Raport z wdrożenia aplikacji mikroserwisowej w Docker oraz Kubernetes

1 Etap I – Wdrożenie aplikacji z użyciem Docker Compose

Aplikacja to system do zarządzania zadaniami, w którym administratorzy tworzą i przydzielają zadania, a użytkownicy mogą je przeglądać i oznaczać jako wykonane. Składa się ona z czterech głównych mikroserwisów:

- Frontend (React)
- Backend (FastAPI)
- Keycloak (uwierzytelnianie)
- PostgreSQL (baza danych)

1.1 Podział na mikroserwisy

- **Backend** – udostępnia REST API dla administratora do obsugi zadań oraz użytkowników (Flask).
- **Backend-express** – udostępnia REST API do obsugi zadań oraz użytkowników (Express).
- **Frontend** – komunikuje się z backendem oraz z Keycloak w celu logowania (React).
- **Keycloak** – serwer uwierzytelniania i zarządzania tożsamościami.
- **PostgreSQL** – baza danych dla całego systemu.

1.2 Plik docker-compose.yml

W pliku `docker-compose.yml` zdefiniowano cztery usługi:

- **postgres** – kontener Postgres 16 z wolumenem `postgres-data`, healthcheckiem i zmiennymi środowiskowymi.
- **backend** – obraz z `./backend/python`, healthcheck HTTP, zależność od Postgresa.
- **express-backend** – obraz z `./backend/express`, healthcheck HTTP, zależność od Postgresa.
- **keycloak** – obraz lokalny z `./keycloak`, import realm, healthcheck, wolumen `keycloak-data`.
- **frontend** – React + NGINX, healthcheck i zależności.

1.3 Docker networks i komunikacja

- `db_network` – łączy backend, Postgres i Keycloak.
- `frontend_network` – łączy frontend, backend i Keycloak.

Komunikacja między mikroserwisami odbywa się przez nazwy usług (DNS Dockera).

1.4 Wolumeny i trwałość danych

- **Postgres** – wolumen `postgres-data`.
- **Keycloak** – wolumen `keycloak-data`.

1.5 Dockerfile i optymalizacja

- Frontend (React) – multi-stage build + serwowanie przez NGINX.
- Keycloak – import realm z `config.json`, oparcie o oficjalny obraz.

Użyto plików `.dockerignore` do redukcji rozmiaru obrazów.

1.6 Multiplatformowość i `docker buildx`

W projekcie zastosowano mechanizm budowania obrazów wieloplatformowych z wykorzystaniem rozszerzenia `docker buildx`. Obrazy są budowane jednocześnie dla dwóch architektur: `linux/amd64` oraz `linux/arm64`. Konfiguracja ta została użyta we wszystkich kluczowych komponentach systemu: `backend`, `frontend` oraz `keycloak`.

- Dla każdej usługi zdefiniowano platformy docelowe poprzez pole `platforms` w rozszerzeniu `x-buildx`.
- Budowanie obrazów korzysta z mechanizmu cache (`cache-from` oraz `cache-to`), co przyspiesza kolejne przebiegi procesu CI/CD.
- Opcja `push: true` zapewnia automatyczne publikowanie manifestu wieloplatformowego do zdalnego rejestru.

Takie podejście umożliwia uruchamianie kontenerów na różnych architekturach CPU (np. standardowych komputerach z procesorami x86_64 oraz urządzeniach ARM, takich jak Raspberry Pi czy MacBooki z procesorami Apple Silicon), bez potrzeby modyfikacji kodu źródłowego lub konfiguracji kontenerów.

1.7 Mechanizmy bezpieczeństwa i monitorowania

- Dla każdej usługi zdefiniowano mechanizm `healthcheck`, który okresowo sprawdza stan aplikacji wewnątrz kontenera. W przypadku bazy danych PostgreSQL wykorzystywana jest komenda `pg_isready`, natomiast usługi HTTP (`backend`, `frontend`) i `keycloak` monitorowane są poprzez zapytania `curl` lub wywołania własnych komend diagnostycznych.

- Wrażliwe dane, takie jak hasło do bazy danych czy klucz JWT, są przekazywane do kontenerów za pomocą mechanizmu `Docker secrets`, który umożliwia bezpieczne przechowywanie danych w postaci plików dostępnych wyłącznie wewnątrz kontenera w katalogu `/run/secrets`. Dzięki temu unika się trzymania haseł bezpośrednio w pliku `docker-compose.yml` lub zmiennych środowiskowych.

2 Etap II – Migracja do Kubernetes

2.1 Manifesty zasobów Kubernetes

W ramach projektu przygotowano pełny zestaw manifestów YAML do uruchomienia aplikacji w środowisku Kubernetes. Skonfigurowano zasoby typu Deployment, Service, ConfigMap, Secret, Ingress, PersistentVolume, PersistentVolumeClaim, `HorizontalPodAutoscaler`. Każdy manifest zawiera poprawnie zdefiniowane etykiety i selektory, umożliwiające bezbłędne powiązanie obiektów w klastrze.

- **Zarządzanie konfiguracją i sekretami:**
  - Plik `backend-config-configmap.yaml` zawiera zmienne środowiskowe dla backendu, przekazywane jako `ConfigMap`, co umożliwia centralne zarządzanie konfiguracją aplikacji.
  - Hasła oraz klucze prywatne przekazywane są jako zasoby typu `Secret`:
    - `pg-password-secret.yaml` – hasło do bazy danych PostgreSQL,
    - `kc-admin-password-secret.yaml` – hasło administratora Keycloak,
    - `jwt-private-key-secret.yaml` – klucz prywatny JWT.
  - Sekrety montowane są do kontenerów jako pliki lub przekazywane jako zmienne środowiskowe z użyciem `valueFrom.secretKeyRef`.
- **Warstwa aplikacyjna:**
  - `backend-deployment.yaml`, `frontend-deployment.yaml`, `keycloak-deployment.yaml` zawierają definicje replik, obrazów kontenerów, wolumenów i zmiennych środowiskowych, wraz z selektorami opartymi o etykiety `app` oraz `component`.
  - `frontend-ingress.yaml` definiuje reguły dostępu HTTP, umożliwiające udostępnienie aplikacji na zewnątrz klastra.
- **Warstwa usług:**
  - `backend-service.yaml`, `express-backend-service.yaml`, `frontend-service.yaml`, `keycloak-service.yaml`, `postgres-service.yaml` definiują zasoby typu `Service` typu `ClusterIP` i `LoadBalancer`, eksponując aplikacje wewnątrz klastra.
  - Porty i selektory w usługach są zgodne z etykietami `app` i `component` w zasobach `Deployment`.
- **Warstwa danych:**
  - postgres-data-persistentvolume.yaml, postgres-data-persistentvolumeclaim.yaml, keycloak-data-persistentvolume.yaml, keycloak-data-persistentvolumeclaim.yaml zapewniają trwałość danych poprzez zewnętrzne wolumeny.

  - Wolumeny montowane są we właściwych ścieżkach w kontenerach PostgreSQL i Keycloak.

- **Skalowalność pozioma (autoskalowanie):**
  - Pliki konfiguracyjne `hpa-backend.yaml` i `hpa-frontend.yaml` definiują zasoby **Horizontal Pod Autoscaler (HPA)**, które umożliwiają automatyczne skalowanie aplikacji (backendu i frontendu) w odpowiedzi na zmiany w obciążeniu systemu, mierzone na podstawie wykorzystania CPU. Dzięki temu, liczba replik podów jest dynamicznie dostosowywana, zapewniając odpowiednią wydajność aplikacji w zależności od zapotrzebowania.