AKADEMIA GÓRNICZO-HUTNICZA IM. STANISŁAWA STASZICA W KRAKOWIE

Wydział Informatyki

INSTYTUT INFORMATYKI

Środowiska Udostępniania Usług

Grupa 4 - czwartek 9:45

Operator Framework - studium przypadku technologii

Operator Framework - a case study in technology

Dominika Bocheńczyk

Mateusz Łopaciński

Piotr Magiera

Michał Wójcik

Kraków, 2024

Spis treści

Rozdział 1

Wprowadzenie

Kubernetes to niekwestionowany lider w segmencie automatyzacji i orkiestracji aplikacji kontenerowych, pełniąc kluczową rolę w skalowalnym wdrażaniu oprogramowania na infrastrukturę sieciową. Doskonale wpisuje się w trend zastępowania architektur monolitycznych przez mikroserwisy, które znacząco zwiększają reaktywność i elastyczność systemów informatycznych. Istnieje jednakże pewien podzbiór aplikacji, dla którego pojawiają się wyzwania związane z automatyzacją ich obsługi, zwłaszcza w sytuacji awarii lub dynamicznego zwiększania obciążenia. Opisane aplikacje, to oprogramowanie typu *stateful*, takie jak bazy danych (Postgres, MySQL, Redis), middleware (RabbitMQ) czy systemy monitorowania (Prometheus), których stan jest krytyczny dla ciągłości działania i nie może zostać utracony w przypadku awarii.

Rysunek 1.1: Przykładowe aplikacje typu stateful

Zaproponowane przez twórców Kubernetes rozwiązania, takie jak Stateful Sets połączone z Persistent Volumes, umożliwiają utrzymanie danych na dysku i relacji master-slave między replikami baz danych, jednakże ich konfiguracja i zarządzanie mogą być złożone i czasochłonne, co utrudnia w pełni automatyczne zarządzanie cyklem życia tych aplikacji. Ponadto, standardowe narzędzia Kubernetes nie zawsze dostarczają wystarczających możliwości zarządzania stanem aplikacji, co może skutkować koniecznością utrzymywania systemów bazodanowych poza klastrem Kubernetes, co jest niezgodne z ideą Infrastructure as Code.

Rozdział 2

Podstawy teoretyczne i stos technologiczny

2.1. Podstawy teoretyczne

Operator Framework rozszerza możliwości Kubernetes, dostarczając narzędzia umożliwiające tworzenie operatorów – specjalistycznych programów, które zarządzają innymi aplikacjami wewnątrz klastra Kubernetes. Operatorzy są projektowani tak, aby w sposób ciągły monitorować stan aplikacji, automatycznie podejmując decyzje o koniecznych działaniach naprawczych, skalowaniu, aktualizacji lub konfiguracji w odpowiedzi na zmieniające się warunki operacyjne.

Istotą Operator Framework jest umożliwienie automatyzacji operacji, które tradycyjnie wymagałyby ręcznego przeprowadzenia przez zespoły operacyjne lub administratorów systemów. Przykładowo, operator bazy danych nie tylko zarządza replikacją danych, ale również może automatycznie zarządzać schematami bazy danych, przeprowadzać rotację certyfikatów, czy realizować procedury backupu i przywracania danych.

Jako że każda aplikacja typu stateful może posiadać specyficzny sposób zarządzania, potrzebuje ona swojego własnego Operatora. Z tego względu istnieje nawet publiczne repozytorium, z którego można pobrać konfiguracje opracowane pod konkretne oprogramowanie (znajduje się ono pod linkiem https://operatorhub.io/).

Wzorzec Operator pozwala na łączenie kontrolerów jednego lub więcej zasobów aby rozszerzyć zachowanie klastra bez konieczności zmiany implementacji. Operatorzy przyjmują rolę kontrolerów dla tzw. Custom Resource. Custom Resource rozszerzają / personalizują konkretne instalacje Kubernetesa, z tym zachowaniem że użytkownicy mogą z nich korzystać jak z wbudowanych już zasobów (np. Pods). [1]

2.2. Stos technologiczny

Głównym komponentem technologicznym naszego projketu jest Operator Framework, który zapewnia zestaw narzędzi, szablonów i wytycznych, ułatwiających programistom tworzenie operatorów, co przyspiesza proces tworzenia nowych aplikacji i usprawnia zarządzanie nimi w środowiskach Kubernetes.

W przypadku wyłącznie prezentacji działania Operatora na klastrze Kubernetesowym, możemy skorzystać z narzędzia jakim jest minikube. Minikube pozwala na szybkie stworzenie lokalnego klastra Kubernetesa w danym systemie operacyjnym. Dzięki temu, mając jedynie kontener Dockerowy lub środowisko maszyny wirtualnej, możemy lepiej skupić się na samej funkcjonalności Kubernetesa dla naszych potrzeb. [2]

Rozdział 3

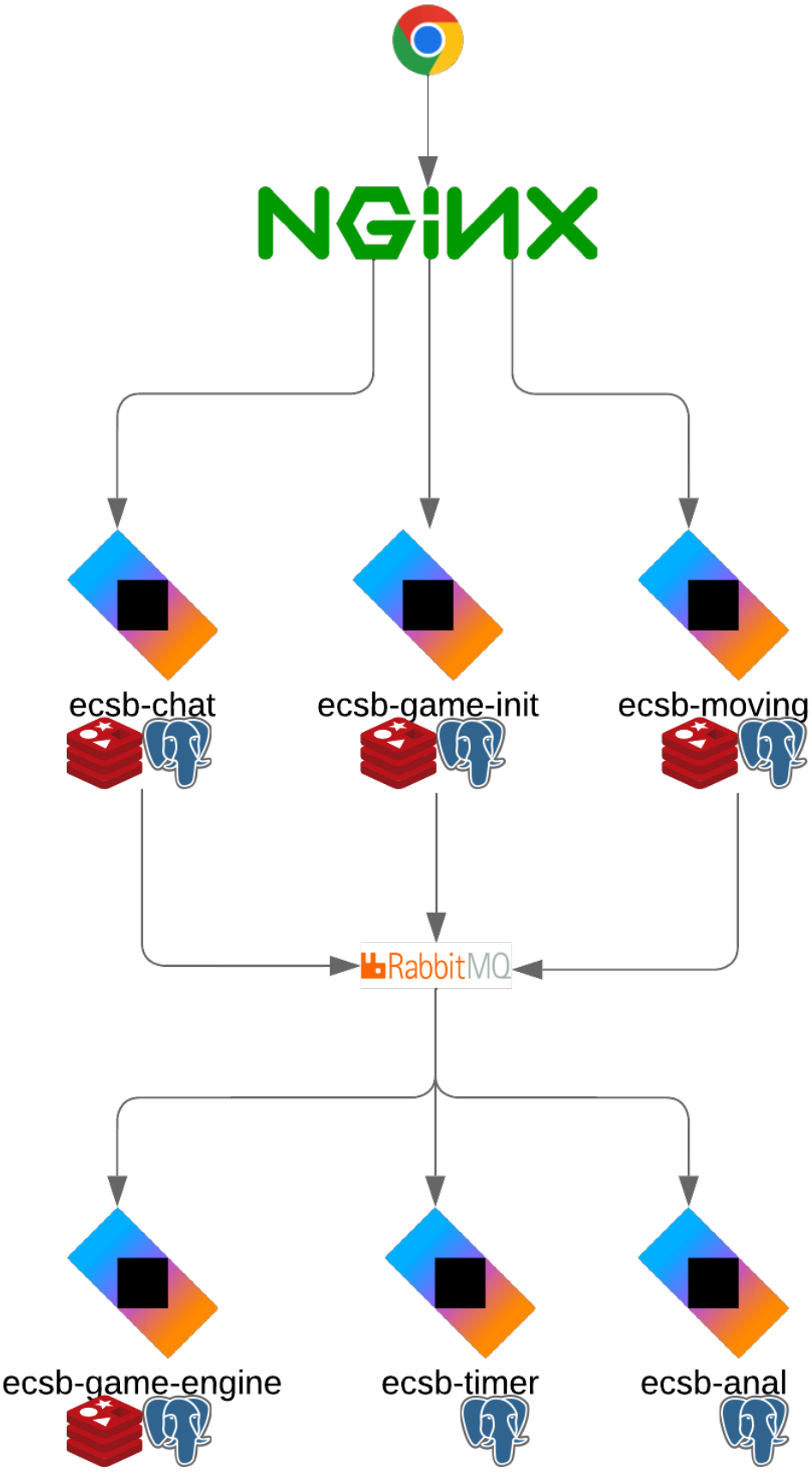
Opis studium przypadku

3.1. Przykład aplikacji typu stateful

Aplikacją, na której pokażemy działanie Operatora, będzie Elektroniczna Chłopska Szkoła Biznesu (eCSB), cyfrowa implementacja gry planszowej 'Chłopska Szkoła Biznesu' wydanej przez Małopolski Instytut Kultury. Aplikacja ta jest pracą inżynierską czworga studentów naszego Wydziału, obecnie kontynuowaną w ramach pracy magisterskiej. Gra polega na produkcji zasobów według przydzielonego zawodu, handlu towarami, zakładaniu spółek oraz odbywaniu wypraw w celu korzystniej wymiany produktów na pieniądze. W czasie kilkunastominutowej rozgrywki każdy z graczy stara się zgromadzić jak największy majątek.

Rysunek 3.1: Ekran powitalny ECSB

eCSB jest aplikacją webową opartą o mikroserwisy oraz komunikację z wykorzystaniem wiadomości. W minimalnym wariancie składa się z 6 bezstanowych modułów napisanych w języku Kotlin, podłączonych do bazy danych Postgres, bazy klucz-wartość Redis oraz brokera wiadomości RabbitMQ, a także aplikacji webowej, komunikującej się z modułami poprzez protokoły REST oraz WebSocket. Dostęp do wszystkich elementów architektury realizowany jest dzięki serwerowi HTTP nginx, który pełni rolę reverse-proxy (zapewniając przy tym certyfikaty SSL i ruch HTTPS) oraz API gateway (przekierowując żądania do odpowiednich mikroserwisów). Pełna architektura rozwiązania przedstawiona jest poniżej 3.2.



Rysunek 3.2: Architektura projektu eCSB

Warto dodać, że niektóre z modułów zostały zaprojektowane z myślą o skalowaniu systemu i rozkładaniu obciążenia. Są to moduły chat, moving oraz game-engine. Pozostałe 3 moduły odpowiadają za usługi stosunkowo rzadkie (tworzenie sesji gry) lub w pełni scentralizowane (zbieranie logów, odświeżanie czasu gry).

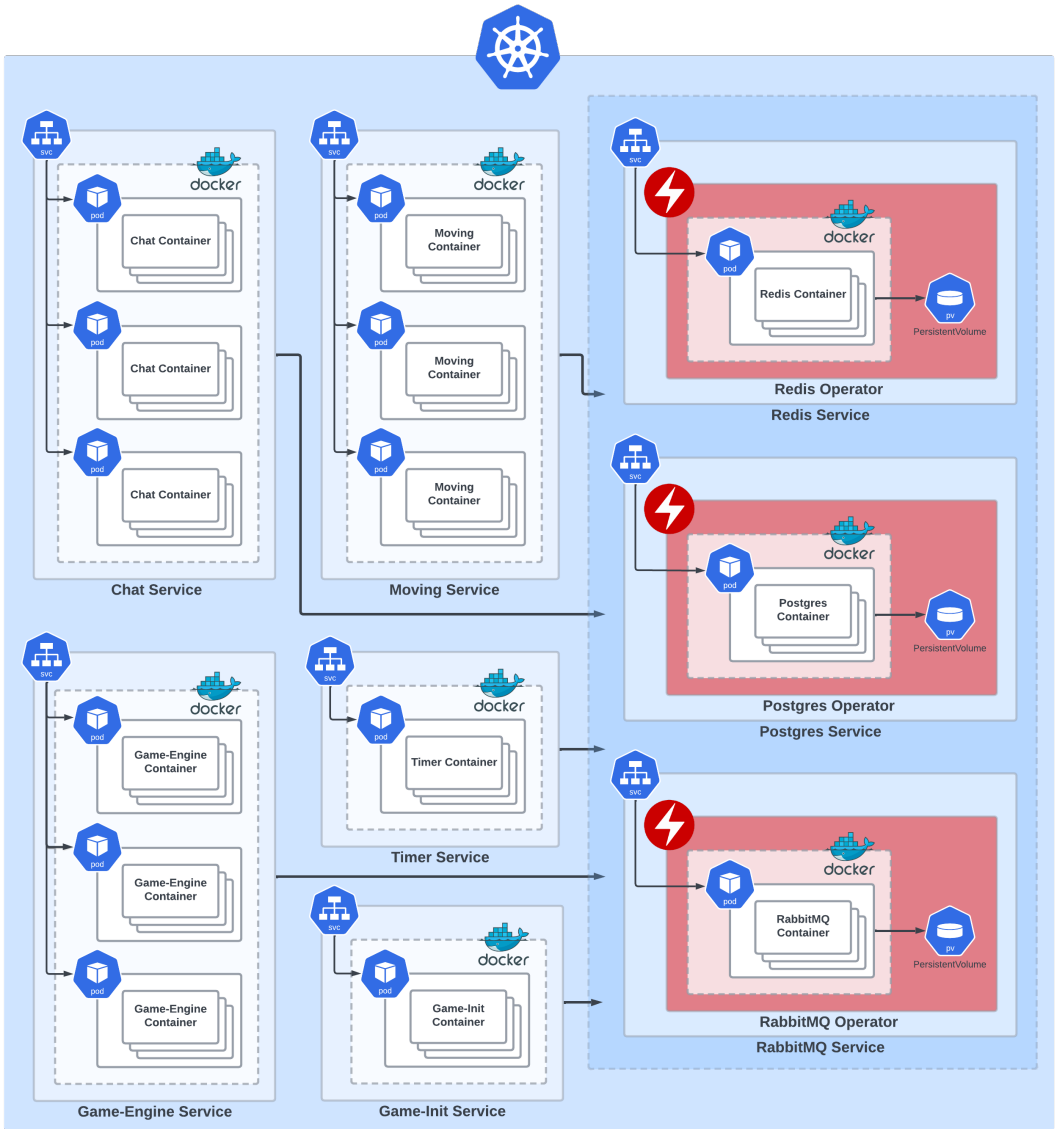
3.2. Przypadek użycia Operatora

Naszym scenariuszem, poprzez który zademonstrujemy działanie Operatora dla aplikacji eCSB, będą następujące operacje:

1. uruchomienie klastra Kubernetesowego z minimalnym wariantem eCSB (5 modułów - pomijamy moduł ecsb-anal, który służył jedynie do zbierania danych analitycznych)
2. potrojenie skalowalnych modułów (moving, chat, game-engine), a następnie powrót do wariantu minimalnego
3. przywrócenie centralnego modułu po awarii (timer lub game-init)

Rozdział 4

Architektura rozwiązania



Rysunek 4.1: Diagram architektury rozwiązania z wykorzystaniem Kubernetesa

Rozdział 5

Konfiguracja środowiska

5.1. Konfiguracja w Docker Compose

Jako punkt wyjścia konfiguracji projektu został wyznaczony zmodyfikowany plik `docker-compose.yml`, za pomocą którego uruchamiane było middleware dla aplikacji backendowych oraz aplikacja webowa. Plik ten został rozszerzony o backendowe mikroserwisy oraz dodano wstępną konfigurację dla PostgreSQL, oraz RabbitMQ:

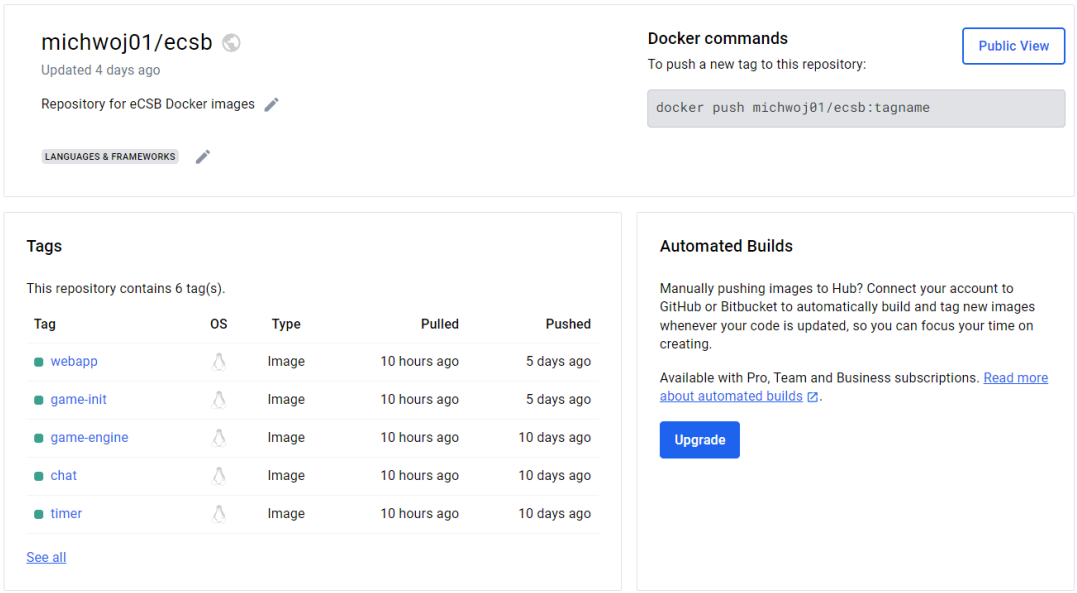
```
FROM gradle:7-jdk11 AS build
COPY .. /home/gradle/ecsb-backend
WORKDIR /home/gradle/ecsb-backend
RUN gradle :ecsb-chat:buildFatJar --no-daemon

FROM openjdk:11-jre-slim
EXPOSE 2138
WORKDIR /app
COPY --from=build /home/gradle/ecsb-backend/ecsb-chat/build/libs/*.jar
    /app/ecsb-chat.jar
ENTRYPOINT ["java", "-jar", "/app/ecsb-chat.jar"]
```

Listing 5.1: Przykład pliku Dockerfile dla mikroserwisu

```
  moving:
    container_name: moving
    image: michwoj01/ecsb:moving
    build:
      context: ecsb-backend
      dockerfile: ./ecsb-moving/Dockerfile
    depends_on:
      - postgres
      - redis
      - rabbitmq
    restart: on-failure
    ports:
      - '8085:8085'
    networks:
      - my_network
```

Listing 5.2: Przykładowa definicja kontenera mikroserwisu w pliku docker-compose.yml

Przy okazji konfiguracji w Docker Compose pojawił się problem w postaci bardzo długiego budowania obrazów serwisów eCSB (ponad 6 minut), dlatego, uwzględniając fakt, że może się zmienić jedynie ich konfiguracja, założyliśmy repozytorium na DockerHub, gdzie wrzuciliśmy wszystkie obrazy eCSB, aby były dostępne do pobrania dla wszystkich członków zespołu. Przy okazji odchudziliśmy bazowe obrazy (openjdk:11 -> openjdk:11-jre-slim).



Rysunek 5.1: Repozytorium obrazów na DockerHub

5.2. Konfiguracja w Kubernetes

Mając z grubsza zdefiniowane kontenery składające się na architekturę, następnym krokiem było opakowanie ich w Kubernetesowe serwisy oraz konfiguracja zmiennych środowiskowych.

W celu przeniesienia konfiguracji do Kubernetes należało stworzyć odpowiednie pliki YAML definiujące zasoby takie jak ConfigMap, Pod, Deployment, Service oraz PersistentVolumeClaim. Przykładowe pliki konfiguracyjne:

5.2.1. Pod dla mikroserwisu

```
apiVersion: v1
kind: Pod
metadata:
  name: chat
  labels:
    app.kubernetes.io/name: chat
spec:
  containers:
    - image: michwoj01/ecsb:chat
      name: chat
      ports:
        - containerPort: 2138
  restartPolicy: OnFailure
```

Listing 5.3: Pod dla mikroserwisu

5.2.2. Deployment dla PostgreSQL

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: postgres-operator
  namespace: default
spec:
  replicas: 1
  selector:
    matchLabels:
      app.kubernetes.io/name: postgres-operator
  template:
    metadata:
      labels:
        app.kubernetes.io/name: postgres-operator
    spec:
      serviceAccountName: postgres-operator-sa
      containers:
        - name: operator
          image: matipl01/ecsb-postgres-operator:v1.0.0
          ports:
            - containerPort: 5432
          args:
            - "--zap-log-level=debug"
            - "--zap-stacktrace-level=info"
            - "--zap-encoder=console"
```

Listing 5.4: Deployment dla PostgreSQL

5.2.3. PersistentVolumeClaim dla PostgreSQL

```yaml
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: postgres-data-pvc
  namespace: default
  labels:
    app.kubernetes.io/name: postgres
spec:
  storageClassName: local-storage
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 1Gi
```

Listing 5.5: PersistentVolumeClaim dla PostgreSQL

5.2.4. Konfiguracja serwisów

Aplikacja frontendowa oraz baza danych wystawione są na zewnątrz klastra poprzez serwisy typu NodePort, dzięki czemu dostępne są lokalnie:

```yaml
apiVersion: v1
kind: Service
metadata:
  name: webapp
spec:
  type: NodePort
  ports:
    - protocol: TCP
      port: 5173
      targetPort: 5173
      nodePort: 30007
  selector:
    app.kubernetes.io/name: webapp
```

Listing 5.6: Konfiguracja serwisu webapp

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: webapp
  namespace: default
  labels:
    app.kubernetes.io/name: webapp
spec:
  replicas: 1
  selector:
    matchLabels:
      app.kubernetes.io/name: webapp
  template:
    metadata:
      labels:
        app.kubernetes.io/name: webapp
    spec:
      containers:
        - image: michwoj01/ecsb:webapp
          name: webapp
          ports:
            - containerPort: 5173
          env:
            - name: "VITE_ECSB_MOVEMENT_WS_API_URL"
              value: "ws://ecsb.agh.edu.pl/moving"
            - name: "VITE_ECSB_CHAT_WS_API_URL"
              value: "ws://ecsb.agh.edu.pl/chat"
            - name: "VITE_ECSB_LOBBY_WS_API_URL"
              value: "ws://ecsb.agh.edu.pl/chat/landing"
            - name: "VITE_ECSB_HTTP_AUTH_AND_MENAGEMENT_API_URL"
              value: "http://ecsb.agh.edu.pl/api"
            - name: "VITE_ECSB_HTTP_SELF_INTERACTIONS_API_URL"
              value: "http://ecsb.agh.edu.pl/chat"
      restartPolicy: Always
```

Listing 5.7: Konfiguracja podów webapp

5.2.5. Konfiguracja Ingress

Moduły backendowe wystawione są poprzez Ingress, do którego żądania przesyła aplikacja frontendowa. Warto zaznaczyć, że ścieżki do serwisów określone w poniższym pliku muszą odpowiadać zmiennym środowiskowym wstrzykiwanym do podów webapp.

```
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: backend
  annotations:
    nginx.ingress.kubernetes.io/use-regex: "true"
    nginx.ingress.kubernetes.io/enable-cors: "true"
    nginx.ingress.kubernetes.io/cors-allow-origin: "*"
    nginx.ingress.kubernetes.io/rewrite-target: /$2
spec:
  ingressClassName: nginx
  rules:
  - host: ecsb.agh.edu.pl
    http:
      paths:
      - path: /api(/|$)(.*)
        pathType: ImplementationSpecific
        backend:
          service:
            name: game-init
            port:
              number: 2136
      - path: /moving(/|$)(.*)
        pathType: ImplementationSpecific
        backend:
          service:
            name: moving
            port:
              number: 8085
      - path: /chat(/|$)(.*)
        pathType: ImplementationSpecific
        backend:
          service:
            name: chat
            port:
              number: 2138
```

Listing 5.8: Konfiguracja Ingress serwisów backendowych

Konfiguracja w Kubernetes pozwala na zarządzanie mikroserwisami w bardziej skalowalny i elastyczny sposób, umożliwiając dynamiczne skalowanie, zarządzanie stanem aplikacji oraz łatwiejszą integrację z innymi komponentami systemu. Wszystkie pliki konfiguracyjne YAML powinny być umieszczone w odpowiednim katalogu (np. `kubernetes`), aby można było łatwo zastosować je w klastrze Kubernetes.

5.3. Operator Framework dla middleware

Mając środowisko minikube i aplikację w Kubernetes, dla odpowiednich serwisów (PostgreSQL, Redis, RabbitMQ) wprowadzeni zostali operatorzy utworzeni przy użyciu Operator Framework bazującym na Go.

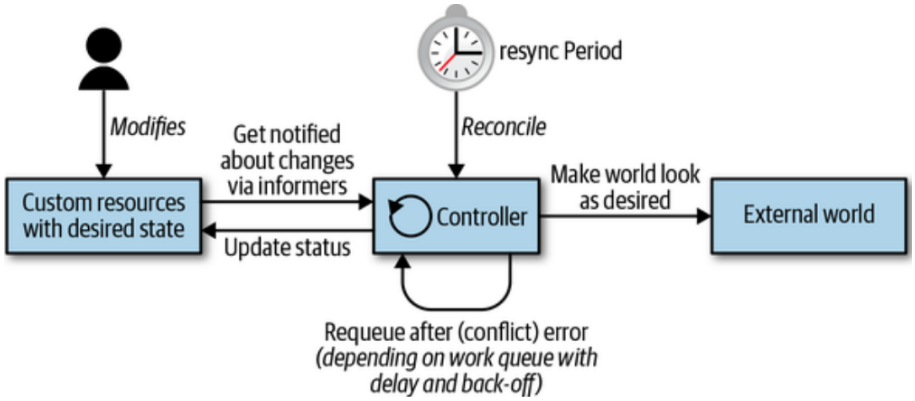
1. **Utworzenie repozytorium Operatora**: `operator-sdk` automatycznie generuje projekt z bazowymi plikami do modyfikacji

```
$ operator-sdk init
```

2. **Stworzenie API oraz kontrolera**: również jest generowany bazowy kod dla resource API oraz kontrolera, np. dla PostgreSQL są to - `postgres_types.go` oraz `postgres_controller.go`, które następnie są dostosowywane do pożądanej funkcjonalności.

```
$ operator-sdk create api --version=v1
```

Controller (kontroler) jest częścią Kubernetesa gdzie znajduje się logika Operatora. Za każdym razem gdy pojawia się jakaś zmiana na obserwowanym zasobie, funkcja `Reconcile` ma doprowadzić, aby aktualny stan CR (CustomResource) odpowiadał pożądanemu stanowi systemu. Dodatkowo, dzięki funkcji `SetupWithManager()` zarządzamy zasobami pod kontrolerem oraz konfiguracjami.



Rysunek 5.2: Schemat działania kontrolera[3]

```
1    func (r *PostgresReconciler) Reconcile(ctx context.Context, req
   ctrl.Request) (ctrl.Result, error) {
2      // Create a logger with context specific to this reconcile loop
3      logger := r.Log.WithValues("namespace", req.Namespace,
   "postgres", req.Name)
4      logger.Info("Reconciling Postgres instance")
5
6      // Fetch the Postgres instance
7      logger.Info("Fetching Postgres instance")
8      postgres := &databasev1.Postgres{}
9      err := r.Get(ctx, req.NamespacedName, postgres)
10     if err != nil {
11       if errors.IsNotFound(err) {
12         logger.Info("Postgres resource not found. Ignoring since
   object must be deleted")
```

```
13          return ctrl.Result{}, nil
14        }
15        logger.Error(err, "Failed to get Postgres")
16        return ctrl.Result{}, err
17      }
18
19      // Ensure PVCs exist
20      if err := r.ensurePVC(ctx, postgres.Spec.DataPvcName,
    postgres); err != nil {
21        logger.Error(err, "Failed to ensure data PVC")
22        return ctrl.Result{}, err
23      }
24
25      // Check if the Pod already exists
26      logger.Info("Checking if the Pod already exists")
27      found := &corev1.Pod{}
28      err = r.Get(ctx, types.NamespacedName{Name: postgres.Name,
    Namespace: postgres.Namespace}, found)
29      if err != nil && errors.IsNotFound(err) {
30        logger.Info("Pod does not exist, will create a new one")
31        // Define a new Pod object
32        logger.Info("Defining a new Pod object")
33        pod := r.newPodForCR(postgres)
34
35        // Set Postgres instance as the owner and controller
36        if err := controllerutil.SetControllerReference(postgres,
    pod, r.Scheme); err != nil {
37          logger.Error(err, "Failed to set controller reference")
38          return ctrl.Result{}, err
39        }
40
41        err = r.Create(ctx, pod)
42        if err != nil {
43          logger.Error(err, "Failed to create new Pod",
    "Pod.Namespace", pod.Namespace, "Pod.Name", pod.Name)
44          return ctrl.Result{}, err
45        }
46        // Pod created successfully - return and requeue
47        logger.Info("Pod created successfully")
48        return ctrl.Result{Requeue: true}, nil
49      } else if err != nil {
50        logger.Error(err, "Failed to get Pod")
51        return ctrl.Result{}, err
52      } else {
53        // If the Pod exists and is not managed by this operator,
    delete it
54        if !metav1.IsControlledBy(found, postgres) {
55          logger.Info("Found existing Pod not managed by this
    operator, deleting it", "Pod.Namespace", found.Namespace,
    "Pod.Name", found.Name)
56          err = r.Delete(ctx, found)
57          if err != nil {
58            logger.Error(err, "Failed to delete existing Pod",
    "Pod.Namespace", found.Namespace, "Pod.Name", found.Name)
59            return ctrl.Result{}, err
60          }
61          logger.Info("Deleted existing Pod", "Pod.Namespace",
    found.Namespace, "Pod.Name", found.Name)
```

```
62            return ctrl.Result{Requeue: true}, nil
63          }
64          logger.Info("Pod already exists and is managed by this
    operator", "Pod.Namespace", found.Namespace, "Pod.Name",
    found.Name)
65        }
66
67        // Update the Postgres status with the pod names
68        logger.Info("Updating Postgres status with the pod names")
69        podNames := []string{found.Name}
70        if !reflect.DeepEqual(podNames, postgres.Status.Nodes) {
71          postgres.Status.Nodes = podNames
72          err := r.Status().Update(ctx, postgres)
73          if err != nil {
74            logger.Error(err, "Failed to update Postgres status")
75            return ctrl.Result{}, err
76          }
77          logger.Info("Postgres status updated", "Status.Nodes",
    postgres.Status.Nodes)
78        }
79
80        return ctrl.Result{}, nil
81      }
```

Listing 5.9: fragment kontrolera dla PostgreSQL

3. **Utworzenie CRD**: wygenerowanie CustomResourceDefinition, który Operator ma obserwować oraz nim zarządzać.

```
$ make manifests
```

```
---
apiVersion: apiextensions.k8s.io/v1
kind: CustomResourceDefinition
metadata:
  annotations:
    controller-gen.kubebuilder.io/version: v0.13.0
  name: postgres.database.pl.edu.agh
spec:
  group: database.pl.edu.agh
  names:
    kind: Postgres
    listKind: PostgresList
    plural: postgres
    singular: postgres
  scope: Namespaced
  versions:
  - name: v1
    schema:
      openAPIV3Schema:
        description: Postgres is the Schema for the postgres API
        properties:
          apiVersion:
            description: 'APIVersion defines the versioned schema of
    this representation
              of an object. Servers should convert recognized schemas
    to the latest
              internal value, and may reject unrecognized values.
    More info:
    https://git.k8s.io/community/contributors/devel/sig-architecture
              /api-conventions.md#resources'
            type: string
          kind:
```

Listing 5.10: Fragment CRD dla PostgreSQL

4. **Uruchomienie Operatora**: lokalnie, przy podłączeniu do klastra Kubernetes

```
$ make install run
```

Po utworzeniu odpowiednich Operatorów oraz ich wdrożeniu na klastrze minikube można obserwować ich działanie dla naszego scenariusza 3.2.

5.4. Napotkane problemy

Jednymi z najistotniejszych elementów konfiguracji poza zwykłym zdefiniowaniem plików YAMLowych okazały się:

- **wystawienie aplikacji na zewnątrz klastra Kubernetesowego** — okazało się, że zastosowanie komponentu Ingress zarówno dla modułów backendowych jak i aplikacji frontendowej było błędem (opcja rewrite-target uniemożliwiała poprawny odbiór plików aplikacji webapp).

- **dostęp do klastra Minikube** - klaster Minikube uruchomiony jako kontener na Docker Desktop skutkował jego niedostępnością z lokalnego hosta. Z tego względu zmieniliśmy środowisko na Kubernetesa zapewnianego przez Docker Desktop, który wszystkie seriwsy typu NodePort wystawia na localhost.

- **skrypty bazodanowe** — Postgres oprócz tworzenia i obsługi instancji wymagał dostępu do skryptów bazodanowych, które tworzyły wszystkie potrzebne tabele. Oprócz zamontowania wolumenów w klastrze Minikube potrzebne było również przekopiowanie samych skryptów do środka klastra, aby pody mogły z nich korzystać przy inicjalizacji. Niestety okazało się, że ten sposób przekazania skryptów nie może być zautomatyzowany, stąd skorzystaliśmy z przekształcenia skryptów do ConfigMap i przekazania ich jako parametry startowe podów Postgresa.

- **dane Postgresa i Redisa** — persystencja na lokalnym klastrze nie jest najlepszym rozwiązaniem (jeśli Minikube odmówi posłuszeństwa, dane przepadną), niemniej pojedyncze pody bywają jeszcze bardziej zawodne. Zamontowanie wolumenów zarówno dla danych statycznych (Postgres) jak i dynamicznych (Redis) było oczywistością.

- **konfiguracja RabbitMQ** — okazało się, że przekazanie plików konfiguracyjnych poprzez zamontowanie ConfigMap spowodowało błędy poda (oczywiście nigdzie nielogowane), które uniemożliwiały podłączenie się serwisów. Konieczne stało się w tym przypadku zastosowanie Operator Framework, aby inicjalizować pody RabbitMQ z odpowiednimi pluginami.

- **zasoby potrzebne do stworzenia gry** — jeden z modułów Chłopskiej Szkoły Biznesu wymagał zapisywania oraz odczytywania plików .jpg, będących elementami mapy lub grafikami postaci. Również w tym przypadku konieczne było zastosowanie persystentnego wolumenu wewnątrz klastra.

Rozdział 6

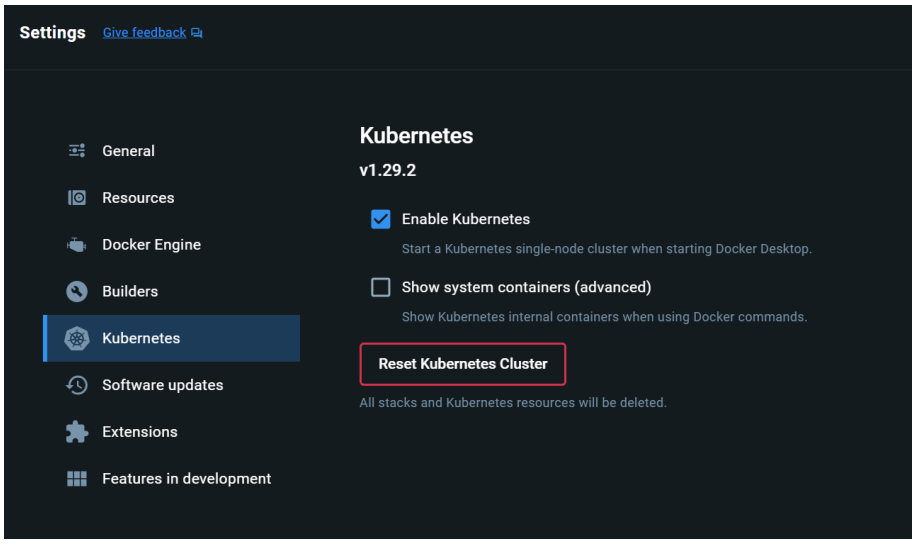
Sposób instalacji

6.1. Podejście z Docker Desktop

1. Należy przejść na stronę Docker Desktop i pobrać odpowiednią wersję dla systemu operacyjnego (Windows, macOS, Linux).

2. Następnie należy postępować zgodnie z opisanymi krokami instalacji dla używanego systemu operacyjnego.

3. Po uruchomieniu przechodzimy do ustawień i włączamy Kubernetesa:



Rysunek 6.1: Włączenie klastra Kubernetes w Docker Desktop

4. Odczekujemy, aż klaster pobierze wszystkie potrzebne obrazy oraz uruchomi wewnętrzne serwisy Kubernetesa i instalujemy kontroler Ingress w klastrze:

```
kubectl apply -f
   https://raw.githubusercontent.com/kubernetes/
ingress-nginx/controller-v1.10.1/deploy/static/
provider/cloud/deploy.yaml
```

6.2. Podejście z Minikube (najlepiej Linux/MacOS)

1. Instalujemy wybrany menedżer kontenerów taki jak Docker, Hyper-V lub VirtualBox.
2. Należy przejść na stronę Minikube i pobrać odpowiednią wersję dla systemu operacyjnego, a następnie postępować zgodnie z krokami instalacji opisanymi w na załączonej stronie internetowej.
3. Uruchomiamy Minikube:

```
minikube start
```

4. Włączamy plugin odpowiedzialny za kontrolę Ingress:

```
minikube addons enable ingress
```

5. Weryfikujemy, czy nasz system poprawnie wyszukuje klaster Minikube poprzez wykonanie następującej komendy:

```
ssh docker@$(minikube ip)
```

Jeśli komenda zakończy się nieznalezieniem hosta, oznacza to, że klaster Minikube jest niewidoczny, w związku z czym wystawiona aplikacja frontendowa nie będzie mogła wykonać żądań do modułów backendowych. W tym przypadku należy zastosować podejście z Docker Desktop.

6.3. Uruchomienie aplikacji na Kubernetes

1. **Pobranie repozytorium z rozwiązaniem**:

```
git clone
  git@github.com:michwoj01/EOSI-Operator-Framework.git
cd EOSI-Operator-Framework
```

2. **Zastosowanie wszystkich plików konfiguracyjnych Kubernetes oraz Custom Resource Definition**:

```
cd kubernetes && ./deploy.sh
```

Jeżeli napotkano błąd `error looking up service account default/default: serviceaccount "default"not found`, należy ponownie uruchomić powyższą komendę.

3. **Dodanie wpisu do /etc/hosts** Do pliku /etc/hosts dodajemy wpis:

```
127.0.0.1 ecsb.agh.edu.pl
```

4. **Sprawdzenie statusu stworzonych elementów**:

```
kubectl get all
```

6.4. Debugowanie

W przypadku problemów, sprawdzenie statusu lub logów podów, ingressa lub serwisów:

```
kubectl describe <pod-name>
kubectl logs <pod-name>
kubectl get ingress backend -o wide
kubectl get svc -o wide
```

Rozdział 7

Odtworzenie rozwiązania

W tym rozdziale opisany jest proces odtworzenia całego rozwiązania krok po kroku, z wykorzystaniem podejścia Infrastructure as Code (IaC). Podejście to znacząco ułatwia proces wdrażania infrastruktury, poprzez jego automatyzację i ogranicza się do wykonania kilku komend, które, na podstawie zdefiniowanych plików konfiguracyjnych, wykonają niezbędną pracę za nas.

7.1. Infrastructure as Code (IaC) - Podejście

Infrastructure as Code (IaC) to praktyka zarządzania infrastrukturą IT przy użyciu kodu, co pozwala na automatyzację procesów wdrożeniowych oraz łatwe zarządzanie środowiskiem. W naszym projekcie wykorzystujemy Kubernetes do zarządzania kontenerami oraz Docker do budowy obrazów kontenerów. W celu zapewnienia infrastruktury dla naszego rozwiązania początkowo korzystaliśmy z lokalnego klastra Minikube, a następnie z Kubernetesa zapewnianego przez Docker Desktop w związku z czym nie było potrzeby korzystać z narzędzi typu Terraform.

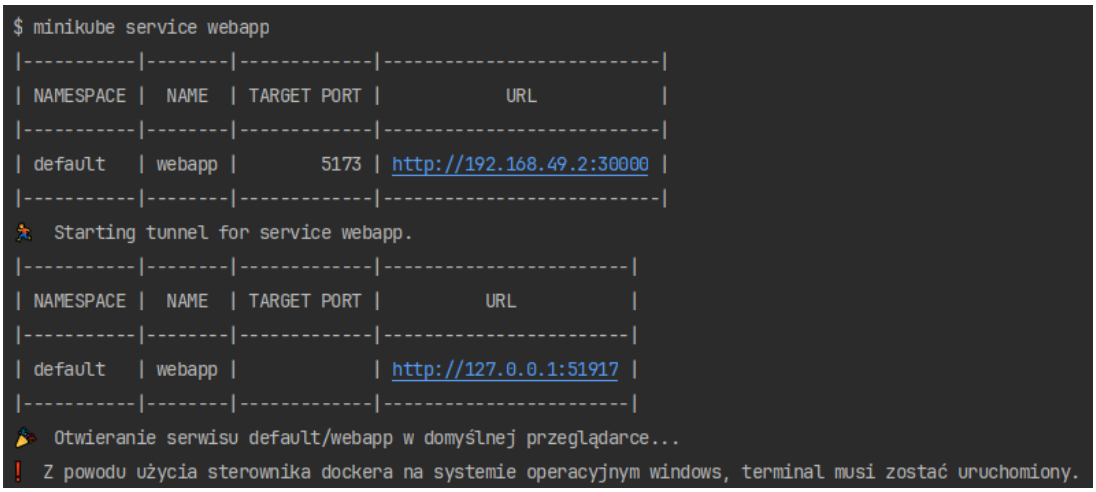
7.2. Kroki odtworzenia rozwiązania

1. **Instalacja narzędzi**: Na wstępie należy zainstalować niezbędne narzędzia, takie jak Docker Desktop lub Minikube zgodnie z instrukcjami podanymi w rozdziale dotyczącym instalacji, a także pobrać repozytorium i uruchomić konfigurację projektu. Dokładniej opisane jest to w rozdziale 6.

2. **Dostęp do aplikacji webowej**: Po poprawnym uruchomieniu wszystkich usług, aplikacja webowa powinna być dostępna. Minikube dostarcza polecenie, które pozwala na dostęp do usług zewnętrznych poprzez otwarcie odpowiednich portów:

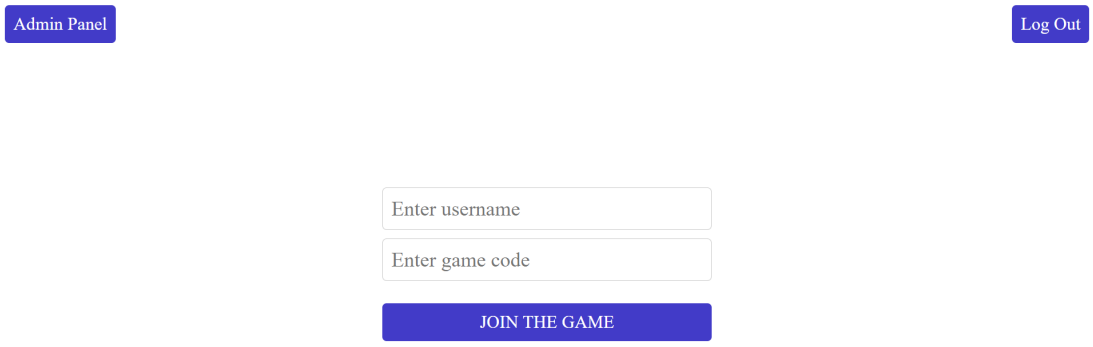
   ```
   minikube service webapp
   ```

   Następnie powinien otworzyć się tunel między klastrem Minikube a naszym lokalnym hostem i serwis aplikacji webowej powinien być dostępny lokalnie pod wyszczególnionym portem:



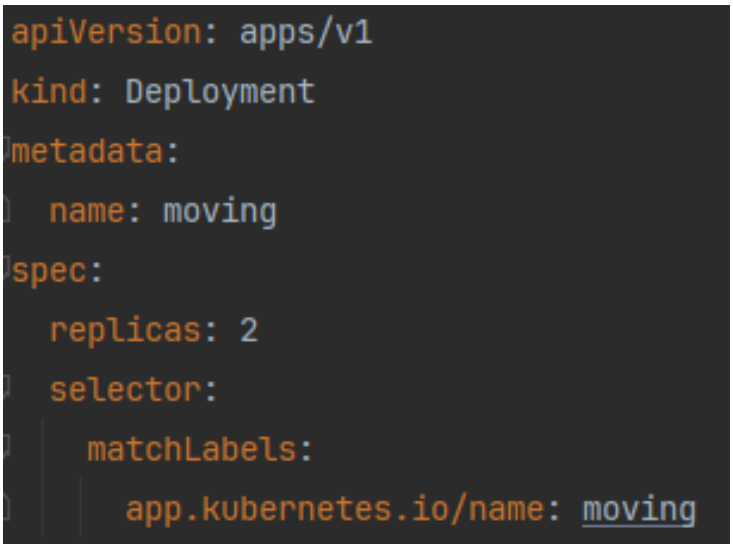
Rysunek 7.1: Otworzenie tunelu między Minikube a naszym hostem

Jeśli natomiast użyliśmy podejścia z Docker Desktop, to nasza aplikacja dostępna jest pod adresem . lokalnie pod wyszczególnionym portem:



Rysunek 7.2: Aplikacja frontendowa po zalogowaniu

3. **Zmiana liczebności skalowalnych serwisów**: Każdy serwis eCSB, który dzięki zastosowaniu shardingu jest możliwy do skalowania, możemy kontrolować za pomocą deklaracji Deploymentów. Dla przykładu możemy zwiększyć liczbę podów serwisu moving zmieniając parametr ilości replik:



Rysunek 7.3: Zmiana liczby podów serwisu moving

Następnie (zakładając, że jesteśmy w folderze root naszego projektu) możemy zastosować nasze zmiany w Kubernetesie:

```
kubectl apply -f kubernetes/deployments/moving.yaml
```

Rozdział 8

Podsumowanie

Operator Framework to przydatne narzędzie do zautomatyzowanego zarządzania aplikacjami Kubernetesowymi. Pozwala m.in. na zarządzanie bazą danych w razie ewentualnej awarii systemu.

Operator Framework umożliwia zautomatyzowane zarządzanie cyklem życia tych komponentów, w tym instalację, skalowanie, konfigurację, oraz monitorowanie. Utworzeni Operatorzy zapewniają wysoką dostępność i automatyczną naprawę ewentualnych awarii.

Dzięki zastosowaniu Operator Framework, nasza aplikacja była w stanie działać bardziej efektywnie i niezawodnie, a proces zarządzania środowiskiem Kubernetes stał się prostszy i mniej podatny na błędy. Projekt pokazał, że Operator Framework jest silnym narzędziem do zarządzania wielokomponentowymi aplikacjami w chmurze.

Spis rysunków

Spis listingów