Projet OPSCI: Infrastructure complète avec objets connectés et architecture événementielle

Objectif général

L'objectif de ce projet est de construire une architecture complète pour une plateforme de gestion de produits intégrant une base de données, un système d'événements et des objets connectés.

Structure du projet

Le projet se compose de trois parties : 1. **Déploiement de l'infrastructure de base** (Strapi, PostgreSQL, Frontend React) 2. **Mise en place d'une architecture événementielle** (Kafka et gestion des flux de données)

Partie 1 : Infrastructure de base (Strapi, PostgreSQL, Frontend React)

Objectif

Déployer une infrastructure de gestion des produits avec une base de données PostgreSQL, un CMS Strapi et un frontend React.

Tâches

- Déployer **Strapi** pour la gestion des produits
- Mettre en place une base de données **PostgreSQL** pour stocker les données
- Configurer et déployer un **frontend React** connecté à l'API de Strapi
- Générer un **token API** pour sécuriser les échanges entre le frontend et Strapi

Architecture

Strapi

La première étape pour déployer l'application est de créer un projet strapi.

```
yarn create strapi-app ${project-name}
```

Configurez l'application: choisissez le mode avancé et séléctionnez postgresql.

Une fois le projet créé (dans le dossier `project-name`) créez un Dockerfile pour créer une image docker du projet.

Strapi nous fourni une documentation et un exemple de dockerfile : https://docs.strapi.io/dev-docs/installation/docker

Certaines de ces configurations doivent être cohérentes avec celle de la base de données `Postgresql`.

Résultat attendu:

Strapi est lancé correctement :

Se connecter avec un administrateur :

Créer la collection `product` :

Attention! si la collection `product` n'est pas créé ou mal créé le frontend pourrait avoir des erreurs

Product

name: short text description: long text stock_available: integer (default 0) image: single media (only image) barcode: short text

Postresql

Pour que Strapi fonctionne il faut lui fournir une base de données.

Pour déployer cette base de données nous allons utiliser docker.

Il y a quelques configuration à ajouter pour être sur de correctement initialiser la base de données :

`POSTGRES_PASSWORD` et `POSTGRES_USER`

Que l'ont peut communiquer à l'image postresql de cette façon :

```
docker run -dit -p 5432:5432 -e POSTGRES_PASSWORD=safepassword -e
        POSTGRES_USER=strapi --name strapi-pg postgres
```

NB: par défaut une base postresql expose ses services sur le port 5432. Libre à vous de modifier cette configuration si besoin.

React

En addition avec le frontend administrateur de strapi un projet react vous est fourni et doit se connecter à l'API de strapi.

Vous devez récupérer le code : https://github.com/arthurescriou/opsci-strapi-frontend.

Vous devez également configurer ce frontend et le compiler/minifier (cf readme).

NB: pour se connecter à l'API de strapi, outre l'URL correct, il faut ajouter un `TOKEN` de sécurité. Il est possible de générer ce token dans l'interface administateur. (https://docs.strapi.io/dev-docs/configurations/api-tokens) Il faudra l'ajouter dans la configuration du frontend.

Résultat attendu (les deux élèments sont des produits créé dans strapi):

Travail attendu

L'objectif de cette partie est de créer toute l'application décrite ci-dessus.

Chacun des ces éléments devra être déployé à partir d'une image docker sur l'infrastructure fourni par l'UE.

Vous devrez configurer correctement les conteneurs pour qu'ils interagissent entre eux.

Partie 2 : Architecture évènementielle avec Kafka

Objectif

Intégrer un système de gestion des évènements basé sur **Kafka** pour traiter des flux de données en temps réel.

Prérequis

Pour facilier le lancement du premier projet vous pouvez utiliser ce docker-compose, ou directement votre travail.

Si vous utilisez votre travail il faudra modifier le type de `product` : Ajouter un champs `status` de type `enumeration` : `safe|danger|empty`

Et créer un type event avec un champ `value` de type `string` et un champ `metadata` de type `JSON`.

Le projet initialise les formats des objets nécessaire au projet 2. Il peut être nécessaire de regénérer un API TOKEN.

```yaml
version: '3'
services:
  strapi:
    container_name: strapi
    image: arthurescriou/strapi:1.0.0
    restart: unless-stopped
```

```
    env_file: .env
    environment:
      DATABASE_CLIENT: ${DATABASE_CLIENT}
      DATABASE_HOST: strapiDB
      DATABASE_PORT: ${DATABASE_PORT}
      DATABASE_NAME: ${DATABASE_NAME}
      DATABASE_USERNAME: ${DATABASE_USERNAME}
      DATABASE_PASSWORD: ${DATABASE_PASSWORD}
      JWT_SECRET: ${JWT_SECRET}
      ADMIN_JWT_SECRET: ${ADMIN_JWT_SECRET}
      APP_KEYS: ${APP_KEYS}
      NODE_ENV: ${NODE_ENV}
      PORT: 8080
    ports:
      - '8080:8080'
    networks:
      - strapi
    depends_on:
      - strapiDB

  strapiDB:
    container_name: strapiDB
    restart: unless-stopped
    env_file: .env
    image: arthurescriou/strapi-pg:1.0.0
    environment:
      POSTGRES_USER: ${DATABASE_USERNAME}
      POSTGRES_PASSWORD: ${DATABASE_PASSWORD}
      POSTGRES_DB: ${DATABASE_NAME}

    ports:
      - '5432:5432'
    networks:
      - strapi

networks:
  strapi:
    name: Strapi
    driver: bridge
```

Tâches

- Déployer un **broker Kafka** et **Zookeeper**
- Créer plusieurs **topics** pour gérer les flux de données :
    - `product` : ajout de nouveaux produits

    - `event` : enregistrement d'évènements liés aux produits
    - `stock` : gestion des mises à jour de stock
    - `error` : stockage des erreurs
- Déployer des **producers et consumers** Kafka pour chaque flux
- Intégrer Strapi avec Kafka pour gérer les mises à jour des produits

Architecture

On souhaite créer un système permettant d'intégrer de grande quantité de données venant de différents flux avec beaucoup de résilience à l'erreur.

Pour ça on utilise un message broker: Kafka.

Kafka

On veut lancer un kafka avec docker (ou kubernetes).

On utilise la même solution qu'au TME7.

```yaml
version: '3'
services:
  zookeeper:
    image: wurstmeister/zookeeper:latest
    container_name: zookeeper
    ports:
      - '2181:2181'
    expose:
      - '2181'

  kafka:
    image: wurstmeister/kafka:2.11-1.1.1
    container_name: kafka
    ports:
      - '9092:9092'
      - '9093:9093'
    environment:
      KAFKA_ADVERTISED_LISTENERS: INSIDE://localhost:
        9093,OUTSIDE://kafka:9092,
      KAFKA_AUTO_CREATE_TOPICS_ENABLE: 'true'
      KAFKA_DELETE_TOPIC_ENABLE: 'true'
      KAFKA_ADVERTISED_PORT: 9092
      KAFKA_ZOOKEEPER_CONNECT: 'zookeeper:2181'
      KAFKA_OFFSETS_TOPIC_REPLICATION_FACTOR: 1
      KAFKAJS_NO_PARTITIONER_WARNING: '1'
      KAFKA_LISTENER_SECURITY_PROTOCOL_MAP:
        INSIDE:PLAINTEXT,OUTSIDE:PLAINTEXT
```

```
    KAFKA_LISTENERS: INSIDE://0.0.0.0:9093,OUTSIDE://
      0.0.0.0:9092
    KAFKA_INTER_BROKER_LISTENER_NAME: INSIDE
    KAFKA_NO_LISTENER_AUTHENTICATION_PLAINTEXT: 'true'
    KAFKA_NO_LISTENER_AUTHENTICATION_SSL: 'true'
    KAFKA_BROKER_ID: 1
    KAFKA_LOG_RETENTION_HOURS: 168
    KAFKA_LOG_RETENTION_BYTES: 1073741824
    KAFKA_LOG_DIRS: /kafka/logs
  volumes:
    - /var/run/docker.sock:/var/run/docker.sock
  depends_on:
    - zookeeper
```

Topics

Notre kafka va nécéssiter plusieurs topics :

- product : un topic dédié à la création de nouveau produit en masse, venant de différentes sources
- event : un topic dédié à la création de nouveau produit en masse, venant de différentes sources
- stock : un topic pour enregistrer et appliquer tous les mouvements de stocks de nos produits
- error : pour récupérer les évènement ayant abouti à une erreur diverse

Attention tous ces topics devront être précisés dans les configurations des consumers et producers. Si le noms n'est pas le même des deux cotées la communication ne marchera pas.

Consumer et Producer

Pour utiliser ces topics plusieurs élèments sont à votre disposition sous la forme d'image docker ou de dépôt github (https://github.com/orgs/opsci-su/repositories).

Il s'agit des consumers et producers pour les différents éléments.

Pensez bien à lire les readme des différents élèments pour bien configurer leurs variables d'environments.

product-producer

https://hub.docker.com/repository/docker/arthurescriou/product-producer/

Pour lancer la création de `product` insérez le fichier dans votre container et spécifiez sa position dans la configuration.

product-consumer

https://hub.docker.com/repository/docker/arthurescriou/product-consumer/

event-producer

https://hub.docker.com/repository/docker/arthurescriou/event-producer/

Pour lancer la création de `event` insérez le fichier dans votre container et spécifiez sa position dans la configuration.

event-consumer

https://hub.docker.com/repository/docker/arthurescriou/event-consumer/

stock-producer

https://hub.docker.com/repository/docker/arthurescriou/stock-producer/

Pour lancer l'update de `stocks` insérez le fichier dans votre container et spécifiez sa position dans la configuration.

stock-consumer

https://hub.docker.com/repository/docker/arthurescriou/stock-consumer/

artificial-intelligence

Rendu attendu

- **Vidéo de démonstration** : Parcours complet du projet, de l'ajout d'un produit à sa mise à jour via Kafka.
- **Dépôt Git** contenant :
  - Les fichiers de configuration (Dockerfiles, docker-compose.yml)
  - Les scripts de lancement et d'arrêt des services
  - Le code source du frontend React
  - Un `readme.md` détaillant le déploiement et l'utilisation de l'application.

Envoi du rendu

- Uniquement via l'espace Moodle de l'UE.

Date de rendu

- ** 31/03/2025 à 23h59**

- Travail en **binôme** avec les numéros étudiants mentionnés dans le `readme.md`.
