TPJAD-HW1: Aplicație Containerizată cu Tomcat, WildFly și Jetty

- **Student:** David Simonel-Olimpiu
- **Grupa:** 244

Introducere

Scopul acestui proiect este să demonstreze utilizarea tehnologiilor moderne precum **Docker** și **Kubernetes** pentru a dezvolta, containeriza și orchestra o aplicație distribuită bazată pe trei servere de aplicații: **Jetty**, **Tomcat** și **WildFly**. Acest sistem modular implementează un flux în care **Jetty** este principalul server ce inițiază cereri către **Tomcat**, care acționează ca un intermediar pentru a primi date de la **WildFly**.

Arhitectura generală

Aplicația este organizată în trei module distincte, fiecare reprezentând un server containerizat, care colaborează pentru a procesa cererile utilizatorului:

1. **Jetty**: Punctul de intrare principal al aplicației. Primește cererile utilizatorului și apelează **Tomcat** pentru a obține răspunsuri.
2. **Tomcat**: Acționează ca intermediar. Primește cereri de la **Jetty**, trimite cereri către **WildFly** și returnează răspunsul.
3. **WildFly**: Scrie un mesaj static.

Fluxul principal:
Client → Jetty → Tomcat → WildFly

Implementare și tehnologii

Containerizare cu Docker

Docker este folosit pentru a crea containere izolate pentru fiecare server. Fiecare modul al aplicației are un fișier `Dockerfile`, unde se specifică:

- Imaginea de bază corespunzătoare serverului (e.g., `jetty:11.0`, `tomcat:9.0`, `wildfly:26.1`).

- Locația fișierului `.war` generat de Maven și configurarea sa în directorul potrivit pentru server.
- Setările necesare pentru porturi și alte configurații.

Fișierele `Dockerfile` pot fi găsite în directorul `Docker/` .

Orchestrare cu Kubernetes

Kubernetes este utilizat pentru a orchestra containerele și a asigura conectivitatea dintre ele. Fiecare modul are definit:

1. **Deployment**: Asigură lansarea și gestionarea podurilor pentru fiecare server de aplicații. Aceasta include specificarea imaginii Docker și a numărului de replici.
2. **Service**: Expune fiecare modul în rețeaua internă a Kubernetes, folosind DNS pentru identificare și porturi pentru comunicare.

De exemplu:

- **Jetty** este configurat să ruleze pe portul 8080 și să fie accesibil la adresa `jetty-service.tpjad-hw1.svc.cluster.local` .
- **Tomcat** este configurat să ruleze pe portul 8081 și să fie accesibil la adresa `tomcat-service.tpjad-hw1.svc.cluster.local` .
- **WildFly** rulează pe portul 8082 și este accesibil la adresa `wildfly-service.tpjad-hw1.svc.cluster.local` .

Fișierele de configurare Kubernetes sunt disponibile în directorul `kubernetes/` .

Codul aplicației

Jetty

Jetty este responsabil pentru inițierea cererilor în cadrul aplicației. Servletul său principal trimite o cerere către **Tomcat**, care, la rândul său, apelează **WildFly**. Codul servletului este disponibil în `src/main/java/com/example/jetty/JettyServlet.java` .

Prototipul metodei servletului:

- `protected void doGet(HttpServletRequest req, HttpServletResponse resp)`
  - Parametri:
    - `req` - Obiect care conține detalii despre cererea HTTP primită.
    - `resp` - Obiect care permite generarea răspunsului HTTP.
  - Return: Void.
  - Excepții:
    - `ServletException` - Eroare în procesarea cererii.

- `IOException` - Eroare de scriere a răspunsului.

În servlet, clasa utilitară `HttpRequestHelper` este folosită pentru a trimite cereri HTTP către **Tomcat**. Aceasta este implementată în `src/main/java/com/example/jetty/HttpRequestHelper.java`.

Exemple de răspunsuri:

- Cerere reușită:
  Utilizatorul primește un mesaj compus care arată colaborarea între module:

```
Jetty received a message from Tomcat:
And Tomcat received a message from WildFly: Hello, all! This is WildFly :)
```

Tomcat

Tomcat servește ca un intermediar între **Jetty** și **WildFly**. Servletul său primește cereri de la **Jetty**, comunică cu **WildFly** și returnează rezultatul.

Exemple de utilizare:

- **Cerere validă:**
  Tomcat primește răspunsul de la **WildFly** și îl trimite înapoi către **Jetty**:

```
And Tomcat received a message from WildFly:
Hello, all! This is WildFly :)
```

WildFly

WildFly este ultimul punct din lanțul de cereri. Răspunsul său este static și reprezintă rezultatul final procesat de aplicație.

Exemplu de răspuns:

```
Hello, all! This is WildFly :)
```

Procesul de implementare

Construirea imaginilor

Imaginile Docker sunt generate folosind scripturi dedicate care:

1. Construiesc proiectul folosind Maven.
2. Copiază fișierele `.war` în locația specificată de Dockerfile pentru fiecare server.
3. Etichetează imaginile pentru a putea fi utilizate în Kubernetes.

Lansarea aplicației

După construirea imaginilor, Kubernetes este utilizat pentru a lansa modulele într-un mediu orchestrat. Resursele Kubernetes sunt definite astfel încât modulele să poată comunica între ele folosind DNS intern.

Testare și validare

Testarea aplicației este automatizată printr-un script (`test.sh`) care verifică:

1. Starea podurilor Kubernetes.
2. Funcționarea fluxului complet al aplicației:
   - Jetty → Tomcat → WildFly.

Exemple de utilizare:

- Răspuns valid:
  Utilizatorul primește răspunsul final compus de la toate modulele.
- Eroare:
  Dacă unul dintre module nu este disponibil, scriptul va afișa mesajele corespunzătoare.

Îmbunătățiri propuse

1. **Tratamentul erorilor:** Deși aplicația gestionează erorile principale, se pot adăuga mesaje mai detaliate și logare centralizată pentru a facilita depanarea.
2. **Expunere publică limitată:** Jetty ar trebui expus permanent pe un ip public. Această schimbare este posibilă folosind un Ingress care mapează cererile externe direct la serviciul Jetty. Aceasta ar crește securitatea și ar limita suprafața de atac.
3. **Scalabilitate:** Modulele pot fi scalate orizontal, adăugând replici pentru a gestiona un număr mai mare de cereri.

Concluzie

Acest proiect demonstrează cum tehnologiile **Docker** și **Kubernetes** pot fi utilizate pentru a crea o arhitectură distribuită, modulară și scalabilă. Integrarea eficientă între modulele Jetty, Tomcat și

WildFly asigură un flux clar și extensibil. Prin utilizarea unui design bine definit, soluția este ușor de înțeles, implementat și extins pentru cerințe viitoare.