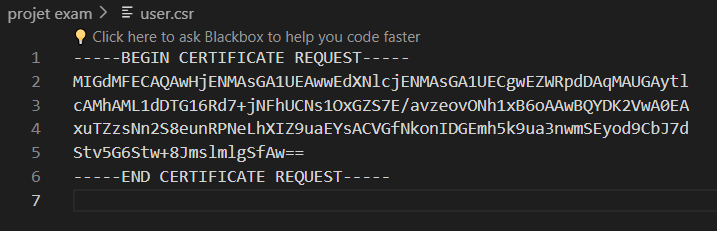
Il existe trois type d’utilisateur à qui on peux définir des roles: des utilisateurs, des administrateurs(avec ClusterRole) et des serviceAccount(pour les applications). Pour répondre à cette question on a pris le cas d’un utilisateur non administrateur et humain.

Tout d’abord créant un utilisateur pour la démonstration ultérieure, en suivant ces étapes :

* 1. générer le certificat des clients pour les utilisateurs :

**openssl genpkey -out user.key -algorithm ed25519**

**openssl req -new -key user.key -out user.csr -subj /CN=user/O=edit**

****

* 1. Créer une demande de signature du certificat :

**cat user.csr | base64 | tr -d "\n"**

* **certificatRequest.yaml :**

apiVersion: certificates.k8s.io/v1

kind: CertificateSigningRequest

metadata:

  name: user

spec:

  request: LS0tLS1CRUdJTiBDRVJUSUZJQ0FURSBSRVFVRVNULS0tLS0NCk1JR2RNRkVDQVFBd0hqRU5NQXNHQTFVRUF3d0VkWE5sY2pFTk1Bc0dBMVVFQ2d3RVpXUnBkREFxTUFVR0F5dGwNCmNBTWhBTUwxZERURzE2UmQ3K2pORmhVQ05zMU94R1pTN0UvYXZ6ZW92T05oMXhCNm9BQXdCUVlESzJWd0EwRUENCnh1VFp6c05uMlM4ZXVuUlBOZUxoWElaOXVhRVlzQUNWR2ZOa29uSURHRW1oNWs5dWEzbndtU0V5b2Q5Q2JKN2QNClN0djVHNlN0dys4Sm1zbG1sZ1NmQXc9PQ0KLS0tLS1FTkQgQ0VSVElGSUNBVEUgUkVRVUVTVC0tLS0tDQo=

  signerName: kubernetes.io/kube-apiserver-client

  expirationSeconds: 172800 #j’ai définit le certificat pour deux jours

  usages:

  - client auth

**kubectl apply -f certificateRequest.yaml**

* 1. accepter le certificate:

**kubectl certificate approve user**

**kubectl get csr/user -o jsonpath='{.status.certificate}' | base64 -d > user.crt**

* 1. Copier le fichier .kube/config
  2. De la copie, supprimer tout à part les informations sur le cluster.
  3. Créer l’utilisateur :

**kubectl --kubeconfig configCopie config set-credentials user --client-key=user.key --client-certificate=user.crt --embed-certs=true**

* 1. Créer le context:

**kubectl --kubeconfig configCopie config set-context user-context --cluster=docker-desktop --user=user**

* 1. Créer le role:
* **role.yaml**

apiVersion: rbac.authorization.k8s.io/v1

kind: Role

metadata:

  namespace: exam

  name: exam-role

rules:

- apiGroups: [""]

  resources: ["pods"]

  verbs: ["get", "list"]

* 1. Lier le role à l’utilisateur :
* **roleBinding.yaml**

apiVersion: rbac.authorization.k8s.io/v1

kind: RoleBinding

metadata:

  name: exam-role-binding

  namespace: exam

subjects:

- kind: User

  name: "user"

  apiGroup: rbac.authorization.k8s.io

roleRef:

  kind: Role

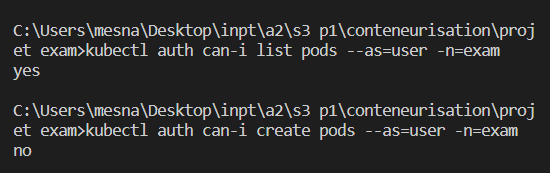
  name: exam-role

  apiGroup: rbac.authorization.k8s.io

**kubectl apply -f role.yaml**

**kubectl apply -f roleBinding.yaml**

**Démonstration :** s’assurant des fonctionnalités de création et de consultations des pods dans le namespace « exam » pour l’utilisateur user :

****

1. Spécifier un budget de perturbation (Disruption Budget) pour votre application.

Un Disruption Budget (budget de perturbation) est une politique qui limite le nombre d'instances en cours d'exécution d'une application qui peuvent être indisponibles simultanément lors d'une mise à niveau ou d'une opération de maintenance.

Pour ce faire, on va créer un manifest PodDisruptionBudget :

* distruptionBudget.yaml :

apiVersion: policy/v1

kind: PodDisruptionBudget

metadata:

  name: db-budget

  namespace: exam

spec:

  maxUnavailable:

  selector:

    matchLabels:

      app: mysql-db

---

apiVersion: policy/v1

kind: PodDisruptionBudget

metadata:

  name: backend-budget

  namespace: exam

spec:

  maxUnavailable: 1

  selector:

    matchLabels:

      app: backend

---

apiVersion: policy/v1

kind: PodDisruptionBudget

metadata:

  name: frontend-budget

  namespace: exam

spec:

  maxUnavailable: 1

  selector:

    matchLabels:

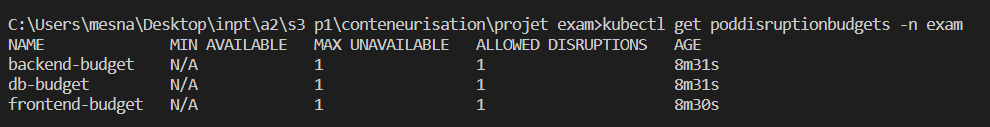
      app: frontend

**maxUnavailable:** Nombre maximal d'instances pouvant être indisponibles simultanément.

**kubectl apply -f disruptionBudget.yaml**

**Démonstration:**

**kubectl get poddisruptionbudgets -n exam**



Le 1 de ALLOWED DISRUPTIONS indique que k8s a calculé le nombre toléré de pods indisponibles qui sera le minimun entre nombre de réplicas du pod et le nombre du MAX UNVALAIBLE.

On peut aussi utiliser la stratégie de RollingUpdate (mise à jour en continu). Dans une mise à jour en continu, les nouveaux pods sont progressivement déployés tout en réduisant progressivement le nombre d'instances des anciens pods, assurant ainsi un déploiement en douceur sans interruption de service.

Cela est fait en ajoutant dans la partie « spec » des trois deployments : du frontend, du backend et de la base de données :

  strategy:

    type: RollingUpdate

    rollingUpdate:

      maxUnavailable: 1

      maxSurge: 1

**maxUnavailable:** indique le nombre maximum de pods qui peuvent être indisponibles pendant la mise à jour.

**maxSurge:** spécifie le nombre maximal de pods supplémentaires qui peuvent être créés par rapport au nombre désiré de pods pendant la mise à jour.

**kubectl apply -f frontend.yaml -n exam**

**kubectl apply -f backend.yaml -n exam**

**kubectl apply -f database.yaml -n exam**

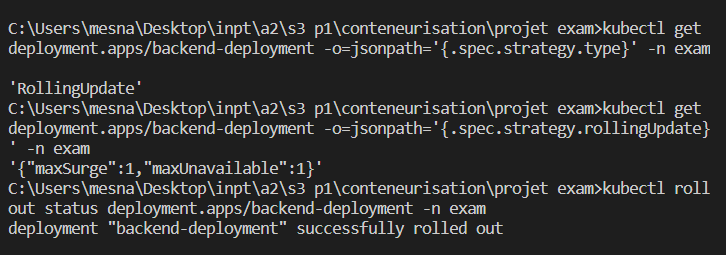
**Démonstration :**

Vérifier l'état du déploiement :

**kubectl get deployment.apps/backend-deployment -o=jsonpath='{.spec.strategy.type}' -n exam**

**kubectl get deployment.apps/backend-deployment -o=jsonpath='{.spec.strategy.rollingUpdate’ -n exam**

**kubectl rollout status deployment.apps/backend-deployment -n exam**



En combinant les deux approches, on peut améliorer et contrôler les mises à jour en continu et d’autres scénarios de perturbations.

1. Créer et configurer pour cette application un : Liveness Prob Readiness Prob Startup Prob.

**Liveness Probe (Sonde de vivacité) :** est utilisée pour déterminer si l'application à l'intérieur du conteneur est en cours d'exécution correctement. Si la sonde échoue, Kubernetes peut redémarrer le conteneur.

**Readiness Probe (Sonde de préparation) :** utilisée pour déterminer si le conteneur est prêt à recevoir du trafic réseau. Si la sonde échoue, le conteneur ne recevra pas de trafic.

**Startup Probe (Sonde de démarrage) :** utilisée pour déterminer quand le conteneur est considéré comme démarré.

Il existe trois types de de sonde : de commande , http et TCP.

On ajoute les sondes dans la section des conteneurs :

          startupProbe:

            httpGet:

              path: /startup

              port: 80

            initialDelaySeconds: 10

            periodSeconds: 5

            failureThreshold: 30

          livenessProbe:

            httpGet:

              path: /health

              port: 80

            initialDelaySeconds: 3

            periodSeconds: 5

          readinessProbe:

            httpGet:

              path: /ready

              port: 80

            initialDelaySeconds: 5

            periodSeconds: 5

**path :** l’endpoint exposé par l’application.

**Port :** le port sur lequel l’application écoute.

**initialDelaySeconds :** délai initial à attendre avant l’envoi des requêtes de vérification.

**periodSeconds :** période entre deux vérifications successives.

Dans mon cas, je vais créer des sondes de type HTTP. Pour configurer ces sondes, nous devons mettre en place des points d'accès au niveau de l'application qui seront interfacés avec les sondes afin de leur envoyer des réponses HTTP.

* Côté frontend :

J’ai délégué cette tâche à nginx. Ainsi j’ai créé un fichier de configuration nginx.conf où j’ai spécifié ces endpoints (voir le code dans github).

Et j’ai reconstruit l’image avec le tag mesnaoui/img-frontend :1.1.0.

J’ai enfin mis à jour l’image et ajouté les séctions des sondes dans la sous-partie containers dans la partie spec dans le déploiment du frontend.

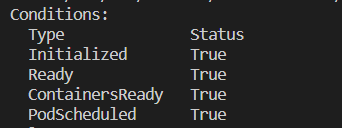
**Vérification :**

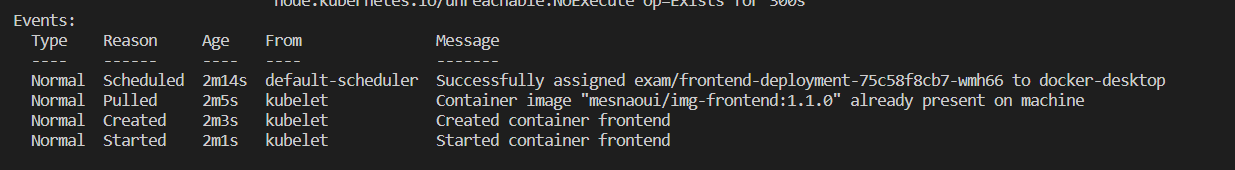
**kubectl describe pod/frontend-deployment-75c58f8cb7-wmh66 -n exam**



On voit bien que les sondes (Liveness Probe, Readiness Probe, Startup Probe) sont configurées pour le conteneur "frontend".

* Le Liveness Probe est configuré pour effectuer une requête HTTP GET sur le chemin /health avec un délai initial de 3 secondes, un délai entre les périodes de 5 secondes et un délai d'expiration de 1 seconde.
* Le Readiness Probe : est configuré pour effectuer une requête HTTP GET sur le chemin /ready avec un délai initial de 5 secondes, un délai entre les périodes de 5 secondes, un délai d'expiration de 1 seconde.
* Le Startup Probe : configuré pour effectuer une requête HTTP GET sur le chemin /startup avec un délai initial de 10 secondes, un délai entre les périodes de 5 secondes, un délai d'expiration de 1 seconde et un succès après une tentative réussie. Le délai d'échec est configuré pour 30 tentatives avant de déclarer un échec.

****

****

1. Créer un Ingress pour accéder à l’application depuis un navigateur moyennant un nom de domaine.

On a créé un ingress avec deux noms domaines : un pour le frontend ‘project-angular.com’ et un autre pour le backend (on a suivi cette approche car tout appel entre le frontend et le backend est traité au niveau du browser et puisque les services ne sont connus qu’au niveau du cluser, la communication entre le browser et le backend n’est jamais possible. Du coup, on a créé un nom de domaine pour le backend mais on a défini CorsOrigin à project-angular.com pour garantir sa sécurité.)

apiVersion: networking.k8s.io/v1

kind: Ingress

metadata:

  name: app-ingress

  annotations:

    kubernetes.io/ingress.class: "nginx"

    nginx.ingress.kubernetes.io/add-base-url: "true"

    nginx.ingress.kubernetes.io/rewrite-target: /

spec:

  rules:

    - host: project-angular.com

      http:

        paths:

          - path: /

            pathType: Prefix

            backend:

              service:

                name: frontend-service

                port:

                  number: 8081

    - host: backend.com

      http:

        paths:

          - path: /

            pathType: Prefix

            backend:

              service:

                name: backend-service

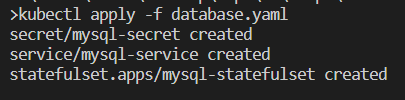
                port:

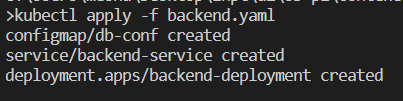
                  number: 8080

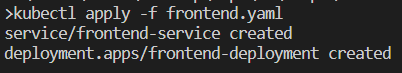
On n’a aussi ajouté ces noms de domaine dans le fichier « hosts » et créé un ingress-nginx controller :

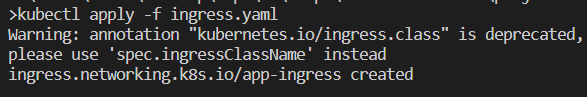
**kubectl apply -f https://raw.githubusercontent.com/kubernetes/ingress-nginx/controller-v1.8.2/deploy/static/provider/cloud/deploy.yaml**

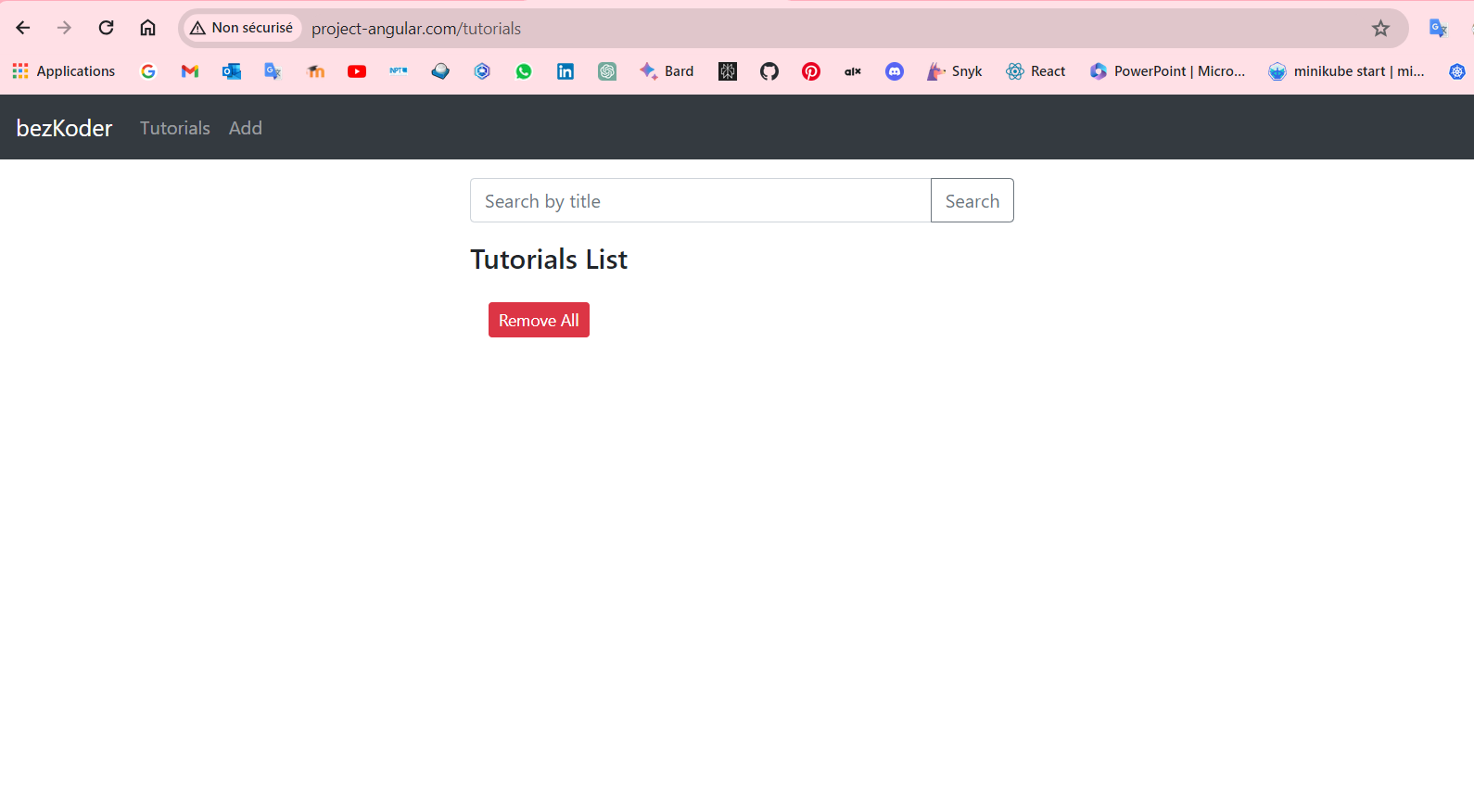
1. S’assurer que l’application a été bien conteneurisée et qu’elle fonctionne correctement.

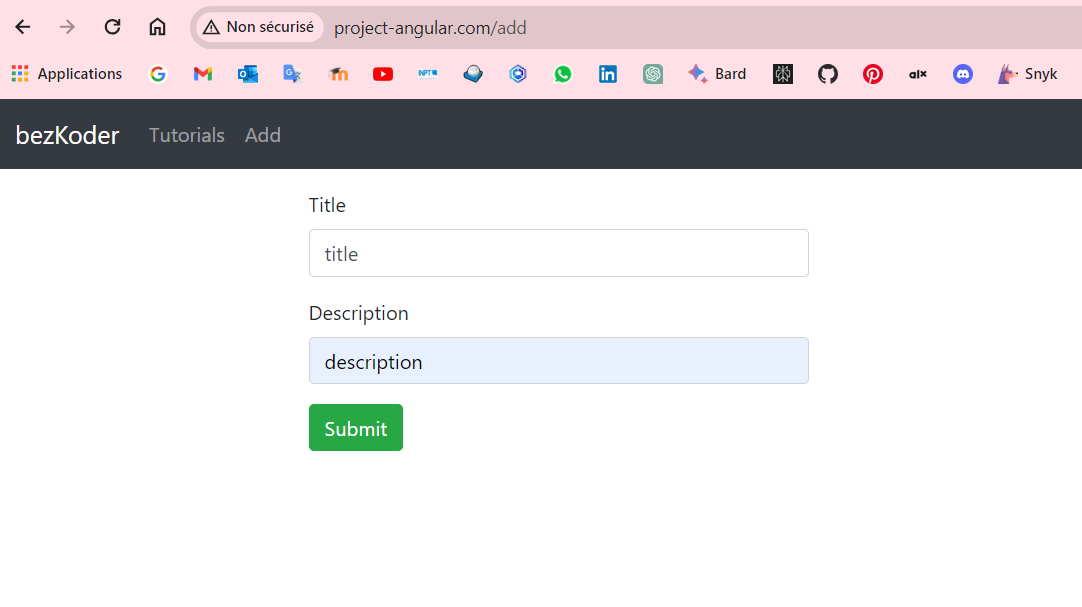
****

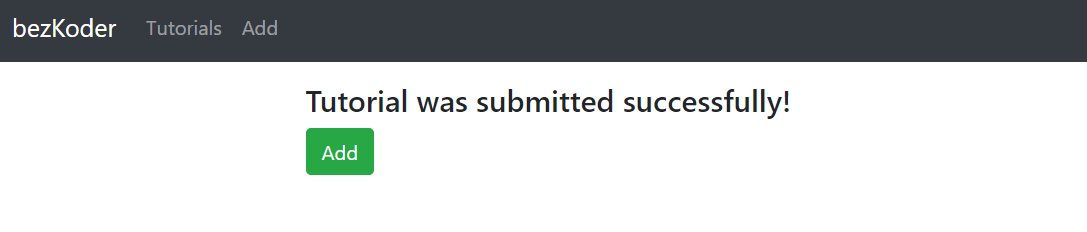


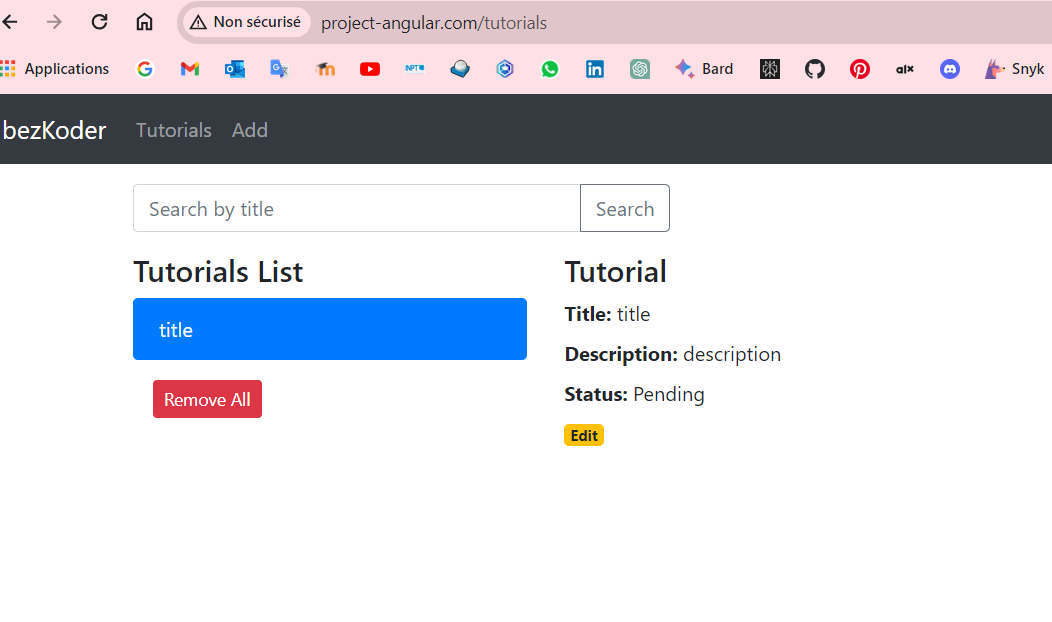
****

****

****

****

****

****