MINISTRY OF EDUCATION

TECHNICAL UNIVERSITY
OF CLUJ-NAPOCA, ROMANIA

WEB CRAWLER

Proiect – Proiectarea Aplicațiilor Web & Sisteme distribuite

Program de studii: Calculatoare

Autor: Avram Darius

2025

Cuprins

1 Introducere

1.1 Context general

Aplicația a fost dezvoltată din nevoia de a procesa volume mari de date web într-un mod eficient și scalabil. Sistemul propus utilizează o arhitectură distribuită, permițând utilizatorilor să trimită job-uri de crawling care sunt procesate în paralel de un cluster de noduri, coordonate printr-un mecanism de stare partajată.

1.2 Obiective

Scopul fundamental al aplicației este de a facilita explorarea recursivă a site-urilor web printr-un sistem robust. Principalele obiective urmărite sunt:

- **Paralelism:** Procesarea simultană a paginilor web de către mai multe instanțe pentru a reduce timpul total de execuție.
- **Scalabilitate:** Posibilitatea de a adăuga noi noduri dinamic, fără a opri sistemul.
- **Feedback în timp real:** Vizualizarea progresului scanării direct în user interface prin WebSockets.
- **Robustețe:** Gestionarea nodurilor care devin inactive (dacă nu găsește heartbeat de la un anumit nod pe o anmită perioadă) și redistribuirea automată a sarcinilor.

1.3 Lista MoSCoW

Pentru prioritizarea funcționalităților, am utilizat metoda MoSCoW, clasificând cerințele în funcție de importanța lor strategică.

Must have	**Should have**	**Could have**	**Won't have**
Sistem distribuit Master-Worker	Autentificare JWT	Export date (CSV/JSON)	Analiză semantică NLP
Crawling recursiv (BFS)	WebSockets (Live Progress)	Grafice vizuale de legături	Indexare tip Motor de Căutare
Bază de date partajată (MySQL)	Dashboard monitorizare noduri	Dockerizare completă	Suport pentru procesare video

Tabela 1: Lista MoSCoW a proiectului

1.4 Cazuri de utilizare

Sistemul deservește două tipuri de actori: Utilizatorul (care dorește date) și Administratorul (care monitorizează clusterul).

Principalele scenarii includ:

1. **Submit Job:** Utilizatorul introduce un URL de start și adâncimea maximă.
2. **Task Processing:** Nodurile backend preiau automat sarcinile din baza de date.
3. **Cluster Monitoring:** Sistemul detectează automat nodurile active prin heartbeat.

2 Arhitectura

2.1 Privire de ansamblu

Arhitectura aplicației este de tip Peer-to-Peer / Master-Worker. Sistemul este compus dintr-un frontend dezvoltat în React (TypeScript) și un backend bazat pe Spring Boot.

Toate nodurile backend comunică printr-un Shared State Pattern, folosind o bază de date MySQL comună pentru sincronizare. Nu există un nod "Master" dedicat hard-codat. Orice nod poate prelua orice sarcină, coordonarea făcându-se prin baza de date.

2.2 Frontend (React)

Frontend-ul este realizat în React și TypeScript, are o interfață simplă, care constă într-o singură pagină. Acesta comunică cu backend-ul prin REST API pentru comenzi și prin WebSockets pentru a primi date în timp real despre progresul job-urilor.

2.3 Backend (Spring Boot)

Backend-ul utilizează Spring Boot și Java. Fiecare instanță pornită se înregistrează automat în cluster și începe să caute sarcini disponibile în baza de date.

2.4 Comunicare și Baza de Date

- **Axios:** Gestionarea cererilor HTTP standard.
- **MySQL:** Baza de date stochează job-uri (practic link-uri care trebuie explorate) și starea fiecărui job (PENDING, PROCESSING, COMPLETED, FAILED).

3 Implementare Frontend

3.1 Tehnologii Frontend

Pentru partea de interfață am folosit React și TypeScript. Această alegere m-a ajutat să mențin codul organizat, făcând dezvoltarea mult mai simplă și mai sigură. Pentru partea de update-uri în timp real (WebSockets) am folosit biblioteca stompjs.

3.2 Structura Componentelor

Aplicația este structurată modular. Principalele componente sunt:

Componentă	Rol
`Dashboard`	Pagina principală, afișează job-urile active și statusul nodurilor.
`SubmitJobForm`	Formular pentru introducerea URL-ului și a adâncimii.
`CrawlProgress`	Componentă care ascultă pe WebSocket și afișează bara de progres.
`ResultsView`	Afișează link-urile extrase într-un format tabelar sau listă.
`Login`	Gestionează autentificarea și stocarea token-ului JWT.

Tabela 2: Componentele React principale

3.3 Interfața Utilizator

Utilizatorul interacționează cu aplicația printr-un flux simplificat:

1. **Login:** Acces securizat în aplicație.
2. **Dashboard:** Vizualizarea stării clusterului (câte noduri sunt online).
3. **Inițiere Job:** Utilizatorul trimite un URL.
4. **Monitorizare:** Frontend-ul actualizează interfața pe măsură ce paginile sunt procesate (prin /topic/progress).

4 Implementare Backend

4.1 Tehnologii Backend

Backend-ul reprezintă nucleul logic al sistemului. Tehnologiile alese asigură performanță și scalabilitate:

- **Spring Boot:** Framework-ul principal.
- **Spring Data JPA:** Pentru interacțiunea cu baza de date și maparea obiect-relațională a entităților CrawlJob și CrawlTask.
- **Jsoup:** Bibliotecă pentru parsing HTML și extragerea link-urilor din paginile descărcate.
- **Spring Security:** Implementarea autentificării stateless folosind JWT.

4.2 Coordonarea Distribuită

Un aspect critic este gestionarea nodurilor. Clasa NodeService implementează logica de heartbeat:

- La pornire, fiecare nod generează un UUID unic.

- La fiecare 5 secunde, nodul actualizează timestamp-ul lastHeartbeat în baza de date.
- Un job programat verifică periodic nodurile care nu au mai trimis heartbeat și le marchează ca "dead", redistribuind sarcinile.

4.3 WebSockets

Pentru a oferi feedback instantaneu, am implementat protocolul STOMP peste WebSockets. Când un nod finalizează procesarea unei pagini, trimite un eveniment:

```
messagingTemplate.convertAndSend("/topic/progress/" + jobId,
    progressUpdate);
```

4.4 API Endpoints

Gestionarea cererilor HTTP se face prin REST

Endpoint	Metodă	Descriere
`/auth/login`	POST	Autentificare utilizator și emitere token JWT.
`/api/crawl/submit`	POST	Crearea unui nou job de crawling (URL + Depth).
`/api/crawl/{id}`	GET	Obținerea detaliilor despre un job specific.
`/api/nodes`	GET	Lista tuturor nodurilor (Active/Inactive) din cluster.
`/api/results/{id}`	GET	Descărcarea rezultatelor pentru un job finalizat.

Tabela 3: Lista principalelor endpoint-uri REST

5 Concluzii

Proiectul demonstrează eficiența arhitecturilor distribuite în rezolvarea problemelor consumatoare de resurse. Prin utilizarea Spring Boot pentru coordonare și a bazei de date ca mecanism de sincronizare (Shared State), am obținut un sistem scalabil.

Implementarea frontend-ului în React cu suport WebSocket asigură o experiență modernă și interactivă, permițând utilizatorilor să vizualizeze procesul de crawling în timp real, transparentizând activitatea clusterului de backend.

6 Bibliografie

1. Cursuri "Proiectarea aplicațiilor Web" și "Sisteme Distribuite".
2. Documentație Spring Boot, `https://spring.io/projects/spring-boot`
3. Documentație React, `https://react.dev/`
4. Documentație Jsoup, `https://jsoup.org/`

5. Ghid Spring Boot WebSockets, https://spring.io/guides/gs/messaging-stomp-websocket/