UNIVERSIDADE TECNOLÓGICA FEDERAL DO PARANÁ

COORDENAÇÃO DE ANÁLISE E DESENVOLVIMENTO DE SISTEMAS

TECNOLOGIA EM ANÁLISE E DESENVOLVIMENTO DE SISTEMAS

ANDERSON BALDUINO JACHINSKI

RAFAEL GELINSKI

FERRAMENTA BASEADA EM WEB SERVICE RESTFUL PARA INDEXAÇÃO E VERIFICAÇÃO DE PLÁGIO EM TRABALHOS ACADÊMICOS

TRABALHO DE DIPLOMAÇÃO

PONTA GROSSA

2011

ANDERSON BALDUINO JACHINSKI

RAFAEL GELINSKI

FERRAMENTA BASEADA EM WEB SERVICE RESTFUL PARA INDEXAÇÃO E VERIFICAÇÃO DE PLÁGIO EM TRABALHOS ACADÊMICOS

Trabalho de Conclusão de Curso apresentado como requisito parcial à obtenção do título de Tecnólogo em Análise e Desenvolvimento de Sistemas, da Universidade Tecnológica Federal do Paraná.

Orientador: Prof. Ademir Mazer Jr.

PONTA GROSSA

2011

Ministério da Educação
Universidade Tecnológica Federal do Paraná
Campus Ponta Grossa

Diretoria de Graduação e Educação Profissional

TERMO DE APROVAÇÃO

FERRAMENTA BASEADA EM WEB SERVICE RESTFUL PARA INDEXAÇÃO E VERIFICAÇÃO DE PLÁGIO EM TRABALHOS ACADÊMICOS

por

ANDERSON BALDUINO JACHINSKI

Este Trabalho de Conclusão de Curso (TCC) foi apresentado em 09 de Novembro de 2011 como requisito parcial para a obtenção do título de Tecnólogo em Análise e Desenvolviento de Sistemas. O candidato foi arguido pela Banca Examinadora composta pelos professores abaixo assinados. Após deliberação, a Banca Examinadora considerou o trabalho aprovado.

_____
Ademir Mazer Junior
Prof. Orientador

_____
Helyane Bronoski Borges
Membro titular

_____
Saulo Queiroz
Membro titular

_____
Helyane Bronoski Borges
Responsável pelos Trabalhos
de Conclusão de Curso

_____
André Koscianski
Coordenador do Curso
UTFPR - Campus Ponta Grossa

Ministério da Educação
Universidade Tecnológica Federal do Paraná
Campus Ponta Grossa

Diretoria de Graduação e Educação Profissional

UTFPR
UNIVERSIDADE TECNOLÓGICA FEDERAL DO PARANÁ

TERMO DE APROVAÇÃO

FERRAMENTA BASEADA EM WEB SERVICE RESTFUL PARA INDEXAÇÃO E VERIFICAÇÃO DE PLÁGIO EM TRABALHOS ACADÊMICOS

por

RAFAEL GELINSKI

Este Trabalho de Conclusão de Curso (TCC) foi apresentado em 09 de Novembro de 2011 como requisito parcial para a obtenção do título de Tecnólogo em Análise e Desenvolviento de Sistemas. O candidato foi arguido pela Banca Examinadora composta pelos professores abaixo assinados. Após deliberação, a Banca Examinadora considerou o trabalho aprovado.

_____
Ademir Mazer Junior
Prof. Orientador

_____
Helyane Bronoski Borges
Membro titular

_____
Saulo Queiroz
Membro titular

_____
Helyane Bronoski Borges
Responsável pelos Trabalhos
de Conclusão de Curso

_____
André Koscianski
Coordenador do Curso
UTFPR - Campus Ponta Grossa

Dedicamos este trabalho a todos aqueles
que acreditaram que conseguiríamos, e
aos que não acreditaram também.

AGRADECIMENTOS

Neste momento tão importante, agradecimentos são necessários, pois certamente não chegaríamos aqui sem o apoio de muitas pessoas. Agradecemos aos nossos familiares por todo apoio, mesmo quando um aparente sentimento de desânimo estava presente. É preciso também agradecer aos amigos, que fizeram parte desta caminhada, tanto no meio acadêmico, quanto fora dele.

Muito obrigado à todos os professores que estiveram presente nestes anos de estudo, especialmente ao professor Ademir Mazer Jr. pela orientação e acompanhamento neste trabalho, e ainda por acreditar que seríamos capazes de entregar este trabalho.

É claro que a tecnologia não é responsável por toda a transformação cultural que ela impulsiona. A mudança tecnológica apenas cria novos espaços de possibilidades a serem, então, explorados. (FIALHO, Francisco, 2001)

RESUMO

GELINSKI, Rafael; JACHISNKI, Anderson Balduino. **Ferramenta baseada em Web Service RESTful Para Indexação e Verificação de Plágio Em Trabalhos Acadêmicos**. 2011. 63. Trabalho de Conclusão de Curso (Tecnologia em Análise e Desenvolvimento de Sistemas) - Universidade Tecnológica Federal do Paraná. Ponta Grossa, 2011.

Cópias em trabalhos acadêmicos são comuns, e com o advento da internet, esta prática foi facilitada, tanto para realizar pesquisas, quanto na cópia de trechos de trabalhos prontos ou livros sem citar a autoria devidamente. Concluir que um trabalho foi ou não plagiado não é tarefa simples, mas pode-se usar ferramentas para contribuir na verificação de ocorrências deste problema. O objetivo deste trabalho é unir tecnologia de Serviços Web em arquitetura REST, com a ferramenta Lucene, utilizada para indexação de texto, aproveitando os conceitos SOA para criar uma ferramenta independente de plataforma, que possibilite a detecção de plágio em trabalhos acadêmicos.

Palavras-chave: SOA. REST. Java. Serviços Web. Lucene. Ferramenta de Combate ao Plágio.

ABSTRACT

GELINSKI, Rafael; JACHISNKI, Anderson Balduino. **RESTful Web Service-Based Tool for Indexing and Plagiarism Verification in Academic Paper**. 2011. 63. Trabalho de Conclusão de Curso (Tecnologia em Análise e Desenvolvimento de Sistemas) - Federal Technology University - Parana. Ponta Grossa, 2011.

Copies in academic studies are usual, and with the advent of the Internet, this pratice was made easier, both to conduct research, such in the copy of snippets from books or completed academic papers without citing the author properly. Conclude that a work was plagiarized or not is no simple task, but you can use this tool to contribute in scanning events of this problem. The objective of this study is to unite technology of Web Services over REST arquitecture, with Lucene tools, used on text-indexing, taking advantage of the SOA's concepts to create a free-platform tool that enables the detection of plagiarism in academic papers.

Keywords: SOA. REST. Java. Web-services. Lucene. Plagiarism Combating Tool.

LISTA DE ABREVIATURAS, SIGLAS E ACRÔNIMOS

API	*Application Programming Interface* (Interface de Programação de Aplicativos*)*
HTTP	*Hyper Text Transfer Protocol* (Protocolo de Transferência de Hipertexto)
IETF	*Internet Engineering Task Force* (Força de Trabalho de Engenharia de Internet)
J2EE	*Java 2 Plataform, Enterprise Edition* (Java Edição Empresarial)
JSON	*JavaScript Object Notation* (Notação de Objeto JavaScript)
PDF	*Portable Document Format* (Formato de Documento Portável)
REST	*Representational State Transfer* (Transferência de Estado Representacional)
RFC	*Request for Comment* (Pedido de Comentário)
RPC	*Remote Procedure Call* (Chamada remota de procedimento)
SOA	*Service-Oriented Arquitecture* (Arquitetura Orientada a Serviços)
SOAP	*Simple Object Acess Protocol* (Protocolo Simples de Acesso a Objetos)
SQL	*Structured Query Language* (Linguagem de Consulta Estruturada)
TCP	*Transfer Control Protocol* (Protocolo de Controle de Transmissão)
URI	*Uniform Resource Identifier* *(*Identificador Uniforme de Recursos*)*
URL	*Uniform Resource Locator* (Localizador Uniforme de Recursos)
WSDL	*Web Services Description Language* (Linguagem de Descrição de Serviços Web)
XML	*eXtensible Markup Language* (Linguagem de Marcação Estendida)

XML-RPC	*XML Remote Procedure Call* (Chamada remota de procedimento com XML)
W3C	*World Wide Web Consortium* (Consórcio World Wide Web)
RMI	*Remote Method Invocation* (Invocação de Métodos Remotos)
CORBA	*Common Object Request Broker Architecture*

LISTA DE QUADROS

SUMÁRIO

1 INTRODUÇÃO

Com a evolução tecnológica, criaram-se inúmeras facilidades para a humanidade. Dentre elas, a Computação e a Internet se destacam por reduzir custos, distância e tempo para trafegar inúmeros tipos de informação. Porém, isso também permitiu o crescimento de alguns pontos negativos em relação à propriedade intelectual. Visto que as informações estão muito mais acessíveis, pode-se observar o crescimento das práticas de plágio, onde a produção de conhecimento por meio da violação de direitos autorais alheios é frequente. (FURTADO, 2002)

De forma geral, “(...) plágio caracteriza-se como uma falsa atribuição de autoria, uma apropriação indevida de trabalho de um autor por outro indivíduo”. (ROMANCINI, 2007)

Diante da necessidade em verificar e reduzir a prática do plágio, surge a possibilidade de implementar uma ferramenta, independente de plataforma, para a busca de semelhanças de padrão de texto em trabalhos e documentos em texto.

De qualquer forma, a avaliação de plágio no âmbito técnico de comparação não se restringe a verificação da forma escrita estritamente idêntica, mas também na tentativa de apropriar-se de idéias alheias sem designar o autor verdadeiro.

Os tipos de avaliação são distinguidos entre análise sintática e/ou semântica. Deste ponto em dianto do trabalho, os termos relacionados à verificação e avaliação da ocorrência de plágio serão adotados apenas como análise sintática, ou seja, a verificação partindo da comparação da forma com que foram escritos.

Para chegar à ferramenta, a solução está implementada em Linguagem Java, com auxílio de uma ferramenta denominada JERSEY, baseada na arquitetura SOA, mais especificamente em Serviço Web seguindo o conceito REST, do inglês *Representational State Transfer* (Transferência do Estado Representacional), o qual foi proposto por Roy T. Fielding em sua dissertação de doutorado, publicada no ano 2000. (SANDOVAL, 2009)

Este tipo de Serviço Web utiliza-se do protocolo HTTP para realizar a troca de mensagens entre clientes e servidor. Outra característica importante é a forma de redirecionamento de serviços, realizado através de um endereçamento URI específico. (RICHARDSON; RUBY, 2007)

Para realizar a tarefa de verificar a existência do plágio, a ferramenta necessita ainda da utilização de bibliotecas para a indexação e pesquisa de texto em arquivos. Apache Lucene é uma biblioteca em Java que realiza busca e indexação de documentos e possui código aberto. A Lucene foi selecionada devido sua grande flexibilidade em receber arquivos não importando seu formato, desde que seja possível a extração do seu conteúdo textual. Outra vantagem da Lucene é a forma de busca em sua base de dados, semelhante a linguagem SQL facilitando a implementação de buscas. (MCCANDLESS; HATCHER; GOSPODNETIC, 2010)

Devido a crescente utilização de serviços em arquitetura SOA baseados em RESTful, como os oferecidos por Amazon, Yahoo!, Google Maps, Ficklr (RICHARDSON; RUBY, 2007), e pela fácil utilização destes serviços em diversas plataformas, torna-se interessante e viável a utilização de tecnologias implementadas sobre a mesma estrutura.

Aliando todos os fatores e tecnologias apresentados anteriormente, torna-se possível o desenvolvimento de uma arquitetura baseada em Serviços Web, fornecendo uma plataforma para buscas textuais com baixos custos computacionais, ou seja, baixo consumo de memória e rápido retorno da pesquisa contendo um alto índice de relevância.

1.1 PROBLEMA

Visto o problema gerado pela evolução tecnológica, em que a grande facilidade de acesso à informações passou a permitir a banalização da propriedade intelectual sobre conteúdos disponilizados pela Internet, gera-se a necessidade de prover alguma forma de averiguar e certificar que nenhuma criação será erroneamente atribuída a alguém que não detém este mérito.

Na perspectiva de que os ambientes computacionais são ferramentas com o intuito de prover funcionalidades diversificadas para facilitar a obtenção de informações, torna-se inviável a remoção, seja parcial ou completa, dos recursos que ela é capaz de prover. Contudo, é necessário o desenvolvimento de métodos de verificação que possibilitem caracterizar o plágio.

De forma geral, não apenas na Internet, mas em qualquer conteúdo digital disponível, a capacidade de replicar informações com um mínimo de esforço torna ainda mais tênue a verificação de autoria. É neste ambiente generalizado que pode-se encontrar grande quantidade de conteúdos que agregam a autoria em indivíduos que não são os proprietários autênticos das informações neles expressas.

1.2 OBJETIVOS

O objetivo deste trabalho é colaborar no combate ao plágio e falsas atribuições de autoria no ambiente acadêmico através do desenvolvimento de uma ferramenta idependente de plataforma capaz de realizar uma análise sintática sobre documentos que forem enviados para ela. Em síntese, a idéia é utilizar os recursos computacionais para avaliar documentos em formato digital, na tentativa de encontrar textos com trechos idênticos em documentos distintos.

Por mais que trabalhe-se sobre um mesmo assunto ou aspecto de pesquisa, indivíduos diferentes tendem a compor o material escrito de forma diversificada. Obviamente que os documentos sobre um mesmo assunto irão convergir em algum ponto, mas isto não caracteriza uma disposição idêntica do conteúdo escrito.

1.2.1 Objetivo Geral

A finalidade mais abrangente abordada, está na criação de uma ferramenta, capaz de verificar e informar a probabilidade da existência de plágio entre documentos de conteúdo escrito, existentes em uma base de dados previamente indexada e continuamente alimentada com novos textos. Esta ferramenta será desenvolvida na tentativa de amenizar a ocorrência da cópia, sem citar a autoria detentora da propriedade intelectual utilizada, buscando evidenciar a ocorrência ou a tentativa de plágio acadêmico.

A construção deverá ser feita atendendo a necessidade de funcionar em uma plataforma distribuída, onde vários elementos (pessoas ou aplicações) deverão ser capazes de contribuir na geração de um repositório unificado de pesquisa, e assegurar também que todos eles poderão pesquisar trechos idênticos de um arquivo com os que estão contidos no repositório.

1.2.2 Objetivos Específicos

- Utilizar arquitetura de Serviços Web e a linguagem de programação Java, aplicando o conceito REST;
- Criar e alimentar um Repositório de Arquivos para testar a eficácia de análise na ocorrência de plágio utilizando a biblioteca Lucene;
- Implementar um algoritmo de varredura que, com o auxílio da Lucene, seja capaz de verificar similaridade entre os textos indexados;

2 FUNDAMENTAÇÃO TEÓRICA

Nesta parte do trabalho serão vistos algumas considerações necessárias e conceituações básicas das tecnologias aplicadas, direta ou indiretamente ligadas ao desenvolvimento proposto, que serão base para todo o trabalho desenvolvido.

2.1 PROTOCOLO HTTP

O Protocolo de Transferência de Hipertexto, convencionalmente chamado apenas de HTTP (proveniente da sigla em inglês para *HyperText Transfer Protocol*), pode ser visto como parte fundamental para o funcionamento da Web.

Kurose e Ross descrevem-no como um protocolo de camada de aplicação, definido no [RFC 1945] e no [RFC 2616]. (KUROSE; ROSS, 2006)

Os chamados RFCs (do inglês *Request for Comments*, Pedido de Comentários em tradução literal), são documentos regulamentados e mantidos pela IETF (*Internet Engineering Task Force* - Força de Trabalho de Engenharia de Internet), e tem por finalidade estipular os padrões que serão utilizados nos protocolos definidos para cada documento. É possível encontrar uma descrição bem técnica e muito detalhada à respeito do protocolo no documento RFC relacionado. (KUROSE; ROSS, 2006 p. 4)

O RFC 1945 (IETF, 1996) define o padrão adotado no HTTP/1.0, enquanto que o RFC 2616 refere-se ao HTTP/1.1, este sendo compatível com aquele. (IETF, 1999)

Tanenbaum (2003) explica que, de forma geral, os servidores de HTTP utilizam o protocolo TCP (*Transmission Control Protocol* - Protocolo de Controle de Transmissão) sobre a porta 80 para evitar preocupações desnecessárias com replicação e perda de pacotes nas transmissões, visto que o protocolo TCP faz o tratamento e controle sobre isso.

Kurose e Ross (2006) ressaltam que o protocolo é um protocolo sem estado, ou seja, não mantém nenhuma informação sobre os clientes após cada requisição. Cada chamada HTTP obtém uma resposta e não é guardado no servidor nada relacionado após a resposta ser concluída.

Tanenbaum (2003) diz que, apesar de ter sido projetado para a Web, o HTTP possui métodos para manipular os recursos disponíveis. Segundo ele, estes métodos foram cruciais para o surgimento das tecnologias como o SOAP.

O HTTP possui seis métodos principais para o controle dos seus recursos. São eles: (TANENBAUM, 2003)

- GET - Responsável pela leitura e recuperação de recursos e páginas existentes no servidor;
- HEAD - Capaz de informar apenas o cabeçalho da mensagem, porém sem receber a página ou recurso associada a ele;
- PUT - Serve para realizar o armazenamento de uma nova página;
- POST - Seu funcionamento é similar ao de PUT, porém, ao invés de gravar um novo conteúdo, ele apenas complementa uma página já existente;
- OPTION - Fornece opções e configurações do servidor ou de um arquivo;
- DELETE - Faz o oposto de PUT, remove páginas do servidor.

2.2 ARQUITETURA SOA

Há arquiteturas com base em estruturas similares às encontradas no mundo real. SOA (*Service-Oriented Architecture*), que pode ser traduzido como Arquitetura Orientada a Serviços, segue este conceito. A arquitetura assemelha-se à uma estrutura convencional de designação de responsabilidades aos elementos que a compõe, onde cada um desempenha um papel específico.

Erl (2007) cita um serviço de entrega como um exemplo trivial de serviço na realidade, onde cada encarregado desenvolve uma atividade específica. Um deles, denominado de despachante, tem o dever de receber as chamadas e organizar as entregas para a chamada recebida. O segundo é o condutor, o qual assume a responsabilidade de realizar a entrega, e o terceiro se encarrega de realizar o cálculo da conta de todo o processo.

Apesar de ser um exemplo trivial que pode ser presenciado facilmente no mundo real, o processo envolvido e a separação em papéis específicos são atribuídos de uma forma extremamente organizada e sistemática, permitindo que tudo funcione corretamente. Da mesma forma, a arquitetura SOA procura estabelecer funcionalidades através da atribuição de serviços específicos para cada parte funcional implementada. Isto quer dizer que, independente da plataforma e

tecnologia utilizada, o funcionamento da aplicação dependerá da disponibilidade de serviços que irão receber instruções, processá-las devidamente, e retornar uma resposta ao requisitante.

É importante destacar que, a correta atribuição de responsabilidades aos elementos, sejam eles humanos ou aplicações informatizadas, não implica em restringir uma única funcionalidade para cada um dos elementos. Na verdade, é mais aplicável que estruturas orientadas a serviço designem múltiplas capacidades ao elemento prestador de serviço.

Com isso, é possível denotar que um único elemento nesta arquitetura poderá prover múltiplos serviços. Observando a estrutura de serviços no contexto de Arquitetura de desenvolvimento computacional, pode-se citar que SOA está muito mais próximo de um paradigma de solução lógica para sistemas distribuídos do que para uma implementação concreta, ou seja, SOA é uma forma de abstrair e conceituar o processo de implementação e não uma tecnologia específica para realizar o desenvolvimento. (ERL, T. 2007)

É possível perceber que, por tratar-se de um estilo arquitetural para sistemas distribuídos, SOA pode ser definida de forma abrangente:

> Arquitetura Orientada a Serviços pode ser entendida como um estilo arquitetural para a construção de sistemas baseados em componentes modularizados, autônomos e fracamente acoplados, denominados serviços. Cada serviço expõe processos e comportamentos, através de contratos, que são compostos de mensagens em endereços detectáveis chamados terminais. O comportamento dos serviços obedece a uma política, externa ao próprio serviço.” (FRONDANA; BARBOSA; GALVÃO, 2009 p. 20)

2.2.1 Princípois da Arquitetura

A arquitetura SOA possui oito princípios para o *design* da aplicação. A descrição oficial de cada um deles é objetiva e sucinta, porém pouco concisa. Contudo, é possível analisar subjetivamente cada uma delas, e com isso, obter uma definição mais plausível.

Utilizando-se de duas referências distintas de Erl (2007, 2009), é possível descrever a definição oficial através de uma e ainda uma descrição subjetiva do autor na outra.

É importante destacar que os princípios estipulados tem a finalidade de melhorar a viabilidade em implantar a Arquitetura Orientada a Serviços, porém isso não torna obrigatória a aplicação de todos os oito princípios descritos.

2.2.2 Contrato de Serviço Padronizado

Em tradução livre, a definição oficial diz que: “Serviços sem o mesmo inventário de serviço estão em conformidade através do mesmo contrato de design padronizado”. (ERL, 2009 p.49)

O Contrato de Serviço Padronizado (Standardized Service Contract) é uma parte essencial na arquitetura SOA, pois é responsável por toda a interface de comunicação com o serviço disponibilizado. É através dele que serão definidas as capacidades e os propósitos de cada serviço bem como todas as suas funcionalidades e tipos de dados que serão utilizados. (ERL, 2007 p. 71)

Frondana *et al* descreve como Contrato, o conjunto de todas as mensagens suportadas e refere-se a ela como interface do Serviço. (FRONDANA; BARBOSA; GALVÃO, 2009 p. 23)

2.2.3 Baixo Acoplamento de Serviço

Este princípio refere-se à mensura do nível de dependência, no que diz respeito aos relacionamentos do serviço. Dispõe da idéia de que os serviços devem ser auto-suficientes, reduzindo a dependência e permitindo que a manutenção seja mais clara e ágil.

“Contratos de serviço requerem um baixo nível de acoplamento e eles mesmos são pouco acoplados com o ambiente ao seu redor.” (ERL, 2009 p.49)

2.2.4 Abstração de Serviço

Erl (2009) relata que na Orientação a Serviços existe um grande vínculo com a Abstração. Este vínculo expõe que os serviços devem fornecer o menor nível de detalhamento que for possível e com isso, assegurar o baixo acoplamento com o resto da estrutura implementada. Na descrição oficial temos que “O Contrato de

Serviço contém somente as Informações Essenciais e as informações sobre os serviços são limitadas ao que será publicado.”

2.2.5 Reusabilidade de Serviço

A descrição oficial define que “O Serviço composto e expresso por uma lógica genérica pode ser considerado como um recurso reutilizável” (ERL, 2009)

Aplicar este princípio pode ser considerado uma vantagem, não pela construção de um código genérico, mas graças a idéia de não perder e possibilitar que o código seja útil em um contexto além do qual ele foi desenvolvido. (FRONDANA; BARBOSA; GALVÃO, 2009)

Em outras palavras, este princípio assegura o levantamento de considerações que deverão assegurar que o Serviço possua capacidades apropriadas, que garantam maior facilidade na reutilização quando esta for necessária. (ERL, 2007 p.72)

2.2.6 Autonomia de Serviço

“Os serviços exercem um alto nível de controle interno sobre seu ambiente em tempo de execução”, este princípio elenca várias questões relacionadas ao design lógico e também à implementação concreta do Serviço. Este princípio visa facilitar a independência dos serviços, tornando mais fácil a reutilização dos mesmos (ERL, 2009).

2.2.7 Serviço sem controle de Estado (*Statelessness*)

Aplica-se no contexto de Escalabilidade e consumo de recursos para o acesso do Serviço. Sua função está voltada a reduzir o número de informações pouco ou completamente desnecessárias no contexto, viabilizando o acesso ao serviço com um baixo consumo de recursos. O princípio procura manter o Controle de Estado somente quando este for estritamente necessário.

2.2.8 Descoberta de Serviços

De forma geral, é possível declarar este princípio como uma forma de assegurar que o Retorno de Investimento do Serviço envolvido será valorizado, e que uma vez disponibilizado, o Serviço poderá ser encontrado por outras soluções que desejem reutilizar os seus métodos.

2.2.9 Composição de Serviços

Visa a prevenção em realizar grandes mudanças na implementação do Serviço para concluir a demanda existente sobre ele. Deve ser previsto e definido a complexidade em que será desenvolvido o serviço, buscando deixá-lo preparado desde o seu início, evitando a reorganização da implementação, a qual exigiria que o Serviço fosse reestruturado constantemente. Este princípio tende a minimizar os esforços dedicados no desenvolvimento da solução. (ERL, 2007 p.73-74)

2.3 *WEB SERVICES* (SERVIÇOS WEB)

Serviços Web são aplicações desenvolvidas na Internet capazes de comunicarem-se com outras aplicações, independente da plataformas em que elas estiverem construídas. Em outras palavras, são implementações realizadas com o intuito de prover conectividade entre aplicações distintas.

De forma geral, a maioria das arquiteturas envolvidas em Serviços Web dispõe do uso de XML, ou Linguagem de Marcação Extensível, porém aplicada de diferentes formas.

Para validar esta afirmação, pode-se citar Cerami (2002), o qual diz que Serviço Web engloba todos os serviços disponibilizados pela Internet, utilizando um sistema de mensagem XML padronizado, sem vínculo com alguma linguagem de programação. Segundo ele, existem diversos formatos de mensagens XML e, dependendo do formato escolhido, é possível trabalhar com XML-RPC, SOAP ou ainda utilizar requisições HTTP básicas para informar os dados XML. Richardson e Ruby (2007) descrevem outras tecnologias refententes à Serviços Web, como por exemplo, WSDL e REST.

2.3.1 Arquitetura RPC

No contexto de interoperabilidade, pode-se citar a arquitetura RPC (*Remote Procedure Call*) ou Chamada de Procedimento Remoto, a qual consiste em conectar diferentes aplicações através de chamadas que podem ser compreendidas por todas as aplicações envolvidas no processo. No entanto, RPC não é tratada no contexto de Serviços Web, já que provê suporte para conexão de aplicações em uma camada de rede local, sem a necessidade da publicação destes métodos em um servidor Web.

O XML-RPC é a derivação da arquitetura RPC que, por meio da utilização de XML, torna-se capaz de fornecer informações padronizadas para os serviços implementados sobre este padrão.

Quanto aos dados, as requisições são feitas por marcações (*tags*) de parâmetro <param>. O que define uma informação é a marcação <value> e o tipo de dado é referenciado através de marcações específicas. Segundo Laurent et al (2001), pode-se citar <boolean>, <string>, <double> e <int> como algumas marcações de tipo de dado (LAURENT; JOHNSTON; DUMBILL, 2001).

2.3.2 Tecnologia SOAP

No contexto de Serviços Web, destaca-se também a tecnologia SOAP, a qual é divida em várias partes. Uma destas partes é o envelope, construído em XML, que servirá para descrever o conteúdo das mensagens e algumas informações de como processá-las. Outra parte da tecnologia é a responsável por referenciar toda codificação nos tipos personalizados de informação. A parte mais crítica é a extensibilidade, que se responsabiliza pela criação do envelope que descreve as regras para representar as chamadas e respostas. Normalmente baseia-se em outros protocolos da Camada de aplicação, mais notavelmente em RPC e HTTP (ENGLANDER, 2002).

Sua especificação provê maneiras para se construir mensagens que podem trafegar através de diversos protocolos, e foi especificado de forma a ser independente de qualquer modelo de programação ou outra implementação específica.

Englander descreve que SOAP pode ser construído baseado em RPC, e que hoje a maioria dos serviços disponíveis está sobre esta arquitetura, porém, a arquitetura não está estritamente vinculada com a tecnologia descrita (ENGLANDER, 2002).

Na visão de Richardson e Ruby (2007), ao contrário do que muitos acreditam, a arquitetura RPC é a primeira competidora do estilo de arquitetura REST, e não a tecnologia SOAP diretamente, pois apesar dos serviços SOAP atuais aplicarem os conceitos da arquitetura RPC, a tecnologia é apenas a forma de empacotar as informações. Em sua descrição sobre SOAP diz ainda que, os programadores SOAP desconsideram a comparação da tecnologia à arquitetura RPC, e preferem referirem-se à ela como “orientada à mensagem” ou “orientada à documento” (RICHARDSON; RUBY, 2007).

2.3.3 Linguagem de definição de interface WSDL

De acordo com Graham, a WSDL (*Web Services Definition Language*), ou Linguagem de Definição de Serviços Web, foi submetida para a W3C (*World Wide Web Consortium*) por um conjunto de empresas, dentre elas a IBM e a Microsoft, com o intuito de criar uma sintaxe padronizada para os métodos de acesso e invocação de chamada à Serviços Web. A descrição do serviço implementado em WSDL é representado em um documento em formato XML, o qual segue o esquema descrito na padronização (GRAHAM *et al,* 2001).

Trata-se de uma IDL (*Interface Definition Language*), ou Linguagem de Definição de Interface. Assim como CORBA, RMI, entre outras IDL’s, a WSDL é responsável por criar uma divisão, separando às responsabilidades entre a especificação abstrata e a implementação.

A WSDL padroniza como um Serviço Web representa os parâmetros de entrada e saída de uma chamada externa, a estrutura da função, a natureza da invocação e a ligação do serviço de protocolo. Possibilita que clientes em diferentes plataformas compreendam automaticamente como interagir com um Serviço Web.

É usada para definir as interfaces (assinaturas de método) e tipos de dados para a programação lógica.

A definição WSDL descreve três propriedades fundamentais: (Graham *et al*, 2001)

- O Funcionamento do Serviço, quais as operações e métodos que ele realiza;
- Os detalhes referentes aos formatos de dados utilizados;
- As questões e protocolos de Endereçamento que serão utilizados;

Tudo definido dentro de um arquivo WSDL é abstrato: é apenas a definição de parâmetros e restrições para como a comunicação deve ocorrer em tempo de execução. A implementação do serviço web tem que seguir as diretrizes definidas no arquivo WSDL, mas tem alguma flexibilidade sobre especificidades. WSDL define extensões *out-of-the-box* de ligação para SOAP 1.1, HTTP GET, HTTP POST, e MIME.

2.3.4 Estilo de Arquitetura REST

O REST (*Representational State Transfer*), ou Transferência de Estado Representacional, é um conjunto de regras e conceitos de como desenvolver um Serviço Web. Proposto na tese de doutorado de Roy Thomas Fielding, o REST não é uma arquitetura de software propriamente dita, mas sim, um conjunto de princípios a serem seguidos com o intuito de estabelecer um estilo de arquitetura de software (SANDOVAL, 2009).

Segundo Fielding:

> "Um estilo de arquitetura é um conjunto coordenado de restrições arquiteturais que restringe os papéis/funções de elementos da arquitetura e as relações permitidas entre estes elementos dentro de qualquer arquitetura que está em conformidade com esse estilo." (FIELDING, 2000 p. 13 - Tradução Livre)

Fielding (2000) aborda o REST como um estilo de arquitetura híbrido, proveniente da combinação de estilos arquitetônicos baseados em rede. Conforme Tanenbaum e van Renesse (*apud* FIELDING, 2000 p. 24), a diferença entre a arquitetura baseada em Rede e os Sistemas Distribuídos está na forma que a aplicação irá interpretar o funcionamento do sistema. Nos sistemas distribuídos, apesar da implementação concreta estar espalhada em diversos componentes, a aplicação verá todo o sistema similar à estrutura de sistema centralizado, trantando-o de forma transparente, interpretando como um sistema convencional. Os sistemas

baseados em Rede, ao contrário, não necessitam que a aplicação trate o sistema de forma transparente, abstraindo a distribuição. Neste ponto, os Sistemas baseados em Rede não precisam tratar tudo como aplicação local, permitindo a existência de endereçamentos para realizar o acesso aos recursos disponíveis.

Em um contexto mais abrangente, pode-se dizer que REST é um estilo arquitetural baseado em SOA (*Service-oriented Arquitecture*) e desenvolvido em plataforma Cliente-Servidor. Para implementar um Serviço Web baseado em REST deve-se, primeiramente, compreender quais são as restrições relacionadas a sua aplicação. A primeira trata a necessidade de desenvolver sobre uma plataforma Cliente-Servidor (SANDOVAL, 2009).

A segunda restrição está relacionada a implementação de um sistema *Stateless*, ou em tradução livre, sem estado. O conceito de sistema sem estado está relacionado a falta de configuração e conservação de informações de sessão. Em outras palavras, cada requisição a um serviço baseado em REST será tratada independentemente, sem a necessidade de qualquer vínculo com as requisições anteriores realizadas pelo mesmo cliente (SANDOVAL, 2009).

As demais restrições para o desenvolvimento tratam a capacidade de suportar um sistema de armazenamento em *cache* disposto pela infraestrutura da rede, a uniformidade de acesso, onde cada recurso deverá ter um endereçamento único e válido, e ainda, suportar escalabilidade. Possui ainda uma restrição opcional, que trata a capacidade em prover o acesso ao código sobre demanda (SANDOVAL, 2009).

Apesar da existência destas restrições, de forma alguma é estipulada uma linguagem específica para o desenvolvimento em REST, existindo apenas os princípios de como os recursos serão maniputados durante a comunição (SANDOVAL, 2009).

O grande diferencial de REST em relação aos outros métodos de desenvolvimento de Serviços Web está na utilização de uma interface uniforme, ou seja, o acesso aos serviços deve ser realizado de forma similar mesmo tratando-se de serviços implementados em servidores diferentes. Por esse motivo, é adotada a utilização do protocolo HTTP (*Hyper Text Transfer Protocol*), pois em essência o mesmo disponibiliza esta uniformidade, visto que é utilizado em todos os sites da Internet (RICHARDSON; RUBY, 2007).

Com isso, nota-se uma maior flexibilidade em relação aos demais Serviços Web, já que utiliza a simplicidade na recuperação de recursos disponibilizada pelo protocolo HTTP.

Todas essas restrições definidas tem como papel principal estipular uma padronização para o tratamento e utilização dos recursos disponibilizados pelo estilo arquitetural. Segundo Sandoval, recurso é tudo aquilo que contém um endereçamento na Web. Em REST, o endereçamento é realizado utilizando-se URI (*Uniform Resource Identifier*) ou Identificador Uniforme de Recursos, o qual é responsável pelo mapeamento de endereços, sendo utilizado para atribuir caminhos lógicos individuais para cada recurso disponível (SANDOVAL, 2009 p. 9).

Para exemplificar o conceito de recurso em REST, adote que um arquivo terá seu acesso disponibilizado na Web. À partir do momento em que este arquivo recebe um URI para ser encontrado diretamente, considera-se o mesmo como um recurso. De forma mais abrangente, não somente um arquivo convencional, mas também serviços e consultas geradas à partir de um URI específico, são considerados como recursos.

Como o acesso aos Serviços Web baseados em REST ocorre através de URI, a extração e estipulação criteriosa dos recursos é extremamente importante, pois é através deles que serão gerados os caminhos úteis para o funcionamento do mesmo.

É crucial entender também como é realizada a manipulação dos recursos disponibilizados pelo serviço. Basicamente, associa-se o método HTTP GET com a recuperação e representação dos dados de um recurso, a remoção dos recursos através do método HTTP DELETE, a alteração de recursos existentes é obtida por meio do HTTP PUT e a criação de novos recursos é feita pelo HTTP POST ou HTTP PUT, este quando não existe a URI do recurso e aquele gera recurso através de uma URI já existente (RICHARDSON; RUBY, 2007).

2.3.4.1 JERSEY: JAX-RS

Java sempre teve alguma aplicação no que diz respeito à Serviços Web, só que estava limitada ao uso de WSDL e SOAP. O projeto JAX-RS surgiu para

contribuir com a linguagem permitindo a implementação RESTful (SANDOVAL, 2009 p. 81).

O termo RESTful é designado para classificar os sistemas que aplicam o estilo REST em seu desenvolvimento (RICHARDSON; RUBY, 2007).

Para facilitar a aplicação do REST, há alguns *frameworks* para auxiliar no desenvolvimento.

> "Um *framework* pode ser definido como 'um projeto reutilizável de uma parte ou de todo um sistema, que é representado por um conjunto de classes abstratas e pelo modo que elas interagem', ou 'um esqueleto de uma aplicação que pode ser customizado para gerar novas aplicações'." (JOHNSON, 1997, p. 39 *apud* LORENZETTI, 2004 p. 15)

O JERSEY é um dos poucos frameworks puros na implementação Java RESTful. Utiliza-se de anotações para realizar o mapeamento em relação ao protocolo HTTP, permitindo que o programador mantenha seu foco mais voltado às regras de negócio. Ele é voltado para implementação J2EE, e pode ser implementado em qualquer Servidor Web compatível. Além disso, o *framework* segue as especificações descritas pelo JAX-RS, também conhecido como JAX-311, que estipula as restrições necessárias para padronizar o funcionamento de REST em Java.

A anotação @PATH é responsável por definir qual é o recurso tratado em questão. Serve para associar, através do *framework*, e demonstrar qual classe será responsável por cada URI.

Quanto às anotações *@GET*, *@POST*, *@PUT* e *@DELETE*, são responsáveis pela geração automática de código, para interpretar o tipo de requisição HTTP que foi utilizado no acesso ao Serviço RESTful. O tipo de requisição dependerá da forma de acesso requerida (visualização, criação, alteração e remoção de recursos), e o funcionamento destes métodos depende necessariamente da existência de uma URI.

As requisições do tipo HTTP GET realizadas em URI do Serviço RESTful serão gerenciadas e respondidas através do método que estiver associado à anotação *@GET*.

@POST e *@PUT* são úteis quando necessita-se criar e alterar novos recursos dentro do Serviço RESTful. Ambos podem ter parâmetros de entrada associados ao método que estão relacionados. Em outras palavras, as informações

contidas na requisição, tanto HTTP POST, quanto HTTP PUT, podem ser manipuladas através de parâmetros de entrada. Em oposição às anotações *@POST* e PUT, a anotação *@DELETE* é responsável pela exclusão de recursos.

Com a necessidade de existir ao menos um URI associado a cada recurso RESTful, algumas informações relevantes são expostas como parte do caminho de acesso. Devido isto, a anotação *@PathParam* serve para avaliar e construir variáveis à partir destas informações.

A manipulação das requisições e respostas é capaz de interpretar além de texto simples, para tanto *@Consumes* e *@Produces* realização a interpretação e formatação do conteúdo no corpo da mensagem, este para a geração da resposta, e aquele para a interpretação dos dados contidos na requisição ao serviço. Dentre os formatos mais utilizados nas requisições e respostas, destacam-se texto plano HTML, XML e JSON. (SANDOVAL, 2009)

@FORMPARAM está associado no contexto de PUT e POST, relacionando os campos de formulário HTML enviados no corpo da requisição com as variáveis que serão utilizadas no serviço implementado. (JBOSS, 2011)

Segundo a especificação JavaDocs contida no núcleo do *framework* Java REST, *@CONTEXT* é a anotação usada para inserir informações à uma variável de classe, uma propriedade bean ou um parâmetro de método. Esta anotação permite manipular dados do request, cabeçalho da requisição, entre outros. (JBOSS, 2011)

2.3.4.2 Restlet

Restlet é um *framework* REST para plataforma java. Ele pode ser usando tanto para aplicações do lado cliente quanto para aplicacões do lado servidor. Ele suporta a maioria dos protocolos de transporte da internet, transporte de dados e descrição de serviços. O *framework* Restlet é formado por uma API (*Application Programming Interface*) ou Interface de Programação de Aplicativos, com conceitos REST para facilitar as chamadas tanto do lado cliente quanto do lado servidor.

Restlet trabalha com dois tipos de conectores para fazer as conexões com a aplicação. Conectores do lado servidor, por exemplo: JAX-RS que são utilizados neste trabalho e conectores do lado cliente como, por exemplo: A biblioteca Apache HTTPClient, utilizada como ferramenta para desenvolver clientes de teste.

2.4 LUCENE

Apache Lucene é uma Biblioteca em Java capaz de realizar indexação e busca de informações à partir da similaridade de textos. Entre os pontos positivos em relação ao uso desta biblioteca, destacam-se a escalabilidade na recuperação de informações, o seu alto desempenho, a simplicidade de uso e o seu livre desenvolvimento em código aberto. Contudo, para o seu funcionamento é necessária a existência de informações derivadas de conteúdo em texto, independente da linguagem ou formato de arquivo em que estão contidas. (MCCANDLESS; HATCHER; GOSPODNETIC, 2010)

Trata-se de uma API que consiste em disponibilizar ao programador a capacidade de indexar e realizar buscas de texto de forma simples, sem a necessidade de entender toda a complexidade envolvida. Em outras palavras, pode-se dizer que toda a complexidade envolvida na indexação será feita de forma transparente ao programador, permitido que ele se concentre nas regras de negócio da aplicação que deseja desenvolver.

No contexto de desenvolvimento deste trabalho, a Lucene será aplicada para realizar a comparação de frases contidas em documentos diferentes, na tentativa de encontrar cópia de conteúdo intelectual, caracterizando o plágio.

3 ARQUITETURA DESENVOLVIDA

Em resumo, o aplicativo desenvolvido divide-se em quatro principais aspectos: os Serviços REST, a Indexação de conteúdo, a pesquisa utilizando-se da Lucene e o retorno para o cliente em formato XML.

A seguir, representada em um Diagrama de Classes, está a estrutura em que o projeto foi desenvolvido.

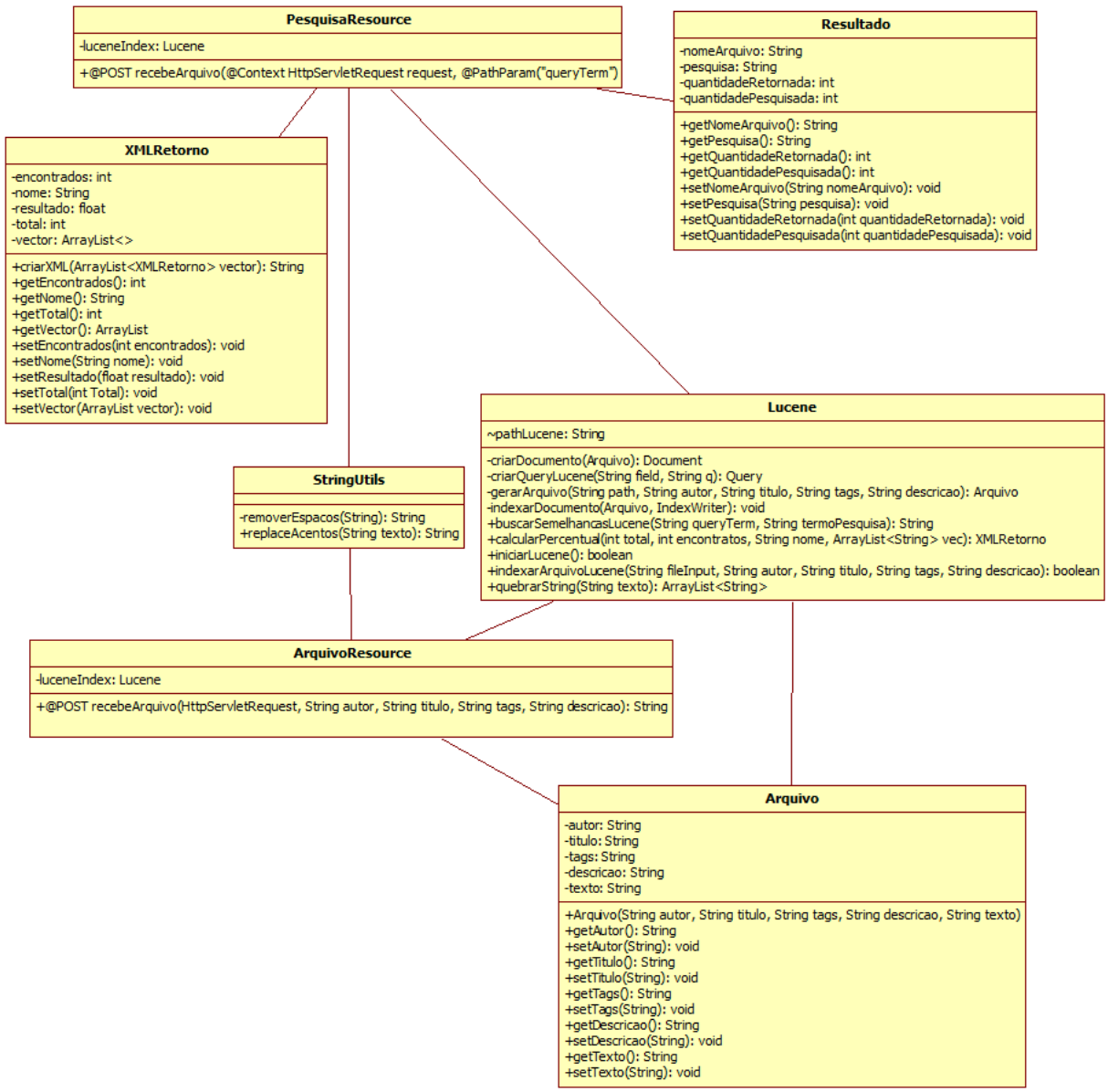

Figura 1 - Diagrama de Classes do Projeto
Fonte: Autoria Própria

De acordo com a Figura 1, as classes responsáveis pelo funcionamento dos Serviços REST são a ArquivoResource e a PesquisaResource, a classe Lucene detém toda a implementação do algoritmo de busca implementado, bem como a associação e utilização das capacidades providas pela biblioteca Lucene. As demais classes expostas, servem de apoio para a interpretação ou geração do conteúdo XML que será devolvido na resposta da conclusão do processamento.

O foco principal deste trabalho está atribuído à implementação de uma ferramenta para auxiliar o combate ao plágio, aliando Serviços Web, um indexador textual e tecnologia Java. Será possível analisar documentos indexados por intermédio da biblioteca Lucene, que, combinada com um algoritmo de busca e verificação de similaridade.

Esta ferramenta torna-se capaz de gerar um relatório contendo os resultados comparativos dentre os documentos já existentes na base de textos, na tentativa de encontrar indícios de cópia de trechos de outros trabalhos contidos nesta. A ferramenta serve de auxílio no combate ao plágio indicando os documentos que podem conter estas ocorrências.

Primeiramente, definiram-se as regras de negócio básicas como: envio de arquivos, recebimento de arquivos, identificação de plágio, tratamento de *Strings*, retorno de resposta para os clientes, forma de indexação e pesquisa que seriam utilizadas em todo *framework*.

Procurou-se manter as regras e serviços o mais simples possível devido ao baixo conhecimento até o momento tanto da tecnologia dos serviços REST quanto da Lucene, estas regras serão descritas adiante, onde serão melhor contextualizadas. Após isso, realizou-se uma avaliação de funcionamento da tecnologia Lucene da Apache, levantando algumas características como suas possibilidades, funcionalidades e limitações.

A análise e as primeiras implementações, com a finalidade de realizar testes, foram feitas independentemente da camada de serviços.

Esta decisão foi tomada para que o aprendizado fosse mais rápido, pois seria levado em conta somente a biblioteca de indexação, sem a preocupação de como ela se ligaria aos Serviços Web, isto garantiu que as dificuldades encontradas neste momento seriam relacionadas somente à Lucene.

A implementação de testes incluiu pequenas funcionalidades que envolviam tarefas de indexação e pesquisa que posteriormente serviram para definir como

seria realizada a ligação entre o Serviço Web e o restante da aplicação desenvolvida, e também, para ter uma visão panorâmica da complexidade no funcionamento da API.

A Lucene é capaz de se relacionar com bibliotecas de extensões, estas podem trabalhar com um ou vários formatos de arquivo, mas cabe ao programador extrair o conteúdo textual dos arquivos em questão. No desenvolvimento desta aplicação, levando em consideração a possibilidade de trabalhar com vários formatos de arquivos, e que cada extensão deve ser tratada individualmente, decidiu-se tratar somente arquivos PDF, pois, limitando a este tipo de arquivo, o tempo de desenvolvimento seria poupado.

Além disso, a conexão com outras extensões é relativamente semelhante ao que foi usado para controlar este formato, tornando-se um trabalho desnecessário no que diz respeito a produção de conhecimento novo. Cada formato deve ser tratado e recebido de maneira diferenciada, dependendo da extensão e com isso, cria-se a dependência da extensão da Lucene em relação ao formato de arquivo envolvido.

Por exemplo, uma extensão pode trabalhar com os formatos de texto como TXT (texto puro trabalhado, por exemplo, pelo Bloco de Notas), DOC (utilizado pelo editor de texto *Word*) e ODT (Editores de Código Livre), mas ao utilizá-la, torna-se indispensável que sejam fornecidos apenas documentos dentre os formatos por ela suportados.

3.1 LUCENE NA ARQUITETURA

Para utilizar a Lucene é necessário entender que trata-se de uma API desenvolvida para armazenar conteúdo textual em uma base de dados chamada de index e, à partir desta, extrair informações relevantes. A API permite a indexação de arquivos desde que sejam informações em origem textual, excluído conteúdos em imagens, assim como construir pesquisas de alto desempenho, se comparado com SQL por exemplo.

Antes do conteúdo ser indexado ele passa por um processo de análise que transforma-o em um texto simples e após a conversão remove conteúdo irrelevante

para pesquisas futuras, caso seja esta função configurada. Este processo passa por um “*Analyzer*”, que nada mais é do que uma classe disponibilizada pela própria API da Lucene para tratamento das informações que serão indexadas, esta classe ainda possui várias configurações, como por exemplo, a escolha do idioma que este analisador irá tratar (McCANDLESS; HATCHER; GOSPODNETIC, 2010).

Neste trabalho utilizou-se por conveniência a classe *BrazilianAnalyzer*, que é preparada trabalhar com textos escritos em português brasileiro, e remove as “*stop words*” referentes a este idioma, que é o utilizado nos trabalhos que serão indexados. Adotando que o *Analyzer* esteja configurado para português brasileiro, ele poderá remover automaticamente as “*stop words*” do idioma, que são palavras irrelevantes considerando-se todo o texto. Por exemplo: ‘a’, ‘o’, ‘as’, ‘os’, ‘para’, ‘com’ estas são palavras irrelevantes para a pesquisa posterior, mas há autonomia para configurar as “*stop words*” que fizerem-se necessárias. (SMILEY; PUGH, 2009)

O próximo passo na construção do indexador é a definição da organização interna do índice da Lucene, para que a pesquisa seja feita de maneira mais veloz, e entender como funcionam as principais Classes da API. Neste momento, é criado uma espécie de “sumário” interno para facilitar a pesquisa, utilizado para a organização do índice deste trabalho. Este é baseado nos *Fields*: autor, título, tags, descrição e o texto extraído, que são os parametros que serão enviados para o serviço pelo cliente.

A Lucene, possui uma classe *Document*, que é uma unidade de indexação e pesquisa que permite armazenar campos com os atributos a serem indexados. Possui também uma classe *Field*, que é o objeto capaz de receber os dados que serão armazenados. Pode-se dizer que um *Field* só pode ser armazenado em um *Document*.

Um *Field* não trabalha sozinho, ele possui um nome e um valor. Além disso, um documento pode conter um ou mais *Fields*. Para o desenvolvimento estes conceitos foram utilizados, o *Document* criado para representar o arquivo que seria indexado com seus respectivos *fields*, os *fields* foram definidos conforme os parâmetros de entrada no serviço. (McCANDLESS; HATCHER; GOSPODNETIC, 2010)

A forma de criação do *Document* é relativamente simples tanto quanto a inclusão de fields no mesmo, como neste exemplo a seguir:

```
Document doc = new Document();
doc.add(new Field("autor", autor, Field.Store.YES, Field.Index.ANALYZED));
doc.add(new Field("titulo", titulo, Field.Store.YES, Field.Index.ANALYZED));
doc.add(new Field("tags", tags, Field.Store.YES, Field.Index.ANALYZED));
doc.add(new Field("descricao", descricao, Field.Store.YES, Field.Index.ANALYZED));
doc.add(new Field("texto",texto, Field.Store.YES, Field.Index.ANALYZED));
```

Quadro 1 - Exemplo de instância da classe *Document*

O trecho: "*Field.Store.YES*" mostra que o campo original será adicionado ao indíce. "*Field.Index.ANALYZED*", demonstra que este campo antes de ser efetivamente indexado passará pelo *Analyzer* escolhido. Estas configurações foram escolhidas devido a problemas com caracteres acentuados que será demonstrado no decorrer do trabalho.

Outra classe presente na Lucene é a *Directory*, responsável por endereçar e salvar o índice no disco rígido do computador em questão. O armazenamento dos *Document's* é feito no *Directory*. A Classe *IndexWriter* é a responsável pela criação do índice, ao qual pode-se adicionar *Documents* e após isto, salvar estes no *Directory*. Foi escolhido utilizar o diretório no próprio disco rígido do servidor, o que trás uma melhor performance na manipulação de arquivos (McCANDLESS; HATCHER; GOSPODNETIC, 2010).

A Classe *IndexSearcher* tem o papel de executar a busca no índice. Os critérios de busca são passados para a função de busca do *IndexSearcher* através do objeto *Query.*

Para a criação deste objeto, utiliza-se a *TermQuery*, para a qual passam-se como parâmetro o nome do campo a ser procurado e o valor que deseja-se realizar a busca no campo descrito.

Este campo é o *field* que foi criado anteriormente. O uso do *TermQuery* é realizado conforme abaixo onde a busca será realizada no *field* "texto", pesquisando pela palavra "aluno":

```
TermQuery = new TermQuery(new Term ("texto","aluno"));
```

É possível descobrir a relevância de um documento no retorno de uma pesquisa através da pontuação retornada para cada arquivo indexado. Esta pontuação é baseado na relevância da pesquisa realizada. Por exemplo, ao pesquisar as palavras "Trabalho de Diplomação", a pontuação de um documento

que contenha essas palavras mais vezes será maior do que outro onde elas aparecem apenas uma vez, porém, não utilizou-se este recurso durante o desenvolvimento.

A exclusão de documentos é feita através do *IndexReader*, que é a classe utilizada para acessar um índice já existente. Caso seja necessária a atualização do índice, será necessário remover o *Document* e indexá-lo novamente, pois a Lucene não dispõe de uma função específica para isso. Na solução proposta a exclusão de textos já adicionados ao índice não é possível, então este recurso não foi implementado.

A escolha da Lucene se deve ao fato do aplicativo a ser desenvolvido basear-se em buscas textuais, as quais ocorrem em trabalhos que serão enviados pelo cliente para a ferramenta. Imaginando a situação em que um grupo de pessoas entregue um trabalho de aproximadamente 20 páginas, a quantidade de palavras somadas nestes trabalhos ultrapassaria um número aceitável para que a busca nestes dados fosse feita por meio de uma SQL, usando o comando. Com a Apache Lucene, obtêm-se respostas rápidas ao fazer pesquisas no seu *Index*. Além de velocidade ela oferece mais opções de busca e personalização de pesquisa nas informações que estão indexadas.

As buscas podem ser feitas utilizando-se de operadores como: (+, -, AND, NOT, OR, * e etc.). Destaca-se que a Lucene faz a indexação de qualquer arquivo que contenha informação textual, mas a extração do texto dos respectivos arquivos não fica cargo dela e, para isso, é necessário encontrar uma ferramenta que seja capaz de fazer essa extração para que, por fim, a indexação seja efetuada, nesta implementação utilizados a biblioteca *PDFTextStripper*, que será detalhada adiante no trabalho. (SMILEY; PUGH, 2009)

Como dito anteriormente, para pesquisar são feitas consultas através de *Query's*, semelhantes às encontradas em SQL, que realizam a pesquisa com ou sem o uso de operadores.

Existem três tipos de *Query's* definidas que são:

- *TermQuery* - realiza a busca por um termo exato no index;

  query = new TermQuery(new Term("texto", "aluno"));

O *TermQuery* pesquisa documentos que contenham a palavra “aluno” indexadas no field “texto”. Este é o modelo de *Query* utilizado neste projeto.

- *PhraseQuery* - faz a busca por uma frase específica no index da lucene, ou ainda, agrupa vários termos a serem pesquisados no objeto para pesquisa;

```
PhraseQuery query = new PhraseQuery();
query.add(new Term("texto", "professor"));
query.add(new Term("texto", "aluno"));
```

Quando executada a query criada com *PhraseQuery* ele vai executar as duas pesquisas por “aluno” e “professor”, juntas sem a necessidade de executa-las separadamente.

- *FuzzyQuery* - busca utilizando-se de operadores e um algoritmo diferenciado.

```
query = new FuzzyQuery(new Term("texto", "aluno"));
```

Com a *FuzzyQuery* se procurada a frase “aluno” em um indice que contenha a palavra “al1no” serão encontrados, mas não serão encontrados caso a palavra indexada seja “alun”.

3.2 CONSTRUINDO OS SERVIÇOS WEB

Antes da construção dos Serviços Web básicos, que permitem que a ferramenta implementada seja capaz de avaliar as similaridades entre os documentos enviados, foi necessário estudar como Serviços Web funcionam. Uma vez concluída a etapa conceitual, conseguiu-se iniciar as atividades de desenvolvimento dentro da arquitetura e tecnologia estipulada.

Esta implementação de Serviços Web baseou-se no estilo de arquitetura REST, o qual compreende a realização das chamadas para o serviço através de uma Interface Padronizada Uniforme (URI). Apesar da definição do REST não estipular exatamente uma tecnologia para implementar esta interface, é comum

utilizar o protocolo HTTP para realizar este procedimento. Já que este é amplamente utilizado na Internet e atende às restrições definidas pelo estilo de arquitetura citado.

Como REST estipula restrições e padrões em relação à arquitetura mas não define qual tecnologia será aplicada, é possível desenvolver aplicações em diversas linguagens e com intuitos diversificados, desde que atendam aos requisitos estabelecidos. Graças à isso, é possível implementar o Serviço Web em linguagem Java, utilizando-se de um *framework* que segue a especificação JAX-RS. Para implementação utilizou-se a linguagem Java devido à extensa base de conhecimento e referencial teórico em implementações de serviços REST disponíveis para ela, também pelo fato da especificação JAX-RS já estar consolidada e ainda por facilitar a integração à Lucene, que tem seu núcleo desenvolvido sobre a mesma tecnologia.

A especificação, em sua versão 1.1, visa padronizar o desenvolvimento dos *frameworks* que realizam a manipulação REST na linguagem Java através de determinadas anotações, as quais servirão como referência na associação das atividades envolvidas na implementação e através delas, possibilitando ao programador manter seu foco voltado às regras de negócio, enquanto o *framework* fica com a responsabilidade de associar as funcionalidades aos métodos HTTP.

Contudo, é importante ressaltar que a principal intenção da especificação é, independente do *framework* utilizado, permitindo a manipulação dos métodos implementados da mesma forma, e com isso, padronizar a implementação dos serviços sem criar vínculos ou restringir o uso de um *framework* específico.

Com esses conceitos assimilados, é necessário implementá-los, procurando colocar na prática todo o conhecimento adquirido. Apesar de todo este embasamento, definiu-se que seria mais prudente iniciar o desenvolvimento com uma aplicação básica, visando conhecer e identificar como o desenvolvimento iria se comportar perante as requisições em REST. Com isso, foi criada a primeira implementação sobre RESTful Java que realizou-se neste projeto, utilizando um método GET para retornar uma mensagem simples, e um método POST para retornar o resultado de um cálculo.

Dada a devida atenção ao funcionamento das regras de implementação, conseguiu-se iniciar às primeiras funcionalidades do Serviço, já pensando na forma com que se conectariam ao restante do projeto e como seriam manipuladas as respostas obtidas pela lógica desenvolvida.

Após completar as primeiras indexações e compreender como realizar as buscas através da API, e entender o comportamento da implementação de Serviço RESTful Java, integrou-se o desenvolvimento da ferramenta, através da combinação dos recursos ofertados pela Lucene à plataforma de Serviço REST.

A estrutura que compõe a construção do projeto, ficou disposta sobre três camas, onde a primeira está focada nos métodos disponibilizados ao cliente, a segunda refere-se ao tratamento das chamadas REST e suas respectivas respostas e a última camada é a responsável pelo tratamento das informações e processamento junto à Lucene.

A figura 2 demonstra como fica distribuída as camadas, e como é realizado o acesso ao Serviço implementado.

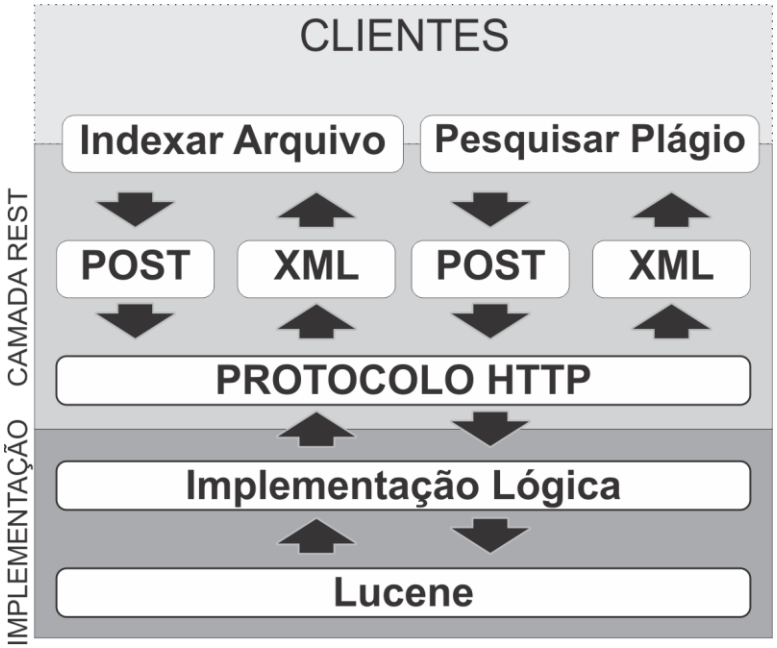

Figura 2 - Camadas de Acesso ao Serviço implementado
Fonte: Autoria Própria

Nesse momento, com a intenção de padronizar o tratamento e recuperação dos arquivos por parte do cliente, limitou-se que o Serviço Web utilizaria apenas arquivos de extensão PDF, esta medida foi tomada para que os esforços de desenvolvimento fossem concentrados apenas em um tipo de extensão de arquivos, para fazer a filtragem das extensões, foi implementada uma verificação tanto no serviço de pesquisa quanto no de recebimento de arquivos com o seguinte código:

```
if (item.getContentType().equals("application/pdf") || item.getName().endsWith(".pdf"))
```

Quadro 2 - Verificação de formato de arquivo

Com isto verifica-se o conteúdo da requisição e o final da extensão do arquivo enviado. Após o arquivo validado é que se da a extração do conteúdo textual através da biblioteca *PDFTextStripper*, para proceder até a indexação, pois a Lucene só aceita o conteúdo textual do arquivo PDF e não o arquivo em si. Caso fosse escolhido aceitar mais extensões de arquivos no serviço seria necessário pesquisar outras bibliotecas além da *PDFTextStripper*, pois está somente trabalha com PDFs.

Com isso, a primeira restrição de acesso serviço desenvolvido ficou relacionada ao formato de arquivo.

3.3 DEFINIÇÃO DA LÓGICA GERAL DA ARQUITETURA IMPLEMENTADA

Dentre as diferenças entre REST e outras implementações de Serviços Web, a que mais se destaca é o estilo de comunicação entre o Serviço e os Clientes.

Esta comunicação é feita através de requisições por intermédio direto do protocolo HTTP, enquanto que em outros estilos que implementam Serviços Web ocorre por meio de uma Linguagem de Definição de Interface (IDL) realizada através da invocação de métodos pelo cliente.

Por utilizar uma camada em sua implementação, sendo esta geralmente construída através de documentos XML contendo definições dos serviços, o desenvolvimento sobre outra arquitetura tem o foco em métodos com vários parâmetros de entrada e saída.

Já na estrutura em arquitetura REST, a comunicação tem foco nos recursos envolvidos, e é feita atráves de métodos HTTP convencionais, permitindo acessar os serviços atráves de URL’s, facilitando consideravelmente a forma de realizar as chamadas e, consequentemente, reduzindo a complexidade na construção de aplicações clientes. Ressalta-se que, por utilizar os métodos padrão do HTTP, existe a possibilidade de realizar o acesso diretamente em um navegador web convencional, informando-se o endereço do recurso como nestas imagens:

Figura 3 - exemplo de chamada utilizando um cliente *desktop* para serviço de indexação
Fonte: Autoria Própria

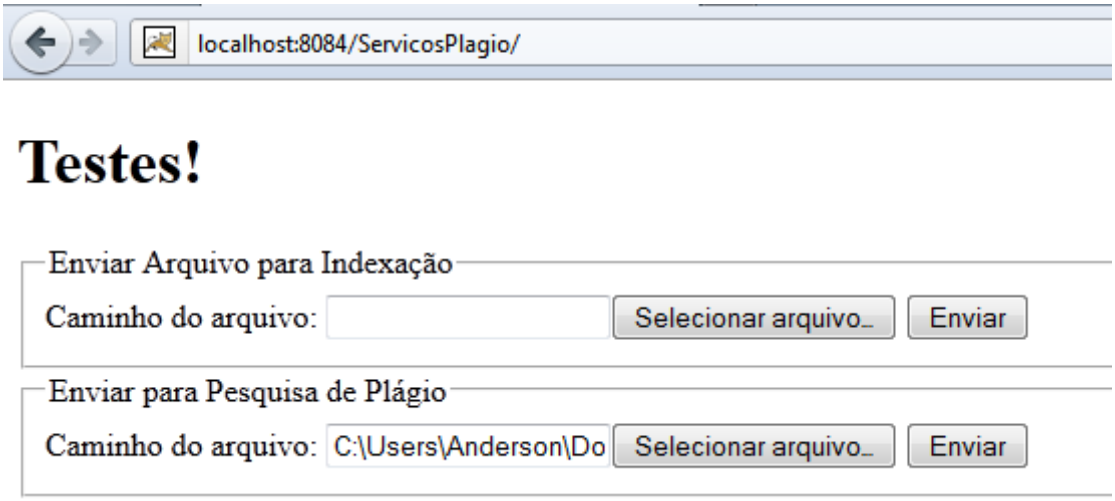

Figura 4 - Exemplo de chamada ao serviço de Pesquisa, utilizando um navegador convencional, o Action do botão enviar está apontando para: "res/pesquisa/texto"
Fonte: Autoria Própria

A figura 5 exemplifica uma chamada ao serviço de pesquisa agora utilizando um cliente *desktop*:

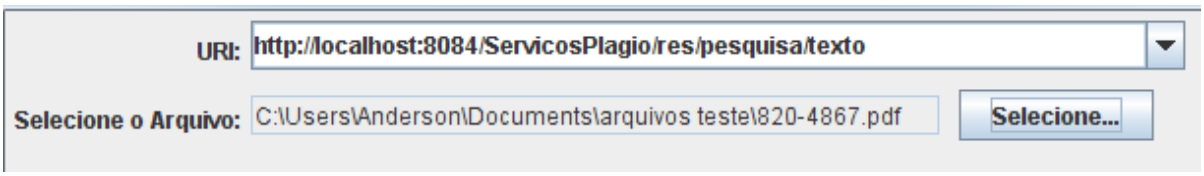

Figura 5 - chamada ao serviço de pesquisa utilizando uma aplicação *desktop*
Fonte: Autoria Própria

Em um primeiro momento, foram modelados os serviços e as funcionalidades que compõe a ferramenta. Modelar estes serviços nem sempre é uma tarefa trivial, pois é preciso considerar alguns aspectos durante a descrição de requisitos e construção dos recursos. Dependendo do nível de complexidade da aplicação, e quanto mais ela se aproximar de um controle de dados no padrão *CRUD* mais simples é a identificação dos recursos. *CRUD* é proveniente das palavras inglesas *Create, Read/Recover, Update, Delete* (Criar, Ler/Recuperar,

Atualizar, Excluir), é um termo adotado para descrever as ações básicas no controle sobre um cadastro ou arquivo. No caso do aplicativo desenvolvido, foram modelados apenas dois serviços, ambos com funções bem definidas.

Para elaborar um bom serviço REST é preciso verificar quais são:

- os recursos
- as URI's desses recursos
- os dados que serão manipulados pelos serviços
- os métodos HTTP que serão aceitos por cada uma das URI's envolvidas.

Por exemplo, uma das URI's definida neste aplicativo é encontrada no caminho:

http://enderecoIP[:porta]/ServicosPlagio/res/arquivo

Quadro 3 - Endereço do Serviço de Envio de Arquivo

Para facilitar a descrição do caminho dos Serviços será adotado, desta parte em diante do trabalho, apenas o caminho relativo ao invés de ser necessário descrever o extenso endereçamento de um recurso. Assumindo o caminho descrito anteriormente como exemplo, o caminho relativo para este recurso será expresso no formato "/res/arquivo", o qual é muito mais prático de assimilar.

Este endereço aceita o método POST para recebimento de um arquivo. Esta definição tem que ser mais intuitiva possível, normalmente o método POST é associado ao envio de informações para servidor. Contudo, nada impede que o método GET seja implementado na mesma URI para que, por exemplo, retorne a situação de um serviço ou outro tipo de informação relacionado ao recurso deste endereço. Na implementação dos serviços deste projeto utilizou-se apenas o método POST, tanto para envio do arquivo para indexação, quanto para envio do arquivo para pesquisa que responde na URI /res/pesquisa.

Definiu-se a Lucene como sendo o motor de busca devido a sua rápida resposta às consultas realizadas. Unindo-se estas duas tecnologias, Serviços Web e Lucene, procurou-se criar um aplicativo com capacidade de expansão e aplicação independente de plataforma. Procurou-se manter a simplicidade dos serviços modelados também durante o desenvolvidos para que, na ocorrência de dificuldades e erros, a solução para os mesmos fosse encontrada mais rapidamente.

Inicialmente o projeto previa um único serviço que procurava atender todas as necessidade na mesma URI's, mas desta maneira o uso da aplicação não era intuitivo. Os serviços utilizariam a mesma URI com dois métodos POST apenas mudando o seu @*Path*. Com uma única URI, por exemplo: .res/arquivo, internamente no Serviço Web seriam encontrados dois @*Path* diferentes, @*Path*("arquivo") e @*Path*("pesquisa"), cada qual com seu respectivo método @*POST*.

Lembrando que cada método tem seu @*Path* interno, cada um é invocado conforme a chamada do cliente. Para facilitar o endereçamento para os serviços, as funcionalidades de enviar arquivo e de comparar com os arquivos existentes foram separadas, cada qual com seus métodos.

Para tratar a sequência de execução do programa, após receber uma requisição, os serviços dispões de métodos que fazem uma pequena validação verificando se o arquivo enviado é de extensão PDF e, só então, a execução passa para classes subsequentes. Foram definidas classes que tem funções bem definidas para tratamento de *Strings*, tratamento da geração do XML para retorno, dentre outras.

Para extração do conteúdo textual dos arquivos PDF foi necessário definir a maneira de realizar esta tarefa, pois existem diversas bibliotecas que fazem este trabalho. A Apache *PDFBox* apresentou vantagens, por ser compacta e simples de utilizar, e também por trabalhar exclusivamente com a extensão PDF.

O processo envolvido na verificação de plágio é concluído através do retorno para o usuário caso existam trabalhos com semelhanças ao que ele está pesquisando. Para este retorno, definiu-se o formato XML, por ser de fácil tratamento e manipulação. Para a geração do mesmo existem diversas bibliotecas disponíveis, dentre elas, destaca-se a JDOM que cria XML's de forma simples, caracterizando o motivo da sua escolha.

3.4 SERVIÇOS WEB DESENVOLVIDOS

Utilizando-se da arquitetura REST, foram criados dois Serviços Web distintos, ambos utilizando como forma de acesso o método POST do protocolo

HTTP. A escolha deste método para receber os documentos se deve ao fato dele dispor da capacidade de manipular os arquivos que são enviados pelos clientes de maneira fácil, resultando em uma baixa complexidade no recebimento e tratamento de arquivos contidos em uma requisição, tratando não apenas requisições realizadas em aplicações Desktop como também as realizadas em plataforma Web com o uso de formulários HTML.

O primeiro Serviço desenvolvido é responsável por receber os arquivos enviados e indexar na Lucene com um retorno em formato XML em caso de sucesso na indexação do arquivo enviado conforme o quadro 4:

```
<Envio>
    <Arquivo id="0">
        <Nome>Voucher2956_Anderson.pdf</Nome>
    </Arquivo>
</Envio>
```

Quadro 4 - Exemplo de resposta em XML para indexação realizada com sucesso

O método POST responde às requisições recebidas. Este Serviço é o responsável por receber os arquivos enviados que serão indexados ao repositório de texto, mantido pela Lucene. É necessário salientar que por ser um Serviço REST, qualquer plataforma que consiga manipular chamadas HTTP, conseguirá também fazer solicitações aos serviços criados sobre essa concepção.

Um dos detalhes deste Serviço Web é sua chamada, respondendo na seguinte URI: “/res/arquivo”, que é caminho ao Serviço propriamente dito, onde “/arquivo” é o *@Path* deste serviço. Foi definido que os parâmetros devem ser informados através da sua inclusão no endereçamento de acesso ao Serviço, tornando-se parte do caminho definido para realizar a chamada. Devido a isso, os parâmetros irão variar conforme o arquivo enviado.

Para construir o caminho de acesso, serão informados juntamente com o caminho do Serviço, o nome do autor, o título do documento enviado, tags ou marcadores e por fim, a descrição do arquivo enviado.

O caminho informado deverá ser construído conforme a seguinte expressão:

http://{CaminhoDoServiço}/res/arquivo/{autor}/{título}/{marcadores}/{descrição}

Quadro 5 - Modelo de Endereçamento dos Serviços

Um exemplo de caminho pode ser descrito da seguinte forma:

“.../res/arquivo/Anderson/Trabalho_Academico/Inteligencia_Artificial/descricao_do_trabalho”.

Quadro 6 - Exemplo de Endereço para a chamada do Serviço

Independente do anseio do cliente em informar esses dados, eles são requisitos mínimos, pois o Serviço só responderá quando receber o caminho de acordo com este formato.

Para receber os arquivos da requisição, utilizou-se a classe *ServletFileUpload,* que verifica se existe algum arquivo no corpo da mensagem da requisição HTTP que chegou ao método e também faz o tratamento da mesma. Caso exista um arquivo, ele efetua a “extração” do mesmo à partir da requisição para que este possa ser manipulado.

É possível receber mais de um arquivo ao mesmo tempo com a utilização desta classe, mas este não é o foco deste Serviço Web. Após o recebimento do arquivo, uma cópia deste é arquivada no disco rígido do servidor para uma futura implementação, para uma análise do mesmo e para que a manipulação dos textos contidos nos arquivos seja feita localmente, reduzindo o uso de recursos e tornando a resposta mais rápida. Após passar por alguns processos, é chegado o momento de adicionar o arquivo à Lucene e, por se tratar de arquivos do tipo PDF, é necessário submetê-los a algumas etapas antes da indexação.

Para realizar a indexação, o primeiro passo é abrir o diretório físico em que se encontra-se o *index* da Lucene. Como dito anteriormente, é necessário criar um objeto do tipo *Document* para que a Lucene faça a indexação. Visando tornar o código mais legível, dividiu-se esta tarefa em etapas.

A primeira delas compreende a criação de uma classe que contém os atributos dos arquivos a serem indexados que são o autor, o título, as marcações, uma breve descrição e o texto obtido através do documento em PDF. Conforme visto, esses valores são conseguidos através do caminho do serviço. Então, instancia-se um objeto da classe criada para a manipulação do arquivo que contém apenas os atributos básicos e, através dele informam-se os valores conforme os que foram descritos no caminho do Serviço Web. É importante ressaltar que é nesta implementação que deve-se abstrair o texto contido no documento PDF e incluí-lo ao objeto.

Para tanto, utilizou-se uma biblioteca com a finalidade específica de extrair o conteúdo textual de arquivos PDF. A biblioteca *PDFTextStripper* se encarrega dessa extração e, para isso, deve-se instanciar um objeto desta classe para, através dele, requisitar o método "*getText*()", passando como parâmetro o arquivo em questão. Neste caso o arquivo já está salvo no disco rígido do servidor e esse método retorna uma *String* contendo o texto extraído.

O problema deste processo de extração é o formato em que a *String* é apresentada após extraída, pois dependendo da formatação de texto no PDF vários espaços desnecessários são adicionados e presenciam-se também alguns problemas com relação aos caracteres especiais.

Para amenizar o problema, criou-se uma classe apenas para resolver esses problemas, com a qual, por meio do método que ficou denominado como *replaceAcentos*, informa-se como parâmetro a *String* que acabou de ser extraída. Após executado, o método retorna a *String* sem espaços excessivos e sem caracteres acentuados.

Essa medida visa melhorar a pesquisa posterior e diminuir problemas com o index da Lucene, que pode não assimilar os caracteres acentuados conforme são utilizados na língua portuguesa.

Uma vez realizadas as devidas atribuições de valores ao objeto da Classe Arquivo, é necessário criar o *Document* para que a Lucene consiga fazer a indexação. Utiliza-se então o objeto da Classe *Arquivo* para simplificar a criação deste *Document*. Um *Document* pode conter vários *Fields*, que serão utilizados para organizar o índice do *index*. Os campos contidos na criação do Documento recebem as seguintes descrições:

- autor: Recebe o autor que foi passado como parâmetro no Serviço Web;
- titulo: Contém o título do trabalho informado na requisição do Cliente;
- tags: Mantém as Marcações associadas ao Documento recebido;
- descricao: Possui uma breve descrição de trabalho enviado pelo Cliente;
- texto: Este último recebe o texto que foi previamente extraído e tratado, obtido à partir do arquivo PDF e será utilizado nas pesquisas.

Após criado o *Document*, utilizou-se o método *addDocument*(), obtido à partir de um objeto da classe *IndexWriter* para finalizar a indexação. Para indexar usou-se também a classe chamada *BrazilianAnalyzer*, que é fornecida pela própria Lucene para tratamento do texto em idioma português brasileiro. Como explicado

anteriormente, esta classe tem por padrão o objetivo de remover as *stop words* do português, ou seja, subtrair as palavras que não tem relevância para a pesquisa futura.

Terminada a tarefa de indexação, é indispensável fechar o objeto *indexWriter* com o método *close*(). O Serviço que detém as responsabilidades relacionadas ao recebimento do arquivo tem uma estrutura relativamente simples, a parte lógica mais robusta fica alheia ao serviço, sendo implementada nas classes responsáveis por extrair o texto do arquivo PDF. Deve garantir que as informações não venham com caracteres que possam atrapalhar a pesquisa no índice.

Completado o desenvolvimento do primeiro Serviço disponibilizado, pode-se concentrar esforços na implementação do Serviço de busca e comparação. Visto que este é dependente das funcionalidades disponíveis no serviço primário.

O segundo Serviço é responsável por receber um arquivo para compará-lo com os que estão indexados, e retornar um resultado em formato XML composto pelas informações adquiridas pela comparação. Os dados da comparação só serão incluídos ao resultado se acaso existirem semelhanças entre o arquivo enviado e algum que já tenha sido previamente indexado à base.

Observando o tratamento estabelecido para obter o arquivo PDF, não há grande diferença na lógica de funcionamento, pois é muito semelhante ao que é realizado no primeiro serviço. Usando também o método POST, o arquivo é extraído da requisição e logo em seguida, tem seu texto subtraído e tratado como descrito anteriormente, e só então, é passado para a função que fará a busca de semelhanças.

A solução encontrada para buscar as semelhanças entre os textos indexados e o arquivo que se deseja comparar consiste em, após se extrair o texto do arquivo enviado para o Serviço Web de pesquisa, utilizar uma função para fragmentar o texto recebido em várias sequências, com cinco palavras cada, e armazenar isto em um vetor de palavras.

O sistema irá pesquisar sequência por sequência comparando-as com as contidas neste vetor, armazenando a quantidade de resultados retornados à cada arquivo comparado para a geração do arquivo XML. Antes da divisão do texto que foi extraído do PDF, para a geração das sequências que serão usadas nas pesquisas, ele deve ser submetido ao mesmo processo do serviço anterior, que remove caracteres acentuados e espaços desnecessários para melhorar a pesquisa.

Tendo o vetor de *Strings* preenchido, lembrando que o tamanho deste vetor varia conforme o tamanho do texto a ser pesquisado, faz-se um laço de repetição para comparar uma a uma no índice da Lucene.

Para este trabalho, inicialmente optou-se por pesquisar pelo termo exato no texto, isso significa que se dentro do vetor de *String* estivesse a frase: “aluno pagou a conta do cartão”, deveria-se procurar exatamente por essa *String* em textos já indexados. Fazendo a pesquisa desta maneira, a ilustração de resultados ficava mais simples, e se acaso fosse decidido pesquisar pelo termo “aluno pagou a conta do cartão” acrescido de um operador “~” ao final da frase, a pesquisa seria feita procurando todos os documento que contivesse frases semelhantes à “aluno pagou a conta do cartão”, o que para a implementação inicial da ferramenta não se encaixava na pesquisa de plagio.

Caso a *String* em questão retorne resultados, é necessário neste caso, instanciar um objeto da classe Resultados, para guardar esse retorno em um vetor de resultados. O próximo passo na busca de semelhanças é analisar o vetor de resultados e verificar quantas vezes um arquivo retornou, e para quantas *Strings* diferentes.

Por exemplo, procurando a *String* “implementanção de software no seculo XX” vamos verificar em quantos arquivos diferentes essa *String* se encontra para que possamos montar um arquivo XML com estar informações e retonar para o cliente com a seguinte estrutura:

```
<Resultados>
    <Arquivo id="0">
        <Total>174.0</Total>
        <Encontrados>45.0</Encontrados>
        <Percentual>25.862068</Percentual>
        <Nome>Proposta TCC.pdf</Nome>
    </Arquivo>
    <Arquivo id="1">
        <Total>174.0</Total>
        <Encontrados>40.0</Encontrados>
        <Percentual>28.862068</Percentual>
        <Nome>Proposta Luiz.pdf</Nome>
    </Arquivo>
</Resultados>
```

Quadro 7 - Resposta XML para pesquisa de plágio

Onde o “Total” representa a quantidade de *Strings* que o texto do arquivo que foi enviado para o serviço foi dividida, “Encontrados” demonstra quantas das *Strings* do “Total” que retornaram resultados, “Percentual” representa a quantidade que o campo “Encontrados” representa do total de *Strings* e “Nome” é o nome do arquivo em que foram encontrados resultados com estes termos.

3.5 CHAMADA AOS WEBSERVICES

Serviços REST são chamados através de URIs e a forma de acesso a estes serviços são os próprios métodos HTTP. Este trabalho de chamada pode ser feito diretamente com requisições HTTP “puras” ou utilizar bibliotecas para facilitar este trabalho. É simples trabalhar diretamente com requisições e respostas HTTP, para enviar as requisições ao serviços, as linguagens de programação mais populares possuem métodos/bibliotecas para manipulação protocolo HTTP. Para Java existem vários, como o projeto HttpClient mantido pela Apache.

Independente da forma que as chamadas sejam construídas, de qualquer linguagem que seja feita a requisição ao serviço REST alguns passos devem ser seguidos e variam conforme a linguagem.

3.5.1 Testes

Para testes desta aplicação foi criado um cliente em Java para desktop, construído para acessar os serviços utilizando-se da classe *HTTPClient*. Além de testar o aplicativo, outro objetivo desta aplicação foi demonstrar que o acesso a estes serviços pode ser realizado a partir de diferentes implementações sem levar em conta a plataforma do cliente. Além do cliente *desktop,* foi utilizada também uma pagina JSP com a mesma finalidade de mostar que podemos fazer acessos aos serviço de outras plataformas. Com isto demonstra-se uma das vantagens de REST que é sua interoperabilidade.

Para acessar os serviços, precisa-se apenas, que o cliente manipule chamadas e respostas HTTP. Para exemplificar, um servidor com os serviços de

indexação e pesquisas disponíveis teria o método de acesso, passando um arquivo PDF no corpo da requisição como já demonstrado, com a seguinte chamada a URI:

gelinski.homedns.org:8084/ServicosPlagio/res/arquivo/anderson/titulo/tags/descricao

Após o trecho de endereço “arquivo/”, os parâmetros podem variar conforme a necessidade do cliente. Caso a indexação do arquivo ocorra normalmente o serviço irá retornar um XML de que ele foi recebido e indexado com sucesso. Para usarmos o serviço de buscar usamos a URI:

http://gelinski.homedns.org:8084/ServicosPlagio/res/pesquisa/texto

Esta fará a busca pelo índice dos textos indexados e retornará um XML para posterior manipulação do cliente. Um dos pontos importantes é a possibilidade de alterar a lógica de consulta ou a geração do XML, sem ter que alterar o caminho dos serviços. Ou ainda sem ter que alterar a forma de acesso dos clientes que podem estar implementados em outras linguagens de programação.

3.6 LÓGICA DE IDENTIFICAÇÃO DE PLAGIO

A identificação de plágio é uma tarefa complicada que demanda tempo para o desenvolvimento de algoritmos capazes disso. Está dificuldade foi notada neste trabalho, mesmo que utilizando-se de uma ferramenta que trabalha exclusivamente com indexação e pesquisa de textos. A Lucene oferece diversas maneiras de pesquisa no seu índice. Testes foram feitos com diferentes formas de pesquisa e a que retornou resultados significativos foi a pesquisa pelo termo exato. Mesmo assim existiram várias falhas que precisaram ser resolvidas neste método de pesquisa.

A abordagem na identificação de plágio iniciou-se procurando a melhor forma de identificar se o texto possui vestígios de trechos copiados. As primeiras pesquisas foram feitas utilizando-se apenas a *String* desejada como parâmetro para a busca. Porém, desta maneira elas não retornam resultados significativos, pois a Lucene faz a busca de uma forma particular, por exemplo, pesquisando pela *String* “o gato roeu a roupa do rei” diretamente em um índice onde mesmo sabendo que não existem palavras ou frases semelhantes a esta, resultados são retornados o que dificultava a análise dos retornos.

Outra alternativa foi adicionar o operador “~” ao final da *String* que retorna resultados similares à *String* pesquisada, por exemplo, ao pesquisar “o gato roeu a roupa do rei~”, vários resultados eram retornados devido a grande quantia de caracteres na frase, e com isso, tornavam-se resultados irrelevantes para este estudo.

Outros testes foram feitos adicionando o operador “*” ao final da *String*, mas os testes feitos não mostraram ser uma alternativa eficiente. Para exemplificar:

Ao pesquisar a frase "Universidade Tecnologica Federal do Parana*" em um texto em inglês, com aproximadamente 400 páginas, a *String* retornava resultados, mesmo que, em teoria, isso não devesse acontecer. Então foi descartada a possibilidade do operador “*” ser aplicado ao projeto. Problemas semelhantes ocorreram quando incluído o operador “~” ao final da *String,* tornando-o descartável para a solução desejada.

Utilizando- se do operador aspas tanto no inicio quanto no final da *String* a ser pesquisada por exemplo: "Universidade Tecnologica Federal do Parana – UTFPR". Pequisando este termo exatamente os resultados são mais precisos e é mais fácil a identificação de onde uma copia foi feita.

Para chegar às *Strings* que serão usadas na pesquisa, é necessário decidir antecipadamente como o sistema irá se comportar. A ideia é enviar um arquivo PDF para que o serviço retorne um resultado, informando se aquele arquivo apresentava trechos copiados de algum outro texto que já está indexado.

Para isto, dividiu-se o texto deste PDF em pequenas *Strings* e realizou-se a pesquisa. A principal questão quanto à essa divisão, é o número de palavras que deveriam ser contidas em cada *String*.

Para a divisão utilizou-se a seguinte metodologia, “quebrar” a *String* extraída em um número de palavras pré-definido, ou seja, dividir-se todo o texto do PDF em diversos trechos de 10 palavras cada, por exemplo. Esta divisão afeta diretamente a pesquisa e o tempo para sua execução. Dessa forma, se pré-fixada a divisão em trechos de 15 palavras, a chance de se encontrar em algum texto esta mesma sequência é pequena, mas ainda, se imaginando o exemplo de utilizar uma sequência formada por apenas 2 palavras, a probabilidade de encontrar nos textos indexados essa sequência é muito maior. Mesmo removendo as *stop words* do texto antes de indexá-lo. Com isso também, o número de comparações seria muito maior,

imaginando um texto com 2000 mil palavras, teria-se 1000 mil *Strings* para se pesquisar, o que tornaria o processo lento.

Depois de diversos testes, chegou-se a uma sequência formada por cinco palavras, pois este é um número razoável de palavras para considerar como uma sequência copiada de outros textos e a divisão com este número de palavras não gera um número de comparações tão elevado.

Depois destes testes, verificar plágio desta maneira ainda está longe de ser um método ideal, imaginando-se o seguinte cenário:

Com apenas um trabalho indexado à Lucene, uma pesquisa é feita por meio do envio deste mesmo trabalho. O ideal seria que todas as *Strings* pesquisadas retornassem resultado, pois sabe-se que cada uma delas se encontra no texto que está indexado, mas não é desta maneira que acontece, pois dependo do ponto em que cada *String* é quebrada, a mesma não retorna resultados. Outro problema relacionado a pesquisa é que a Lucene não suporta *Strings* com determinados caracteres especiais e, mesmo removendo acentuações para amenizar o problema, ainda pode ocorrer o aparecimento, de aspas e outros caracteres especiais no meio das pesquisas, fazendo com que a Lucene não pesquise o termo desejado.

Isso limita o número de resultados, tornando-se um problema maior quando executado na procura de plagio em livros por exemplo, onde a quantidade de texto é maior e a presença destes caracteres torna-se mais comum.

3.7 PROBLEMAS ENCONTRADOS

Apesar da implementação de um serviço REST ser relativamente simples, uma boa análise inicial é imprescindível. O serviço sobre esta arquitetura dá margem para várias implementações diferentes, dependendo do método HTTP em questão.

Apesar de que, em um primeiro momento a escolha pode parecer óbvia, nem sempre é desta maneira que acontece durante o desenvolvimento. Para fazer a geração do retorno em XML, foi cogitada a possibilidade de utilizar o método GET, porém essa escolha foi descartada, pois este método não é capaz enviar arquivos para o Servidor. Neste caso, utilizando-se do método POST, tanto no serviço de envio quanto no serviço de pesquisa, possibilitou-se que os arquivos pudessem ser

recebidos e tratados, sem comprometer a resposta em XML para o cliente que realizou o envio.

É necessário entender como os serviços REST tratam as requisições para a criação de um serviço que possa ser acessado de várias plataformas. Para tanto, não deve-se limitar ao cliente uma forma de acesso específica, pois sem isso, a portabilidade de acesso fica comprometida.

Primeiramente ao implementar o serviço de recebimento de arquivo, utilizou-se a anotação *@FormParam*. Com esta anotação tornou-se possível passar os parâmetros contidos em um formulário HTML como atributos de entrada para o serviço implementado, entretanto, tornava-se obrigatório que o serviço sempre receba as informações mediante um formulário, o que acabava por deixar o serviço atrelado demais a uma interface voltada para a plataforma Web, dificultando a implementação de um cliente *Desktop*. Por este motivo, percebeu-se que, para evitar uma maior dificuldade na implementação no cliente, o serviço deveria ser modificado. Neste momento o serviço foi implementado novamente, para com isso, estar apto a receber tanto o formulário, quanto o arquivo contido diretamente no corpo da requisição.

Para tanto, ao invés da anotação *@FormParam*, utilizou-se como parâmetro de entrada a própria requisição, através da anotação *@Context*. A anotação *@Context* foi utilizada para manipular o objeto da classe *HttpServletRequest, pois* com ela é possivel buscar o arquivo inserido na requisição, independente da forma em que ele foi anexo a ela, ou seja, tanto por formulários HTML ou por diferentes formas de programação.

Este era apenas uma parte do todo, com isso resolveu-se apenas o problema de envio do arquivo por diversas plataformas, mas ainda era necessário a construção do Serviço de forma que seguisse a definição inicial, onde é necessário enviar também quem é o autor, o título do trabalho e as outras informações referentes ao arquivo.

Inicialmente procurou-se uma solução que fizesse o envio destes dados diretamente no requisição HTTP junto com o arquivo.

Por se tratar de uma requisição, tentou-se utilizar o método *getParameter*("parametro"), onde “parametro” seria o nome do parâmetro utilizado para envio das informações, mas não conseguiu-se resultados satisfatórios desta maneira, pois ao realizar a chamada ao serviço, o valor do parâmetro sempre estava

nulo. A solução foi utilizar o conceito recursos REST, onde parte do caminho estipulado na URI serve como identificador do conteúdo, desta forma, as informações relevantes passaram a ser infomadas, por exemplo, desta maneira:

caminhoServidor/res/arquivo/autor/titulo/tags/descricao

À partir da parte “arquivo/” do endereço, foram informados os dados dos parâmetros, que variam conforme o arquivo enviado pelo cliente. Para receber os parâmetros desta forma alterou-se a estrutura interna do serviço para que ele pudesse interpretrar cada trecho do caminho como parâmetro.

REST permite que use-se a anotação *@Path* internamente a um método como neste exemplo:

```
@POST
@Path("/{autor}/{titulo}/{tags}/{descricao}")
```

Ou seja quando é executada uma requisição ao serviço de envio de arquivos que está no *@Path*("arquivo"), por meio do método POST, o sistema interpreta que o que for informado após “/arquivo” são parâmetros, mas para manipular os parâmetros, é necessário que o método seja criado da seguinte forma:

```
public String receberArquivo(@Context HttpServletRequest request,
                             @PathParam("autor") String autor,
                             @PathParam("titulo") String titulo,
                             @PathParam("tags") String tags,
                             @PathParam("descricao") String descricao)
```

Quadro 8 - Método executado na chamada do Serviço de envio de arquivo

Usando a anotação *@Context*, é associado que o objeto que contém a requisição da classe *HttpServletRequest* será obtido partindo do contexto em execução.

Conforme dito anteriormente, é por meio do *@Context* que consegue-se recuperar o arquivo independente da forma que foi enviado, e agregado às anotações *@PathParam*, é possivel interpretar os demais dados conforme desejar.

Outro problema enfrentado estava na obtenção do arquivo contido na requisição, pois não basta apenas receber o arquivo, sendo necessário também, salvá-lo em disco rígido para que facilitar sua manipulação. Primeiramente tentou-se

salvar os arquivos enviados utilizando a classe *MultipartRequest* com o seguinte código:

MultipartRequest multipartRequest = new MultipartRequest(request, path);

Onde a variável *path* seria o caminho físico onde o arquivo deveria ser gravado. Depois disto é usado o objeto multipartRequest para salvar o arquivo da seguinte maneira:

multipartRequest.getFile("nome");

O parâmetro “nome” é que foi recebido na requisição juntamente com o arquivo. A grande vantagem deste código é que torna possível receber todos os parâmetros da requisição sem ter que alterar a o URI do serviço a cada envio, por exemplo, para pegar o autor, descrição seria possível proceder, informando cara um dos campos como um parâmreto, desta forma:

multipartRequest.getParameter("autor");

Entretanto, o mapa de parâmetros acabava por retornar sempre vazio ao serviço quando era utilizado um objeto da classe MultipartRequest. Então optou-se por utilizar a Classe *ServletFileUpload,* com ela é possível verificar se existem arquivos na requisição HTTP conforme o código abaixo:

```
if (ServletFileUpload.isMultipartContent(request)) {
        FileItemFactory factory = new DiskFileItemFactory();
        ServletFileUpload upload = new ServletFileUpload(factory);
        upload.setSizeMax(1073741824); //limite do arquivo em bytes
        List<FileItem> items = null;
        try {
            items = upload.parseRequest(request);
        } catch (FileUploadException e) {
            resultStatus = resultStatus + " Erro: " + e.getMessage();
        }
        for (FileItem item : items) {
            if (item.getFieldName().equals("nome")) {
                if (item.getContentType().equals("application/pdf")
                        || item.getName().endsWith(".pdf")) {
                try {
                        String filename = item.getName();
                        item.write(new File(fileRepository + filename));
                } catch (Exception e) {
                resultStatus += "Erro: " + e.getMessage();
                }
            } else {
                resultStatus += "Esse formato de arquivo”
                        + “não suportado.\n Desculpe pelo transtorno.";
```

```
                        }
                    }
                }
        } else {
                resultStatus += "Forma de envio não suportada!";
                }
```

Quadro 9 - Código para analisar a existência de arquivo em uma requisição REST

O grande problema em utilizar este código, se comparado com a classe *MultipartRequest*, está em sua complexidade na implementação.

Em outro ponto do projeto, surgiram dúvidas quanto a como criar os índices na Lucene e que parâmetros utilizar para isto. Quando deseja-se indexar um texto, o mesmo vai passar por um analisador antes, porém não obrigatoriamente este analisador vai retirar *stop words* ou fazer outros ajustes, o que significa que quando indexa-se um texto, pode-se realizar esta tarefa sem que ele sofra modificações.

É correto dizer que isto acarretou em um problema, pois quando é realizada a indexação de um arquivo removendo as *stop words* e pesquisa-se este mesmo arquivo somente quebrando o texto extraido em várias *Strings,* lembrando que este texto não passou pelo analisador, ou seja, ele continua com as *stop words*.

As *strings* geradas para a pesquisa, não batem com as que foram indexados, que estão sem as *stop words*, por exemplo: ao procurar o segmento “o rato roeu a unha” em textos indexados onde a esta mesma frase foi adicionada ao índice, ela foi adicionada como "rato roeu unha", não batendo com o termo que estava sendo pesquisado, devido a remoção das *stop words*. O que acarretou em problemas.

Foi necessário remover também as *stop words* antes de pesquisar para melhorar os resultados.

Verificaram-se problemas ao indexar conteúdo com caracteres acentuados, eles ocorriam no momento da recuperação das informações destes textos, onde os caracteres que foram acentuados retornavam sem um padrão, com caracteres trocados por outros aleatórios, similar ao que acontece como o “lixo de memoria”.

A utilização do *BrazilianAnalyzer* resolveu parte do problema, mas ainda encontrou-se problemas com caracteres acentuados. A solução foi então remover manualmente estes caracteres utilizando uma classe para substituir os caracteres acentuados por seu equivalente sem acento.

Mesmo aplicando o *BrazilianAnalyzer* para indexar os textos, ainda existiam palavras que não retornavam nas pesquisas por detalhes de artigos gramaticais. Para garantir que estes detalhes gramaticais não interferissem na pesquisa a solução foi sua remoção, para não ocorrerem divergênvias do que estava indexado do que se estava procurando. Esta classe criada exclui das *Strings* palavras comuns que não fariam diferença para nossas pesquisas como: teu, teus, todas, todo, tudo, um, uma e etc, além das *stop words*. Com ela conseguiu-se deixa as pesquisas e o texto indexado mais consistente.

Outro problema encontrado foi a criação das *Querys* para pesquisa na Lucene, após quebrada as *Strings* para a pesquisa continuam caracteres especiais que a Lucene não aceita quando inseridos na pesquisa, como por exemplo: ?, $, &, *, [,], {, }, <, >, :, “e etc, isso porque estes caracteres são utilizados para fazer um “*range*” de pesquisas ou no caso do aspas que é utilizado para pesquisar pelo termo exato. Então quando criaram-se as *Strings* de pesquisas precisou-se remover estes caracteres ou a Lucene acusará erro de sintaxe na pesquisa como neste exemplo: “d template=Artist{artist.id field column” a Lucene retorna o seguinte erro: “*Cannot parse 'd template=Artist{artist.id field column': Encountered " <RANGEEX_GOOP> "column "" at line 1, column 34.*” ou seja, devido a *String* conter o caracter “{“ a Lucene não consegue criar a sua *query* o que gera problemas, pois os caracteres acima citados são muito comuns em trabalhos acadêmicos, mas irrelevantes para a pesquisa.

Para reduzir estes problemas estes caracteres foram retirados tanto das pesquisas quanto dos textos que seriam indexados. Entender os critérios de busca da Lucene é uma tarefa que demanda tempo, em uma analise superficial é difícil entender quais são os critérios usados nas buscas.

A maior dificuldade encontrada foi a definição de como seria feita a pesquisa para que fosse um trabalho fosse considerado com plágio, esta detecção não é tão simples quanto parece, a cópia é apenas uma forma de plagiar o conteúdo de um autor. Pode-se considerar plágio também a cópia de uma ideia geral ou uma simples mudança de palavras, mas sem alterar o sentido geral do texto.

A detecção de plágio mais simples é verificar se o autor copiou a obra de algum outro trabalho devido ao fácil acesso a outros trabalhos aliado a facilidade por meio de um computador de simplesmente copiar e colar o conteúdo desejado.

Este trabalho focou neste tipo de identificação devido ao seu baixo nível de complexidade na procura se comparada a outros tipos de plagio, mas com os critérios utilizados a pesquisa ainda é falha devido a diversos fatores, o principal deles é a forma com que a Lucene faz a busca com suas *querys*.

Ela oferece diversas opções de implementações de pesquisa e para uma busca mais eficiente é estudar mais a fundo todo o sistema que envolve as Querys. Da maneira que foi implementada ainda é uma pesquisa ainda superficial se comparado ao que a API da Lucene pode nos oferecer.

4 CONCLUSÃO

O Trabalho com arquitetura SOA se mostra muito vantajoso, aliando SOA com serviços web REST, tem-se uma aplicação distribuída de fácil acesso. Seus principios se encaixaram na definição inicial do projeto e durante seu desenvolvimento. Conseguiu-se aliar os serviços com implementação Java, fazendo a transição de camadas para troca de informações, como arquivos e dados em formato XML sem maiores problemas.

Percebeu-se durante este projeto que REST tem uma fácil e rápida aprendizagem, e sua estrutura permite flexibilidade em sua aplicação. É possivel acessar estes serviços de diversas plataformas sem grandes problemas apenas utilizando bibliotecas adequedas ou HTML simples.

Além da tecnologia para disponibilização de Serviços Web, notou-se que pode-se dispensar menos tempo no desenvolvimento da indexação de arquivos, através da utilização da ferramenta Lucene na indexação e busca de textos, apesar de que esta possui algumas complicações para obter resultados.

Finalizando, considera-se que durante o desenvolvimento, os resultados obtidos estão longe de serem vistos como solução ótima para o objetivo proposto. Entretanto, aceitam-se como satisfatórios, pois possibilitaram o entendimento sobre diversas tecnologias. Também permite-se por meio deste trabalho, servir de embasamento para uma implementação futura com algoritmos mais otimizados que aproximem-se de atender as necessidades no Combate ao Plágio.

4.1 TRABALHOS FUTUROS

Este trabalho abordou principalmente o desenvolvimento do aplicativo em si, a forma como os serviços web se relacionariam com os clientes e com a Lucene. Procurou-se explorar a melhor maneira de montar uma arquitetura que seja a mais genérica possível e que se encaixe em vários contextos com sua ideia principal de utilizar Lucene e Serviços Web. O trabalho da forma que foi conduzido deixou várias possibilidades de trabalhos futuros.

Dentre os trabalhos que podem ser desenvolvidos baseados nesta ferramenta, destacam-se novas regras e métodos de verificação e comparação entre

os arquivos. Um trabalho dirigido somente a algoritmos de busca de plágio por palavras com mesmo sentido, ou uma verificação que utilize vários métodos de busca combinados e que faça um balanceamento entre eles para que em cada situação de pesquisa o melhor método seja aplicado. Ou ainda adicionar uma funcionalidade que utilize sites de busca para fazer a busca em conjunto com os métodos já existente no *framework*.

A construção de clientes em diversas linguagem e plataformas, extendo o alcance da ferramenta já que a mesma se utiliza de serviços REST que são facilmente acessíveis de diversas plataformas.

O modo em que a ferramenta foi construída permite ainda a criação de extenções para sistemas ou ambientes já existentes, como o ambiente colaborativo MOODLE, por exemplo.

A criação de respostas mais complexas para serem usadas em relatórios mais completos, ou criar métodos de geração de XMLs de retorno com mais informações e personalização, por exemplo, um cliente gostaria de retornar somente o autor dos textos, retornar somente o títulos dos mesmos, isso seria possível com a personalização dos retornos.

Aumentar a possibilidades de formatos de arquivo que o sistema aceita além de PDFs. incluir também DOC, TXT, RTF e etc.

Explorar a fundo todas as possibilidades que a Apache Lucene oferece, pois foi trabalhado muito superficialmente com o que a ferramenta é capaz de fazer. Direcionar um estudo que busque a melhor forma de indexar os conteúdos e de recuperar as informações contidas no seu index.

Desenvolver novos serviços além dos já existentes para indexação e pesquisa. Como por exemplos, serviços que façam pesquisas alem da base de dados já estabelecida.

É possível também a implementação de uma pesquisa que retorne resultados considerando um percentual pré-definido, para que trabalhos ou textos que tenham partes de pouca relevância em comum não sejam acusados nos resultados de plágio, pois, em trabalhos acadêmicos sempre há partes escritas fixas, e no método atual de pesquisa, estas podem ser consideradas como plágio.

REFERÊNCIAS

CERAMI, Ethan. **Web Services Essential.** Distributed Applications with XML-RPC, SOAP, UDDI & WSDL. O'Reilly, 2002.

ENGLANDER, Robert**. Java and SOAP.** Sebastopol: O'Reilly, 2002.

ERL, Thomas. **SOA: Principles of Service Design.** Crawfordsville :Prentice Hall. 2007

ERL, Thomas. **SOA: Design Patterns.** Crawfordsville :Prentice Hall. 2009

FIELDING, Roy T. **Architectural Styles and the Design of Network-based Software Architectures.** 2000. 180 f. Dissertação (Doutorado) - Information and Computer Science, University of California. Irvine. 2000.

FRONDANA, Giovani Ten.; BARBOSA, Victor D. T. J. Ten.; GALVÃO, Rodrigo F. Al. **Arquitetura Orientada a Serviços para Gestão de Processos Acadêmicos na Web.** 2009. 173 f. Projeto de Final de Curso (Graduação) – Curso de Graduação em Engenharia de Computação, Instituto Militar de Engenharia. Rio de Janeiro, 2009.

FURTADO, José A. P. X**. Trabalhos acadêmicos em Direito e a violação de direitos autorais através de plágio**. Disponível em: <http://jus.uol.com.br/revista/texto/3493/trabalhos-academicos-em-direito-e-a-violacao-de-direitos-autorais-atraves-de-plagio>. Acesso em: 24 de Agosto de 2011.

IETF. **RFC 1945:** Hypertext Transfer Protocol -- HTTP/1.0. [sine loco], 1996.

IETF. **RFC 2616:** Hypertext Transfer Protocol -- HTTP/1.1. [sine loco], 1999.

GRAHAM, Steve. et al. **Building Web Services with Java:** Making Sense of XML, SOAP, WSDL and UDDI. Sams Publishing, 2001.

Jboss Enterprise. **RESTEasy:** Java Docs. <http://docs.jboss.org/resteasy/docs/2.3-beta-1/javadocs/index.html>. Acesso em: 27 de Setembro de 2011.

KUROSE, James F.; ROSS, Keith W. **Redes de Computadores e a Internet:** Uma Abordagem Top-down. São Paulo: Pearson Addison Wesley, 2006.

LAURENT, Simon St.; JOHNSTON, Joe; DUMBILL, Edd. **Programing Web Services with XML-RPC.** O'Reilly, 2001.

LORENZETTI, Celso. **Framework para Persistência de Dados.** 2004. 102 f. Trabalho de Conclusão de Curso – Curso de Ciência da Computação, Centro Universitário Feevale. Novo Hamburgo, 2004.

MCCANDLESS, Michael; HATCHER, Erik; GOSPODNETIĆ, Otis**. Lucene in Action.** 2nd. Edition. Stamford: Manning, 2010.

RICHARDSON Leonard; RUBY, Sam. **RESTful Web Services.** Sebastopol: O'Reilly, 2007.

ROMANCINI, Richard. A praga do plágio acadêmico. **Revista Científica FAMEC / FAAC / FMI / FABRASP.** Ano 6, n. 6, p. 44-48. Disponível em: <http://www.wannydigiorgi.com.br/paginas/publi/revista2007.pdf#page=44>. Acesso em: 24 de Agosto de 2011.

SANDOVAL, Jose**. RESTful Java Web Services.** Birmingham: Packt Publishing, 2009.

SMILEY, David; PUGH, Eric. **Solr 1.4 Enterprise Search Server.** Birmingham: Packt Publishing, 2009.

TANENBAUM, Andrew S. **Redes de Computadores.** 4ª edição. Editora Campus (Elsevier), 2003.