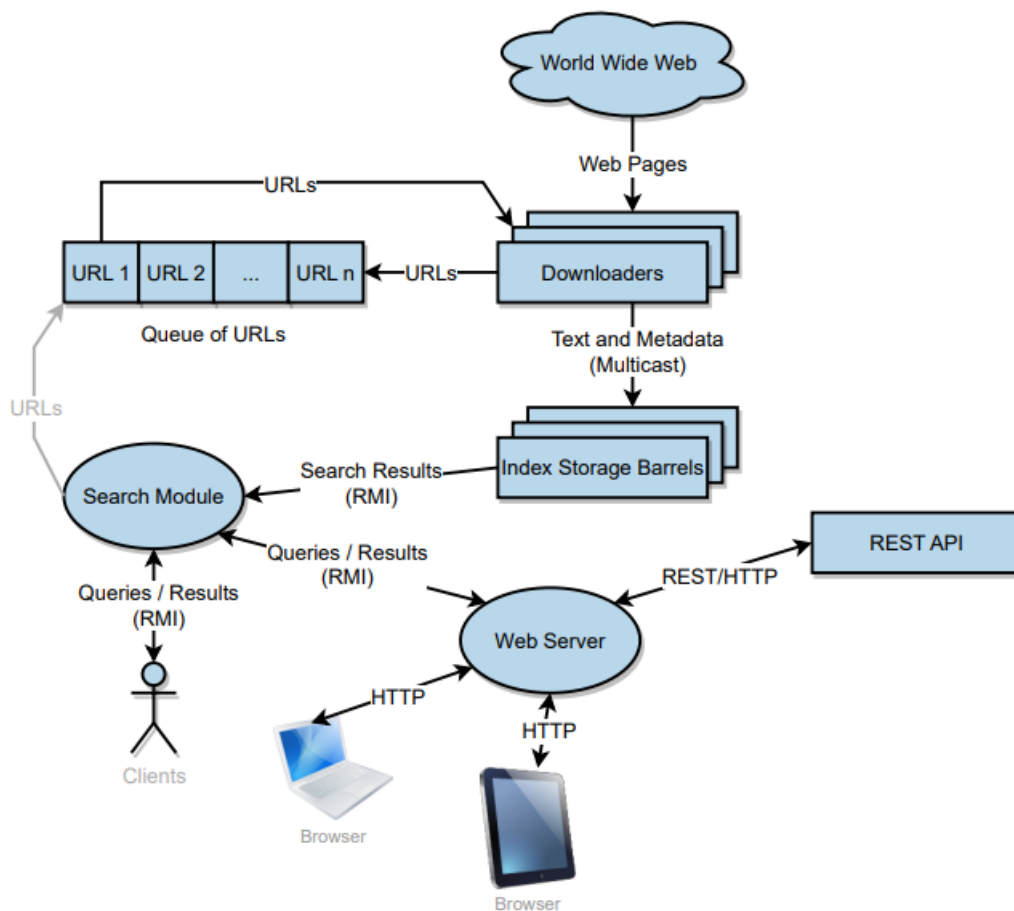


## Introdução

Nesta segunda meta do trabalho foi-nos pedido para desenvolver uma interface Web para integrar com a aplicação Googol desenvolvida na primeira meta.

## Arquitetura



---

## Arquitetura

O programa consiste em 2 principais processos:

- **Main** - Processo responsável por inicializar a Queue, os Downloaders e os Barrels
- **Search Module** - Processo com os métodos RMI a serem chamados para executar as tarefas e funcionalidades desejadas.

Usámos um modelo **MVC** com o controlador **App\_Controller**. Neste controlador utilizámos uma spring boot. Para o frontend do programa também tivemos de utilizar html e css (views).

Para obtermos as stories do Hacker News tivemos de integrar a **API** do *Hacker News*.

Foi utilizada uma **Web Socket** para receber as informações da página de administração em tempo real.

Utilizámos **RMI (Remote Method Invocation)** para estabelecermos uma ligação entre o **Search Module** e o **spring boot**, de forma a conseguir executar as funcionalidades no **Web Server**.

## Spring Boot

Começamos por integrar a **Spring Boot** com o programa já feito na meta 1. Para tal, usamos a comunicação *RMI* para que seja possível a chamada de várias funções do **RMIsearchModule** para a interface web. No ficheiro java **SdProjectApplication** criamos uma função **connect()** com uma anotação **@bean** para garantir que existe comunicação *RMI* entre o servidor e o controlador. Se esta comunicação não existir, o controlador não consegue correr. Usamos como endereço ip o do servidor e usamos um porto específico para estabelecer a tal conexão.

---

No ficheiro java **App\_Controller** temos uma variável ( **logged\_in** ) que controla se o cliente está logado ou não. Quanto ao **login**, criamos um ficheiro de texto ( **login.txt** ), sempre que um cliente faz o **register**, colocamos o *username* e a *password* nesse ficheiro de texto. No **login** verificamos se o *username* e a *password* existem no ficheiro de texto (as informações estão guardadas da forma " *username;password* " sendo cada informação de **login** separada por um **\n** ). Sempre que o **login** estiver correto, atualizamos a variável **logged\_in** para true.

Relativamente aos métodos criados neste ficheiro, temos:

## App\_Controller

Método que recebe um parâmetro do tipo **ServerInterface**. Utilizamos **@Autowired** para instruir o Spring a injetar automaticamente uma instância "**ServerInterface**" quando uma instância "**AppController**" é criada permitindo assim utilizar o objeto "**ServerInterface**" sem a necessidade de o criar manualmente ou gerenciar a instância.

## Menu

Este método é chamado quando uma requisição **GET** for feita para o endpoint **"/menu"**. Também verifica o estado da variável **login** de maneira a apresentar o *username* .

## Register

Este método como referido acima, guarda as informações do registo para poder ser usado mais tarde no **login**.

## Login

Método para um *user* poder efetuar o **login**. Se a combinação *username/password* não existir no **login.txt** é feito o redirect para o endpoint **"/error"** a dizer que houve erro no **login**.

## Index

---

Método para um *user* poder inserir um **url** para este ser indexado nos **barrels**. Se o *url* não existe, o *user* é redirecionado para endpoint **"/error"**. Para um *url* ser indexado, usamos a função **opcaoUm()**. Este é um caso que explica o porquê de ser necessária a comunicação **RMI**, visto que esta função faz parte do **servidor**.

## Word

Método para um *user* poder escrever uma palavra/contexto e posteriormente listar os sites em que esta aparece. Se não existe nenhum site em que a palavra/contexto apareça o *user* é redirecionado para o endpoint **"/error"**.

Para um contexto ser indexado, usamos a função **opcaoDois()**. *Mais uma vez este é um caso do porquê de ser necessário comunicação RMI, visto que esta função faz parte do servidor.* Se não ocorrerem erros, o *user* será redirecionado para um endpoint do tipo **"/word/palavra/?pagina=0"**. Como era preciso mostrar as ocorrências do contexto de 10 em 10, tivemos de criar várias *"paginas"* para poder mostrar todas as ocorrências.

## WordPage

Após um *user* procurar uma palavra com sucesso, será redirecionado para este método. Começa na página zero, mas o *user* tem a possibilidade de navegar por estas. Para tal ,vamos guardando o número da página para saber quais são as ocorrências a mostrar. Recorremos também ao método **opcaoDois()** do servidor.

## Url

Método que ao receber um *url* vindo do input do *user* utiliza o método **opcaoQuatro()** que devolve uma lista com os *links* que apontam para o *url* pesquisado. Se der erro o *user* é redirecionado para o endpoint **"/error"**. Mais uma vez, o método **opcaoQuatro()** é do servidor e só foi possível o seu uso a partir do RMI.

---

## Error

Método para apresentar ao *user* o erro.

## Logout

Método para permitir ao um *user* efetuar o logout.

## UserStories

Neste método tivemos de utilizar funções de uma **api rest**. Após usarmos a função **userStories** da **api rest**, vemos se o id existe. Se não existir, o *user* é redirecionado para **"/error"**. Caso contrário, indexamos todos os links encontrados com a **opcaoUm()** do servidor.

## Adminpage

Método para redirecionar o *user* para a página de administração.

## TopStories

Mais um método que utilizamos para funções da **api rest**. Neste caso usamos a função **topStories** que devolve vários links para serem indexados.

## OnMessage

Este método é ativado quando uma mensagem é recebida, sendo responsável por enviar essa mesma mensagem para a **Web Socket** em **"topic/messages"**.

Em maior parte destas funções, usamos a anotação **@GetMapping** para definir que o método em questão vai ser executado quando há uma requisição **HTTP GET** feita por uma determinada por um *URL*.

Tivemos de criar vários templates e ficheiros do tipo **.js** e **.css** para um melhor frontend da **springBoot**.

---

## WebSockets

Para a configuração da **Web Socket** criamos o ficheiro **WebSocketConfig.java**, onde definimos o **endpoint** como **"/my-websocket"**, o prefixo da aplicação como **"/app"** e o prefixo de destino **"/topic"**.

De seguida, para integrar os **web sockets** com a nossa página de administração, criamos o ficheiro **adminpage.js**. Neste ficheiro utilizamos 3 métodos:

### Connect()

Neste método ligamos à nossa **socket**, definimos como conectado e de seguida mostramos a **mensagem** recebida pela **socket**.

### SetConnected()

Método responsável por definir a ligação como conectada ou não conectada

Chamamos a função **connect()** ao carregar a página de administração de forma a estabelecer a ligação com a **Socket** e mostrar as **mensagens** recebidas.

## REST

Relativamente à parte da integração com serviço **REST** criamos um ficheiro java para tratar das duas funcionalidades pedidas.

Primeiramente criamos a função **topStories**. Nesta função, começamos por criar uma conexão **HTTP** com o **url** das **top stories**. Depois fazemos um **"GET"** para receber os **topStories** e fechamos a conexão para começar a tratar dos dados recebidos. De seguida, removemos todos os caracteres a mais dos resultados obtidos para ficarmos com uma

---


string que contém os números todos. Para cada número obtido fazemos outro pedido “GET” onde obtemos os **links** que correspondem àquele número. Tal como fizemos anteriormente, tratamos dos dados para obtermos apenas os **links**. Para cada número fechamos a conexão que tivemos de fazer para obter o *url* correspondente ao número.

Criamos também outra função, **userStories()** onde recebemos o *id* de um dado utilizador do *Hacker News* e obtemos as **top stories**. Tal como na função anterior tivemos de fazer um “GET” e obter uma conexão para obtermos os users stories. Depois de obtermos os **user stories** fechamos a conexão e tratamos dos dados. Por fim, obtemos os *urls* com outro “GET” e uma nova conexão e enviamos estes.

## Testes Realizados

Descrição	 
Indexar um novo URL	
Pesquisar páginas que contenham um conjunto de termos	
Consultar lista de páginas com ligação para uma página específica	
Pesquisar páginas que contenham um conjunto de termos	
Resultados de pesquisa ordenados por importância	
Comunicação RMI entre o search module e o spring-boot	
Programa a correr na mesma máquina	
Programa a correr em diferentes máquinas	
[WebSockets] Página de administração atualizada em tempo real	
[REST] Indexar URLs das top stories que contenham os termos de pesquisa	
[REST] Indexar todas as “stories” de um utilizador	
Correr programa com vários barris	
Login e logout	
Programa não correr sem search module	

---

Programa não correr sem main (barris e downloaders ativos)	
Programa não correr sem search module e sem main (barris e downloaders ativos)	