Aprimorando o Buscadir: Sistema para Buscas em Tribunais Regionais Federais

Lucas Bastos Suchorski, Alexandre Zamberlan
Curso de Ciência da Computação
UFN - Universidade Franciscana
Santa Maria - RS
lucas.bastos@ufn.edu.br, alexz@ufn.edu.br

***Resumo*—Este trabalho integra pesquisas do Direito na Universidade Franciscana (UFN) sobre Tecnologia da Informação. Anteriormente, um Sistema *Web Crawler* foi desenvolvido, implementado e disponibilizado para buscar informações jurídicas em sites de Tribunais de forma autônoma. No entanto, implicações surgiram (principalmente CAPTCHA) que fizeram com que o *Crawler* não pudesse ser totalmente explorado. Para isso, duas novas alternativas, utilizando a linguagem Python, são propostas: envio de requisição via HTTP/Request e utilização da API do Datajud (base nacional de dados do poder judiciário). Ponto de destaque é que o robô não foi totalmente inviabilizado e ainda pode ser necessário para manipulação da resposta de alguns tribunais.**

***Palavras-chave*: Injeção/Extração de dados; *Scraping*; Python; *Requests*; Tribunais de Justiça**

I. Introdução

Este trabalho está inserido nas pesquisas realizadas na área do Direito da Universidade Franciscana (UFN) que trabalham com questões que envolvam Tecnologia da Informação, em especial a Inteligência Artificial. Os projetos existentes são: *Técnicas de Inteligência Artificial Aplicadas ao Direito: Representação de Conhecimento e Raciocínio Automatizado*; e *Direito à Saúde e à Educação de Crianças e Adolescentes em Tempos De Pandemia: a atuação dos Entes Públicos Brasileiros na efetivação de Direitos Fundamentais*.

Esses projetos possuem fases que necessitam de buscas relevantes em sites específicos, como Tribunais de Justiça. Para auxiliar as buscas, foi projetado, implementado e disponibilizado o Sistema *Web Crawler* para pesquisas relevantes em sites da área do Direito [1], disponível em https://buscadir.lapinf.ufn.edu.br/. Entretanto, há inúmeras melhorias a serem projetadas, implementadas e avaliadas nesse sistema, como por exemplo, otimizações de processamento, tratamento automatizado de alguns campos, tratamento de *CAPTCHA*, entre outros.

Para uma compreensão aprofundada, um site do Tribunal de Justiça Federal geralmente incorpora diversos mecanismos de busca abrangendo diferentes áreas dos tribunais, incluindo processos e julgamentos, bem como campos como pesquisa livre, ementa, número do recurso, assunto e data do julgamento[1]. O propósito desses mecanismos é promover a transparência na operação do Poder Judiciário, disponibilizando informações pertinentes tanto para os profissionais da área jurídica quanto para o público em geral. Essa transparência é essencial para garantir a acurácia e a confiança no sistema judicial, além de facilitar o acesso à justiça e promover a democracia.

[1]Para exemplificar, há o site do Tribunal de Justiça Federal do Estado de São Paulo - https://esaj.tjsp.jus.br/cjsg/consultaCompleta.do.

A. Justificativa

A pesquisa realizada por Eduardo Palharini e Alexandre Zamberlan [1] deixou alguns trabalhos futuros, que são limitações do processo de busca em Tribunais Federais e necessidades de melhorias na ferramenta desenvolvida. Dessa forma, faz-se necessário trabalhar qualificando alguns pontos, como melhorar a interface enquanto o sistema executa uma busca, detecção automática de campos pelo *crawler*, entre outros.

B. Objetivos

Portanto, o objetivo principal deste trabalho é projetar, implementar e avaliar soluções adicionais e possíveis para as limitações de *backend* presentes no Sistema *Web Crawler* para buscas relevantes em sites da área do Direito.

Os objetivos específicos são:

- Pesquisar sobre *Web Crawler* e *CAPTCHA*;
- Buscar alternativas de códigos que auxiliem na centralização das pesquisas realizadas pelo sistema;
- Estudar e testar o ambiente do Sistema *Web Crawler* para buscas relevantes em sites da área do Direito [1];
- Localizar nos códigos do sistema os pontos onde as soluções devem ser implementadas;
- Definir e aplicar um ambiente de avaliação e teste.

Por fim, o texto foi dividido em 4 seções: Revisão Bibliográfica, Metodologia do Trabalho, Conclusões e Referências Bibliográficas. Na primeira seção, é apresentado uma compilação geral dos conceitos necessários para o entendimento do estudo, bem como ferramentas utilizadas para este processo, além dos desafios encontrados. Na segunda, a modelagem do trabalho desenvolvido, tecnologias para organização, modelagem da solução e protótipos de interface. A terceira contém conclusões acerca do que foi trabalhado e apontamentos considerados importantes. Na última seção, são mostradas as fontes da pesquisa em questão.

II. Revisão bibliográfica

Nesta seção, encontram-se explicações sobre os fundamentos necessários para o cumprimento do objetivo deste trabalho.

A. Monitoramento, análise e extração de conteúdo

De acordo com Lawson [2], considerando que uma pessoa tenha uma loja qualquer e deseja manter o controle de preços dos produtos dos competidores, ela teria que acessar os catálogos de preços dos produtos desses competidores, todos os dias, para comparar valores, demandando tempo considerável, principalmente se há um número de concorrentes significativo.

Para se ter um bom controle dessas informações, é necessário que estejam dispostas de forma organizada. Para tanto, é importante que se busque informações condizentes com aquilo que se deseja conhecer. Partindo desta premissa, surge a ideia de utilizar *Web Crawler*, que consiste, basicamente, em um software que acessa páginas de Internet, extrai e traz informações de forma rápida e eficiente [3]. O processo de extração e coleta de dados tem alguns nomes, como por exemplo: *scraping*; rastreamento; *crawling*; desmembramento, etc.

B. O papel da Inteligência Artificial no monitoramento, análise e extração de conteúdo

É fundamental registrar que existe uma relação entre a técnica de *Web Crawler* e a área de Inteligência Artificial (IA). A relação com a IA ocorre principalmente na análise e processamento dos dados coletados pelo *crawler*. Citam-se algumas formas da relação com IA:

- Processamento de Linguagem Natural (PLN): após coletar dados de páginas da Web, é comum que esses dados precisem ser analisados e compreendidos. A PLN, uma subárea da IA, é utilizada para extrair significado do texto coletado, identificar entidades, realizar análises de sentimento, entre outras tarefas [4];
- Aprendizado de Máquina para Classificação e Categorização: o aprendizado de máquina é frequentemente usado para classificar e categorizar os dados coletados pelo *crawler*. Por exemplo, um *crawler* pode ser usado para coletar resumos de processos jurídicos julgados em Tribunais de Justiça e, em seguida, um modelo de aprendizado de máquina pode ser treinado para classificar automaticamente esses processos julgados em diferentes categorias do Direito, como Ambiental, Civil, Contratual, Digital, Consumidor, Eleitoral, entre outras;
- Recomendação de Conteúdo: Com base nos dados coletados pelo *crawler* e nas preferências do usuário, sistemas de recomendação baseados em IA podem ser desenvolvidos para sugerir conteúdo relevante. Como exemplo, pode-se citar, o portal de buscas em Tribunais de Justiça, em que numa determinada área do Direito o sistema pode recomendar as melhores jurisprudências[2] utilizadas em processos listados pelo *crawler*.

C. Ferramentas para Web Crawler

As ferramentas apresentadas, em sua maioria, têm em comum a capacidade de acessar uma página da Web de destino e extrair elementos de texto no formato *Hypertext Markup Language* (HTML), com base nos parâmetros fornecidos de pesquisa. Cada uma dessas ferramentas possui suas próprias características distintas e é projetada para funcionar de maneira eficaz em diversos cenários.

1) Scrapy: é um *framework* de conexão e desmembramento em Python eficiente para extrair dados de sites de forma estruturada e automatizada. É especialmente útil quando se necessita extrair grandes volumes de dados de várias páginas de Internet (desmembramento), sendo ideal para projetos mais complexos. Suas características incluem o gerenciamento automático de *cookies*, agendamento de solicitações, suporte a *proxies*, tratamento de erros sem interromper a execução e outras funcionalidades. Além disso, é capaz de interagir de forma integrada com ferramentas de processamento e mineração de dados, ampliando ainda mais sua utilidade [6].

Segundo a documentação do *framework* [6], há classes e métodos importantes para a construção de um *Web Crawler*:

- classe *Spider*: responsável para extrair dados dos sites. Ela define como o *scraping* deve ser realizado, incluindo a estrutura das *Uniform Resource Locator* (URLs)[3] a serem rastreadas, como os dados devem ser extraídos e como os dados extraídos devem ser processados ou armazenados;
- classe *Request*: faz as solicitações *Hypertext Transfer Protocol* (HTTP) para URLs específicas. Pode ser usada para iniciar o processo de *scraping* e para seguir links para outras páginas. Um pouco mais elaborada que a nativa do Python;
- método *selector*: seleciona partes específicas do HTML de uma página da Web para extração. Pode ser usado para encontrar elementos específicos, como *forms*, *inputs*, *submits*, links, texto, entre outros;
- classe *Pipeline*: processa, armazena, valida e limpa os itens extraídos em diferentes formatos, como *Comma-Separated Values* (CSV), *JavaScript Object Notation* (JSON) ou em um banco de dados;
- componente *middleware*: processa solicitações e respostas HTTP, adicionando funcionalidades, como roteamento de solicitações, manipulação de *cookies* ou uso de *proxies*.

[2]Em um Tribunal Federal, jurisprudência refere-se ao conjunto de decisões judiciais tomadas por esse tribunal em casos anteriores. Essas decisões estabelecem precedentes legais que orientam futuras decisões judiciais sobre questões semelhantes [5].

[3]Um URL refere ao endereço de rede no qual se encontra algum recurso computacional, como por exemplo um arquivo de computador ou um dispositivo periférico [5].

2) Beautiful Soup: biblioteca Python para analisar documentos HTML e *Extensible Markup Language* (XML), também focada em extrair dados e manipular a estrutura HTML de forma mais flexível ou projetos menores. Pode não ser uma boa alternativa para projetos grandes devido à simplicidade. Além disso, como Scrapy, ela consegue exportar os objetos para um arquivo JSON ou CSV [7]. As principais classes utilizadas nessa biblioteca são:

- classe Beautiful Soup: responsável por controlar elementos extraídos por meio dos métodos *find*, *find_all* e atributos como *text*;
- classe Requests: utilizada para acessar determinado *link*, bem como servir de parâmetro para a biblioteca.

Vale ressaltar, que no momento da inicialização do objeto de invocação da biblioteca, determina-se qual tipo de objeto será utilizado (XML, HTML, HTML, etc).

3) Selenium: biblioteca de automação suportada por diversas linguagens como JavaScript, Python, C++, HTML (por meio de um *driver* de navegador). No caso do contexto deste trabalho, ela permite emular interações humanas com uma página Web. Ela é útil quando se precisa extrair e/ou injetar elementos de sites que são dinâmicos e interativos, como páginas geradas por JavaScript. O Selenium pode ser usado para simular cliques, preenchimento de formulários, *scroll*, entre outras interações [8]. Para isso, utiliza-se a classe *Webdriver*. Primeiramente, define-se qual navegador é utilizado (Chrome, Firefox, Edge) e, em seguida, ela conecta, extrai e armazena todo conteúdo do site dentro de um objeto *Webdriver*.

Com isso, basta selecionar o elemento da página que se deseja interagir por meio dos métodos *send_keys* (com auxílio da classe *Keys* do *Webdriver*) e *find_element_by_id*, *find_element_by_name*, *find_element_by_tag*, *find_element_by_link*.

É possível concluir que Selenium é útil para páginas dinâmicas carregadas com JavaScript. O Scrapy é uma ferramenta robusta para grandes projetos de *web scraping*. Por fim, Beautiful Soup é simples e direto para análise de HTML estático. Dessa forma, a escolha da ferramenta deve ser conforme as necessidades específicas com base na complexidade do projeto de *Web Crawling*.

D. Exemplos

Com base nas Figuras 1, 2 e 3 é possível observar a simplicidade de se acessar um site, neste caso o https://www.python.org/ e obter dados a partir do *crawler*, utilizando as três tecnologias citadas neste estudo.

Um ponto que deve ser levado em conta é que a *Beautiful Soup* sozinha não acessa o site, é preciso utilizar a biblioteca *requests*.

Vale ressaltar que o Selenium é o único que possui a funcionalidade de injeção/interação com a página, por meio da classe *Keys*.

```
import scrapy
from scrapy.crawler import CrawlerProcess

class PythonOrgSpider(scrapy.Spider):
    name = 'python_org_spider'
    start_urls = ['https://www.python.org/']

    # Método de comunicação entre a saída e a response (crawler)
    def parse(self, response):
        title = response.css('title::text').get()
        yield {
            'title': title,
        }

process = CrawlerProcess()
process.crawl(PythonOrgSpider)
process.start()
```

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS
2024-05-22 15:14:37 [scrapy.extensions.logstats] INFO: Crawled 0 pages (at 0 pag
2024-05-22 15:14:37 [scrapy.extensions.telnet] INFO: Telnet console listening or
2024-05-22 15:14:37 [scrapy.core.engine] DEBUG: Crawled (200) <GET https://www.p
2024-05-22 15:14:37 [scrapy.core.scraper] DEBUG: Scraped from <200 https://www.p
{'title': 'Welcome to Python.org'}
2024-05-22 15:14:37 [scrapy.core.engine] INFO: Closing spider (finished)
```

Figura 1. Exemplo de código Scrapy [6].

```
import requests
from bs4 import BeautifulSoup

#Método que etorna um objeto Response
pagina = requests.get('https://www.python.org/')
# Atribuição do html
html = pagina.text
# Conversão do html para um objeto Beautiful Soup
soup = BeautifulSoup(html, 'html.parser')
# Exibição do título da página
print(soup.title.string)
```

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS

PS C:\Users\lsbas\OneDrive\Desktop> & C:/Users/lsbas/AppData/L
Welcome to Python.org
```

Figura 2. Exemplo de código Beautiful Soup [7].

```
from selenium import webdriver
# Inicializando o driver
driver = webdriver.Edge()
# Acessando o site e obtendo os dados da página
driver.get('https://www.python.org/')
# Exibição do título da página
print(driver.title)
```

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS
PS C:\Users\lsbas\OneDrive\Desktop> & C:/Users/lsbas/AppData

DevTools listening on ws://127.0.0.1:59254/devtools/browser/
[37428:38108:0522/151603.627:ERROR:edge_auth_errors.cc(520)]
econdary Error: kAccountProviderFetchError, Platform error:

Welcome to Python.org
```

Figura 3. Exemplo de código Selenium [8].

E. Desafios do Web Crawler

Existe um estudo na área do Direito chamado Jurimetria. Este estudo consiste, de forma simplificada, em uma mineração de dados para área jurídica. Serve como ferramenta de otimização para processos e administração de tribunais, bem como cálculo de riscos [9]. Para adquirir e analisar estes dados, nada melhor que um *Web Crawler*, tendo em vista a necessidade de se congregar informações.

A Associação Brasileira de Jurimetria (ABJ) possui um estudo que utiliza a linguagem R como ferramenta. Ela enfrenta, em diversos cenários o mesmo problema: o CAPT-

CHA. Os tribunais, segundo a associação [9], costumam utilizar este mecanismo para não onerar os sistemas ou para propagar uma ideia de proteção aos usuários. Entretanto, não faz muito sentido tendo em vista que os arquivos são de acesso público.

Apesar do argumento, vale ressaltar que os sites possuem suas respectivas políticas e diretrizes estabelecidas para uso. Além da sobrecarga, o CAPTCHA funciona como uma proteção contra atividades maliciosas como manipulação de dados e possíveis informações sensíveis. Portanto, deve-se respeitar as diretrizes.

Em fóruns de discussão, Pedro Guimarães [10] realiza uma reflexão bastante interessante com base no artigo "154-A Invadir dispositivo informático alheio, conectado ou não à rede de computadores, mediante violação indevida de mecanismo de segurança e com o fim de obter, adulterar ou destruir dados ou informações sem autorização expressa ou tácita do titular do dispositivo ou instalar vulnerabilidades para obter vantagem ilícita". Ele menciona que esse processo de verificação se torna uma forma de contorno burocrático, complementa que o propósito de quem burla não é de obter, adulterar ou destruir dados sem autorização, haja vista que estas informações já serão disponibilizadas; serve mais como um bloqueio para acesso em lote (*batch*) de dados. Além disso, destaca sobre a oferta de APIs para "burlar"o CAPTCHA pelo próprio site. Uma analogia/resposta, feita por Felipe Sanches [10], simplifica bem a reflexão "(...) captcha não é cadeado. Captcha é lombada.".

A biblioteca Scrapy não possui recursos internos para manipular diretamente com CAPTCHAs, afinal, eles são projetados para impedir a automação de atividades na Web, como a extração de dados por meio de *Web Crawlers*.

Ao mesmo tempo, o Selenium, que é uma ferramenta muito utilizada para automação de testes em navegadores Web (incluindo a interação com elementos da página, preenchimento de formulários e clique em botões) também não possui funcionalidades específicas para lidar com CAPTCHAs.

No entanto, há algumas abordagens que se pode considerar para tratar CAPTCHAs ao usar o Scrapy ou Selenium, como: i) uso de serviços terceirizados de resolução de CAPTCHA. Alguns exemplos são 2Captcha e Anti-Captcha; ii) ou desenvolver soluções personalizadas que envolva o uso de técnicas de visão computacional[4] e aprendizado de máquina para identificar e resolver automaticamente.

É importante ressaltar que resolver de forma automatizada CAPTCHAs pode violar os termos de serviço de alguns sites e pode ser considerado antiético ou ilegal.

F. Trabalhos Relacionados

Esta seção tem como objetivo elencar e discutir alguns trabalhos que possuem tecnologia semelhante a tecnologia utilizada na solução ou por possuírem contexto técnico deste estudo.

O trabalho realizado por Palharini e Zamberlan [1], contempla o projeto de um sistema automatizado de rastreamento de conteúdo em sites e/ou portais jurídicos. O sistema aplica um *crawler* (bot de pesquisa) para coletar dados, como título, descrição e data de publicação. Ele foi desenvolvido para facilitar a busca de informações por profissionais do Direito nos Tribunais de Justiça do Brasil. O projeto empregou a linguagem Python, os *frameworks* Django, Bootstrap e Scrapy, a biblioteca Beautiful Soup e a API Selenium. Como forma de estruturação, foi utilizado o modelo MVT (*Model-View-Template*), que, no Django, consiste numa abordagem de projeto (*design*) que separa as diferentes partes de um aplicativo Web para facilitar o desenvolvimento e a manutenção de sistemas.

Como resultado há um sistema piloto disponível em https://buscadir.lapinf.ufn.edu.br/ com gestão de usuários, áreas do Direito, fontes de pesquisa, instituições e buscas.

Outro trabalho relacionado é o sistema Digesto para busca em portais do Direito [11], que é uma ferramenta desenvolvida pela Jusbrasil[5]. O sistema é uma plataforma que utiliza Inteligência Artificial e tecnologia de Processamento de Linguagem Natural para as buscas e os acessos a conteúdos jurídicos. O objetivo do Digesto, via seu portal, é fornecer aos seus usuários uma maneira simples de encontrar informações relevantes em documentos legais distribuídos na Web. Ele permite que os usuários realizem buscas por termos específicos, temas, jurisprudência, doutrina e legislação, entre outros aspectos relacionados ao Direito. Por meio de computação, o Digesto é capaz de analisar e classificar os documentos legais, identificando sua relevância e contexto. Entretanto, esse sistema tem custo de assinatura que variam conforme entrevista realizada pela empresa[6], além de ter foco em empresas de médio e/ou grande porte.

G. Contornando Limitação de CAPTCHA

Além do *Crawler*, é possível utilizar um mecanismo de requisições que os navegadores utilizam. O principal motivo da escolha desta abordagem deve-se ao fato de o *Selenium* não trazer a resposta após a busca devido o CAPTCHA/token. Os campos são preenchidos, entretanto a resposta não aparece com os dados da busca. Os navegadores, para comunicar com os sites solicitados, utilizam requisições HTTP. Esta comunicação possui duas partes principais: *Request* e *Response*, que são, basicamente, pergunta/requisição (cliente) e resposta (servidor). Dentro da requisição, existem campos que são preenchidos com valores que serão tratados pelo servidor, resultando na montagem do HTML pelo navegador (resposta). Sendo eles:

[4]Reconhecimento ótico de caracteres ou *Optical Character Recognition* (OCR).

[5]Jusbrasil é uma empresa brasileira que oferece serviços jurídicos e informações legais.

[6]Para descobrir valores de planos, é preciso solicitar orçamento informando dados da empresa contratante.

- Método: na grande maioria, os mais utilizados são "*POST*[7]" ou "*GET*[8]";
- URL: caminho o qual a requisição passará;
- *headers*: campos que o servidor espera receber para validar a requisição;
- *payload*: campos que serão tradados pelo servidor para montagem da resposta.

Os tipos de valores que são enviados nos *headers* e *payloads* variam de acordo com cada site, mas normalmente são utilizados no formato JSON[9] (*JavaScript Object Notation*), comumente descritos no campo *Content-Type*. Os elementos citados nesta seção podem ser encontrados pelo navegador através do *console* na aba *network*, normalmente pelo atalho F12.

H. Utilização de API Datajud

Uma terceira solução é possível, utilizando uma API disponibilizada pelo Conselho Nacional de Justiça (CNJ), o Datajud [12]. Ela funciona de forma semelhante ao sistema deste trabalho, onde os dados podem ser adquiridos com base na seleção do tribunal ou até mesmo por região do Brasil.

Entretanto, apesar de retornar um grande número de informações, a relevância destas não aparenta ser suficiente para o que foi solicitado e proposto para este sistema. Ainda assim, existem outras maneiras de se utilizar a API através de extensões como o *Kibana* e a API *Elastic*, que sugerem ter informações mais complexas. Contudo, elas não são cedidas a desenvolvedores individuais, é necessário um intermédio de órgão do Poder Judiciário, segundo a documentação [12].

I. Considerações sobre Revisão Bibliográfica

Apesar da solução principal para a proposta do sistema utilizar um *Crawler* como ferramenta, em muitos tribunais, o *CAPTCHA* ou utilização de *tokens*[10] acabou limitando a interação com os tribunais. Por conta disso, tornou-se necessário uma nova abordagem para captar os processos com base na pesquisa realizada. Importante ressaltar que o *scraping* ainda será utilizado, pois alguns tribunais emitem como resposta HTML que podem e serão manipulados pelas ferramentas citadas neste trabalho.

A Tabela I, mostra o mapeamento realizado, avaliado caso a caso, da possibilidade de utilização do método proposto via requisição HTTP, bem como as respostas encontradas de cada tribunal. A terceira coluna (promissor) entende-se como a viabilidade de solução, devido ao fato de que alguns campos que são enviados nas solicitações, apesar de serem chaves *token*, por exemplo, não são gerados a cada acesso. Alguns, inclusive, são fixos, o que possibilita a implementação.

Os critérios utilizados para definir se a solução é válida (ou não) são:

- presença *token* gerados a cada busca;
- se a resposta possibilita tratamento dos dados;
- presença de CAPTCHA para realização da busca; e
- se todos os campos do *header* e/ou *payload* podem ser preenchidos;

Tabela I
MAPEAMENTO/PLANEJAMENTO DOS TRIBUNAIS MÉTODO HTTP

Tribunais	Captcha/Token	Response	Promissor
TJAC	Não	HTML	Talvez
TJAL	Não	HTML	Talvez
TJAM	Não	HTML	Talvez
TJBA	Não	JSON	Sim
TJCE	Sim	HTML	Não
TJES	Não	HTML	Sim
TJGO	Sim	HTML	Não
TJMA	Sim	HTML	Não
TJMT	Não	JSON	Sim
TJMS	Sim	HTML	Não
TJMG	Sim	HTML	Não
TJPA	Não	JSON	Sim
TJPB	Sim	JSON	Não
TJPR	Não	HTML	Sim
TJPE	Não	HTML	Talvez
TJPI	Não	HTML	Sim
TJRS	Não	HTML	Sim
TJRJ	Sim	JSON	Talvez
TJRN	Não	JSON	Sim
TJRO	Não	HTML	Talvez
TJRR	Sim	JSON	Talvez
TJSC	Não	HTML	Sim
TJSP	Sim	HTML	Não
TJSE	Sim	HTML	Não
TJTO	Não	HTML	Sim
TJDF	Sim	HTML	Talvez

Com base no texto de revisão, no contexto do trabalho do Sistema *Web Crawler* para buscas relevantes em sites da área do Direito [1] e na análise dos trabalhos relacionados, destacam-se alguns pontos de decisão:

- o *Web Crawler* deve conter ações automatizadas para extração, injeção e interação em páginas Web, logo a ferramenta Selenium e o *framework* Scrapy são interessantes, uma vez que o sistema possui complexidade (sites com grande quantidade de dados e com recursos

[7]Parâmetros enviados de forma "escondida" dentro da URL
[8]Parâmetros enviados de forma "exposta" dentro da URL
[9]Estrutura de dados {"chave":"valor'} agrupados em pares para comunicação
[10]Emaranhado de letras e símbolos que servem como chave para alguma ação.

de *cookies*[11], *proxies*[12], *CAPTCHA*);

- o *Web Crawler* deve possuir suporte para tecnologias Web HTML e JavaScript, já que os sites dos Tribunais de Justiça as utilizam;
- com base na segunda solução proposta, foi mapeado (Tabela I) todos os tribunais quanto ao tipo de resposta e formato de entrada, a fim de organizar o planejamento e tratamento das funcionalidades para a comunicação entre o sistema e todos os tribunais selecionados na busca;
- embora haja a solução utilizando a API Datajud, os dados retornados por ela não são suficientes para a finalidade do sistema para desenvolvedores individuais. A documentação [12] cita os mecanismos Kibana e API Elastic disponibilizados pelo CNJ (Conselho Nacional de Justiça), porém eles não são oferecidos sem uma solicitação direta por um órgão do Poder Judiciário.

III. Metodologia do Trabalho

Em relação à realização da pesquisa, a metodologia *Scrum* [14] foi utilizada, bem como a técnica *Kanban* para a gestão das atividades assumidas no cronograma. Como ferramentas e técnicas de apoio, citam-se:

- GitHub: repositório para os códigos construídos;
- Visual Studio Code;
- Linguagem Python;
- Biblioteca Selenium;
- *Framework* Scrapy.

A. Modelagem da Solução

Como já mencionado, o sistema desenvolvido por Palharini e Zamberlan [1] possui pontos que precisam ser refatorados e melhorados, como:

1) No cadastro de busca, implementar a injeção de datas inicial e final de publicação para o *crawler*;
2) No cadastro de busca, dividir o cadastrar a busca e o realizar a busca em dois processos distintos;
3) No processo de realizar a busca, fazer que o *crawler* navegue/entre nas abas das páginas resultados das pesquisas nos Tribunais Federais;
4) No processo de mostrar resultado da busca, adequar o link do PDF trazido no resumo do resultado;
5) No cadastro de fonte de pesquisa, criar um processo melhorado para identificar os campos de um formulário de pesquisa de um Tribunal de Justiça para que o *crawler* faça a injeção dos dados;
6) No cadastro de fonte de pesquisa, testar outros *frameworks* e/ou *Application Programming Interface* (API) para identificar automaticamente os campos de pesquisa a serem utilizados pelo *crawler*;
7) No cadastro de fonte de pesquisa, testar outros códigos baseados em *frameworks* e/ou API (Selenium, Scrapy e Beautiful Soup) para qualificar a identificação ou não da presença de *CAPTCHA* (*Completely Automated Public Turing test to tell Computers and Humans Apart*) e suas variações.

Devido ao fato de o projeto estar iniciado, possuir um esqueleto, tanto da parte visual, quanto estrutural, a proposta é que seja aprimorada a parte funcional do Buscadir. Para ilustrar essa implementação, associa-se um pacote ao caso de uso proposto para o trabalho primário.

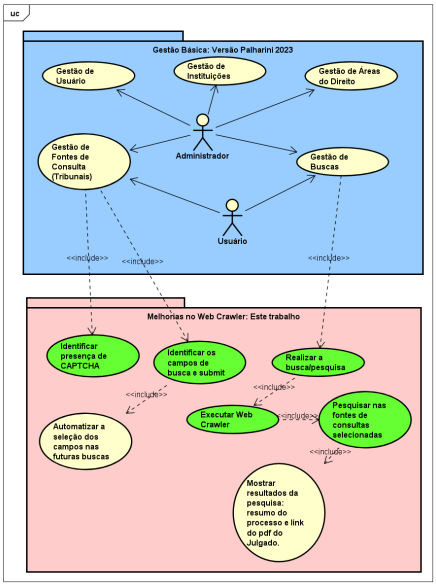

Figura 4. Casos de Uso com Funcionalidades antigas [1] e novas.

Na Figura 4, há dois pacotes: um representando o projeto desenvolvido por Palharini e Zamberlan [1] (parte superior) e outro contendo funcionalidades de aprimoramento do *Web Crawler* (parte inferior), foco deste trabalho. Destaca-se que as funcionalidades em verde (*Identificar presença de CAPTCHA*; *Identificar os campos de busca e submit*; *Realizar a busca/pesquisa*; *Executar Web Crawler*; e *Pesquisar nas fontes de consultas selecionadas*) foram projetadas e parcialmente implementadas por Palharini e Zamberlan [1], mas que também são o objetivo deste trabalho, pois devem receber melhorias como as funcionalidades: *Automatizar a seleção dos campos nas futuras buscas* e *Mostrar resultados da pesquisa*.

Na Figura 5, é possível visualizar como o sistema *Web Crawler*, previamente desenvolvido, está organizado conforme as boas práticas do modelo MVT. Registra-se que as melhorias devem ocorrer nas pastas apps (camada de modelo) e na pasta aplicativo (camada de regra de negócio, conhecida como *view*).

[11] Arquivos que os navegadores guardam contendo informações pessoais para uso posterior [13].

[12] Mecanismo de desvio de tráfego de rede para registrar atividades de tráfego ou economizar conexão [13].

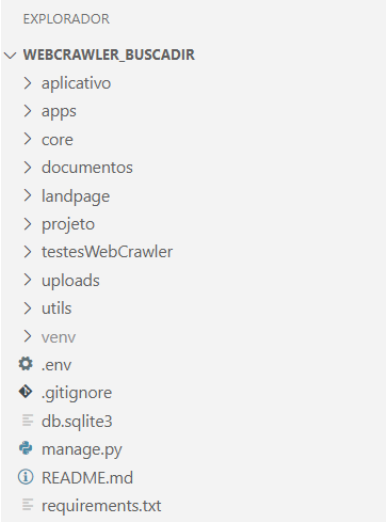

Figura 5. Organização de pastas e arquivos (modelo MVT) [1].

Para um melhor entendimento da organização estrutural do projeto:

- pasta aplicativo: arquivos das camadas *Template* (.html e .css) e *View* (.py);
- pasta apps: arquivos da camada *Model* (.py);
- pasta core: arquivos e pastas de gestão da comunicação básica do sistema com o usuário;
- pasta documentos: contém a documentação do sistema, como diagramas e lista dos Tribunais de Justiça;
- pasta *landpage*: arquivos e pastas para o site de abertura (boas vindas) do sistema;
- pasta projeto: arquivos e pastas de configuração do sistema, por exemplo, idioma, fuso horário, banco de dados e link de acesso ao arquivo .env;
- pasta testesWebCrawler: arquivos Python para testes diversos;
- uploads: pasta contendo todos os arquivos carregados no sistema *Web Crawler* pelos usuários;
- pasta utils: arquivos Python com classes responsáveis por geração de *hash*, validação email, *decorators*, entre outros;
- pasta venv: conjunto de arquivos e pastas que simulam um ambiente virtualizado, como um *container* ou *docker*;
- arquivo .env: contém usuários, senhas e *drivers* de comunicação do sistema;
- arquivo .gitignore: relação de arquivos e pastas que não devem ser enviados ao servidor Git;
- arquivo db.sqlite3: arquivo que contém as informações salvas no banco de dados;
- arquivo manage.py: contém as principais funcionalidades de gestão de um sistema Python-Django. Com ele é possível gerar migrações, executar servidor local e acessar o ambiente shell (terminal) do banco de dados;
- arquivo README.md: arquivo de informações gerais, como contexto do sistema;
- arquivo requirements.txt: arquivo com bibliotecas a serem instaladas na venv para instalação/utilização do sistema;

B. *Resultados e Discussões*

No decorrer da pesquisa, mais precisamente no ano de 2024, vários sites de Tribunais Federais mudaram suas políticas de segurança e uso de CAPTCHA, como por exemplo o site do Tribunal Federal do Estado de São Paulo, que até 2023 não trabalhava com CAPTCHA para buscas nos seus campos de pesquisa, que agora, utiliza. Isso gerou implicações principalmente para o uso da biblioteca *Selenium*, que durante os testes retornava o HTML sem os resultados da pesquisa. Buscou-se então apresentar alternativas paliativas como a modelagem pela biblioteca Python nativa *requests* e uso da API Datajud. Portanto, dos 27 tribunais, mesmo com as melhorias deste trabalho, há inúmeros tribunais que não receberão as buscas automatizadas do sistema desenvolvido.

Apesar de ser um dos objetivos a automatização nas buscas dos campos, a solução via HTTP/*requests* não foi implementada de forma automática, ou seja, para a montagem das fontes, o preenchimento do *payload*, até o momento, deve ser feito de forma manual realizando uma busca "real" e procurando os resultados para a montagem da requisição pelo sistema. Porém, ainda há espaço para tal ação em trabalhos futuros, pelo fato do trecho de código onde isso ocorre estar genérico.

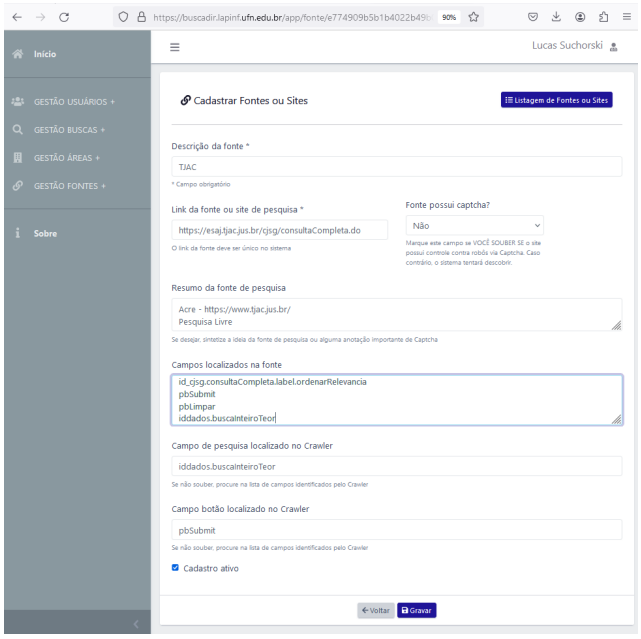

Figura 6. Interface inicial do formulário de Fontes de Busca [1].

As melhorias pretendidas no trabalho passam por interfaces já projetadas, implementadas e disponibilizadas, como o caso da tela da Figura 6. Já na Figura 7, há

melhorias vinculadas às funcionalidades *Identificar presença de CAPTCHA; Identificar os campos de busca e submit; e Automatizar a seleção dos campos nas futuras buscas.*

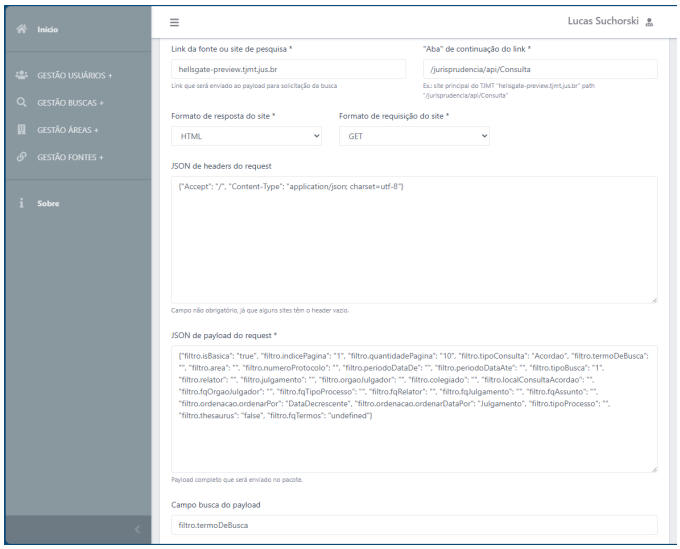

Figura 7. Interface do Formulário Cadastro ou Atualização de Fonte de Busca [1].

Como apresentado na Seção III - Modelagem da Solução, buscou-se implementar e discutir as soluções. Para a resolução via requisição, além de alterar os campos das Fontes para se adequar aos novos campos da requisição, há também a lógica genérica para captação da resposta, no arquivo views.py do aplicativo Pesquisa, conforme Figura 8.

```
for fonte in fontes_form:
    # Preparação da requisição
    payload = fonte.request_payload
    headers = fonte.request_headers
    path = fonte.request_path
    # Adicionando campos do form dentro do payload
    palavra_chave = formulario.palavras_chave
    if fonte.request_method == 'GET':
        palavra_chave = formulario.palavras_chave.replace(' ',f'%20')
    payload[fonte.campo_busca] = palavra_chave
    if formulario.data_publicacao_inicio:
        payload[fonte.campo_data_inicial] = formulario.data_publicacao_inicio.strftime('%Y-%m-%d')
    if formulario.data_publicacao_fim:
        payload[fonte.campo_data_final] = formulario.data_publicacao_fim.strftime('%Y-%m-%d')
    # Busca propriamente
    conn = http.client.HTTPSConnection(fonte.link)
    if fonte.token:
        payload['token'] = fonte.token
    if fonte.request_method == 'GET':
        path += dict_values_to_str(payload)
        payload = ''
    conn.request(fonte.request_method, path, payload, headers)
    response = conn.getresponse()
    bresult = response.read()
    text = bresult.decode()
    formulario.resultado_texto = text
    formulario.save()
    print(f'{fonte.sigla} finalizado com sucesso.')
```

Figura 8. Código para envio de requisição na pesquisa.

Também foi adicionado uma função auxiliar para os casos onde o método é o GET pela Figura 9; um método simples onde se cria um texto seguindo os valores do dicionário que no fim é concatenado à URL.

```
def dict_values_to_str(payload):
    txt = '?'
    for key,value in payload.items():
        txt+=key+'='+value+'&'
    return txt[:-1]
```

Figura 9. Função para converter dicionário em *string* para parâmetro.

Houveram dúvidas durante a implementação se o método deveria ter sido inserido dentro da classe Pesquisa ou dentro do arquivo *views.py*. Porém, foi definido dentro do *views* pelo fato de ser necessário utilizar e alterar o campo antes que ocorra o salvamento do formulário.

Na Figura 10, está solução que seria a ideal, porém necessita maior análise por conta dos dados recebidos e da atual inviabilidade de se obter uma versão mais completa da API Datajud.

```
for fonte in fontes_form:
    headersList = {
        'Authorization': API_KEY,
        'Content-Type': 'application/json'
    }
    payload = json.dumps({
        'query':{
            # por hora meramente ilustrativa
        }
    })
    url = f'https://api-publica.datajud.cnj.jus.br/api_publica_{fonte.descricao.lower()}/_search'
    response = requests.request('POST', url, payload, headersList)
    formulario.resultado_texto = response.text
    if formulario.resultado_texto == '':
        formulario.resultado_texto = 'deu nao'
    formulario.save()
```

Figura 10. Próximo à solução anterior, porém envolvendo API Datajud.

Outro ponto importante, que deve ser levado em conta, é o fato da necessidade de se ter uma estrutura específica para cada tribunal no tocante aos dados recebidos. Como cada site dispõe da sua maneira, um método de visualização deve ser criado para cada um deles, local também onde pode-se utilizar o *crawler* dependendo da resposta do site. Esta era uma situação existente já na versão de Palharini [1]. Nas Figuras 11 e 12 estão a modelagem da exibição e o resultado no buscadir para o Tribunal do Mato Grosso (TJMT), desta vez no arquivo *models.py* do aplicativo Pesquisa.

```
def get_json_to_texto(self):
        data = json.loads(self.resultado_texto)
        texto_saida = ''
        # for fonte in self.fontes_pesquisa:
        acordoes_list = data['AcordaoCollection']
        if not acordoes_list:
            return '<br><br><b>Nenhum processo encontrado</b>'
        for acordoes in acordoes_list:
            texto_saida += '<br>Processo: <b>%s</b><br><br>' % acordoes['Processo']['NumeroUnico']
            texto_saida += '%s<br><br>' % acordoes['Conteudo']
            texto_saida += '----------------------------------------------------<br><br>'
        return texto_saida
```

Figura 11. Código de tratamento JSON para HTML para um dos tribunais.

É possível observar na primeira linha do método que o valor passado como "JSON" é o próprio texto recebido da *response*, convertido para um objeto para ser tratado. Este é, também, o momento onde pode ser aplicado o *crawler* para fazer a varredura dos dados. Um detalhe interessante desta abordagem é que não consome o servidor, uma vez que o texto pode ser tratado no próprio cliente.

Figura 12. Exibição do resultado da busca.

IV. Conclusões

Como discutido ao longo do texto, este trabalho integra as pesquisas colaborativas entre o Laboratório de Práticas da Computação da Universidade Franciscana (UFN) e a área do Direito da mesma instituição, no campo da Tecnologia da Informação. Este estudo dá continuidade ao projeto de Palharini e Zamberlan [1], o que significa que o sistema já possuía uma base estruturada, com funcionalidades implementadas e disponibilidade operacional.

O presente estudo buscou aprimorar a interface e expandir as funcionalidades, incluindo mecanismos para realizar buscas automatizadas em sites de Tribunais que utilizam CAPTCHAs como medida de segurança. Para isso, duas novas alternativas, utilizando a linguagem Python, foram propostas e implementadas: o envio de requisição via HTTP/Request e a utilização da API do Datajud (base nacional de dados do Poder Judiciário). O processo de adaptação e aprimoramento foi facilitado pela adoção do modelo arquitetural MVT (Model-View-Template) do *framework* Django, que proporciona uma estrutura organizada, promovendo a reutilização de código e simplificando a manutenção do aplicativo, além de contribuir para a eficiência do desenvolvimento.

As novas propostas de solução visam aumentar o número de tribunais disponíveis para busca dentro do sistema implementado. Ressalta-se que o modo como estava anteriormente não está mais totalmente operacional para o tribunal testado, o que fez com que novas linhas de pensamento fossem adicionadas ao modelo.

Para implementação da solução via HTTP/*requests*, será necessário um mapeamento dos campos desejados, campos estes que podem ser facilmente descobertos por qualquer navegador, utilizando a aba *console* - *network* (normalmente tecla F12), analisando os dados expostos na listagem. Este mapeamento, em trabalhos futuros, pode ser realizado de forma automática, não implementado neste trabalho. Para a confecção da Tabela I, os valores não foram estudados e testados, foram vistos de forma rápida apenas para determinar se a solução seria viável.

Dos objetivos propostos, a utilização do *crawler*, pelo código implementado por este trabalho, foi descartada, no sentido de busca pelas respostas das pesquisas, porém ainda é interessante que seja montado para exibição dos dados onde a resposta é um HTML. Já a automatização de descoberta utilizando Inteligência Artificial, segue apenas como sugestão para a solução.

Referências

[1] Eduardo Pavani Palharini e Alexandre Zamberlan. *Web Crawler Para Portais Da Área Do Direito*. Santa Maria, RS, Brasil. Disponível em https://tfgonline.lapinf.ufn.edu.br: Trabalho de Conclusão de Curso Sistemas De Informação - Universidade Franciscana (UFN), 2023.

[2] Ricarch Lawson. *Web Scraping with Python*. PACKT, 2005.

[3] Moaiad Ahmad Khder. *Web Scraping or Web Crawling: State of Art, Techniques, Approaches and Application*. 2021. URL: https://ijasca.zuj.edu.jo/PapersUploaded/2021.3.11.pdf (acesso em 19/03/2021).

[4] Hannes Hapke, Cole Howard e Hobson Lane. *Natural Language Processing in Action: Understanding, analyzing, and generating text with Python*. Simon e Schuster, 2019.

[5] *Dicionário Priberam da Língua Portuguesa*. URL: https://dicionario.priberam.org/ (acesso em 27/03/2024).

[6] *Framework Scrapy*. URL: https://scrapy.org/ (acesso em 02/03/2024).

[7] Leonard Richardson. *Biblioteca Beautiful Soup*. URL: https://beautiful-soup-4.readthedocs.io/ (acesso em 02/03/2024).

[8] *Selenium Web Driver*. URL: https://www.selenium.dev/ (acesso em 02/03/2024).

[9] Associação Brasileira de Jurimetria. *R para jurimetria*. 2018. URL: https://abjur.github.io/r4jurimetrics/index.html (acesso em 10/04/2024).

[10] Pedro Guimarães. *burlar captchas é contra a lei?* 2018. URL: https://groups.google.com/g/thackday/c/iUcz0v21axY/m/C_cyaIIRAgAJ?pli=1 (acesso em 06/05/2024).

[11] *Inteligência Jurídica para sua empresa*. 2023. URL: https://insight.jusbrasil.com.br/.

[12] *Datajud-Wiki*. URL: https://datajud-wiki.cnj.jus.br/ (acesso em 04/10/2024).

[13] Glaydson de Farias Lima. *Manual de Direito Digital: fundamentos, legislação e jurisprudência*. Appris Editora e Livraria Eireli-ME, 2016.

[14] Tomasz Wykowski e Justyna Wykowska. *Lessons learned: Using Scrum in non-technical teams*. Set. de 2019. URL: https://www.agilealliance.org/resources/experience-reports/lessons-learned-using-scrum-in-non-technical-teams/.