FACULDADE DE CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE D COIMBRA

Sistemas Distribuídos

Googol: Motor de pesquisas de páginas web

André Magalhães de Carvalho | 2020237655 | PL4
Gonçalo Gameiro Neves | 2020239361 | PL4
Henrique José Gouveia Lobo | 2020225959 | PL4

Introdução

Neste projeto, desenvolvemos uma página de pesquisas web, semelhante ao motor de busca Google com as seguintes funcionalidades:

- Indexar um URL novo, sendo que o motor trata de indexar os URLs que encontrar dentro deste.
- Pesquisar páginas que contenham um conjunto de termos, ordenando-as por relevância (número de páginas que referenciam esta página)
- Consultar as páginas que têm ligação para uma página específica
- Página de administração, com informações importantes sobre os diversos componentes.
- Indexar as stories de um utilizador do site de notícias Hacker News.
- Login/Logout e registo de utilizadores.
- Indexar as top stories do site de notícias Hacker News.

Todas as funcionalidades foram implementadas e são acessíveis através de uma interface web criada com springboot, html e css.

Arquitetura

Na implementação da arquitetura foi seguido o modelo MVC sendo o controlador o GoogolController e as views implementadas em html e css.

É importante notar que o model não é persistente, visto que essa responsabilidade é do projeto implementado na meta anterior.

Para as funcionalidades do lado do servidor (meta anterior), usamos uma arquitetura distribuída com :

- Search Module
- Múltiplos Downloaders
- Múltiplos Storage Barrels
- Queue de URLs a indexar.

Do lado do controlador foi usado o springboot e para as views foi usado html e css.

O websocket foi usado para obtermos atualizações em tempo real da página de administração, juntamente com ajax.

Para que as notícias do site de notícias Hacker News fossem indexadas, o nosso projeto foi integrado com a API do Hacker News.

Usamos o RMI para fazer a conexão entre o search module e o controlador, de modo a que este possa chamar os métodos na sua interface. Como isto foi avaliado na meta 1 não iremos explicar a fundo como foi implementado.

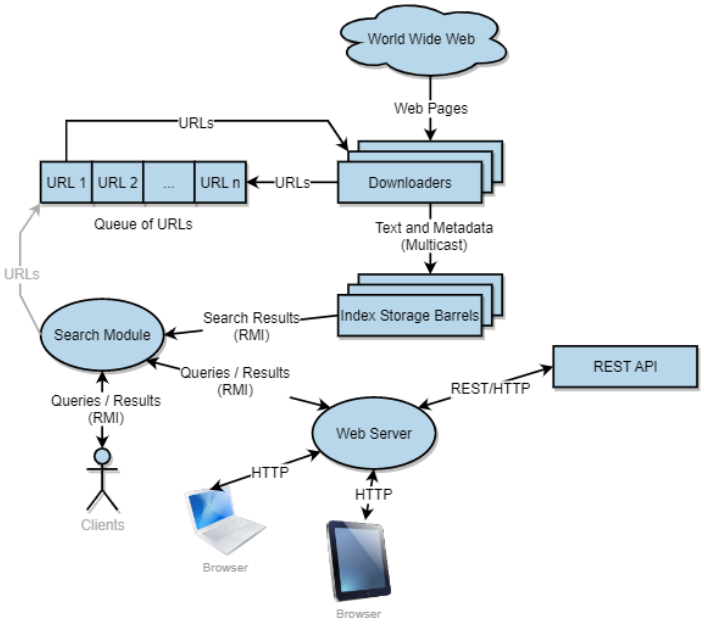

Figura 1: Arquitetura da aplicação

Controlador

O controlador é responsável por receber os pedidos do utilizador e encaminhá-los para o SearchModule que, por sua vez, devolve a informação necessária para as diferentes views.

Neste componente, foi implementado o websocket para obtermos atualizações em tempo real da página de administração, juntamente com ajax. Estas atualizações são feitas, intencionalmente, de 1 em 1 segundo, de modo a não sobrecarregar o servidor.

Na implementação do GoogolController, foram usados alguns parâmetros em autowired, para que o springboot fizesse a injeção de dependências automaticamente.

Websocket

O WebSocket é um protocolo de comunicação bidirecional que permite a troca de mensagens em tempo real entre um navegador (cliente) e um servidor. Ele é projetado para ser uma alternativa mais eficiente e de baixa latência em comparação com as técnicas tradicionais de comunicação baseadas em HTTP.

No nosso caso é usado um método de tratamento de mensagem do servidor para a rota ***"/hello".*** O cliente envia uma mensagem para esse endpoint e o servidor responde enviando atualizações periódicas para o cliente.

Na função é usado o *@MessageMapping("/hello")* que define o mapeamento da rota ***"/hello"*** para esse método de tratamento de mensagem. Isso significa que quando o cliente envia uma mensagem para "/hello", essa função será acionada para processar a mensagem.

A anotação *@SendTo("/topic/admin")* especifica o tópico de destino para o qual a resposta será enviada. Nesse caso, a resposta será enviada para o tópico ***"/topic/admin"***.

Dentro da função sendMessage() a mensagem é convertida em formato JSON usando o método *convertToJSON()*. Em seguida, a mensagem é encapsulada em um objeto Mensagem e enviada para o tópico ***"/topic/admin"*** usando o *messagingTemplate.convertAndSend()*.

Do lado do cliente, é feita uma subscrição ao tópico ***"/topic/admin"*** usando o método *connect()*. De seguida, é definido um callback para processar as mensagens recebidas do servidor. O callback é definido usando o método *stompClient.subscribe()*. Após a receção de uma mensagem, o callback é acionado e a mensagem é processada e mostrada na página.

Para implementar o envio de atualizações periódicas para o cliente de 1 em 1 segundo é usado um scheduler.

Endpoints

Para cada endpoint usamos a anotação **@GetMapping** seguido do nome do endpoint. Neste projeto implementamos os seguintes endpoints

"/" *(GET)*: Página inicial que tem um menu para todas as funcionalidades do motor de busca, bem como as opções de login, logout e registo.

"/login" *(GET)*: Página de login, onde o utilizador pode fazer login com o seu username e password. No nosso caso, o login é feito lendo e verificando se o username e password introduzidos estão no ficheiro de texto onde estão guardados os utilizadores registados. Após este, o utilizador tem acesso a todas as funcionalidades do motor de busca.

"***/logout***" *(GET)*: Fazer logout do utilizador, redirecionando-o para a página inicial do menu.

"/register" *(GET)*: Página de registo, onde o utilizador pode fazer registo com o seu username e password. No nosso caso, o registo é feito escrevendo para um ficheiro de texto.

"/indexNewUrl" *(GET)*: Página onde o utilizador pode indexar um novo URL. Para isso, basta escrever o URL e clicar no botão "Indexar URL". Após isto, o url introduzido é enviado para o SearchModule, que trata de indexar o URL e os URLs que encontrar dentro deste. No final, caso tenha corrido tudo bem, é apresentada uma mensagem de sucesso.

"/search" *(GET)*: Página onde o utilizador pode pesquisar páginas que contenham um conjunto de termos, ordenando-as por relevância (número de páginas que referenciam esta página). Para isso, basta escrever os termos e clicar no botão "Pesquisar". Após isto, os termos introduzidos são enviados para o SearchModule, que trata de pesquisar as páginas que contêm esses termos e ordená-las por relevância. Os resultados da pesquisa são apresentados numa nova página, que mostra de 10 em 10.

"/getSearchResults" *(GET)*: Este endpoint é usado para obter os resultados da pesquisa, 10 a 10 como referido acima.

"/listPages" *(GET)*: Página onde o utilizador pode consultar as páginas que têm ligação para uma página específica. Para isso, basta escrever o URL e clicar no botão "Consultar". Após isto, o URL introduzido é enviado para o SearchModule, que trata de

consultar as páginas que têm ligação para esse URL. É importante notar que para aceder a esta página é necessário estar autenticado (login).

"/IndexHackersByUsername *(GET)*: Página onde o utilizador pode escrever um username do site de notícias tech Hacker News e quando clicar no botão serão indexadas as stories deste utilizador. Para que isto aconteça o nosso projeto foi integrado com a API do Hacker News. Após todas as informações terem chegado ao controlador, este mostra-as ao utilizador e envia-as para o SearchModule.

"/IndexHackerNews" *(GET)*: Endpoint que é usado para indexar as top stories do site de notícias Hacker News, através da API do Hacker News. Isto acontece porque é feito um request com o método GET para a API do Hacker News, que nos devolve as top stories. Após isto, o controlador envia as informações para o SearchModule.

Integração com serviço REST

Este projeto foi integrado com o Hacker News. Para isso foi utilizada a API (Application Programming Interface) deles, para conseguir obter as principais histórias e histórias de usuários específicos. Usámos o GitHub da API da Hacker News para saber para que URL's devíamos fazer pedidos e quais as respostas que deveríamos esperar.

Para isto, foi criada uma classe chamada **HackerNewsAPI**, que contém os métodos necessários para obter tanto as histórias principais quanto as histórias de um usuário específico.

O método responsável por obter as URLs das histórias principais usa um **HTTP GET** para a URL da API que retorna as principais histórias. Depois temos um método auxiliar que faz outro pedido **HTTP GET** à API para obter os links de cada uma dessas histórias.

O outro método faz também um pedido GET, sendo necessário fornecer o nome do usuário desejado. Com isto obtemos uma resposta JSON em que vamos buscar o que nos interessa (os ID's das histórias do utilizador) e depois chamamos o método auxiliar que vai buscar os URL's de cada uma das histórias.

O método auxiliar utiliza threads, para conseguir ir buscar os URL's das histórias de forma mais rápida, usando também pedidos GET para a API.

Testes realizados à plataforma

Para tentar minimizar os possíveis erros que o cliente pudesse obter enquanto utiliza a nossa plataforma realizámos vários testes de controlo de qualidade.

- Realizar login e registo de forma correta e garantir que não dá para usar funcionalidades exclusivas a utilizadores registados sem o estarem.
- Ter o servidor a trabalhar numa máquina e ter o cliente numa máquina diferente e funcionou tudo corretamente (Testamos com mais do que dois clientes ao mesmo tempo)
- Garantir que as funcionalidades core do projeto funcionam:
    - Realizar uma procura (garantindo que os resultados aparecem agrupados 10 em 10)
    - Indexar páginas
    - Listar páginas que tenham ligação para outras
    - Página de administração atualizada em tempo real (neste caso com um delay intencional de 1 segundo).
    - As top stories da Hacker News são indexadas corretamente
    - As stories de um utilizador específico são indexadas corretamente

Conclusão

Com este projeto, conseguimos implementar um motor de busca web, semelhante ao Google, com todas as funcionalidades pedidas. Para além disso, conseguimos integrar o nosso projeto com a API do Hacker News, de modo a indexar as top stories e as stories de um utilizador. Com isto aprendemos a criar um sistema distribuído, com dois componentes que se comunicam entre si. Para além disso, aprendemos a usar o springboot para criar uma interface web, bem como a usar o websocket para obter atualizações em tempo real da página de administração, juntamente com ajax.