

Googol: motor de pesquisa de páginas Web

Sistemas distribuíds - meta 2

David Silva – 2020217642

Eduardo Carneiro 2020240332

[Introdução 2](#_Toc363883867)

[Arquitetura de software 3](#_Toc1916914506)

[Protocolo de comunicação multicast 4](#_Toc803430777)

[Chamadas RMI e callbacks 5](#_Toc994498586)

[Componentes 6](#_Toc790783510)

[Models, Views e Controller 8](#_Toc1058061234)

[Models 9](#_Toc1748642318)

[Views 9](#_Toc28850662)

[CONtrollers 9](#_Toc1296376563)

[Endpoints 10](#_Toc938704)

[Integração SpringBoot com SearchModule RMI 10](#_Toc1393079186)

[Websocket 10](#_Toc300135479)

[Integração com API REST do HackerNews 11](#_Toc648822996)

[Tratamento de exceções e failover 11](#_Toc2074040053)

[Exceções 11](#_Toc1135407402)

[Failovers 12](#_Toc168057220)

[Testagem 12](#_Toc1582341738)

[Informações adicionais 13](#_Toc1679398575)

[Conclusão 12](#_Toc316308974)

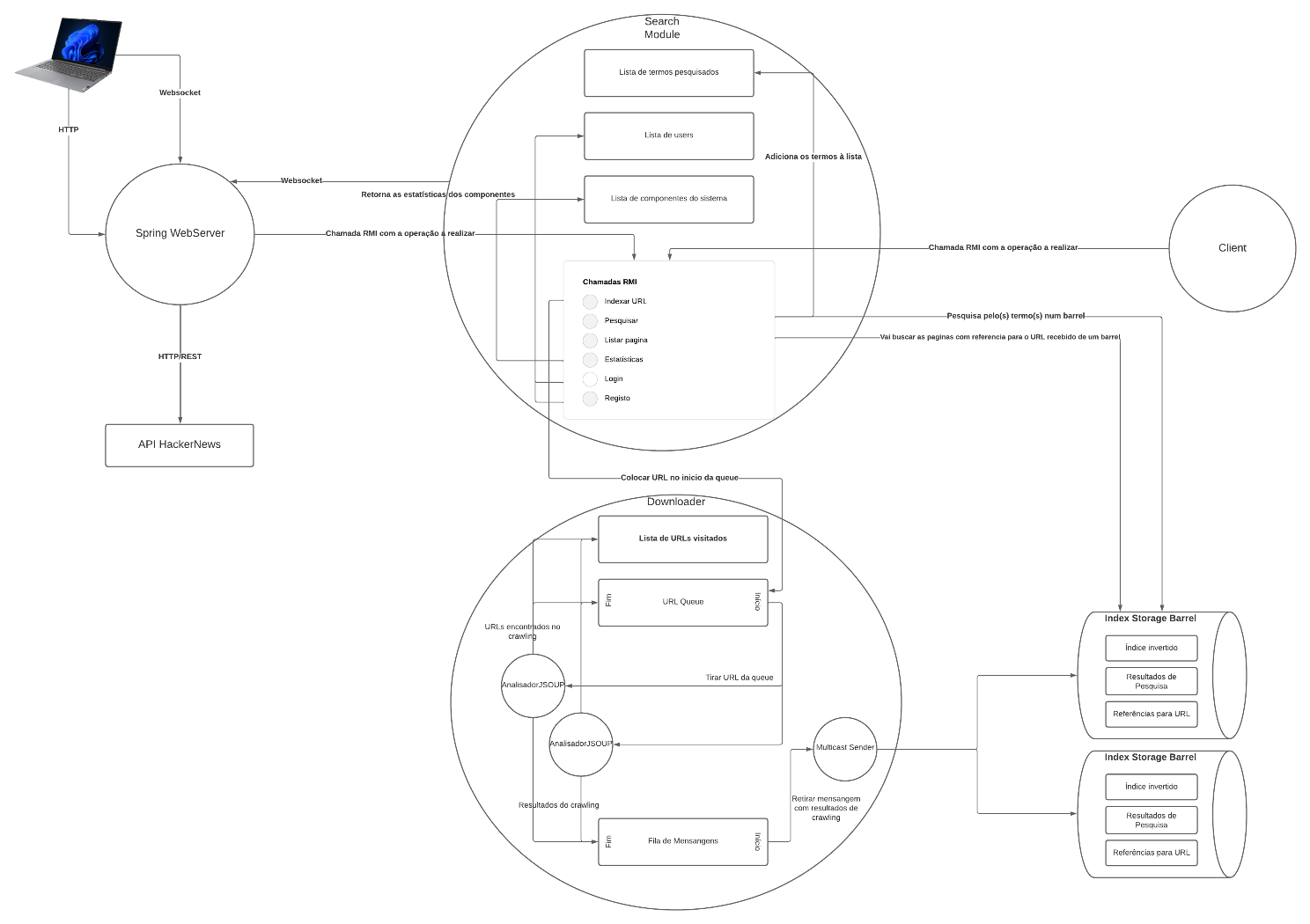
# Introdução

Este projeto tem como objetivo o desenvolvimento de um motor de pesquisa, semelhante às funcionalidades primitivas do Google.com.

Na meta 1, o maior foco esteve relacionado com as ligações RMI e o desenvolvimento das funcionalidades dos componentes. Nesta meta, com o auxílio da framework Spring desenvolveu-se a interface gráfica do Googol, como também, a introdução de novas funcionalidades, nomeadamente, a indexação de urls através da API do HackerNews.

# Arquitetura de software

O sistema está dividido em 4 componentes centrais: Cliente, Search Module, Downloader e Index Storage Barrels(ISB). Além disso, para suportar uma interface gráfica é necessário um WebServer que permita que um cliente consiga fazer pedidos HTTP para efetuar as funcionalidades suportadas pelos 4 componentes centrais. Em baixo, está presente um diagrama de alto nível das componentes deste sistema.



Arquitetura alto nível do sistema

Cada um destes componentes é fundamental para o funcionamento do sistema. Começando pelo Cliente, este apresenta uma estrutura simples. Realiza o lookup RMI para receber uma referência ao Search Module. Seguidamente, recebe input do utilizador sobre que operação realizar e, dependendo desta, faz a respetiva chamada, através de RMI, ao Search Module para a efetuar essa mesma operação. Em relação aos resultados recebidos, são processados de acordo com a sua especificação.

O Search Module é a porta de entrada do cliente ao sistema. Neste é recebida a operação a realizar pelo cliente e, dependendo dessa, acede às outras componentes do sistema (Downloader e ISB). É também onde se encontra a lista de utilizadores, com os respetivos usernames e passwords, a lista de componentes do sistema, utilizada para manter informações relativas a alguns componentes, e a lista de termos pesquisados, recorrida para obter os termos mais pesquisados.

O Downloader é a componente responsável pela realização do crawling dos URLs, tanto introduzidos pelos utilizadores, bem como os encontrados aquando do crawling de outros URLs. Existe nesta componente uma queue de URLs, na qual são introduzidos novos URLs para futuro crawl, bem como uma fila de mensagens e uma lista de URLs visitados (explicados mais adiante). Embora seja um só processo, são criadas várias threads do tipo ‘AnalisadorJSOUP’ (subcomponente onde é efetivamente efetuado o crawl do URL), de modo a realizar processamento em paralelo, aumentando a eficiência do sistema, e ainda uma thread ‘MulticastSender’ (subcomponente utilizada para envio de informação para os ISB). Estas threads ‘AnalisadorJSOUP’ retiram concorrentemente URLs da queue de URLs, verificam se estes já foram visitados (averiguando se estes se encontram na lista de URLs visitados) e, caso não, dão crawl utilizando o JSOUP, adicionando os URLs encontrados ao fim da queue de URLs. Quanto à informação sobre a página, é colocada numa estrutura própria e colocada na fila de mensangens. A thread ‘MulticastSender’ encarrega-se de retirar mensagem a mensagem da fila e enviar por multicast para todos os ISB.

Por fim, os Index Storage Barrels são a componente de armazenamento do sistema. É aqui que se consegue manipular o índice invertido, com informação das páginas como título, uma citação curta, URL, bem como o conjunto de URLs acessíveis através dessa página. Existe também um HashMap com resultados de pesquisa, utilizado para fracionar a pesquisa enviada para o utilizador.

# Protocolo de comunicação multicast

O Multicast é utilizado neste sistema distribuído para enviar a informação recolhida pelas threads AnalisadorJsoup (com recurso à biblioteca JSOUP) para os Barrels, para estes indexarem a informação e armazenarem-na.

Assim, quando a thread AnalisadorJSOUP acaba de analisar um URL adiciona para uma fila de mensagens (todas as threads AnalisadorJSOUP têm acesso direto a essa fila) a classe JSOUPData (contém a informação do url, o título da página associada ao url, uma citação da página, a lista de termos que essa página contém e a lista de URLs que podem ser chamados através desta página).

Posteriormente, a thread MulticastSender retira da lista uma classe JSOUPData a enviar para todos os barrels.

Para enviar a informação para os Barrels, primeiramente, serializamos a classe JSOUPData e enviamos o tamanho para o Barrel, de seguida, comprimimos o array de bytes obtido na serialização, e enviamos a classe comprimida para o Barrel.

Assim, primeiramente, o Barrel quando recebe a mensagem do tamanho guarda numa variável o tamanho da classe não comprimida que vai receber de seguida, e posteriormente recebe então a classe comprimida, descomprime e faz a desserialização.

Importa ainda referir que foram introduzidas medidas para tornar o Multicast o mais fiável possível, nomeadamente, o envio de uma mensagem quando o Barrel recebe a mensagem enviada por Multicast, como também um byte de controlo para distinguir se a mensagem se refere ao tamanho ou a uma classe.

Assim, quando enviamos uma mensagem Multicast e o Barrel recebe-a, o Barrel envia um ACK (neste caso, com o seu hashCode(), o seu identificador) para o MulticastSender (que o guarda num HashSet<Integers>, o MulticastSender conhece o número de ACKs que espera (quando se adiciona um Barrel é atualizada via RMI uma variável do tipo AtomicInteger presente no Downloader, que a thread MulticastSender tem acesso que permite saber quantos barrels estão ativos. Logo, no caso da primeira vez o tamanho do HashSet não coincidir com o número de Barrels disponíveis, tenta-se enviar novamente a mensagem e esperar as respostas do número do Barrels que faltaram enviar os ACKs, caso numa segunda tentativa não consigamos, assumimos que o Barrel está desligado, e, portanto, enviamos o HashSet de ACKs que recolhemos para o SearchModule eliminar os Barrels que não estão presentes nesse HashSet.

Quanto à parte dos bytes de controlo, escrevemos na primeira posição do array de bytes, um 0, caso o tipo da mensagem a enviar seja um tamanho, e escrevemos na priemira posição do array de bytes um 1, caso o tipo de mensagem a enviar seja uma classe, e de seguida escrevemos a informação que queremos enviar propriamente. Assim, quando o Barrel está à espera de uma mensagem do tipo 1 (classe) e receber uma mensagem do tipo 0 (tamanho), não tenta descomprimir o array de bytes (neste caso a string que indica tamanho) e desserializar a classe descomprimida.

Além disso, caso o número de Barrels seja 0 não retiramos nenhuma classe da fila de mensagens e enviamos por Multicast, pois, não temos nenhum Barrel para receber esta informação. Assim, de forma a evitar a espera ativa, fazemos um wait() no while que retira as mensagens da fila de mensagens e “acordamos esse wait” quando o número de barrels passar a ser 1, ou seja, quando o Barrel é adicionado, o SearchModule envia via RMI o número de Barrels conectados, e caso o número de Barrels conectados seja igual a 1 notifica o objeto que está em wait(), e assim a thread (re)começa a retirar classes da fila de mensagens e a enviar para os barrels.

# Chamadas RMI e callbacks

São utilizadas chamadas RMI em todas as componentes do sistema. É possível efetuar 6 tipos diferentes de chamadas entre o cliente e o SearchModule: indexar um URL, efetuar uma pesquisa, listar páginas com referência para um URL recebido, mostrar as estatísticas do sistema, login e registo. Para cada uma destas operações, existe um método correspondente no SearchModule. Nesta mesma componente, de seguida e caso seja necessário, estas operações fazem também chamadas RMI tanto ao Downloader como aos ISB. No caso de indexar um URL, é efetuada uma chamada RMI ao Downloader, de modo a introduzir este URL no início da queue de URLs. Para realizar uma pesquisa ou obter a lista de URLs que fazem referência para um URL específico(informações contidas nos ISB), é também efetuada uma chamada RMI respetiva aos métodos nos ISB (o SearchModule escolhe 1 barrel aleatoriamente).

Em relação a callbacks, estes existem entre o Search Module e o Downloader e entre o Search Module e os ISB. Como já foi referido, existem chamadas do SearchModule para ambas estas componentes, no entanto também foram introduzidas chamadas RMI destas mesmas componentes para o SearchModule, as quais são utilizadas para atualizar informações relativas a estatísticas do sistema, isto é, se certa Thread do Downloader está livre ou não, ou se certo ISB está livre ou não.

# Componentes

Como descrito na arquitetatura do sistema, o sistema está dividido em 4 compoentes essenciais: Cliente, Search Module, Downloader e Index Storage Barrels(ISB).

**SearchModule**

O SearchModule tem como objetivo servir de “ponte” entre os diversos componentes.

Assim, o SearchModule utiliza algumas estruturas de dados que permitem armazenar a informação, descritas de seguida:

* Uma lista de Barrels conectados, quando iniciamos um Barrel, o barrel envia via RMI a sua referência da sua interface, de forma ao SearchModule puder fazer pedidos via RMI a qualquer barrel ativo. Esta lista é armazenada utilizando a estrutura de dados, CopyOnWriteArrayList<BarrelRMI>, dado que é uma estrutura que é Thread-Safe e lida com concorrência.
* Uma lista de pesquisas com todas as pesquisas feitas por todos os utilizadores, assim, conseguimos saber os termos que foram mais pesquisados, a lista é armazenada utilizando estrutura de dados, CopyOnWriteArrayList<BarrelRMI>, dado que é uma estrutura que é Thread-Safe e lida com concorrência.
* Uma HashMap que tem como valor o username e como chave a password, permite armazenar os users, utiliza-se um ConcurrentHashMap, de forma a lidar com problemas de concorrência (mudar no codigo))
* Uma HashMap que tem como valor o nome do componente e a classe que representa o componente (contém o IP e um booleano que representa a disponibilidade do componente), utiliza-se um ConcurrentHashMap, de forma a lidar com problemas de concorrência.

**Downloader**

O Downloader tem como principal objetivo analisar os URLs e enviar a informação recolhida por Multicast para os Barrels.

Assim, o Downloader necessita de guardar alguma informação, descrita de seguida:

* Lista de URLs a analisar, podendo ser introduzidos pelo utilizador (neste caso, é introduzido à cabeça da lista, de forma a indexar de forma instantânea a página introduzida) ou então introduzidos pelo crawler quando encontra um url no ulr que está a analisar (neste caso, é introduzido na cauda da lista). A estrutura que usamos é uma LinkedBlockingDequeue, pois permite inserções à entrada ou à saída da lista, além disso, implementa mecanismos thread-safe, como também espera bloqueante (útil para quando a fila está vazia e queremos retirar algum URL, assim, só quando a fila voltar a ter algum elemento é que é devolvido)). Esta lista é usada entre as diferentes threads.
* Lista de URLs visitados, de forma a não repetir o crawl do mesmo URL, contudo, esta implementação não funciona bem com páginas dinâmicas, pois não voltaremos a analisar este URL. A estrutura dados que utilizamos é um Set<String> inicializado com um ConcurrentHashMap.newKeySet() de forma a ser thread-safe a assim lidar com problemas de concorrência.
* Fila de mensagens Multicast, as threads AnalisadorJsoup inserem a classe JSOUPData na fila (que contém as informações recolhidas pelo crawl) e a thread MulticastSender retira a classe JSOUPData da fila e envia por Multicast. Assim, os diferentes tipos de threads têm acesso a esta fila de mensagens. Além disso, a estrutura usada é um LinkedBlockingQueue pois é thread-safe e bloqueante (ou seja, quando a lista não tem elementos não fica em espera ativa a procurar um elemento).

Importa ainda referir que este componente se baseia essencialmente em 2 subcomponentes, uma thread MulticastSender que envia a info para os barrels por Multicast e um número de threads definido que faz o crawl, propriamente dito, aos URLs que estão na fila de URLs.

**Index Storage Barrels**

Os Barrels servem para guardar o índice invertido dos urls, isto é, para cada termo guardar os urls em que esse termo aparece, o core para fazer pesquisas de forma rápida e eficiente.

Nesta meta, de forma a tornar o sistema mais robusto, fizemos alterações na forma de como a informação era guardada, desta forma, a maior parte da informação é agora guardada recorrendo a base de dados (neste caso, SQLITE).

Assim, o barrel (DB) armazena as diferentes informações, descritas de seguida:

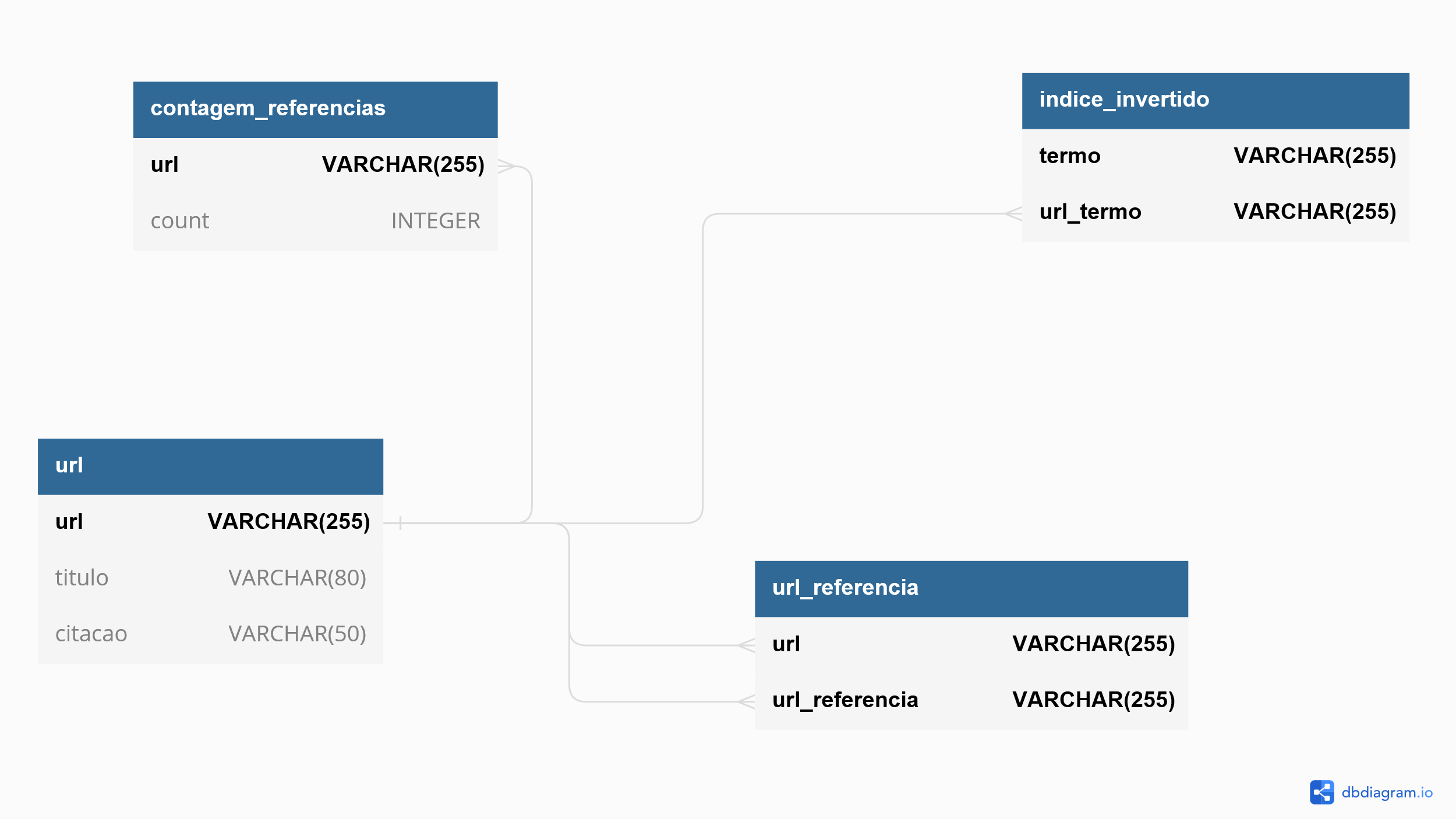


Diagrama ER da base de dados que guarda a informação necessária para os barrels

Estas tabelas armazenadas em DB têm como principal objetivo obter o índice invertido, isto é, para cada termo obter o conjunto de URLs em que aparece esse termo, como também obter os URLs que são referenciados por um determinado URL.

A tabela contagem\_referencias é importante para ordenar de forma eficiente quando se faz uma pesquisa e os URLs mais referenciados têm maior importância.

Importa, ainda frisar que são utilizadas transações, de forma, a lidar com problemas de concorrência, como também, tornar os dados mais consistentes.

Além disso, o próprio Barrel armazena um HashMap que tem como chave o identificador de uma pesquisa (um id de pesquisa que é gerado no endpoint Spring que permite que o utilizador faça uma pesquisa por um termo) e como valor os resultados da pesquisa que faltam enviar para o cliente. Esta informação é armazenada usando um ConcurrentHashMap de forma a lidar com problemas de concorrência.

# Models, Views e Controller

O WebServer neste projeto, implementado em Java com SpringBoot, segue uma arquitetura MVC.

Este tipo de arquitetura divide o servidor em 3 componentes principais: Model, View e Controller. O Model é responsável pelas trocas de dados, a View é a camada de representação apresentada ao utilizador e o Controller interface de pedidos entre o utilizador e o resto do sistema.

## Models

São utilizados dois Models: InputText e CredentialsInput. O InputText apenas contém um atributo: uma string chamada “*inp”*. Este é utilizado para receber qualquer input simples do utilizador em string, tal como os termos da pesquisa, o URL a indexar, o username do utilizador do HackerNews, etc... O CredentialsInput apresenta dois atributos, ambos strings: “user” e “password”. É utilizado para receber inputs de login e registo dos utilizadores.

## Views

As views criadas utilizam Thymeleaf para a renderização do HTML, facilitando na passagem de informação entre os controllers e as views.

Assim, não existe necessidade de renderizar o HTML através de JavaScript ou de concatenação de strings HTML.

Desta forma, as views são devolvidas para o utilizador pelos controllers, e caso existam atributos específicos, estes atributos são adicionados pelo controller.

## CONtrollers

Para o desenvolvimento deste projeto decidimos utilizar 3 controllers: HomeController, UserController e WebsocketController. Esta divisão foi efetuada com base nos diferentes tipos de funcionalidades a realizar pelo utilizador.

* O HomeController é o controller “principal”, sendo responsável pelo handling da home page, bem como todos os endpoints de pesquisa de termos e indexação. É neste controller que está contida a maior parte da lógica do servidor.
* O UserController é o controller que contém os endpoints relacionados com login, registo e logout.
* O WebsocketController, por fim, é apenas responsável por um único endpoint que está ligado, como o nome indica, à funcionalidade do websocket.

# Endpoints

Os endpoints definidos podem ser divididos em 2 grupos: “page loaders” e “action makers”. Estes grupos podem ser explicados de maneira simples:

* Page Loaders: estes endpoints apresentam nenhuma ou pouca lógica e servem apenas para devolver uma View ao utilizador, como resposta ao clique num botão ou mudança de página. Neste grupo estão incluídos os seguintes endpoints (todos através de GET resquest): “/”, “/index-url”, “/reference-url”, “/login”, “/register” e "/index-hackernews-username-page".
* Action Makers: estes são os endpoints que executam, de facto, todo o “trabalho”. É neles que está toda a lógica e de onde são efetuadas as chamadas RMI ao SearchModule. Estão incluídos neste grupo todos os endpoints que não foram mencionados no ponto anterior.

# Integração SpringBoot com SearchModule RMI

A integração do servidor web SpringBoot com o servidor do SearchModule é bastante simples e *straightforward.*

O SpringBoot, ao inicializar o HomeController e o UserController, faz um lookup ao RMI do SearchModule, ficando com uma referência para este. Deste modo, a partir deste ponto, ambos os controllers conseguem fazer chamadas RMI ao SearchModule como se fossem um cliente, apresentando então um funcionamento semelhante ao do RMI Client da meta 1 (cliente na consola).

# Websocket

Nesta meta uma das funcionalidades a implementar era permitir que a página de administração atualizasse em tempo real.

Assim, para cumprir esta funcionalidade foi necessário usar um WebSocket, para que quando o Search Module altere as estruturas de dados responsáveis por armazenar as estatísticas do sistema (components e as pesquisas mais frequentes) a página de administração renderizada por Spring seja atualizada de forma automática, isto é, sem ter de fazer *refresh* de forma manual à página.

Neste sentido, quando a página das estatísticas é renderizada, utilizando JS o cliente conecta-se ao WebSocket (SockJS) criado quando o Spring Boot inicia, e fica em escuta para receber mensagens novas nesse SockJS.

Do lado do servidor (backend), o Search Module todas as vezes que manipula uma estrutura de dados relacionada com a página de administração, envia as duas estruturas para o WebServer (utilizando a função sendMessage da classe WebSocketClient). Assim, quando o Search Module envia a estrutura de dados os clientes que estão em escuta nesse SockJS ao receber a mensagem, o HTML é alterado automaticamente, e assim têm acesso à informação atualizada.

# Integração com API REST do HackerNews

A API REST do HackerNews dispõe de endpoints que devolvem exatamente a informação necessária a ser aplicada neste projeto: endpoint com informação sobre uma story,endpoint com informação sobre um user e um endpoint com as Top Stories.

Para facilitar a compreensão, segue-se uma explicação sucinta dos endpoints da API do HackerNews utilizados:

1. Story Info: para obter informação acerca de uma story, é apenas necessário fazer um GET ao endereço [https://hacker-news.firebaseio.com/v0/item/<ID\_DA\_STORY>.json?print=pretty](https://hacker-news.firebaseio.com/v0/item/%3cID_DA_STORY%3e.json?print=pretty), o qual devolve um JSON com a mesma.
2. Top Stories: os ID's das top stories são devolvidos como um array JSON ao fazer um GET ao endereço <https://hacker-news.firebaseio.com/v0/topstories.json?print=pretty>
3. User info: executando um GET ao endereço [https://hacker-news.firebaseio.com/v0/user/<USERNAME>.json?print=pretty](https://hacker-news.firebaseio.com/v0/user/%3cUSERNAME%3e.json?print=pretty) é possível obter informação acerca de um user em formato JSON.

Através deste endpoints foram implementadas todas as funcionalidades pedidas. Para a funcionalidade de indexar URLs de stories de um utilizador foi utilizado o endpoint que devolve informação sobre esse utilizador, nomeadamente as stories que publicou. A partir daqui, apenas é necessário percorrer essas stories uma a uma e fazer um request ao endpoint que devolve informações sobre uma story, para saber qual o URL desse mesmo. Em relação à funcionalidade de indexação de stories contendo os termos da pesquisa, o processo é similar. São obtidas as top stories através do endpoint específico para tal, são percorridas estas top stories e feito um request para obter informações sobre estas e, caso reúnam condições, são indexadas.

# Tratamento de exceções e failover

## Exceções

Para o tratamento de exceções, no geral, utilizamos o simples try/catch do Java, adequando depois a falha ao contexto, isto é, caso seja uma falha que necessite de ser recuperada, como uma falha numa pesquisa de um ISB, realizamos novamente a pesquisa, mas utilizando outro ISB. Mais especificamente:

* Se Search Module falhe a meio de uma operação, o cliente (via terminal) recebe uma exceção e volta a tentar realizar a mesma operação automaticamente.
* Caso um ISB falhe a meio de uma operação, o Search Module recebe uma exceção e tenta executar a mesma operação noutro ISB.
* Caso o Downloader falhe a meio da operação de introduzir um novo URL na queue através da chamada RMI, o Search Module volta a tentar fazer novamente esta operação.
* Caso o Search Module envie alguma exceção, o cliente (conectado via HTTP) é redirecionado para uma página de erro.

## Failovers

Até agora foram referidos problemas intra-processo, isto é, caso ocorresse algum problema na execução normal do programa. Tendo em conta que é suposto simular um sistema distribuído, isto é, ter programas a correr em locais diferentes e computadores diferentes, sabemos que podem existir falhas externas que intercetam o funcionamento normal do programa, tais como falhas de energia ou quebras de hardware. Desse modo, efetuamos algumas proteções ao nosso código de modo a, caso exista uma destas falhas, seja possível recuperar informação já processada.

Deste modo, introduzimos a escrita de informação crucial para disco, utilizando objetos serializáveis como também um ficheiro base de dados para cada barrel. Assim, por exemplo, caso um ISB falhe, ao ser reiniciado toda a informação sobre as páginas será recuperada ao escrever o nome do ficheiro base de dados correspondente ao Barrel que falhou. O Downloader e o Search Module são recuperados, utilizando os ficheiros de objetos. Como estes componentes poderiam falhar a meio da escrita, utilizámos um mecanismo de duplicação de ficheiros aquando da escrita, de modo a existir sempre um ficheiro íntegro dos dados. A escrita para disco é efetuada apenas após algumas iterações do programa, minimizando o overhead de escrever a cada alteração, no entanto gerando alguma perda em caso de falha. No caso do Barrel, como foi utilizado um ficheiro base de dados foram utilizados mecanismos de transição, e, portanto, só depois de toda a informação ser escrita é que é feito o *commit* para base de dados, e, portanto, não existe informação inconsistente.

# Testagem

|  |  |  |
| --- | --- | --- |
| **id** | **Descrição** | **Resultado** |
| 1 | Indexar o URL ‘https://www.uc.pt’ | O cliente recebe a informação de que o URL foi indexado e o url é adicionado à lista de URLs a fazer crawl |
| 2 | Indexar um URL não valido | O cliente recebe a informação de que a string introduzida não é um url |
| 2 | Pesquisar pelo termo ‘uc’ | O termo existe no barrel e é mostrada a página de resultados ao utilizador |
| 3 | Avançar para a próxima página dos resultados | A próxima página de resultados é mostrada |
| 4 | Recuar para a página anterior dos resultados | A página anterior de resultados é mostrada |
| 5 | Recuar e avançar novamente  na página de resultados | A página anterior é mostrada, e depois ao avançar a página onde estávamos é mostrada novamente |
| 6 | Efetuar registo | O cliente é redirecionado para a página de login para entrar no sistema |
| 7 | Efetuar Login com um username e password válidos | É apresentada a mensagem de boas-vindas ao utilizador |
| 8 | Efetuar login com um username e password não validos | O utilizador recebe a informação de que o username e a password não é valido |
| 9 | Lista páginas com ligação para o URL ‘https://www.uc.pt’ | São apresentados os URL com ligação para ‘https://www.uc.pt’ |
| 10 | Pedir estatísticas do sistema e o utilizador está logado | É apresentada uma tabela com as estatísticas do sistema |
| 11 | Pedir estatísticas do sistema e o utilizador não está logado | O utilizador é redirecionado para a página de login |
| 12 | Adicionar um barrel enquanto o user está na página de stats | A informação contida na página é atualizada automaticamente, e a informação deste barrel é mostrada |
| 13 | Fail do Downloader e recuperação dos dados | É recuperado o estado anterior do componente |
| 14 | Fail do ISB e recuperação dos dados | É recuperado o estado anterior do componente |
| 15 | Indexar stories de um utilizador do HackerNews | Aparece uma mensagem de sucesso e o número de stories indexadas, ou uma mensagem de erro caso contrário |
| 16 | Indexar stories de um utilizador que não existe do HackerNews | Aparece uma mensagem de erro a indicar que o utilizador não existe |

# Conclusão

Através do desenvolvimento deste projeto foi-nos possível aprofundar os nossos conhecimentos acerca de RMI, bem como entender melhor o funcionamento de arquiteturas que apresentam redundância e processamento paralelo, de modo a aumentar a eficiência e o *uptime*. Para além disto, foi também possível compreender a importância da utilização de uma arquitetura MVC, de modo a simplificar a estrutura do projeto, separando as partes lógicas, evitando duplicação de código e facilitando a manutenção

Ao longo do desenvolvimento do projeto foi possível compreender a importância de uma abordagem distribuída para lidar com requisitos de escalabilidade, desempenho e disponibilidade.

Em suma, parece-nos que a solução desenvolvida pelo grupo seja adequada ao problema, apresentando todas as funcionalidades pedidas, bem como praticamente todos os requisitos não-funcionais e *failover*.