**Documentația Proiectului: Blog de Călătorii cu Chat și IA**

**Structura Generală a Proiectului**

Proiectul constă în trei componente principale care rulează în containere Docker gestionate de Kubernetes:

1. **CMS (Content Management System)** - PayloadCMS pentru gestionarea conținutului blog-ului
2. **Chat** - Sistem de chat în timp real folosind WebSocket
3. **Aplicație IA** - Pentru procesarea imaginilor (va fi implementată ulterior)

Fiecare componentă are propria bază de date și servicii asociate.

**Componente Implementate Până Acum**

**1. Infrastructura MongoDB și PayloadCMS**

**1.1 MongoDB Deployment (mongodb-deployment.yaml)**

Acest fișier YAML definește resursele necesare pentru MongoDB:

yaml

apiVersion: apps/v1

kind: Deployment

metadata:

name: mongodb

labels:

app: mongodb

spec:

replicas: 1

selector:

matchLabels:

app: mongodb

template:

metadata:

labels:

app: mongodb

spec:

containers:

- name: mongodb

image: mongo:latest

ports:

- containerPort: 27017

volumeMounts:

- name: mongodb-data

mountPath: /data/db

volumes:

- name: mongodb-data

persistentVolumeClaim:

claimName: mongodb-pvc

---

*# Service definition and PVC follow...*

**Explicații detaliate:**

* apiVersion: apps/v1 - Specifică versiunea API Kubernetes pentru resursa Deployment
* kind: Deployment - Definește tipul resursei (un Deployment)
* metadata - Conține informații despre deployment (nume, etichete)
* spec.replicas: 1 - Specifică faptul că dorim o singură instanță a MongoDB
* spec.selector.matchLabels - Definește etichetele folosite pentru a identifica pod-urile
* spec.template - Șablonul pentru pod-uri:
  + metadata.labels - Etichetele atașate pod-urilor (trebuie să corespundă cu matchLabels)
  + spec.containers - Lista containerelor din pod:
    - name - Numele containerului
    - image - Imaginea Docker folosită (imaginea oficială MongoDB)
    - ports - Porturile expuse de container (27017 este portul standard MongoDB)
    - volumeMounts - Definește montarea volumului în container pentru persistența datelor
  + spec.volumes - Volumele atașate pod-ului (folosind un PersistentVolumeClaim)

Service-ul și PVC-ul urmează în același fișier, separate prin ---.

**1.2 PayloadCMS Deployment (payload-deployment.yaml)**

Acest fișier configurează PayloadCMS pentru blog-ul de călătorii:

yaml

apiVersion: apps/v1

kind: Deployment

metadata:

name: payload-cms

labels:

app: payload-cms

spec:

replicas: 5 *# Conform cerinței*

selector:

matchLabels:

app: payload-cms

template:

metadata:

labels:

app: payload-cms

spec:

containers:

- name: payload-cms

image: payloadcms/payload:latest

ports:

- containerPort: 3000

env:

- name: MONGODB\_URI

value: "mongodb://mongodb-service:27017/payload-cms-db"

- name: PAYLOAD\_SECRET

value: "random-secret-key-for-payload"

- name: PAYLOAD\_CONFIG\_PATH

value: "/payload/config/payload.config.js"

volumeMounts:

- name: config-volume

mountPath: /payload/config

volumes:

- name: config-volume

configMap:

name: payload-config

**Explicații detaliate:**

* replicas: 5 - Specifică 5 replici (pod-uri) pentru PayloadCMS, conform cerinței
* containers.env - Definește variabilele de mediu pentru container:
  + MONGODB\_URI - URI-ul de conexiune la MongoDB, folosind numele service-ului (mongodb-service)
  + PAYLOAD\_SECRET - Cheie secretă pentru criptare și securitate
  + PAYLOAD\_CONFIG\_PATH - Calea către fișierul de configurare
* volumeMounts și volumes - Montează un ConfigMap ca volum pentru configurare

ConfigMap-ul include o structură detaliată pentru blog-ul de călătorii:

yaml

apiVersion: v1

kind: ConfigMap

metadata:

name: payload-config

data:

"payload.config.js": |

module.exports = {

collections: [

{

slug: 'posts',

labels: {

singular: 'Post',

plural: 'Posts',

},

admin: {

useAsTitle: 'title',

},

fields: [

{

name: 'title',

type: 'text',

required: true,

},

/\* Additional fields... \*/

],

},

/\* Destinations collection... \*/

],

};

**Explicații detaliate:**

* ConfigMap-ul definește configurația PayloadCMS în JavaScript
* collections - Array-ul de colecții:
  + posts - Pentru articolele blog-ului, cu câmpuri pentru:
    - title - Titlul articolului (text)
    - content - Conținutul articolului (rich text)
    - destination - Destinația articolului (text)
    - publishedDate - Data publicării (date)
  + destinations - Pentru destinațiile de călătorie, cu câmpuri pentru:
    - name - Numele destinației
    - description - Descrierea destinației
    - continent - Continentul (selectabil din opțiuni predefinite)

**2. Backend-ul pentru Chat**

**2.1 Fișierul principal al aplicației (ChatBackendApplication.java)**

java

package com.travelblog.chat\_backend;

import org.springframework.boot.SpringApplication;

import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication

public class ChatBackendApplication {

public static void main(String[] args) {

SpringApplication.run(ChatBackendApplication.class, args);

}

}

**Explicații detaliate:**

* package com.travelblog.chat\_backend - Definește pachetul în care se află clasa
* import - Importă clasele necesare din framework-ul Spring
* @SpringBootApplication - O meta-adnotare care combină:
  + @Configuration - Marchează clasa ca sursă de definiții de bean-uri
  + @EnableAutoConfiguration - Activează auto-configurarea Spring Boot
  + @ComponentScan - Permite detectarea automată a componentelor
* main - Metoda principală care pornește aplicația:
  + SpringApplication.run - Creează și configurează aplicația Spring

**2.2 Modelul pentru mesaje (ChatMessage.java)**

java

package com.travelblog.chat\_backend.model;

import lombok.AllArgsConstructor;

import lombok.Data;

import lombok.NoArgsConstructor;

import org.springframework.data.annotation.Id;

import org.springframework.data.mongodb.core.mapping.Document;

import java.time.LocalDateTime;

@Data

@NoArgsConstructor

@AllArgsConstructor

@Document(collection = "messages")

public class ChatMessage {

@Id

private String id;

private String username;

private String content;

private LocalDateTime timestamp;

public ChatMessage(String username, String content) {

this.username = username;

this.content = content;

this.timestamp = LocalDateTime.now();

}

}

**Explicații detaliate:**

* @Data - Adnotare Lombok care generează automat:
  + Getteri pentru toate câmpurile
  + Setteri pentru toate câmpurile non-finale
  + Metode equals(), hashCode() și toString()
  + Constructor implicit
* @NoArgsConstructor - Generează un constructor fără parametri
* @AllArgsConstructor - Generează un constructor cu parametri pentru toate câmpurile
* @Document(collection = "messages") - Specifică că această clasă va fi mapată la colecția "messages" din MongoDB
* @Id - Marchează câmpul ca fiind identificatorul primar (echivalent cu cheia primară din SQL)
* private String id - ID-ul mesajului (generat automat de MongoDB)
* private String username - Numele utilizatorului care trimite mesajul
* private String content - Conținutul textual al mesajului
* private LocalDateTime timestamp - Momentul când a fost creat mesajul
* Constructor cu username și content - Un constructor convenabil care setează automat timestamp-ul la momentul actual

**2.3 Repository pentru mesaje (ChatMessageRepository.java)**

java

package com.travelblog.chat\_backend.repository;

import com.travelblog.chat\_backend.model.ChatMessage;

import org.springframework.data.mongodb.repository.MongoRepository;

import org.springframework.stereotype.Repository;

import java.util.List;

@Repository

public interface ChatMessageRepository extends MongoRepository<ChatMessage, String> {

List<ChatMessage> findAllByOrderByTimestampAsc();

}

**Explicații detaliate:**

* @Repository - Indică Spring că această interfață este un repository (un component responsabil pentru persistența datelor)
* extends MongoRepository<ChatMessage, String> - Extinde interfața generică MongoRepository:
  + ChatMessage - Tipul entității
  + String - Tipul identificatorului (ID)
* MongoRepository furnizează automat metode pentru:
  + Salvarea entităților (save())
  + Ștergerea entităților (delete())
  + Găsirea entităților după ID (findById())
  + Găsirea tuturor entităților (findAll())
* findAllByOrderByTimestampAsc() - O metodă personalizată (derivată) care:
  + Returnează toate mesajele de chat
  + Le ordonează ascendent după timestamp (de la cel mai vechi la cel mai nou)
  + Spring Data generează automat implementarea acestei metode pe baza convenției de denumire

**2.4 Service pentru mesaje (ChatService.java)**

java

package com.travelblog.chat\_backend.service;

import com.travelblog.chat\_backend.model.ChatMessage;

import com.travelblog.chat\_backend.repository.ChatMessageRepository;

import org.springframework.beans.factory.annotation.Autowired;

import org.springframework.stereotype.Service;

import java.util.List;

@Service

public class ChatService {

private final ChatMessageRepository chatMessageRepository;

@Autowired

public ChatService(ChatMessageRepository chatMessageRepository) {

this.chatMessageRepository = chatMessageRepository;

}

public ChatMessage saveMessage(ChatMessage message) {

return chatMessageRepository.save(message);

}

public List<ChatMessage> getAllMessages() {

return chatMessageRepository.findAllByOrderByTimestampAsc();

}

}

**Explicații detaliate:**

* @Service - Adnotare Spring care marchează clasa ca fiind un serviciu (component specializat în logica de business)
* private final ChatMessageRepository chatMessageRepository - O referință la repository-ul de mesaje
  + final indică faptul că referința nu se va schimba după inițializare
* Constructor cu @Autowired - Spring va injecta automat o instanță a repository-ului
  + Aceasta este "constructor injection", o practică recomandată pentru injecția de dependențe
  + Este mai bună decât injecția prin câmp (@Autowired direct pe câmp) pentru că:
    - Face dependențele explicite și obligatorii
    - Face testarea mai ușoară
    - Permite declararea câmpurilor ca final
* saveMessage(ChatMessage message) - Salvează un mesaj în baza de date și îl returnează (cu ID generat)
* getAllMessages() - Returnează toate mesajele în ordine cronologică, folosind metoda definită în repository

**2.5 Configurație WebSocket (WebSocketConfig.java)**

java

package com.travelblog.chat\_backend.config;

import org.springframework.context.annotation.Configuration;

import org.springframework.messaging.simp.config.MessageBrokerRegistry;

import org.springframework.web.socket.config.annotation.EnableWebSocketMessageBroker;

import org.springframework.web.socket.config.annotation.StompEndpointRegistry;

import org.springframework.web.socket.config.annotation.WebSocketMessageBrokerConfigurer;

@Configuration

@EnableWebSocketMessageBroker

public class WebSocketConfig implements WebSocketMessageBrokerConfigurer {

@Override

public void registerStompEndpoints(StompEndpointRegistry registry) {

registry.addEndpoint("/ws")

.setAllowedOriginPatterns("\*")

.withSockJS();

}

@Override

public void configureMessageBroker(MessageBrokerRegistry registry) {

registry.setApplicationDestinationPrefixes("/app");

registry.enableSimpleBroker("/topic");

}

}

**Explicații detaliate:**

* @Configuration - Indică Spring că această clasă conține configurații Bean
* @EnableWebSocketMessageBroker - Activează procesarea mesajelor WebSocket prin broker
* implements WebSocketMessageBrokerConfigurer - Interfață pentru personalizarea configurației WebSocket
* registerStompEndpoints - Definește endpoint-urile STOMP:
  + addEndpoint("/ws") - Creează un endpoint WebSocket la calea "/ws"
  + setAllowedOriginPatterns("\*") - Permite conexiuni de la orice origine (CORS)
    - În producție, ar trebui să restricționezi originile pentru securitate
  + withSockJS() - Adaugă suport pentru SockJS (un fallback pentru browserele care nu suportă WebSocket)
* configureMessageBroker - Configurează broker-ul de mesaje:
  + setApplicationDestinationPrefixes("/app") - Mesajele destinate aplicației vor începe cu prefixul "/app"
    - Aceste mesaje vor fi direcționate către metode adnotate cu @MessageMapping
  + enableSimpleBroker("/topic") - Activează un broker simplu pentru destinații care încep cu "/topic"
    - Aceste destinații sunt folosite pentru a trimite mesaje către clienți

**2.6 Controller pentru mesaje (ChatController.java)**

java

package com.travelblog.chat\_backend.controller;

import com.travelblog.chat\_backend.model.ChatMessage;

import com.travelblog.chat\_backend.service.ChatService;

import org.springframework.beans.factory.annotation.Autowired;

import org.springframework.messaging.handler.annotation.MessageMapping;

import org.springframework.messaging.handler.annotation.SendTo;

import org.springframework.stereotype.Controller;

import org.springframework.web.bind.annotation.GetMapping;

import org.springframework.web.bind.annotation.ResponseBody;

import java.util.List;

@Controller

public class ChatController {

private final ChatService chatService;

@Autowired

public ChatController(ChatService chatService) {

this.chatService = chatService;

}

@MessageMapping("/chat.sendMessage")

@SendTo("/topic/public")

public ChatMessage sendMessage(ChatMessage chatMessage) {

return chatService.saveMessage(chatMessage);

}

@GetMapping("/api/messages")

@ResponseBody

public List<ChatMessage> getMessages() {

return chatService.getAllMessages();

}

}

**Explicații detaliate:**

* @Controller - Adnotare Spring care marchează clasa ca fiind un controller
* Constructor cu injecția serviciului de chat
* @MessageMapping("/chat.sendMessage") - Specifică că această metodă va gestiona mesajele trimise la destinația "/app/chat.sendMessage"
  + WebSocket-ul primește mesajul de la client
  + Destinația completă este "/app" (prefixul aplicației) + "/chat.sendMessage"
* @SendTo("/topic/public") - Specifică că rezultatul returnat va fi trimis la destinația "/topic/public"
  + Toți clienții abonați la această destinație vor primi mesajul
* public ChatMessage sendMessage(ChatMessage chatMessage) - Metodă care:
  + Primește un obiect ChatMessage (deserializat automat din JSON)
  + Salvează mesajul în baza de date
  + Returnează mesajul salvat, care va fi trimis la toți clienții abonați
* @GetMapping("/api/messages") - Specifică că această metodă va gestiona cereri HTTP GET la calea "/api/messages"
* @ResponseBody - Indică că rezultatul metodei va fi serializat direct în corpul răspunsului HTTP
* public List<ChatMessage> getMessages() - Returnează toate mesajele din istoricul chat-ului
  + Este folosit de obicei când un client se conectează prima dată și trebuie să încarce mesajele anterioare

**2.7 Fișier de configurare (application.properties)**

properties

server.port=8080

spring.data.mongodb.host=mongodb-service

spring.data.mongodb.port=27017

spring.data.mongodb.database=chat-db

**Explicații detaliate:**

* server.port=8080 - Specifică portul pe care va rula aplicația Spring Boot
* spring.data.mongodb.host=mongodb-service - Adresa MongoDB (nume service Kubernetes)
* spring.data.mongodb.port=27017 - Portul MongoDB (portul standard)
* spring.data.mongodb.database=chat-db - Numele bazei de date MongoDB

**2.8 Dockerfile pentru backend**

dockerfile

*# Etapa de build*

FROM maven:3.8.4-openjdk-17-slim AS build

WORKDIR /app

COPY pom.xml .

*# Descarcă dependențele*

RUN mvn dependency:go-offline

COPY src ./src

*# Compilează aplicația*

RUN mvn package -DskipTests

*# Etapa de rulare*

FROM openjdk:17-slim

WORKDIR /app

COPY --from=build /app/target/chat-backend-0.0.1-SNAPSHOT.jar /app/chat-backend.jar

EXPOSE 8080

CMD ["java", "-jar", "chat-backend.jar"]

**Explicații detaliate:**

* Prima etapă - Build:
  + FROM maven:3.8.4-openjdk-17-slim AS build - Folosește o imagine Maven cu Java 17
  + WORKDIR /app - Setează directorul de lucru
  + COPY pom.xml . - Copiază doar pom.xml pentru a beneficia de caching
  + RUN mvn dependency:go-offline - Descarcă toate dependențele
    - Acest pas este separat pentru a profita de mecanismul de caching Docker
    - Dacă pom.xml nu se schimbă, acest strat va fi reutilizat
  + COPY src ./src - Copiază codul sursă
  + RUN mvn package -DskipTests - Compilează aplicația fără a rula testele
* A doua etapă - Rulare:
  + FROM openjdk:17-slim - Folosește o imagine minimală cu Java 17
  + COPY --from=build ... - Copiază doar JAR-ul compilat din prima etapă
  + EXPOSE 8080 - Documentează faptul că aplicația va asculta pe portul 8080
  + CMD ["java", "-jar", "chat-backend.jar"] - Comanda care va fi executată la pornire

Această abordare "multi-stage" reduce semnificativ dimensiunea imaginii finale, deoarece:

* Nu include Maven și toate dependențele de build
* Include doar runtime-ul Java și JAR-ul compilat
* Elimină codul sursă și alte artefacte intermediare

**Cum Funcționează**

**Fluxul de date pentru blog**

1. Utilizatorii accesează blog-ul la http://<ip-extern>:80
2. PayloadCMS servește conținutul blog-ului (articole, destinații)
3. Datele sunt citite și scrise în MongoDB

**Fluxul de date pentru chat**

1. Backend-ul Java cu WebSocket gestionează comunicarea în timp real
2. Când un client se conectează:
   * Se stabilește o conexiune WebSocket la /ws
   * Clientul obține istoricul mesajelor prin API REST la /api/messages
   * Clientul se abonează la canalul /topic/public pentru a primi mesaje noi
3. Când un client trimite un mesaj:
   * Mesajul este trimis la /app/chat.sendMessage
   * Backend-ul salvează mesajul în MongoDB
   * Mesajul este trimis la toți clienții abonați la /topic/public

**Comenzi pentru Compilare și Testare**

**Compilarea proiectului Java**

bash

cd chat-backend

mvn clean package

Această comandă:

* Șterge directorul target/ cu artefactele compilate anterior (clean)
* Compilează codul sursă
* Rulează testele
* Creează un JAR executabil în directorul target/

**Construirea imaginii Docker pentru backend**

bash

cd chat-backend

docker build -t chat-backend:test .

Această comandă:

* Construiește o imagine Docker bazată pe Dockerfile-ul din directorul curent
* Etichetează imaginea cu numele chat-backend:test
* . la final indică faptul că contextul de build este directorul curent

**Verificarea imaginii Docker create**

bash

docker images

Aceasta afișează toate imaginile Docker disponibile local, inclusiv chat-backend:test.

**Aplicarea configurațiilor Kubernetes**

bash

kubectl apply -f mongodb-deployment.yaml

kubectl apply -f payload-deployment.yaml

Aceste comenzi:

* Creează resursele definite în fișierele YAML
* Kubernetes va crea deployment-urile, service-urile și alte resurse
* Vor fi create pod-uri care rulează containerele specificate

**Verificarea pod-urilor create**

bash

kubectl get pods

Aceasta afișează toate pod-urile din namespace-ul curent, inclusiv status-ul lor.

**Verificarea serviciilor create**

bash

kubectl get services

Aceasta afișează toate serviciile din namespace-ul curent, inclusiv IP-urile și porturile.

**Următorii Pași**

1. Crearea fișierului YAML pentru deployment-ul backend-ului chat în Kubernetes
2. Implementarea Nginx pentru a expune backend-ul chat pe portul 88
3. Implementarea frontend-ului pentru chat în React
4. Implementarea componentei de IA pentru procesarea imaginilor
5. Crearea unui fișier principal (main.yaml) care să includă toate resursele

Retry

[Claude can make mistakes.   
Please double-check responses.](https://support.anthropic.com/en/articles/8525154-claude-is-providing-incorrect-or-misleading-responses-what-s-going-on)

3.7 Sonnet