ALOM - TP 9 - Patterns Cloud

Table of Contents

1. Présentation et objectifs

Le but de ce TP est de mettre en place quelques mécaniques pour les développements orientés cloud.

Nous allons :

- déployer nos applications chez Clever Cloud
- créer des profils pour chacun de nos micro-services
- exposer des métriques avec `spring-boot-actuator`
- charger les properties d'accès à notre base de données depuis un Vault
- envoyer des emails avec un nouveau micro-service d'envoi d'emails

1.1. Pré-requis

Les pré-requis à ce TP sont :

- Avoir terminé la partie *Le controlleur* du TP 4 Persistence

2. Déploiement chez Clever Cloud

Clever Cloud a accepté de nous sponsoriser en nous offrant une organisation avec des crédits illimités .

Nous allons déployer nos TP chez Clever Cloud en utilisant 2 de leurs services :

- Pour la persistence : les bases de données PostgreSQL managées
- Pour le code : les applications Docker

2.1. Votre compte Clever Cloud

Créez un compte sur https://www.clever-cloud.com, en utilisant votre adresse mail d'étudiant !

En principe, vous devriez avoir aussi reçu une "invitation" à rejoindre l'organisation Clever Cloud *Université de Lille*. Acceptez cette invitation pour y avoir accès.

Vous pouvez accéder à l'organisation avec ce lien direct.

2.2. La base de données

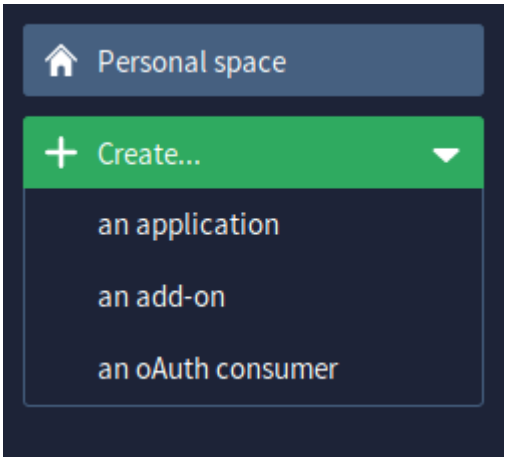
Pour remplacer notre base de données embarquée ou en docker, nous pouvons nous connecter sur une base de données réelle, que nous allons instancier sur un cloud public.

2.2.1. Créer la base de données

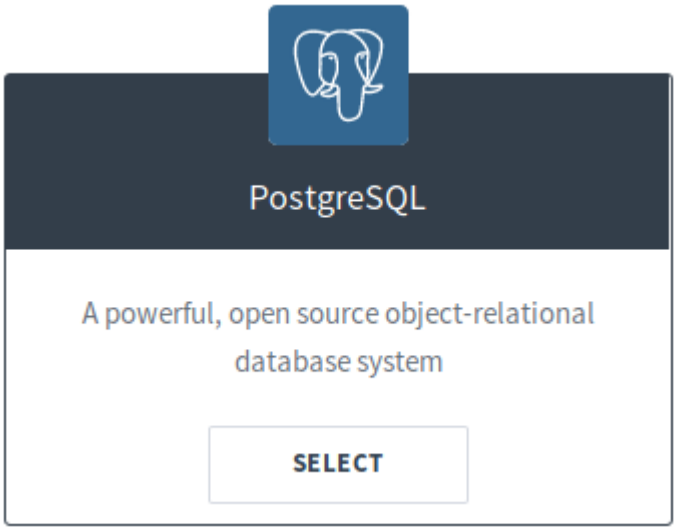
Les droits d'accès que vous avez ne permettent pas de créer des bases de données, je les ai créées en avance de phase pour vous. Cette partie est purement documentaire, vous pouvez la lire si ça vous intéresse, ou passer à la partie suivante Déploiement des applications.

Une fois votre compte créé, vous pouvez instancier une base de données en quelques clics !

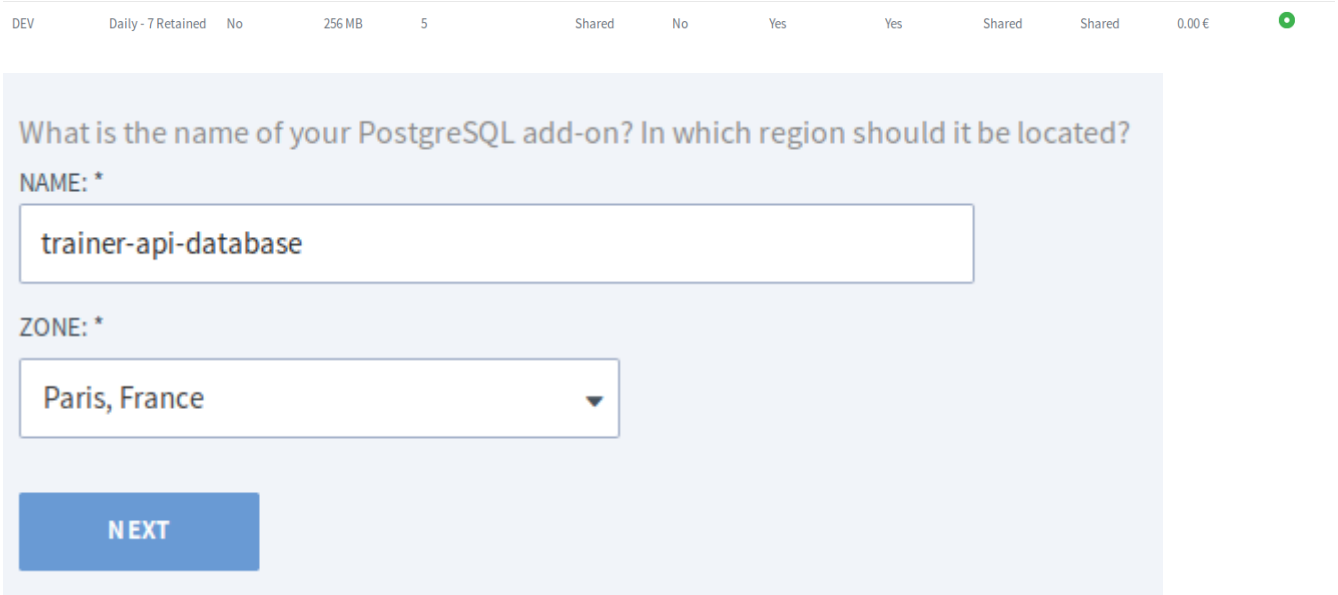
- Dans la console, sélectionnez `Create > an add-on`.



- Sélectionnez la base de données `postgresql`



- Sélectionnez le plan `DEV`, qui est gratuit
- Donnez un nom à votre base de données, et sélectionnez la région `Paris` (un hébergement de notre base de données à Montréal créerait des temps de latence importants !)



- Validez, et attendez quelques secondes. Votre base de données est prête !

Accédez au dashboard de votre base de données. Vous pourrez y trouver:

- Les informations de connexion à votre base de données
- Des menus permettant de réinitialiser votre base, re-généré de nouveaux identifiants de connexions, ou effectuer un backup.
- Vous pouvez également accéder à une interface "PGStudio" vous permettant de naviguer dans votre base de données.

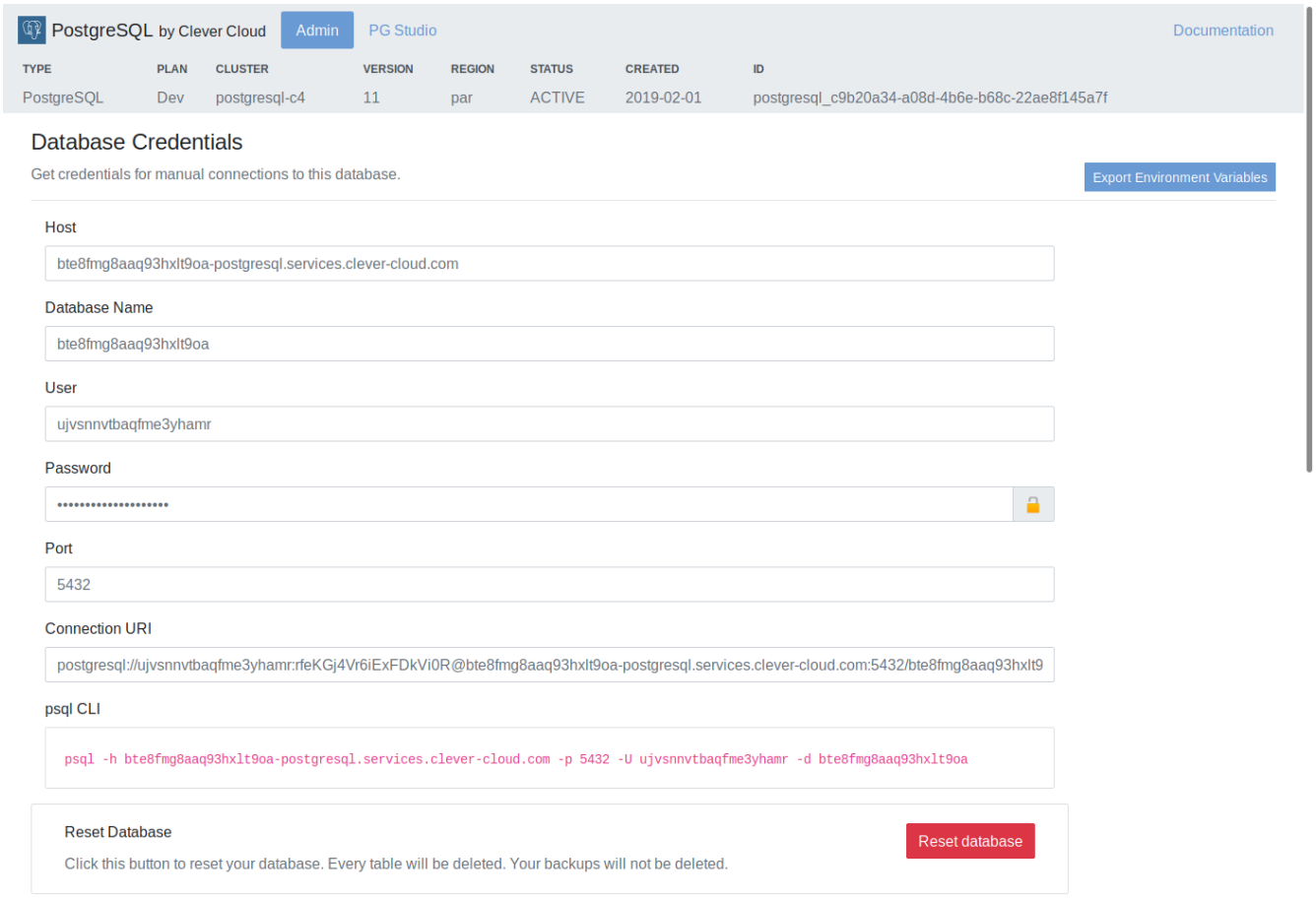


Figure 1. La page d'informations de votre base de données !

2.3. Déploiement des applications

Pour cette partie, je dois vous donner les droits d'accès à l'organisation. Appelez-moi pour que je puisse le faire avec vous si vos accès ne sont pas ouverts. J'ai aussi créé pour vous les applications sur Clever Cloud pour vous faciliter la vie. Vous pouvez les utiliser, ou en créer d'autres.

2.3.1. Création d'un token Clever Cloud

Générez un access token et un secret Clever Cloud pour que le pipeline GitLab puisse s'authentifier.

Rendez-vous à cette URL : https://console.clever-cloud.com/cli-oauth

Récupérez le Token et Secret affichés.

Rendez-vous dans votre projet GitLab, dans la section *Settings* / *CI/CD* / Variables.

Créez deux variables `CLEVER_TOKEN` et `CLEVER_SECRET`, de type *Variable*, avec les valeurs récupérées (attention aux espaces en début et fin de ligne).

2.3.2. Ajout d'un Dockerfile

Dans le code de *chacun* de vos micro-services, ajoutez le `Dockerfile` suivant (il n'est pas parfait, mais fais bien le travail) :

Dockerfile

```
# Stage 1: Build the application
FROM maven:3-eclipse-temurin-25-alpine AS build

WORKDIR /app

# Copy the pom.xml file
COPY pom.xml .
# Copy your source code
COPY src ./src

# Build the application
RUN mvn package -DskipTests

# Stage 2: Create the runtime image
FROM eclipse-temurin:25-alpine

WORKDIR /app

# Copy the jar file from the build stage
COPY --from=build /app/target/*.jar app.jar

# Expose the port the app runs on
EXPOSE 8080

# Run the jar file
ENTRYPOINT ["java","-jar","app.jar"]
```

2.3.3. Ajout d'un pipeline de CI

Sur Clever Cloud, récupérez le `app_id` en allant dans l'onglet *Information* de votre application, ou en haut à droite de l'écran *Overview* de votre application.

Modifiez vos fichiers `.gitlab-ci.yml` pour y ajouter une étape *deploy*, ainsi qu'une variable `CC_APP_ID` :

.gitlab-ci.yml

```
variables:
  CC_APP_ID: ①
```

```
deploy:
  image:
    name: clevercloud/clever-tools
    entrypoint: ["/bin/sh", "-c"]
  stage: deploy
  script:
    - clever link $CC_APP_ID
    - clever deploy --force --same-commit-policy restart
```

① : Renseignez ici l'application id de votre application sur Clever Cloud

Pushez sur Git votre `Dockerfile` et votre `.gitlab-ci.yml` modifié.

Si toutes les étapes sont correctes, chaque `git push` occasionnera un déploiement de votre application !

Répétez les opérations pour chacune de vos applications.

3. Création de profils et activation

Aujourd'hui, nos micro-services devront tourner sur plusieurs environnements distincts :

- notre poste de développeur
- un déploiement d'application Docker chez *Clever Cloud*

On pourrait aussi imaginer vouloir créer un troisième environnement, de recette métier par exemple.

3.1. Extraction des profils

Pour chacun de vos micro-services :

- Créez un fichier de configuration `application-clever.properties` Ce fichier contiendra toutes les properties liées à l'environnement d'exécution Clever-Cloud, par exemple les URL des autres micro-services, et l'URL de connexion à la base de données.
- Créez un fichier de configuration `application-local.properties` Ce fichier contiendra toutes les properties liées à l'exécution en local de votre projet, par exemple les URL des autres micro-services en `localhost`, ainsi que les properties `server.port`

À cette étape, vous pouvez vider vos fichiers `application.properties`, dont le contenu a dû être migré dans les deux fichiers `application-local.properties` et `application-clever.properties`.

Il est parfois utile d'avoir des properties communes dans le `application.properties`. Attention par contre, dans le cas d'utilisation d'un profil, les properties du `application.properties` sont d'abord chargées, et ensuite les properties du profil `application-{profil}.properties`.

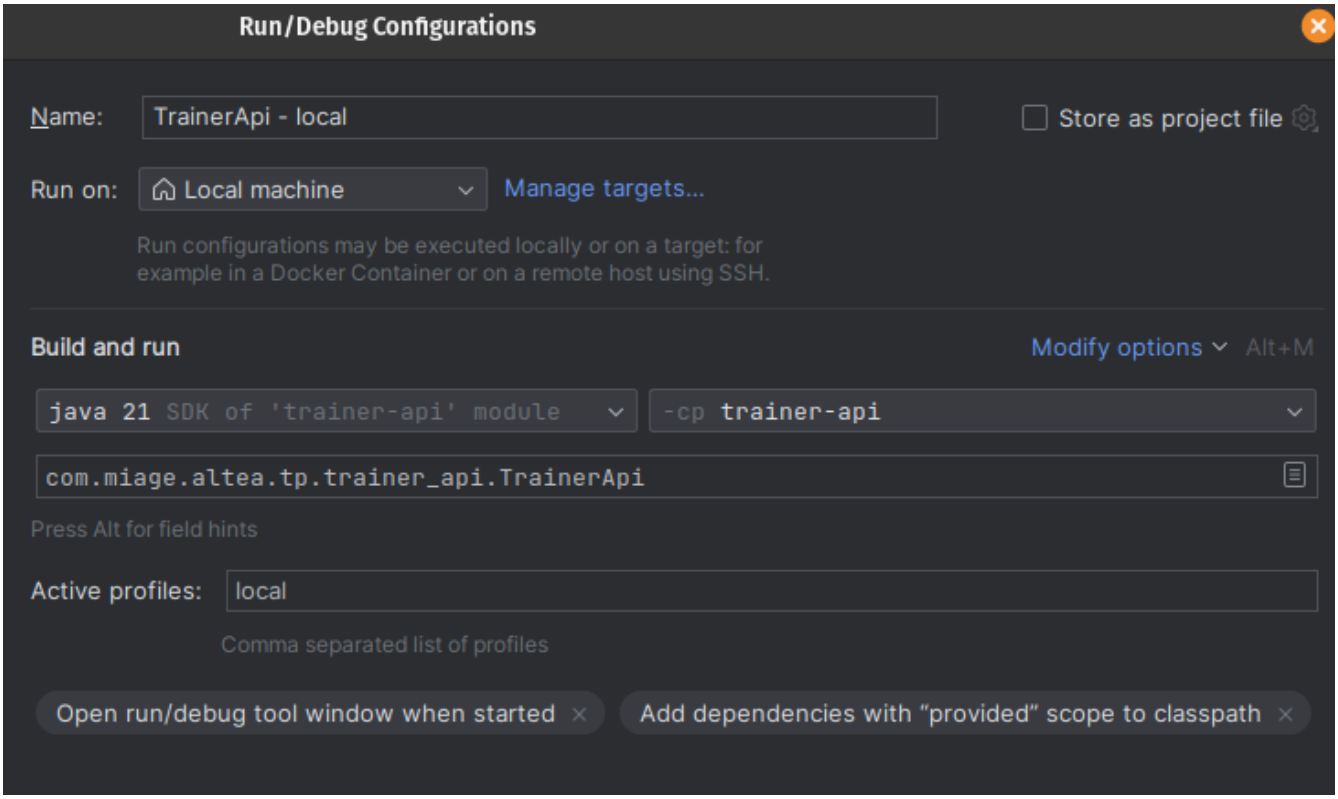
3.2. Activation des profils

3.2.1. Sur votre poste

Lorsque vous démarrez vos micro-services sur votre poste, il vous faut maintenant utiliser le profil `local`. Pour ce faire, vous pouvez indiquer à Spring que le profil local est le profil à utiliser par défaut en absence de tout autre profil.

Pour ce faire, nous allons ajouter un paramètre au lancement de notre application.

Dans IntelliJ, ce paramètre s'ajoute dans la fenêtre de lancement :



Pour les autres IDE, le paramètre peut être passé à la ligne de commande java :

```
java -jar trainer-api.jar --spring.profiles.active=local
```

ou positionné avec une variable d'environnement :

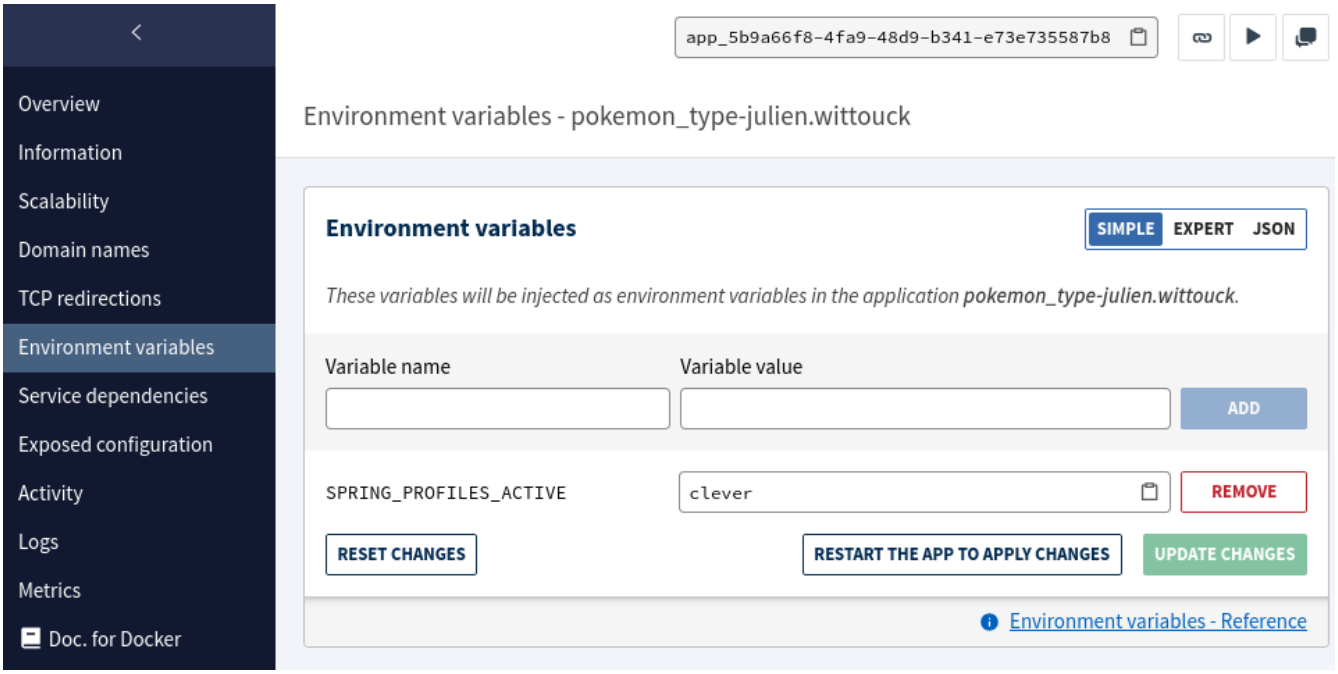
```
export SPRING_PROFILES_ACTIVE=local
java -jar trainer-api.jar
```

3.2.2. Sur l'environnement Clever

Pour vos applications déployées sur Clever-Cloud, nous allons utiliser des variables d'environnement.

L'ajout d'une variable d'environnement se fait directement depuis l'onglet *Environment variables*

d’une application :



Ajoutez à vos applications la variable `SPRING_PROFILES_ACTIVE=clever`.

Vous pouvez aussi ajouter la variable `SERVER_PORT=8080` pour forcer l’utilisation de ce port si vous ne l’avez pas déclaré dans vos fichier `application-clever.properties`

4. Exposition de métriques

L’exposition de métriques pour nos applications se fait avec `spring-boot-actuator`.

4.1. Dépendance maven

Ajoutez la dépendance maven suivante dans vos projets :

pom.xml

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-actuator</artifactId>
</dependency>
```

Ajouter la dépendance suffit à *spring-boot* pour configurer des routes par défaut.

Démarrez ensuite vos applications. Vous devriez y observer des logs dédiés à actuator au démarrage :

```
INFO 28827 --- [main] o.s.b.a.e.web.EndpointLinksResolver      : Exposing 2
endpoint(s) beneath base path '/actuator'
TRACE 28827 --- [main] s.b.a.e.w.s.WebMvcEndpointHandlerMapping : Register "{GET
```

```
/actuator/health, produces [application/vnd.spring-boot.actuator.v3+json ||
application/vnd.spring-boot.actuator.v2+json || application/json]}" to
java.lang.Object
org.springframework.boot.actuate.endpoint.web.servlet.AbstractWebMvcEndpointHandlerMap
ping$OperationHandler.handle(javax.servlet.http.HttpServletRequest,java.util.Map<java.
lang.String, java.lang.String>)
TRACE 28827 --- [main] s.b.a.e.w.s.WebMvcEndpointHandlerMapping : Register "{GET
/actuator/health/**, produces [application/vnd.spring-boot.actuator.v3+json ||
application/vnd.spring-boot.actuator.v2+json || application/json]}" to
java.lang.Object
org.springframework.boot.actuate.endpoint.web.servlet.AbstractWebMvcEndpointHandlerMap
ping$OperationHandler.handle(javax.servlet.http.HttpServletRequest,java.util.Map<java.
lang.String, java.lang.String>)
TRACE 28827 --- [main] s.b.a.e.w.s.WebMvcEndpointHandlerMapping : Register "{GET
/actuator/info, produces [application/vnd.spring-boot.actuator.v3+json ||
application/vnd.spring-boot.actuator.v2+json || application/json]}" to
java.lang.Object
org.springframework.boot.actuate.endpoint.web.servlet.AbstractWebMvcEndpointHandlerMap
ping$OperationHandler.handle(javax.servlet.http.HttpServletRequest,java.util.Map<java.
lang.String, java.lang.String>)
TRACE 28827 --- [main] s.b.a.e.w.s.WebMvcEndpointHandlerMapping : Register "{GET
/actuator, produces [application/vnd.spring-boot.actuator.v3+json ||
application/vnd.spring-boot.actuator.v2+json || application/json]}" to public
java.util.Map<java.lang.String, java.util.Map<java.lang.String,
org.springframework.boot.actuate.endpoint.web.Link>>
org.springframework.boot.actuate.endpoint.web.servlet.WebMvcEndpointHandlerMapping$Web
MvcLinksHandler.links(javax.servlet.http.HttpServletRequest,javax.servlet.http.HttpSer
vletResponse)
```

Consultez enfin pour vos services le endpoint "/actuator/health" :

GET localhost:8080/actuator/health

```
{"status":"UP"}
```

Modifiez le port d'appel en fonction du port d'exposition de votre application !

Sur l'API Trainer, le endpoint `/actuator` est peut-être sécurisé si vous avez terminé la partie 2 du TP Security. Auquel cas, ajoutez une règle pour autoriser les requêtes non-authentifiées à `/actuator`.

4.2. Activer d'autres endpoints

`spring-boot-actuator` propose de nombreux endpoints par défaut, dont la liste est documentée dans la section Production Ready Features / Endpoints de la documentation de Spring Boot.

Exposez au moins les endpoints suivants :

- `health`
- `env`
- `metrics`

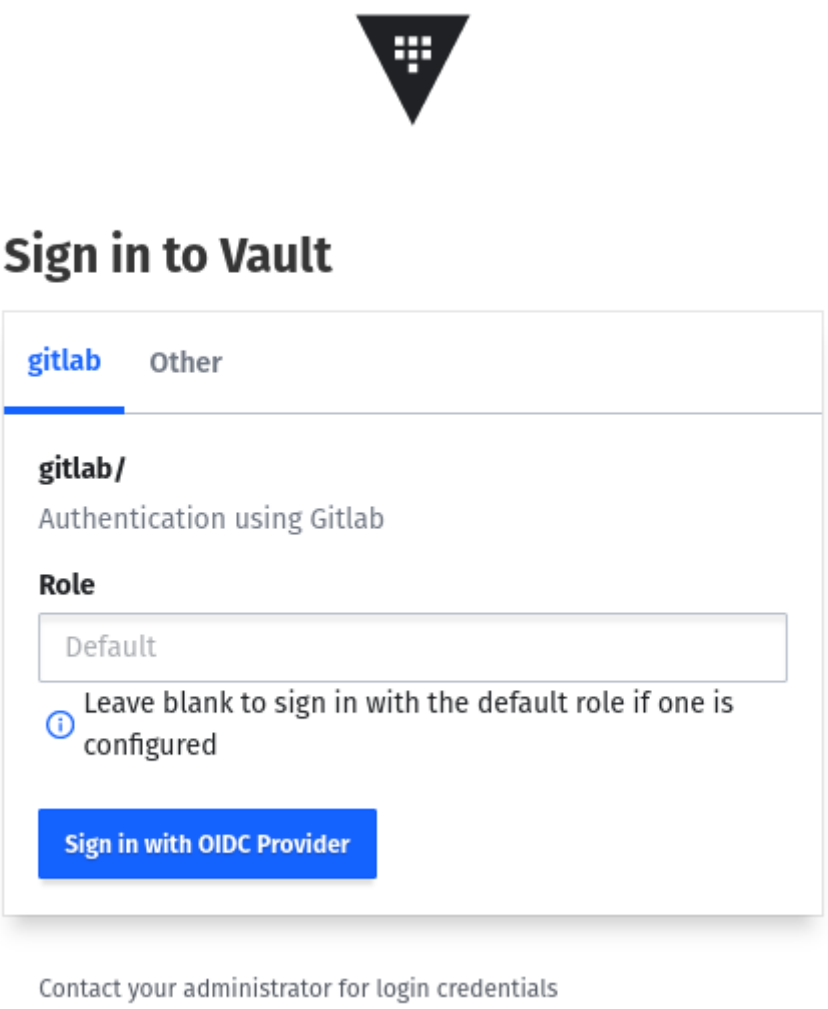
Allez jeter un œil aux endpoints suivants :

- http://localhost:8080/actuator/env
- http://localhost:8080/actuator/metrics
- http://localhost:8080/actuator/metrics/process.cpu.usage

5. Connection au Vault

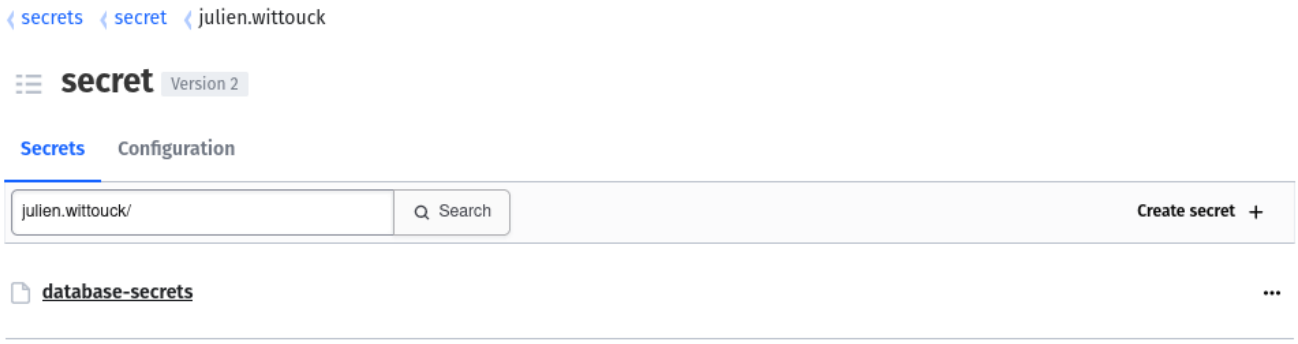
Un serveur *Vault* est disponible à l'adresse suivante : https://vault-alom.cleverapps.io

Vous pouvez vous y connecter avec vos identifiants GitLab (laissez le rôle vide) :

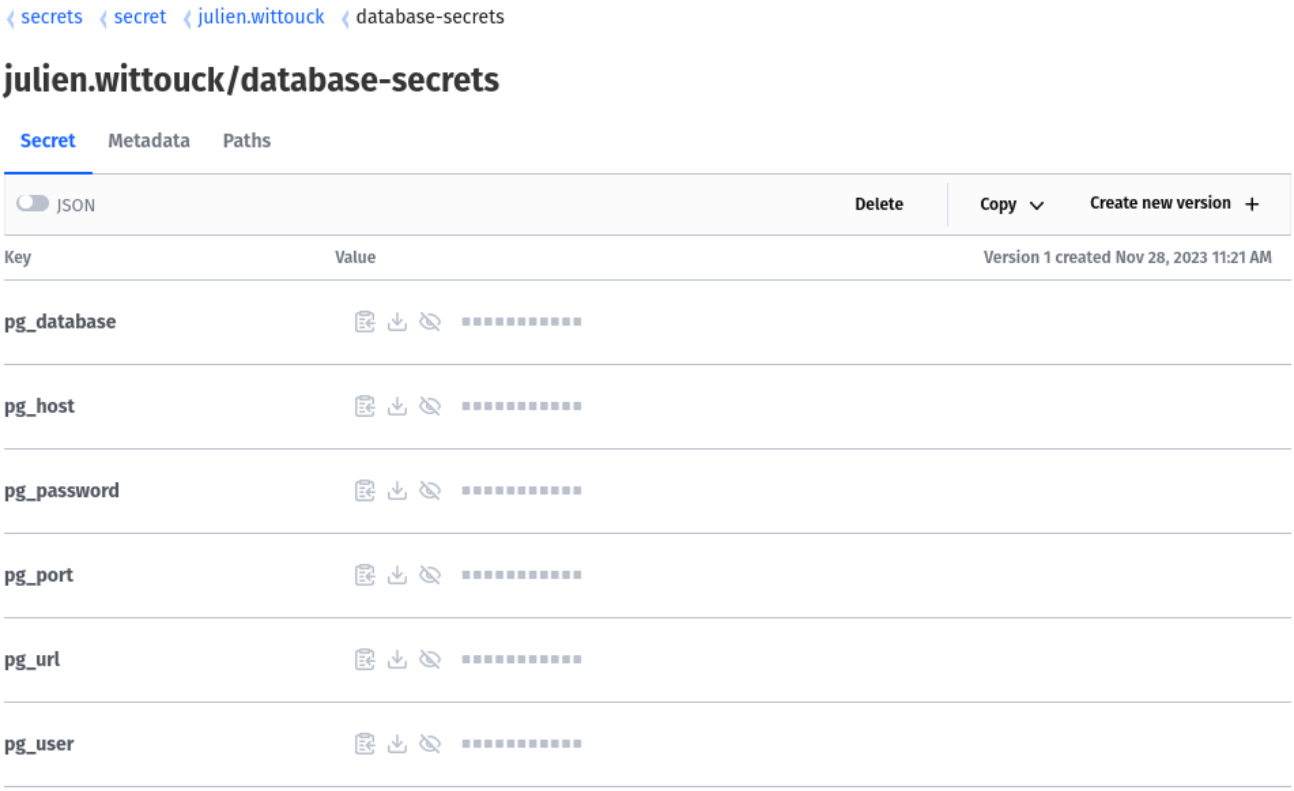


Une fois authentifié, ouvrez le *Secret Engine* nommé *secret/*.

Vous y trouverez un espace pour chacun d'entre vous. Vous avez les droits pour consulter / modifier les secrets qui vous appartiennent.



Un secret *database-secrets* a déjà été initialisé pour vous, avez les informations liées à votre base de données.



5.1. Spring Cloud Vault

Pour connecter votre application au Vault, ajoutez la dépendance suivante à vos projets :

pom.xml

```
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-vault-config</artifactId>
    <version>4.3.0</version>
</dependency>
```

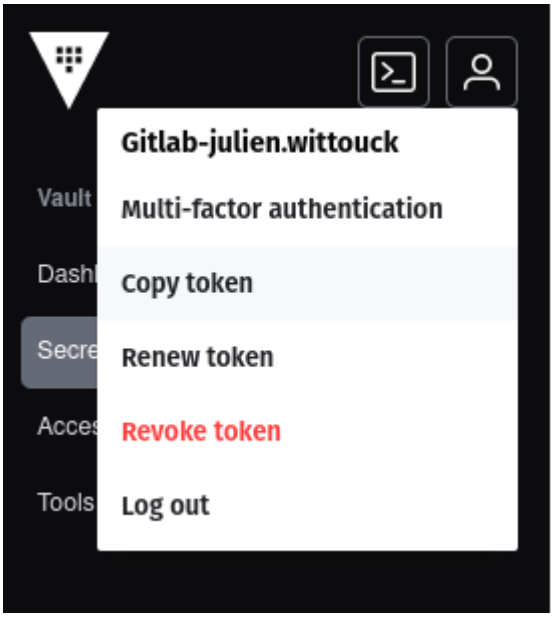
Configurez ensuite les properties suivantes :

application-clever.properties

```
spring.cloud.vault.uri=https://vault-alom.cleverapps.io/
spring.cloud.vault.token=
spring.config.import=vault://secret/${gitlab.user.name}/database-secrets
gitlab.user.name=
```

Indiquez votre nom d'utilisateur GitLab (se terminant par `.etu`), dans la properties `gitlab.user.name`.

Pour le token, une fois connecté au Vault, vous pouvez en récupérer un en dans le menu.



Dans vos properties locales, si vous ne voulez pas utiliser le Vault, vous pouvez aussi ajouter la properties `spring.cloud.vault.enabled=false`.

Comme Clever Cloud exécute les tests avec maven au démarrage de l'application, tout en ayant la variable d'environnement `SPRING_PROFILES_ACTIVE` injectée, vous pouvez aussi ajouter un fichier `src/test/resources/application-clever.properties` vide pour éviter que les tests consomment les properties de prod.

5.2. Reconfiguration des properties

Reconfigurez vos properties, en particulier l'accès à la base de données (url, user, mot de passe). Les properties utilisées par Vault sont accessibles directement, vous pouvez par exemple écrire la properties suivante : `spring.datasource.username=${pg_user}`, la valeur `pg_user` de votre Vault sera chargée au démarrage de l'application.

6. Envoi d'emails

Pour l'envoi d'emails, initialisez un nouveau projet avec le lien suivant : GitLab Classrooms.

Ajoutez dans le `pom.xml` les dépendances suivantes :

- spring-boot-starter-mail
- spring-cloud-starter-vault-config

Importez le secret nommé `mail-secrets` depuis le Vault.

Pour envoyer un email, recevez en injection de dépendance une instance de `JavaMailSender`.

Voici un exemple de code qui envoie un email :

SendMail.java

```
var message = mailSender.createMimeMessage();
message.setFrom("no-reply@gitlab-classrooms.org");
message.setRecipients(Message.RecipientType.TO, "");
message.setSubject("Test From Spring Boot");
message.setText("This is a test from a Spring Boot application.");
mailSender.send(message);
```

Exposez des routes HTTP dans votre service de mailing qui permettent d'envoyer différents mails :

- un email de bienvenue (reçoit le nom d'un Trainer en paramètre, ainsi que son email)
- un email de fin de combat (reçoit les noms des 2 adversaires, leurs emails, et l'information de qui a gagné le combat)

Modifiez en conséquence le `game-ui` pour appeler ces deux nouvelles routes depuis `game-ui` et `battle-api`.