

UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
CURSO DE ENGENHARIA DE COMPUTAÇÃO

GABRIEL CASAGRANDE STABEL

**Estudo de Tecnologias para o Desenvolvimento
de Aplicação Web para Gerenciamento de
Corretora de Seguros**

Monografia apresentada como requisito parcial para
a obtenção do grau de Bacharel em Engenharia de
Computação.

Orientador: Prof. Dr. Marcelo Soares Pimenta

Porto Alegre

Junho de 2014

UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL

Reitor: Prof. Carlos Alexandre Netto

Vice-Reitor: Prof. Rui Vicente Oppermann

Pró-Reitor de Graduação: Prof. Sérgio Roberto Kieling

Diretor do Instituto de Informática: Prof. Luís da Cunha Lamb

Coordenador do Curso de Engenharia de Computação: Prof. Marcelo Götz

Bibliotecária-Chefe do Instituto de Informática: Alexsander Borges Ribeiro

AGRADECIMENTOS

Eu agradeço aos meus colegas da corretora Stacas pela colaboração com as ideias que deram origem a esse projeto. Em especial agradeço à minha mãe, Ana Maria Casagrande, e meu colega Carlos Guilherme Fleck. Espero que esse esforço gere bons frutos no futuro.

Agradeço também ao meu orientador, professor Pimenta, por toda a ajuda e por aturar minha ansiedade e confusões.

Ao meu amor, agradeço por corrigir meu português e por mandar eu ir trabalhar sempre que eu precisei de um empurrão.

RESUMO

Este trabalho apresenta um projeto para desenvolvimento de uma aplicação web para o gerenciamento de uma corretora de seguros. No trabalho serão ressaltadas as necessidades do sistema, suas principais características e como ele integra as tecnologias específicas que foram escolhidas para resolver essas necessidades, incluindo MongoDB, AngularJS, e Solr.

Palavras-chave: Corretora de Seguros. Aplicação Web. MongoDB. AngularJS. Solr.

Web Application for Insurance Brokers Management

ABSTRACT

This paper presents a project to develop a web application for managing an insurance broker. In this paper will be highlighted the system needs, its main features, and how it integrates specific technologies that were chosen to address these needs, including MongoDB, AngularJS and Solr.

Keywords: Insurance Broker. Web Application. MongoDB. AngularJS. Solr.

LISTA DE FIGURAS

Figura 1 - Diagrama geral da arquitetura e tecnologias associadas.....	16
Figura 2 - Exemplo de comunicação cliente-servidor no debug do Chrome.....	18
Figura 3 - Interface gráfica simples implementada com Bootstrap e AngularJS.....	20
Figura 4 - Definição de colunas de uma tabela de clientes usando MySQL.....	23
Figura 5 - Comando SQL para a criação de uma tabela de clientes.....	23
Figura 6 - Exemplo de documento em MongoDB.....	24
Figura 7 - Função definida pelo usuário para auto-incremento.....	24
Figura 8 - Exemplo de documento em MongoDB com listas e objetos.....	25
Figura 9 - Relação entre três tabelas SQL: clientes, contatos e endereços.....	26
Figura 10 - Relação de clientes, contatos e endereços em uma coleção de MongoDB.....	26
Figura 11 - Relação muitos-para-muitos em MongoDB.....	27
Figura 12 - Exemplo de herança em seguros de ramos diferentes.....	29
Figura 13 - Código para criação de índice de busca em SQL e MongoDB.....	29
Figura 14 - Comando SQL para definição de uma restrição entre a tabelas.....	30
Figura 15 - Exemplo de código em JQuery modificando o HTML.....	32
Figura 16 - JQuery se comunicando com o servidor e modificando o HTML.....	33
Figura 17 - Exemplo de código em AngularJS sem modificar o HTML.....	34
Figura 18 - AngularJS sem modificar o HTML e se comunicando com o servidor.....	36
Figura 19 - Declaração de controlador e hierarquia de escopo em AngularJS.....	37
Figura 20 - Exemplo de código de template em AngularJS.....	38
Figura 21 - Exemplo de diretiva customizável.....	38
Figura 22 - Criação de serviço e uso com dependency injection.....	39
Figura 23 - Sistemas da Liberty e Porto Seguro não possuem busca unificada.....	40
Figura 24 - Busca atual do sistema da Stacas.....	41
Figura 25 - Comando de busca em duas colunas, em SQL e MongoDB.....	42
Figura 26 - Busca em SQL usando match-against.....	42
Figura 27 - Diagrama de busca unificada em SQL com tabela auxiliar.....	44
Figura 28 - Servidor e interface administrativa do Solr sendo executada.....	45
Figura 29 - Definição de esquema e documentos correspondentes.....	46
Figura 30 - Exemplo de pesquisa utilizando Solr.....	48
Figura 31 - Solicitação de busca unificada de “joao silva”, e debug no Chrome.....	50
Figura 32 - Requisição HTTP para o servidor Solr de “joao silva”.....	51
Figura 33 - Resposta do Solr à aplicação-servidor e consulta ao MongoDB.....	52
Figura 34 - Resultado da busca na aplicação-cliente.....	52
Figura 35 - Formulário de novo cliente.....	54
Figura 36 - Código HTML com AngularJS e variáveis correspondentes no controller.....	54
Figura 37 - Código de inserção e documento armazenado no MongoDB.....	55

SUMÁRIO

RESUMO	6
LISTA DE FIGURAS.....	8
1 INTRODUÇÃO	9
2 CONTEXTO E REQUISITOS DO SISTEMA	11
2.1 Flexibilidade na manipulação do banco de dados.....	13
2.2 Interface ágil separada da lógica do programa	13
2.3 Busca unificada.....	14
3 ARQUITETURA DO SISTEMA	16
3.1 Aplicação-servidor.....	17
3.1.1 Considerações sobre segurança	18
3.2 Interface cliente-servidor	18
3.3 Aplicação-cliente.....	19
4 ADOTANDO O BANCO DE DADOS MONGODB	21
4.1 Estrutura e formato dos campos.....	21
4.2 Chave primárias	23
4.3 Funções definidas pelo usuário	24
4.4 Listas e documentos embutidos	25
4.4.1 Implementando relações um-para-muitos (1..N)	26
4.4.2 Implementando relações muitos-para-muitos (M..N)	27
4.5 Implementação de heranças.....	28
4.6 Índices de buscas	29
4.7 Restrições e chaves estrangeiras.....	29
5 CRIANDO INTERFACES COM USUÁRIO DE FORMA DECLARATIVA COM ANGULARJS	31
5.1 Interfaces de usuário declarativas (<i>Declarative View</i>).....	33

5.2	Vinculação de dados no dois sentidos (<i>Two-way data binding</i>)	35
5.3	Arquitetura modelo-visão-controlador (MVC, <i>Model-View-Controller</i>)	36
5.3.1	Controladores (<i>Controllers</i>) e Escopo (<i>Scope</i>)	37
5.4	Templates	37
5.5	Diretivas customizáveis	38
5.6	Dependency Injection	39
6	REALIZANDO BUSCA UNIFICADA COM SOLR	40
6.1	Busca unificada em SQL	41
6.1.1	Buscas com índices FULL TEXT em MySQL	42
6.1.2	Relevância dos campos e múltiplas buscas	43
6.1.3	Lógica para identificar campos	43
6.1.4	Campos específicos para buscas	43
6.2	Busca unificada com SOLR	44
6.2.1	Configuração dos dados	45
6.2.2	Índices de busca	47
6.2.3	Pesquisas com eDisMax	48
7	EXEMPLOS DE USO DAS TECNOLOGIAS NO SISTEMA	50
7.1	Localizando um cliente com a busca unificada	50
7.2	Criando um cliente com AngularJS e salvando no MongoDB	53
8	CONCLUSÕES	57
8.1	Resumo de resultados	57
8.2	Limitações do trabalho	57
8.3	Perspectivas futuras	58
	REFERÊNCIAS	59
	ANEXO A – TRABALHO DE GRADUAÇÃO 1	60

1 INTRODUÇÃO

As tecnologias de software para a internet evoluíram muito, e rapidamente, nos últimos anos. Em especial, houve um aumento em tecnologias voltadas para permitir complexas e ricas aplicações baseadas em navegadores, aplicações web, que não eram possíveis no passado. Surgiram novas linguagens, bibliotecas e frameworks que abriram muitas possibilidades em termos do que pode ser desenvolvido na web.

Apesar dos avanços, nosso mercado de software no Brasil, na média, ainda está longe de absorver e transformar todos esses novos potenciais em realidade dentro das empresas. No ramo específico de seguros isso não é diferente. Tanto as seguradoras como as corretoras de seguros no geral ainda utilizam tecnologias antigas, e, infelizmente, muitas vezes com implementações de baixa qualidade. Isso impacta diretamente no desempenho do trabalho.

O objetivo desse trabalho é o estudo de novas tecnologias e a implementação de uma moderna e ágil aplicação web para o gerenciamento de uma corretora de seguros chamada Stacas Corretora de Seguros. Dependendo do resultado do trabalho, da capacidade de investimento da Corretora ou de parcerias comerciais, o sistema desenvolvido será comercializado para outras corretoras.

O trabalho apresentará uma estrutura geral para o sistema, mas enfocará alguns tópicos mais importantes encontrados durante seu desenvolvimento. A partir das necessidades da Corretora para o sistema, as abordagens tradicionais serão comparadas com as alternativas modernas. Por ter um viés mais prático, não tão teórico, esse trabalho pretende se focar mais em como as novas tecnologias se aplicam nos casos concretos neste sistema, e não em todo potencial que essas tecnologias possuem.

A divisão do trabalho ocorrerá da seguinte forma: no capítulo 2 será apresentado o contexto no qual a Corretora se insere e como esse novo sistema pode auxiliá-la. Serão explicadas quais são as necessidades da Corretora em relação ao seu sistema atual e como o novo sistema pretende inová-las, sob a perspectiva tanto do usuário, como também do desenvolvimento e manutenção.

No capítulo 3 será apresentada a arquitetura geral do sistema e as diversas tecnologias associadas a cada parte. Será explicado como essas partes se interligam, e onde estão, no conjunto, as partes inovadoras que serão aprofundadas do decorrer do trabalho.

Três problemas principais foram identificados durante o desenvolvimento. Eles serão investigados nos capítulos seguintes.

O capítulo 4 tratará sobre a flexibilidade na manipulação do banco de dados. Será apresentado o MongoDB, um banco de dados sem esquema de colunas e sem interface SQL. Diferentemente dos bancos relacionais, em que cada registro é armazenado com uma estrutura rígida, MongoDB armazena-os em documentos flexíveis capazes de suportar arrays e outros documentos embutidos. Analisaremos o impacto desses novos conceitos em relação aos bancos de dados relacionais comuns, e ainda como ficam as relações entre tabelas, a possibilidade de salvar novas abstrações, por exemplo, salvar heranças no próprio banco de dados, e diversas outras diferenças.

No capítulo 5, analisaremos como podemos construir uma interface ágil e separada do resto da lógica do programa. Veremos como podemos utilizar o novo conceito de interface declarativa com o AngularJS para construirmos um código que represente estaticamente, de forma não-imperativa, os conceitos dinâmicos da interface. Analisaremos como essas mudanças propostas pelo AngularJS se comparam com as alternativas convencionais de manipulação direta da DOM, que teve seu uso popularizado com a biblioteca JQuery.

Por último, no capítulo 6, vamos abordar a necessidade de uma busca unificada. Vamos discutir como as técnicas tradicionais implementam isso em bancos relacionais com SQL, e como a ferramenta Solr pode simplificar o desenvolvimento com ótimo resultado. Com ela pode-se, além de encontrar quaisquer dados, ter os mais relevantes como prioridade.

Ao final, no capítulo 7 e 8, vamos fazer um apanhado geral do trabalho e analisaremos os resultados obtidos no desenvolvimento.

2 CONTEXTO E REQUISITOS DO SISTEMA

Esse projeto visa ao desenvolvimento de um Sistema Informacional Computadorizado - também denominado de Sistema de Informação ou Sistema de Gerenciamento - para uma corretora de seguros, títulos de capitalização e previdência privada. **O sistema será utilizado como a base de organização de toda atividade-fim da empresa.** Será usado para gerenciar todos os processos internos (exceto controle de caixa, bancário e fiscal), desde as negociações em andamento até o arquivamento digital das apólices dos seguros fechados.

A corretora em questão chama-se Stacas Corretora de Seguros, fundada em 2003 por Ana Maria Casagrande, e tem como sócio seu filho e autor deste projeto. Apesar de sócio, o autor não exerce a atividade-fim da Corretora de seguros desde 2007, mas sim trabalho de suporte e desenvolvimento de sistema de banco de dados da Corretora.

A Corretora Stacas tem hoje um banco de dados com registro de mais de 2.300 clientes e 9 mil seguros contratados. Do total de clientes da Corretora, cerca de mil são clientes ativos no momento, isto é, com apólices de seguros vigentes. A Stacas hoje conta com apenas um escritório em Porto Alegre, mas no período de 2003 até 2007 contava, além desse escritório, com duas lojas em concessionárias de veículos, uma em Porto Alegre e outra em Canoas. Esse fato, de a Corretora ter sede em três locais diferentes, foi o principal motivo que levou à necessidade da criação de um sistema pela internet.

Dependendo do resultado e da capacidade de investimento da Corretora, a ideia futura será transformar esse novo sistema a ser desenvolvido em algo mais customizável, para poder ser comercializado com outras corretoras de seguros. Existe a possibilidade também de realizar alguma parceria com alguma seguradora para comercializar o sistema, mas essas opções serão avaliadas em um segundo momento do projeto.

O sistema utilizado hoje em dia pela Corretora é uma aplicação baseada na internet (*web-based*) desenvolvida entre 2005 e 2007. Houve uma pequena expansão em 2009 e outra em 2013, mas a parte central do sistema permaneceu inalterada desde a primeira versão. O servidor web está atualmente hospedado na Localweb, e é utilizada uma aplicação implementada em PHP3 utilizando um banco de dados MySQL. Já a aplicação cliente foi desenvolvida em HTML4 e JavaScript, ambas versões compatíveis somente com o navegador Mozilla Firefox. Na época do

desenvolvimento, o navegador Internet Explorer 6 apresentava muitos problemas, o Chrome não existia, e o Safari e outros eram pouco utilizados. Tanto na aplicação do servidor quanto no cliente não foram utilizadas bibliotecas externas adicionais, apenas as funcionalidades nativas das respectivas linguagens de programação. A interface gráfica foi desenvolvida com CSS2 puro e poucas imagens, também sem auxílio de bibliotecas externas.

Do ponto de vista técnico, um sistema de gerenciamento de uma corretora de seguros (uma empresa do ramo de serviços) pode ser visto como uma clássica aplicação de banco de dados (*database application*). Em poucas palavras, a função do sistema pode ser resumida em cadastrar, atualizar e obter informações de um banco de dados, com o intuito de organizar as atividades internas da empresa e poder realizar o serviço, ou sua atividade-fim, aos seus clientes.

Esse tipo aplicação segue uma arquitetura cliente-servidor, na qual o servidor concentra os dados e processa as requisições de diversos clientes. Durante muito tempo esses sistemas foram desenvolvidos com aplicações específicas no servidor e no cliente. Geralmente desenvolvidas para apenas um sistema operacional - no caso do Brasil, o Windows sempre foi o mais comum - com alto de manutenção e instalação associados. A cada atualização todas as aplicações-cliente precisavam ser atualizadas, gerando novos problemas de compatibilidade e configurações. As aplicações no servidor normalmente precisavam ser interrompidas para ser atualizadas.

Com a ubiquidade da internet e dos equipamentos com navegadores (computadores, notebooks, tablets, smartphones, etc), e com a capacidade dos hardwares atuais, das linguagens de programação, e todo o imenso conjunto de bibliotecas e tecnologias desenvolvidas para a web, que permitem aplicações ilimitadas nesse ambiente, uma aplicação moderna de banco de dados deve ser desenvolvida para esse ambiente. Mesmo que o sistema não seja usado na internet propriamente, mas apenas em uma rede local, as tecnologias se transpõem sem nenhum grande problema. Ainda há de fato problemas nas implementações e padronizações entre navegadores e sistemas operacionais diferentes, mas iniciativas como a normatização do HTML5 garantem hoje um grau de compatibilidade cruzada muito grande.

Construir uma aplicação web significa, na verdade, desenvolver dois aplicativos, com características, linguagens, e preocupações diferentes. O software no servidor necessita, no processo, lidar diretamente com o banco de dados, garantir a consistência dos dados, lidar com múltiplos acessos e assegurar a segurança das informações. É a base da aplicação, já que lida com os dados na sua forma crua, manipula as solicitações do usuário e as traduz para os detalhes reais do armazenamento.

O software cliente, por sua vez, define a interface gráfica com o cliente. No modelo web, isso será feito por páginas HTML com scripts sendo executados no navegador. Essa aplicação precisa ter um aspecto visual para tornar a apresentação

dos dados clara. Além disso, precisa filtrar as entradas do usuário, que são feitas por um apontador (mouse ou telas sensíveis ao toque) ou por uma entrada de texto. Essas duas aplicações precisam se comunicar por uma interface clara que está sujeita a todos os problemas de uma rede, como instabilidades e quedas na conexão e, ainda, a insegurança.

Definidos o ambiente de utilização e as necessidades do sistema podemos nos centrar agora nos principais problemas que vão ser abordados nesse trabalho.

2.1 Flexibilidade na manipulação do banco de dados

Tradicionalmente, a modelagem e estruturação dos bancos de dados são uma tarefa complexa. As informações a serem armazenadas no nível do usuário precisam ser mapeadas para as estruturas de dados correspondentes do banco de dados, e, assim, configuradas em tabelas e campos.

Dentre os bancos mais utilizados de código aberto estão MySQL e PostgreSQL. Ambos utilizam interface SQL para comunicação com a maioria dos bancos de dados. Nessas tecnologias, as definições dos campos costumam ser tecnicamente rígidas (um campo assume apenas um tipo de dado) e muito específicas (ocupa exatamente tantos bytes). Além disso, para poderem ser manipuladas pela aplicação no servidor essas definições precisam ser mapeadas e convertidas para outros formatos.

Do mesmo modo, o desenvolvimento não só necessita definir essa organização, mas também precisa ser feito pensando na necessidade de mudanças no banco de dados, e na aplicação como um todo, ao longo do tempo de vida útil do sistema. Novos dados precisam ser armazenados, formatos precisam ser alterados, vinculações precisam ser mudadas etc. Essa característica é referida muitas vezes como manutenibilidade (*maintainability*). As mudanças podem ser necessárias tanto por alterações nos produtos e serviços prestados, como na própria forma de organização do trabalho na Corretora. Como exemplo dessa dificuldade, o sistema atual da Corretora, que sofreu pouca manutenção, apresenta diversos casos de campos faltando, de organizações incompletas e vinculações entre dados desatualizadas. Essas informações costumam ser armazenadas em texto puro para suprir a necessidade dos usuários.

A necessidade de manutenção associada à complexidade natural dos bancos de dados representa muito esforço no desenvolvimento. Para lidar com esse entrave será utilizado o banco de dados MongoDB, que tem uma estrutura de armazenamento diferente, e é o mais utilizado banco de dados que não utiliza uma interface SQL (também chamado de NoSQL).

2.2 Interface ágil separada da lógica do programa

Como esse sistema será utilizado no dia-a-dia da Corretora, sua interface gráfica deve ser o mais ágil para o usuário quanto possível. Se possível, tão rápida quanto um aplicativo convencional executado localmente. Deve, também, ser simples de

utilizar tendo o mínimo de opções necessárias. Ambas características têm um impacto direto no modo como os usuários interagem com o sistema, no tempo de adaptação desses usuários e, conseqüentemente, ainda que de maneira indireta, em sua produtividade.

Como a aplicação do cliente será executada em navegadores, definir a interface significa lidar com suas características tecnológicas. No início do uso de sistemas *web-based*, a interface era criada como um conjunto de páginas estáticas e cada mudança na interface significava uma requisição ao servidor de uma nova página. Nesse processo, havia todo o tempo de comunicação e transferência somado ao de processamentos para a exibição de uma nova página. Há algum tempo isso não é mais desejável, nem necessário. O próprio sistema atual da Stacas - assim como outros sistemas mais modernos - já utiliza scripts em JavaScript, que são carregados durante a inicialização para dinamicamente modificar a página, sem a necessidade de comunicação com o servidor.

Essa manipulação direta via JavaScript (ou usando sua biblioteca mais popular JQuery) possibilita uma interface ágil, mas com um grande esforço em termos de programação, especialmente na organização do código fonte. Pois sendo o JavaScript uma linguagem imperativa, quando uma mudança na interface é necessária, ela fica normalmente fragmentada em diversas instruções e misturando elementos da lógica da aplicação com elementos visuais.

Ao utilizar uma abordagem declarativa para a interface, com o auxílio da biblioteca AngularJS, podemos obter a mesma agilidade com uma clareza maior na definição e no código, que contemple seus aspectos dinâmicos, e separe melhor a lógica da interface.

2.3 Busca unificada

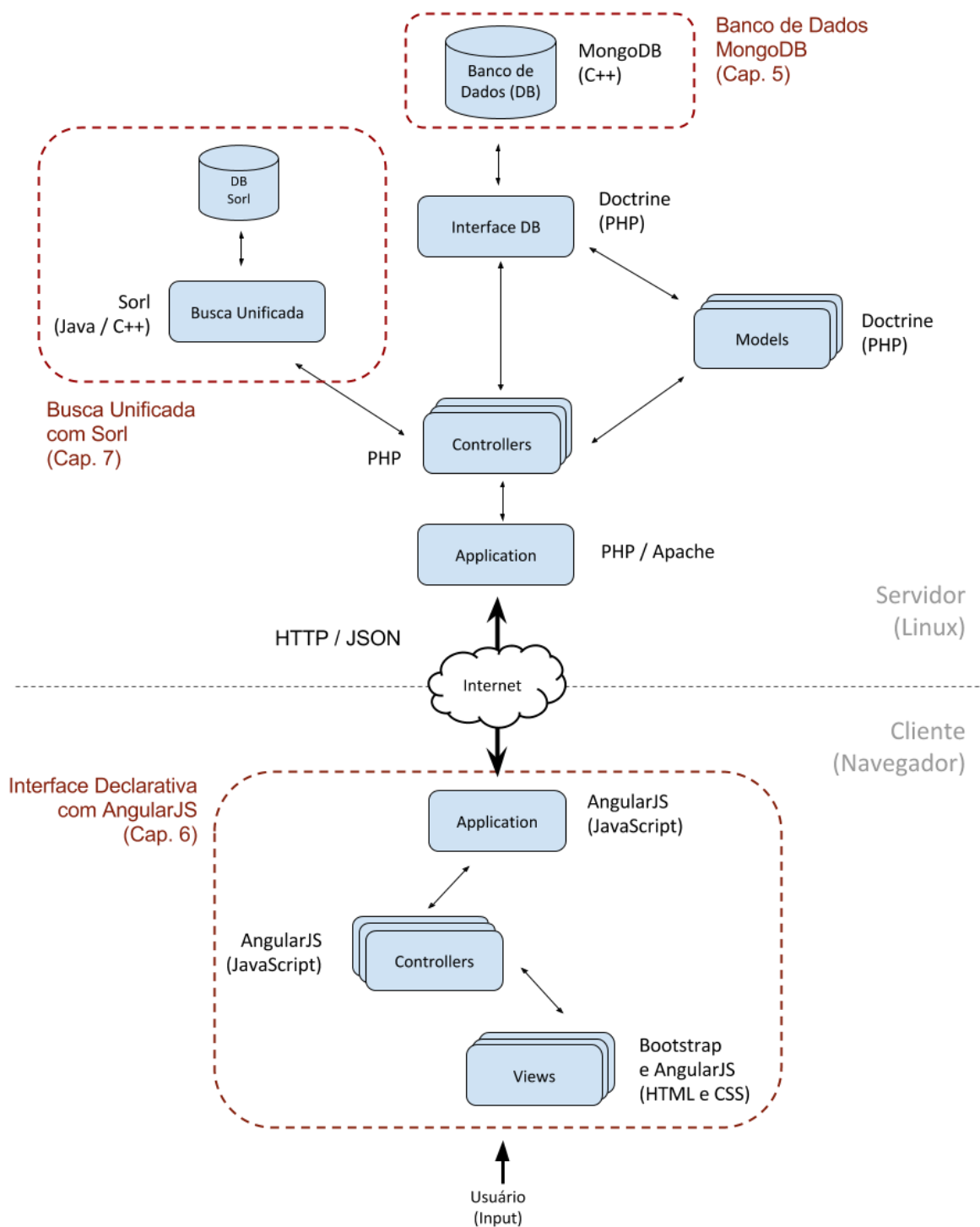
Uma situação comum na Corretora é ter de localizar as informações de um seguro, ou entender a situação de uma negociação em andamento, ou de um sinistro, a partir de pouca informação inicial. Os seguros, na maioria das vezes, envolvem diversas pessoas. Um seguro de automóvel, por exemplo, pode estar em nome de uma pessoa, sendo que outra pessoa da família é o motorista e um terceiro é o proprietário do veículo. Fianças locatícias normalmente envolvem duas empresas com seus diferentes diretores e secretários. Geralmente, das pessoas relacionadas ao seguro, o cliente que contratou o serviço não recorda dos detalhes quando precisa de um auxílio ou está com uma dúvida. Como o seguro é feito por um prazo longo, e só é utilizado em caso de emergência, as informações relativas ao seguro costumam ser esquecidas pelos envolvidos e cabe à Corretora localizar a informação de forma eficiente. Soma-se a isso o fato de existir vários funcionários na Corretora, que lidam com diferentes etapas do processo de um seguro separadamente, e temos uma situação muitas vezes trabalhosa para localizar as informações.

Por isso, um sistema de busca unificado, no qual diversas informações possam ser localizadas, e com uma interface que possa apresentar um resumo da situação dos clientes se torna tão necessário. Pode-se desenvolver um sistema de busca de diversas maneiras utilizando os sistemas de busca padrão dos bancos de dados, mas todas as alternativas envolvem um programação complexa. Ferramentas específicas, como o Solr, uma plataforma do Apache Lucene, pode simplificar em muito essa tarefa da aplicação-servidor.

3 ARQUITETURA DO SISTEMA

Antes de adentrarmos nos problemas principais, precisamos apresentar a arquitetura geral do sistema.

Figura 1 - Diagrama geral da arquitetura e tecnologias associadas.



Como dito anteriormente, o sistema como um todo pode ser visto como duas aplicações: servidor e cliente, que se comunicam por uma interface comum.

3.1 Aplicação-servidor

A aplicação-servidor foi desenvolvida utilizando a linguagem PHP, com o paradigma de orientação a objetos, rodando em um servidor Apache, em uma plataforma Ubuntu/Linux.

Com auxílio da biblioteca Doctrine ODM (*Object Document Mapper*) foi implementada uma arquitetura MVC (*Model-view-controller*), tradicional do ponto de vista do servidor. Toda a interface gráfica do usuário, isto é, a aplicação-cliente no navegador, é vista como a *view*, que solicita as requisições que são tratadas pelo servidor. Já o servidor, em sua classe principal, recebe todas as requisições de usuários e repassa ao *controller* correspondente, que, por sua vez, acessa os respectivos *models* necessários em sua lógica, tudo isso utilizando o mapeamento do Doctrine para tornar os dados permanentes no banco de dados MongoDB.

As características da linguagem PHP e da arquitetura MVC não serão abordadas nesse trabalho por serem temas bem debatidos e explicados. Já o uso e os detalhes do MongoDB serão apresentados no capítulo seguinte.

O sistema de busca unificado também será tratado como um assunto à parte. Foi implementado com auxílio da aplicação Solr, que será analisada no capítulo 6.

3.1.1 Considerações sobre segurança

Por se tratar de um sistema na rede, não há nenhuma garantia de que o cliente irá se comportar como previsto. Qualquer requisição pode ser adulterada pelo usuário com um pouco de conhecimento técnico. Por isso, além do seu papel principal de manipular as informações, a aplicação no servidor precisa ser robusta para lidar com qualquer requisição espúria.

Para atender a isso, na arquitetura proposta há apenas um ponto de entrada para todas as requisições ao servidor, com uma classe específica para o tratamento da entrada. Além disso, há um tratamento geral de exceções (*exceptions*) para garantir que qualquer erro no servidor seja filtrado antes de retornar ao usuário alguma informação suscetível.

Por segurança, o servidor Apache é configurado especialmente para garantir que nenhum outro script seja executado além do *index.php*.

3.2 Interface cliente-servidor

A comunicação entre cliente-servidor ocorre trocando arquivos no formato JSON utilizando o protocolo HTTP. Por ser nativo do JavaScript, o processo de conversão entre objetos em *string* nesse formato, e vice-versa, é simples e eficiente. As *flags* de status já presentes no protocolo HTTP são utilizadas pelo servidor e o cliente para sinalizar problemas de autenticação e erros não tratados no servidor.

Todas as comunicações ocorrem respeitando um padrão simples de troca de mensagens inspirado no protocolo RPC. A requisição é sempre realizada informando o *model* e a *action* desejados, passando um objeto com os *values*, que são os parâmetros da solicitação. A resposta consta sempre de um campo booleano informando o *status* da requisição e um campo *result* que é um objeto cujo conteúdo depende do que foi solicitado.

Figura 2 - Exemplo de comunicação cliente-servidor no debug do Chrome.

```
▼ Request Payload view source
▼ {model:cliente, action:read, values:{clienteId:9}}
  action: "read"
  model: "cliente"
  ▼ values: {clienteId:9}
    clienteId: 9
```

× Headers **Preview** Response Cookies Timing

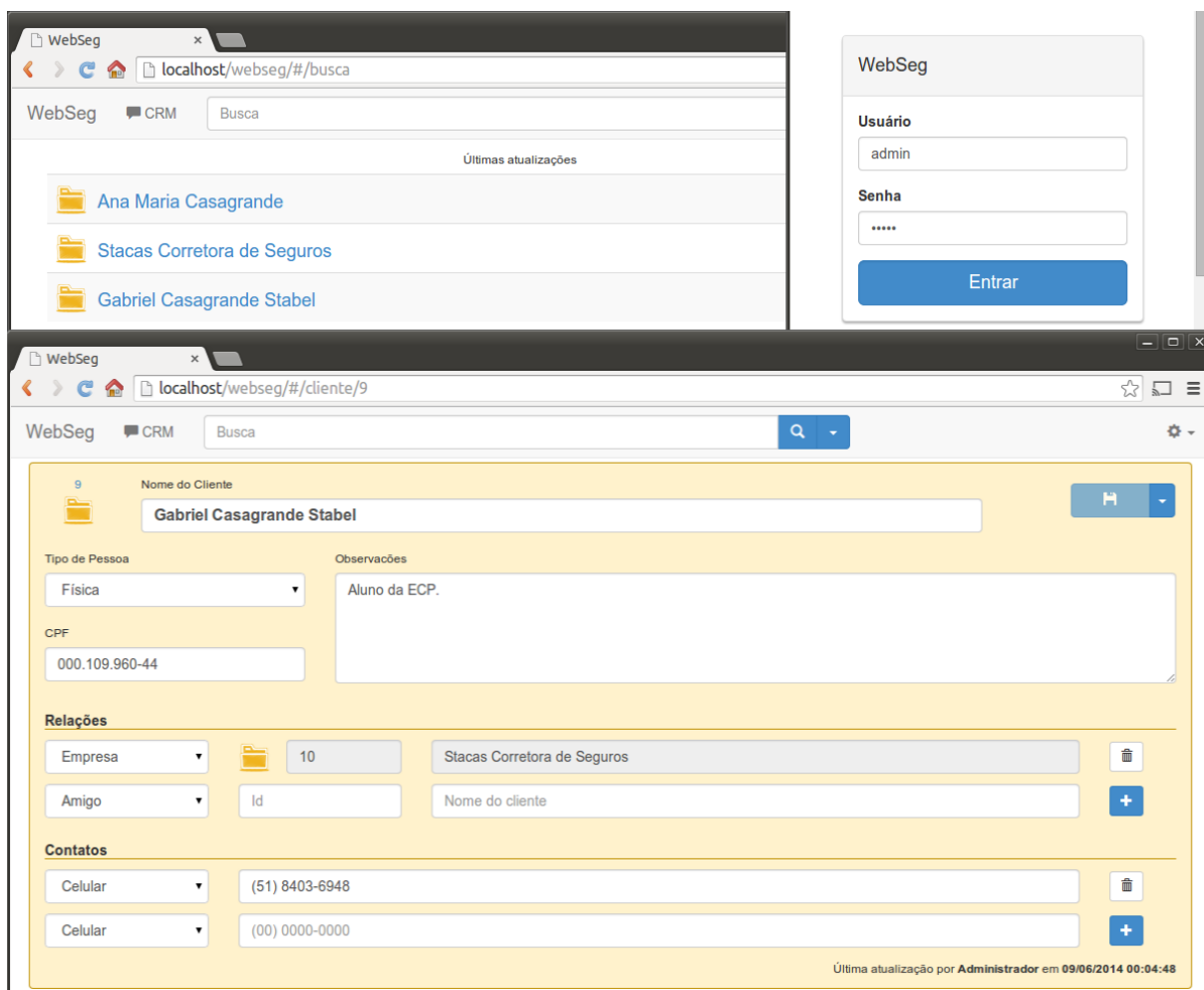
```
▼ {status:true,...}
▼ result: {id:539522e0f727a8451e551412, clienteId:9, nome:Gabriel Casagrande Stabel, tipo:fisica,...}
  cgc: "000.109.960-44"
  clienteId: 9
  ▼ contatos: [{tipo:celular, valor:(51) 8403-6948}]
    ► 0: {tipo:celular, valor:(51) 8403-6948}
      id: "539522e0f727a8451e551412"
      nome: "Gabriel Casagrande Stabel"
      obs: "Aluno da ECP."
  ▼ relacoes: [{tipo:empresa, clienteId:10, nome:Stacas Corretora de Seguros}]
    ► 0: {tipo:empresa, clienteId:10, nome:Stacas Corretora de Seguros}
      tipo: "fisica"
      updatedAt: "09/06/2014 00:04:48"
      updatedBy: "Administrador"
  status: true
```

3.3 Aplicação-cliente

Toda a interface de usuário, a interface gráfica, é tratada pela aplicação-cliente no navegador. A aplicação foi desenvolvida com auxílio da biblioteca gráfica Bootstrap e em conformidade com as normas do HTML5 e CSS3. Por se tratar de uma aplicação de banco de dados, os elementos usados no geral são formulários simples e listas, apenas com cores e imagens para facilitar a identificação do usuário.

A questão mais inovadora da aplicação-cliente será a utilização da biblioteca AngularJS, que será explicada com maior detalhamento no capítulo 5.

Figura 3 - Interface gráfica simples implementada com Bootstrap e AngularJS.



4 ADOTANDO O BANCO DE DADOS MONGODB

Bancos de dados relacionais (*relational database*) com interface SQL (*Structured Query Language*) se tornaram muito populares, em especial as implementações código aberto MySQL e PostgreSQL. Caracterizados por suas estruturas em formas de tabelas e colunas, esses bancos são amplamente utilizados, maduros e estáveis. Há muitos exemplos de implementações disponíveis, boa documentação, e ferramentas auxiliares para trabalhar com eles.

Apesar disso, existem características estruturais, especialmente ligadas ao modelo implícito da interface SQL, que são adversidades nas implementações desses bancos. Por esse motivo, surgiram diversos bancos alternativos sem essa interface de comunicação, ou utilizando ela apenas em parte, em um conceito chamado de NoSQL. Nesse novo grupo, a implementação MongoDB é o maior banco de dados de código aberto.

Não usar SQL não é apenas uma característica, mas também uma necessidade do MongoDB que o distingue dos bancos de dados relacionais. MongoDB trabalha com um conceito de banco de dados orientado a documentos (*document-oriented database*). Nesse tipo de organização os registros são armazenados em estruturas que se assemelham a documentos, sem que necessariamente esses documentos sejam armazenados como arquivos individuais.

O formato interno desses documentos está em um formato chamado de BSON (*Binary JSON*), que é um formato binário serializado de documentos no estilo JSON (*JavaScript Object Notation*). Estilo JSON (*JSON-style*) faz referência ao fato de o formato não ser formalmente um JSON, conforme a norma, mas baseado fortemente nela, adicionados alguns novos objetos (especialmente podemos citar *ObjectId*, que representa uma chave primária única, e *Date*, que representa data, hora e fuso horário). Esses registros, documentos flexíveis, são agrupados em coleções (*collection*), que seriam equivalentes às tabelas em bancos relacionais comuns.

4.1 Estrutura e formato dos campos

A principal característica que distingue as coleções das tabelas é a de não haver uma estrutura predefinida. Em outras palavras, não existe o conceito de colunas, geralmente bem rígidas, para serem definidas. Em MySQL é necessário definir a chave primária da tabela, e os formatos e o tamanho dos dados de cada campo que define uma coluna.

Figura 4 - Definição de colunas de uma tabela de clientes usando MySQL.

Nome	Formato	Características	Descrição
<code>id</code>	<code>INTEGER</code>	- Chave primária	Número único que identifica a linha no banco de dados e o cliente para o usuário.
<code>nome</code>	<code>VARCHAR(256)</code>	- Índice	Nome completo do cliente.
<code>tipo</code>	<code>VARCHAR(8)</code>	- valores: <code>'fisica'</code> ou <code>'juridica'</code>	Tipo legal do cliente.
<code>cgc</code>	<code>VARCHAR(32)</code>	- Exemplo: <code>'003.109.960-44'</code>	Número do documento do cliente.
<code>obs</code>	<code>TEXT</code>		Texto com observações diversas sobre o cliente.

Figura 5 - Comando SQL para a criação de uma tabela de clientes.

```
CREATE TABLE `stacas`.`clientes` (
  `id` INTEGER NOT NULL AUTO_INCREMENT PRIMARY KEY,
  `nome` VARCHAR( 256 ) NOT NULL,
  `tipo` VARCHAR( 8 ) NOT NULL,
  `cgc` VARCHAR( 16 ),
  `obs` TEXT,
  INDEX ( `nome` )
) ENGINE = INNODB;
```

Ao utilizarmos MongoDB, não há a etapa de definição dos campos no processo de desenvolvimento. Desse modo, os formatos são definidos diretamente na linguagem da aplicação do servidor, contanto que haja uma implementação de comunicação entre a linguagem e o MongoDB. No nosso sistema, a aplicação do servidor será desenvolvida em PHP, que já disponibiliza nativamente acesso a esse tipo de banco. MongoDB disponibiliza bibliotecas em todas as grandes linguagens, e geralmente essas bibliotecas já estão incluídas como nativas nas linguagens ou nos seus pacotes de bibliotecas oficiais. Não havendo implementação nativa da interface, ela pode ser feita utilizando a interface de texto padrão do MongoDB (chamada de *shell* ou *console*), que utiliza a linguagem JavaScript, ou ainda utilizando *wrapper* para as bibliotecas em C.

4.2 Chave primárias

Como não há estruturas predefinidas de tabelas em MongoDB, por padrão, cada documento inserido recebe uma chave primária em um campo chamado “_id” (o nome desse campo não pode ser alterado). Além disso, é criado um índice

automático para esse campo. O valor do campo é um tipo especial denominado de ObjectId. Um valor gerado automaticamente com 12 bytes garante a unicidade dentro da coleção.

Em SQL, não é obrigatório (mas é extremamente recomendado) que cada tabela tenha uma chave primária. A chave primária pode ter qualquer nome em SQL.

Figura 6 - Exemplo de documento em MongoDB.

```
{
  "_id" : ObjectId ( "51e866e48737f72b32ae4fbc" ),
  "clienteId" : 1234,
  "nome" : "Fulano da Silva",
  "tipo" : "fisica",
  "cgc" : "003.109.960-44",
  "renda" : 1234.56,
  "obs" : "Prefere falar depois das 20:00 hrs.",
  "updatedAt" : ISODate ( "2013-10-15T21:26:17Z" )
}
```

Como as chaves primárias não são sequenciais, e sim um número não representativo em termos humanos, não é possível usá-las para exibir ao usuário um número simples que identifique os clientes em nosso sistema. Por isso foi criado o campo *clienteld*.

É interessante observar que não há maneira nativa de criar um campo auto-incremental, o que é muito comum em bancos de dados SQL. A documentação oficial sugere uma solução usando uma coleção de contadores, uma função criada pelo usuário e uma operação especial de escrita.

4.3 Funções definidas pelo usuário

Como em SQL o usuário também pode criar funções próprias em MongoDB, é utilizada a sintaxe de JavaScript com objetos especiais definidos pelo banco de dados.

Figura 7 - Função definida pelo usuário para auto-incremento.

```

function getNextSequence ( name ) {
    var ret = db.counters.findAndModify ( {
        query: { _id: name },
        update: { $inc: { seq: 1 } },
        new: true
    }
    );
    return ret.seq;
}

```

Para criar uma função auto-incrementadora, é preciso realizar uma operação de leitura do valor atual, incrementar e escrever o novo valor no banco. Essas etapas causariam uma condição de corrida sobre o registro usado como índice, quando houvesse múltiplas conexões no banco. Para resolver essa situação MongoDB disponibiliza várias funções atômicas de leitura e escrita, como *findAndModify*, usadas no auto-incrementador.

4.4 Listas e documentos embutidos

Outra característica que também desperta interesse sobre MongoDB é a possibilidade de incluir listas e de criar documentos embutidos (embedded documents), que são chamados de objetos, e podem ser vistos como registros dentro de registros. Ambas não existem em bancos relacionais comuns ou são de difícil implementação neles.

As listas podem receber valores simples (números, booleanos, strings etc.) ou complexos (objetos e outras listas), que mantêm a ordem em que os objetos foram inseridos. Com isso, pode-se implementar diversas estruturas de dados diretamente no banco como filas e pilhas. Nota-se, contudo, que a ordem das propriedades dos documentos, ou das propriedades de qualquer objeto dentro de um documento, não é garantida, já que o formato BSON altera a ordem para otimização. Entretanto, pode-se facilmente utilizar uma lista auxiliar para armazenar a ordem das propriedades se isso for necessário, digamos, para se criar um dicionário ordenado (ordered dictionary).

Figura 8 - Exemplo de documento em MongoDB com listas e objetos.

```

{
  "_id" : ObjectId ( "51e866e48737f72b32ae4fbc" ),
  "nome" : "Fulano da Silva",
  // Lista
  "categorias" : [ "homem", "meia-idade", "detalhista", 45, null ],
  // Objeto (documento embutido)
  "identidade" : {
    "orgao" : "ssprs",
    "data_expedicao" : ISODate ( "1975-04-15T00:00:00Z" ),
    "numero" : "6575801934"
  },
  // Lista de Objetos
  "contatos" : [
    { "tipo" : "celular", "contato" : "51 9678-4455" },
    { "tipo" : "residencial", "contato" : "51 3278-0000" },
    { "tipo" : "email", "contato" : "fulano.silva@email.com" },
  ]
}

```

4.4.1 Implementando relações um-para-muitos (1..N)

Uma das aplicações úteis de lista de objetos é a implementação de relações do tipo um-para-muitos (*one-to-many relationship*), em que não seja necessário utilizar os dados relacionados sozinhos. Geralmente, essa situação ocorre com relacionamentos simples entre dados, com uma cardinalidade baixa. Nesse caso, os elementos secundários podem ser colocados dentro dos principais como objetos embutidos.

Figura 9 - Relação entre três tabelas SQL: clientes, contatos e endereços.

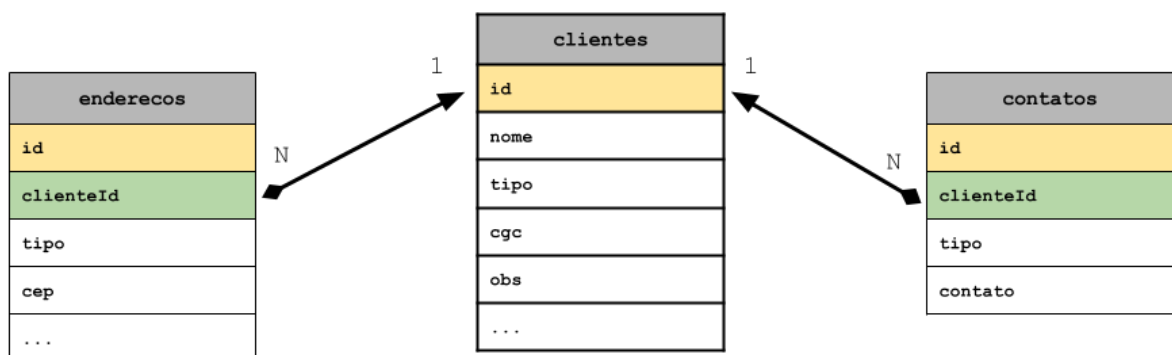


Figura 10 - Relação de clientes, contatos e endereços em uma coleção de MongoDB.

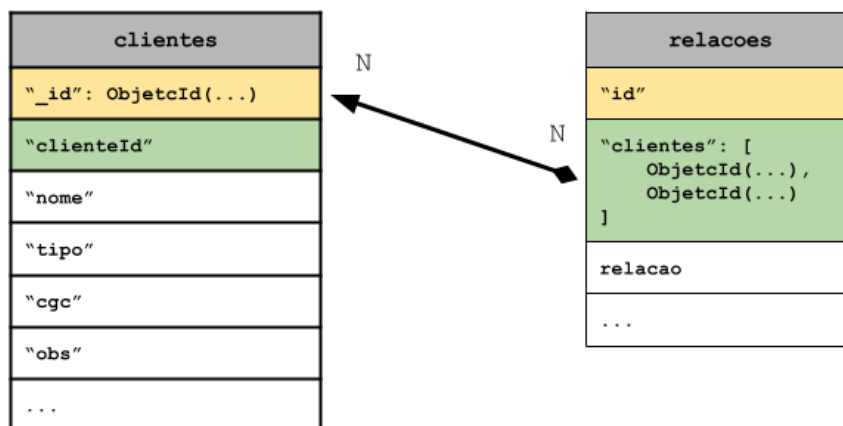
clientes
"_id": ObjectId()
"clienteId"
"nome"
"tipo"
"cgc"
"obs"
"contatos": [{ "tipo", "contato" }, ...]
"enderecos": [{ "tipo", "cep", "logradouro", "municipio", "uf", ... }, ...]
...

Importante ressaltar que MongoDB permite que buscas possam ser feitas por propriedades do documento principal ou dos embutidos.

4.4.2 Implementando relações muitos-para-muitos (M..N)

Para implementar relações de muitos-para-muitos (*many-to-many relationship*), por exemplos, amizades entre clientes, pode-se utilizar listas de índices em ambos os lados da relação, em ambos os clientes. Ou pode-se criar uma coleção auxiliar para armazenar essa relação, de uma forma muito parecida como se faria com uma tabela de um banco relacional comum.

Figura 11 - Relação muitos-para-muitos em MongoDB



4.5 Implementação de heranças

O conceito de armazenar os registros como documentos flexíveis, sem usar o conceito de tabelas estruturadas, permite a fácil implementação do conceito de herança da programação orientada a objetos. Essa é uma tarefa complexa encontrada em bancos de dados estruturados tradicionais.

No caso concreto dessa aplicação, existem vários ramos de seguros distintos (cerca de uma dezena) que compartilham apenas alguns dados em comum. Esse compartilhamento pode ser facilmente implementado como herança, em que cada seguro de um ramo específico herda características de um registro de seguro genérico.

Figura 12 - Exemplo de herança em seguros de ramos diferentes.

<pre>// Seguro de automóvel { "_id" : ObjectId ("... "), "seguroId" : 1233, "tipo_seguro" : "automovel", "placa" : "IJK 1234", "modelo" : "Ford KA", //... }</pre>	<pre>// Seguro de vida { "_id" : ObjectId ("... "), "seguroId" : 2137, "tipo_seguro" : "vida", "idade_inicial" : 35, "doenca_previa" : null, //... }</pre>
---	---

4.6 Índices de buscas

Apesar de ser possível realizar pesquisas em todos os campos sem nenhuma configuração prévia, como em SQL, índices de buscas ainda precisam ser definidos para se obter bom desempenho. Em alguns casos, como busca em campos de texto, especialmente em coleções grandes, índices são fundamentais.

Figura 13 - Código para criação de índice de busca em SQL e MongoDB.

```
// Índice na tabela "clientes" na coluna "nome".
CREATE INDEX `index_name` ON `clientes` ( `nome` );

// Índice na coleção "clientes" na propriedade "nome".
db.clientes.ensureIndex ( { nome : 1 } );
```

4.7 Restrições e chaves estrangeiras

Uma das limitações de MongoDB, se comparado a bancos relacionais, é o fato de não existir restrições (*constraints*) entre tabelas, também chamadas de chaves estrangeiras (*foreign key*). Essas restrições garantem a integridade dos registros apontados. Permitem também a atualização dos registros, que podem ser excluídos ou atualizados automaticamente pelo próprio banco, se desejado.

Com isso, todas essas verificações e atualizações precisam ser feitas pela aplicação-servidor no nosso sistema. Qualquer falha provocará uma inconsistência nos dados armazenados no banco.

Figura 14 - Comando SQL para definição de uma restrição entre a tabelas.

```
ALTER TABLE `contatos` ADD FOREIGN KEY ( `clienteId` )  
REFERENCES `stacas`.`clientes` ( `id` ) ON DELETE  
CASCADE ON UPDATE CASCADE ;
```

Apesar disso, existem índices de unicidade (*unique*) em MongoDB. Quando definidos sobre uma propriedade o banco rejeitará automaticamente, na inserção ou atualização, valores repetidos naquela propriedade entre os documentos da coleção.

5 CRIANDO INTERFACES COM USUÁRIO DE FORMA DECLARATIVA COM ANGULARJS

A linguagem HTML (*HyperText Markup Language*) surgiu em um ambiente muito diferente do atual. No começo dos anos 90, o HTML foi projetado para criação de páginas estáticas que exibiam conteúdo pela Internet com pouca formatação e imagens. Juntamente com esses arquivos, surge o protocolo HTTP (*Hypertext Transfer Protocol*) para sua transmissão. No final dos anos 90, surge o CSS (*Cascading Style Sheets*), que possibilitou melhor formatação gráfica. Ocorreram mudanças também no HTML e HTTP, que viabilizaram mais modos de exibição e novos modos de envios de dados do cliente para o servidor, como o Ajax (*Asynchronous JavaScript and XML*).

É nesse ambiente que nascem as primeiras aplicações web (*web-based application*), nas quais as páginas, exibidas pelos navegadores (*browsers*), passam a ser usadas como interface gráfica de sistemas cliente-servidor, ao invés de aplicações desenvolvidas especificamente para serem clientes. Durante muito tempo, essas aplicações web foram de qualidade muito inferior às executadas localmente. Precisou-se de mais de uma década até que as linguagens, bibliotecas e convenções de formatos - e por que não dizer o próprio hardware -, melhorassem ao ponto de as aplicações web serem capazes de substituir completamente as locais.

Com o passar do tempo, para atingir um desempenho ágil, o gargalo de fazer sistemas web passou a ser cada vez mais a conexão. Por mais rápida que seja a transferência de dados, transferir informação para atualização da interface gráfica a cada interação do usuário ainda hoje é ineficiente.

Por isso a ideia das aplicações evoluiu para a transferência não só de páginas estáticas, mas também para a transferência de scripts, normalmente em JavaScript, para atualizar as páginas dinamicamente sem a necessidade de se comunicar com o servidor. A biblioteca JQuery, em especial, implementada em JavaScript, tornou-se muito popular para executar essas modificações dinâmicas.

Figura 15 - Exemplo de código em JQuery modificando o HTML.

```
// HTML
<div>
  <button onclick="novoCliente();" type="button">Novo Cliente</button>
  <div id="myContainer"></div>
</div>

// JavaScript usando JQuery
function novoCliente()
{
  //Seleciona a <div> myContainer.
  var container = jQuery("#myContainer");
  //Insere dinamicamente um <form> na <div>.
  container.html("<form id='formCliente'> Formulário de Novo
Cliente </form>");
}
```

A estrutura em forma de árvore, na qual os elementos do HTML (isto é, os blocos que o compõem) são organizados chama-se DOM (*Document Object Model*). A manipulação da DOM usando JavaScript é uma técnica que permite muita liberdade de remodelagem da interface, mas aumenta a complexidade do código. O código da interface passa estar separado entre o HTML e os Scripts que o modificam.

Com a criação do Ajax e XMLHttpRequest, no final de 1999 e início dos anos 2000, os scripts passaram a poder fazer requisições diretamente ao servidor, tornando essa técnica ainda mais utilizada.

À medida que essas técnicas foram mais e mais utilizadas, o conceito se expandiu e criou-se a ideia de aplicação de página única (*Single Page Application*). Nesse tipo de aplicação existe apenas uma página, geralmente com apenas um endereço web acessível no servidor, e todas as alterações na interface são feitas pelos scripts manipulando a DOM. Como um bom exemplo disso temos aplicações como GMail e Google Maps.

O sistema atual da Stacas já utiliza em parte conceitos de páginas dinâmicas e aplicação em apenas uma página. A maior parte da interface é carregada na inicialização.

Um dos principais problemas dessa abordagem é seu impacto no código fonte. Como toda a modificação da interface é feita pelo script, e esse script está constantemente se comunicando com o servidor para obter dados (e, assim, processar a lógica da aplicação), o comum é que os códigos fiquem entrelaçados.

Por ser assim, não pode haver uma separação simples entre equipes de programadores da lógica e da interface, que normalmente são profissionais

diferentes. Além disso, com os arquivos-fontes misturados, no geral a manutenção e os testes se tornam muito complexos.

Figura 16 - JQuery se comunicando com o servidor e modificando o HTML.



Nesse contexto, surgem diversas bibliotecas e frameworks para endereçar essas dificuldades, dentre elas umas das mais notáveis é o AngularJS, um framework de código aberto desenvolvido pela Google e pela comunidade desde 2009.

Como é uma abordagem mais inovadora, a adoção do AngularJS ainda é pequena, pois há uma curva de aprendizagem para os desenvolvedores, e a utilização com códigos antigos costuma ser complicada. Apesar disso, os problemas abordados por ela a tornam bastante promissora.

5.1 Interfaces de usuário declarativas (*Declarative View*)

Dentre as principais características diferentes do AngularJS a ideia que mais se destaca é a separação entre a forma de desenvolver a interface e lógica. A ideia é que devemos usar programação declarativa para a construção de interfaces de usuário, enquanto a programação imperativa é melhor para implementar a lógica do

programa. Ou seja, contrasta diretamente com as técnicas tradicionais de manipulação da DOM por JavaScript, que normalmente utiliza JQuery, e especialmente faz isso junto com o resto da lógica.

Para fazer a programação da interface de forma declarativa, AngularJS adapta e estende os elementos do HTML tradicional para poder representar o conteúdo dinâmico das interfaces. Acaba, dessa maneira, quase que completamente com a necessidade de manipulação direta da DOM.

Figura 17 - Exemplo de código em AngularJS sem modificar o HTML.

```
<!doctype html>
<html ng-app="webseg">
<head>
  <script src="//path/to/angular.js"></script>
</head>
<body>
  <button ng-click="novoCliente = true">Novo Cliente</button>
  <form ng-show="novoCliente">
    <!-- Formulário de Novo Cliente -->
  </form>
</body>
</html>
```

A extensão do HTML é feita através da definição de novos atributos aos elementos de HTML (*elements* ou *tags*). A especificação do HTML5 já prevê atributos e elementos customizados, definidos pelo usuário. Logo, o documento de AngularJS continua um HTML5 válido. Esses atributos especiais são chamados de diretivas (*directives*). Eles funcionam como marcadores de blocos ou definem algum comportamento dinâmico daquele elemento que o contém.

Alguns dos atributos importantes que ilustram o conceito são:

ng-app: define a raiz, o elemento inicial de toda a aplicação. Serve como marcador. Todos os demais elementos precisam estar dentro deste para terem efeito. O valor do atributo será o nome da aplicação.

```
<html ng-app="webseg"> ... </html> //HTML
```

ng-bind: define que aquele elemento terá seu conteúdo definido pelo valor do atributo

```
$scope.nome = "Fulano da Silva"; //JavaScript
<div ng-bind="nome"></div> //HTML
```

Será exibido como:

```
<div>Fulano da Silva</div> //Resultado
```

ng-show e **ng-hide**: o valor do atributo é uma expressão booliana que define se aquele elemento irá ser exibido ou não (a propriedade display do CSS é utilizada).

```
$scope.exibir = true; //JavaScript
<div ng-show="exibir">Show!</div> //HTML
<div ng-hide="exibir">Hide!</div> //HTML
```

A div que contem "Show!" será exibida; a outra, não.

```
<div>Show!</div> //Resultado
```

ng-class: o valor do atributo pode ser uma variável que define a classe a ser aplicada naquele elemento, ou uma expressão booliana que determina se a classe deve ser aplicada ou não. Combina ng-class com ng-bind:

```
$scope.classe = "strike"; //JavaScript
$scope.nome = "Fulano da Silva"; //JavaScript
<div ng-class="classe" ng-bind="nome"></div> //HTML
```

Será exibido como:

```
<div>Fulano da Silva</div> //Resultado
```

5.2 Vinculação de dados no dois sentidos (*Two-way data binding*)

Outra característica importante do AngularJS é a vinculação de dados nos dois sentidos, especialmente para os elementos de formulários no HTML, que permitem entrada de dados do usuário: input, select, checkbox, radio, textarea etc. O atributo que define essa característica é o ng-model, e o valor do atributo é o nome da variável. Quando um elemento recebe esse atributo, em toda alteração, seja pela interface HTML com o usuário ou pela variável no JavaScript, o outro valor é atualizado automaticamente.

ng-model: similar ao ng-bind, mas nas duas direções. Se a variável for modificada, o valor no campo é modificado; se o campo for modificado, a variável é modificada automaticamente.

```
$scope.nome = "Fulano da Silva"
<input ng-model="nome" type="text">
```

Essa funcionalidade poupa bastante tempo de desenvolvimento em aplicações *web-based*, pois elas são fortemente baseadas no uso de formulários. Além disso, o

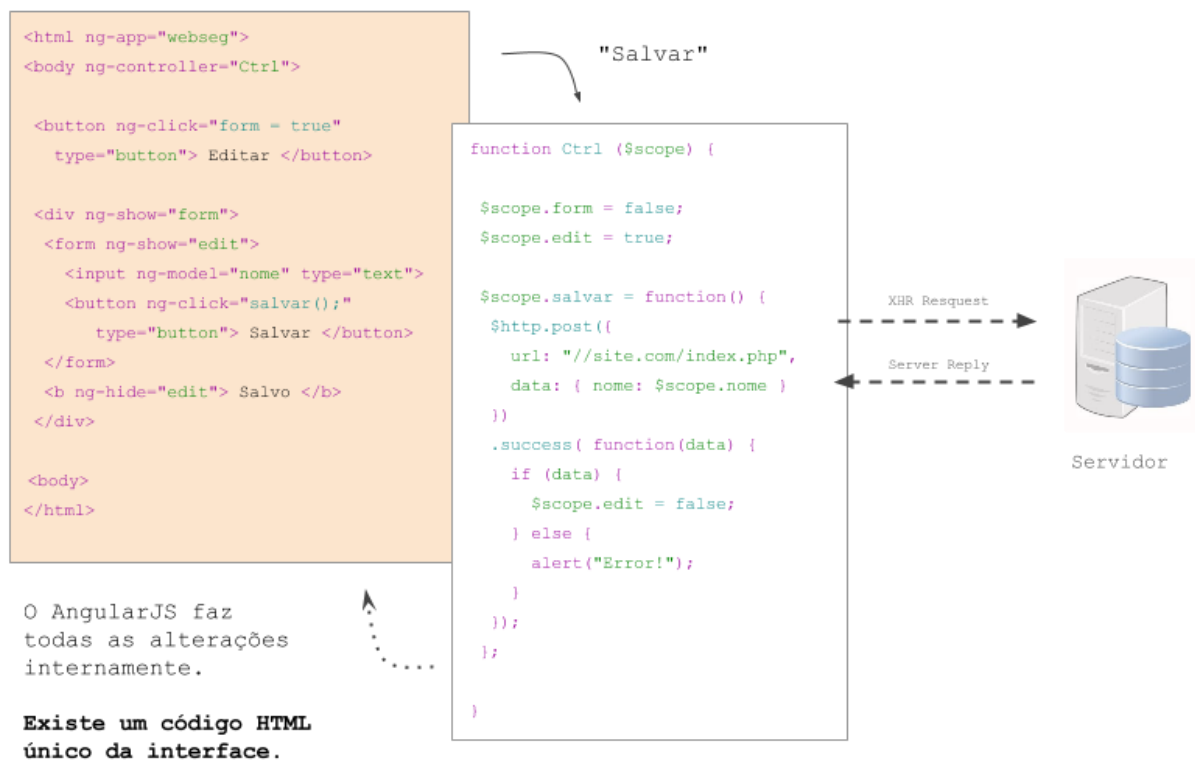
código fonte torna-se mais legível pois não é necessário códigos de leitura e escrita específicos para cada campo.

5.3 Arquitetura modelo-visão-controlador (MVC, *Model-View-Controller*)

AngularJS impõe o uso de uma arquitetura estilo MVC por construção. Os controladores são implementados nos códigos em JavaScript, separados das visões que são implementados em código HTML. Assim, utilizam os atributos especiais para lidar com o conteúdo dinâmico. O modelo fica bem simples, sendo definido pelas variáveis que são atualizadas pela vinculação nos dois sentidos.

O modelo pode ser visto se compararmos a mesma funcionalidade implementada com manipulação direta da DOM e com AngularJS.

Figura 18 - AngularJS sem modificar o HTML e se comunicando com o servidor.



Essa arquitetura imposta também é referida por alguns autores como MVVM (*model-view-viewmodel*). Nessa arquitetura, é dito que a lógica está no *model* (formalmente implementado por um objeto do tipo *controller* no AngularJS), e a interface continua na *view* como na arquitetura genérica. A diferença está nessa nova definição chamada de *viewmodel*, que são as entidades compartilhadas no *model* e na *view*, o que representa muito bem as variáveis com a vinculação dupla que temos em AngularJS.

5.3.1 Controladores (*Controllers*) e Escopo (*Scope*)

Os controladores são os elementos básicos de lógica. Cada controlador define um escopo, no qual são declaradas variáveis e funções que podem ser acessadas pela *view*. Os controladores podem ser reusados e aninhados conforme necessidade, criando uma hierarquia de escopos. Desta forma, AngularJS permite criar blocos reutilizáveis.

Figura 19 - Declaração de controlador e hierarquia de escopo em AngularJS.

```
appWebSeg.controller('clienteController', function($scope) {
    //Declarações e lógica.
});
```

```
<!DOCTYPE html>
<html ng-app class="ng-scope" $scope
  <head>...</head>
  <body>
    <div>
      <div ng-controller="GreetCtrl" class="ng-scope ng-binding" $scope
        Hello World!
        name='Wold'
      </div>
      <div ng-controller="ListCtrl" class="ng-scope" $scope
        <ol>
          <!-- ngRepeat: name in names -->
          <li ng-repeat="name in names" class="ng-scope ng-binding" $scope
            Igor
            name='Igor'
          </li>
          <li ng-repeat="name in names" class="ng-scope ng-binding" $scope
            Misko
            name='Misko'
          </li>
          <li ng-repeat="name in names" class="ng-scope ng-binding" $scope
            Vojta
            name='Vojta'
          </li>
        </ol>
      </div>
    </div>
  </body>
</html>
```

Fonte: <https://docs.angularjs.org/guide/scope>

5.4 Templates

Diversas vezes é necessário iteirar sobre uma lista ou uma coleção de itens para exibir uma relação com um determinado formato, sendo que esses dados são obtidos dinamicamente. Para esses casos, AngularJS disponibiliza *templates*.

Com o atributo `ng-repeat` pode-se definir uma apresentação genérica, um trecho de HTML e CSS, que será aplicado para cada item durante a execução. Notemos que é possível combinar essa diretiva com outras para obter uma apresentação complexa.

Figura 20 - Exemplo de código de *template* em AngularJS.

```
<ul ng-controller="ctrl">
  <li ng-repeat="cliente in clientes">
    <span ng-bind="cliente.nome"></span>
    <div ng-show="cliente.hasPhoto">
      
    </div>
    <div ng-hide="cliente.hasPhoto">Imagem não disponível.</div>
  </li>
</ul>
```

5.5 Diretivas customizáveis

Outra característica que garante grande flexibilidade ao AngularJS é a possibilidade dos desenvolvedores criarem novas diretivas customizadas para a aplicação. Com elas, pode-se definir um novo elemento ou atributo de HTML com um significado próprio, garantindo assim mais legibilidade ainda ao código fonte da view, mesmo com elementos dinâmicos.

Figura 21 - Exemplo de diretiva customizável.

```
// JavaScript
angular.module('components', [])
.directive('link', function() { //Nome: link
  return {
    restrict: 'E', //Tipo: Element
    template: '<div class="link">'
      + '<a ng-href="//path/to/cliente/{{clienteId}}">'
      + ''
      + '</a>' + '</div>'
  };
});
// HTML. Uso da Diretiva criada.
<link clienteId="123">
```

Note que na criação de diretivas próprias volta-se a manipular a DOM diretamente. Por isso, deve-se usar esse tipo de recurso o mínimo necessário para não recair no problema original que queremos evitar. Esse é o tipo de manipulação que o AngularJS faz internamente com as suas diretivas nativas.

5.6 Dependency Injection

O AngularJS dispõe de diversos recursos, chamados de módulos ou serviços, para lidar com situações específicas do desenvolvimento de aplicações *web-based*. Para o uso dessas funcionalidades, AngularJS possui um sistema de Dependency Injection (injeção de dependência, em tradução livre), que carrega e inicializa automaticamente esses serviços quando são chamados. Cada serviço é associado a uma variável com um nome especial (`$scope`, `$routeParams`, `$http` etc). Quando essas variáveis são usadas, o serviço é inicializado.

Usando esse mesmo sistema, podemos criar nossos próprios serviços. Esses serviços criados pelo desenvolvedor podem ser registrados e serão inicializados automaticamente como os nativos.

Figura 22 - Criação de serviço e uso com *dependency injection*.

```
app.factory('$notificacao', function($window) {
    return {
        alerta: function(text) { $window.alert(text); },
        erro: function(text) { console.error(text); }
    };
});

app.controller('clienteCtrl',
    function($scope, $routeParams, $http, $notificacao) {
        //Os serviços são carregadas on-the-fly.
        $notificacao.alerta("Alerta!");
    });
});
```

6 REALIZANDO BUSCA UNIFICADA COM SOLR

Em um sistema repleto de informações espalhadas em diversas colunas e tabelas, é desejável um sistema unificado de busca para que o usuário possa encontrar qualquer informação a partir de qualquer fragmento disponível. Esse sistema de busca deve ser rápido e simples de usar, e deve retornar ao usuário as respostas de forma a destacar as informações mais relevantes.

Além de simplificar o uso do sistema, esse tipo de busca está se tornando cada vez mais uma expectativa do mercado, isto é, uma demanda dos usuários. Além dos sistemas de buscas online - notavelmente temos como exemplo o Google -, os sistemas operacionais, como Windows 7+ e Ubuntu 12.04+, também passaram a disponibilizar esses mecanismos. Neles, pode-se encontrar qualquer programa, nome de arquivo ou, em alguns casos, até conteúdo de arquivos a partir do mesmo campo de busca.

Apesar disso, por exemplo, as grandes seguradoras ainda utilizam sistemas de buscas limitados e simples, nos quais o usuário precisa escolher o campo exato que deseja buscar, e muitas vezes, ainda, é necessário especificar o formato exato que o sistema requer para aquele campo. Como exemplo de limitação, diversos sistemas não conseguem localizar o cliente “João da Silva”, se for realizada uma busca apenas por “joão silva”.

Figura 23 - Sistemas da Liberty e Porto Seguro não possuem busca unificada.

Liberty Seguros

Apólices

Clique no número da apólice para ver detalhes.

Filial: Moinhos de Vento

Corretor: 99015584 / 1

Critério de Busca: Seleccione

Nome do Segurado

Apólice

Contrato

Endosso

CPF/CNPJ

Placa do Veículo

Chassi do Veículo

Pendentes de Emissão

Emitidas no Período

Canceladas por Período

Preencha a informação a ser pesquisada:

Consultar

PORTO SEGURO **Itaú** **Azul SEGUROS**

seguros **auto residência**

Tipo de Pesquisa

Lista de Suseps: ▼

Produtos **Seguros** **Meus Negócios**

Serviços

- ▶ Administração do Corretor
- ▶ Agendamentos
- ▶ Atendimento Online
- ▶ Cálculos, Propostas e Emissões
- ▶ Campanhas e Marketing
- ▶ Cobrança
- ▶ Comissões

Nome:

Data de Protocolo: até Intervalo de até uma semana

Proposta:

Proposta Porto Print: ▼

O sistema atual da Stacas também não possui esse opção de busca unificada. Na verdade, a limitação é ainda maior. Diversos campos sequer podem ser pesquisados.

Apesar disso, a busca pelo nome do cliente, a mais importante do sistema, é bem completa. Ela pode encontrar qualquer fragmento do nome, em qualquer ordem. Por exemplo, uma busca por “gabri gra” encontrará o cliente “**Gabriel Casagrande Stabel**” (a parte grifada marca a correlação encontrada).

Figura 24 - Busca atual do sistema da Stacas.

STACAS CORRETORA DE SEGUROS

Usuário: [\[Sair\]](#)
Gabriel C. Stabel
 Acesso: 17:06 - 22/06
[\[Clientes Pesquisa\]](#)
 Id:

Pesquisa Seguros

Nome: Nº:

Placa: Modelo: Apólice:

	8505 Gabriel Casagrande Stabel	20/Ago/13	Liberty	Renovação Res. Habitual	Sim
	8456 Gabriel Cruz Grando	09/Jul/13	Generali	Renovação Automóvel	Sim
	8346 Gabriel Casagrande Stabel	09/Abr/13	Icatu	Novo Previdência	Sim

6.1 Busca unificada em SQL.

A ideia de criar um busca unificada é muito interessante, mas a sua implementação pode ser desafiadora. Os bancos de dados, tanto os relacionais com interface SQL como os sem esquema como MongoDB, se baseiam em campos bem definidos (colunas ou propriedades) e agrupados (em tabelas ou coleções). Por essa razão, seus sistemas de busca nativos se baseiam fortemente em definir onde será realizada a busca, na definição de quais colunas e tabelas para que sejam realizadas. Isso é claramente um entrave quando se deseja buscar em todos os dados ao mesmo tempo.

Se desejarmos realizar uma busca diretamente usando o sistema nativo de SQL, devemos escrever a query explicitamente definindo a busca de todas as colunas em todas as tabelas. Essa seria a abordagem de “força-bruta”. Teríamos de ter cuidado ao definir uma função de conversão para todos os campos cujo formato seja diferente do que temos originalmente, que é normalmente *string*. Além disso, após a busca realizada, seria necessário um pós-processamento pela aplicação-servidor

para que os dados fossem agrupados e ordenados por algum critério de prioridade, pois isso não estaria contemplado na query SQL.

Essa abordagem é inviável em termos práticos em um banco de dados de tamanho razoável. A query seria muito grande para cobrir todas as tabelas e colunas. Nesse cenário, seu processamento seria mais lento e sua manutenção de código complicada. Além disso, para termos uma busca rápida, deveríamos criar índices para todas as colunas. Isso aumentaria o tamanho do banco e deixaria as atualizações mais lentas, para manter os índices atualizados. Dependendo da quantidade e tipo de colunas, e dos critérios de ordenação necessários, o pós-processamento dos resultados seria ainda mais complexo que a busca.

Figura 25 - Comando de busca em duas colunas, em SQL e MongoDB.

<pre>// SQL SELECT * FROM clientes WHERE nome LIKE "%gab%" OR descricao LIKE "%gab%"</pre>	<pre>// MongoDB db.clientes.find({ nome: /. *gab.*/, descricao : /. *gab.*/ });</pre>
--	--

Pode-se imaginar diversas abordagens mais elegantes ainda usando SQL diretamente ou com um pouco de auxílio da aplicação servidor. Assim, haveria garantia de melhor inteligibilidade do código e de sua boa manutenção.

6.1.1 Buscas com índices FULL TEXT em MySQL

Desde sua versão 5.6, lançada no começo de 2013, o banco de dados MySQL disponibiliza um nova opção de índice chamado de *full-text*, e uma cláusula de busca específica para utilizá-lo. Com ele, pode-se indexar automaticamente todos os campos de *string* (*char*, *varchar*, e *text*) de uma dada tabela. Esse índice não funciona para nenhum outro formato (numéricos, booliano, enums etc) e está disponível apenas para tabelas com o formato MyISAM, que não permitem *constraints*. Como exemplo, o banco de dados atual da Stacas usa formato InnoDB.

Mesmo com esse índice, ainda é necessário especificar cada tabela e cada coluna para realizar uma busca. Seu uso como critério de pesquisa é feito com a cláusula *match-against*.

Por essas limitações, oriundas em parte de uma implementação ainda nova, essa alternativa ainda não serve como uma solução completa, mas provavelmente se tornará uma opção mais interessante no futuro. No banco de dados onde o formato do banco e das colunas permite utilizá-la, essa é uma alternativa um pouco melhor do que a força-bruta.

Figura 26 - Busca em SQL usando *match-against*.

```
SELECT * FROM clientes WHERE MATCH (nome, descricao)
AGAINST ("%gabri%", "%gra%");
```

6.1.2 Relevância dos campos e múltiplas buscas

Geralmente os campos têm prioridades diferentes no critério de exibição e ordem dos resultados encontrados, relativas à sua relevância ao usuário. No sistema da Corretora, uma correspondência (match) no nome do cliente é prioritária ao campo observações que, por sua vez, é prioritária ao campo de relacionamento entre clientes. Isso é muito importante, pois se esse critério não for respeitado a busca unificada pode se tornar menos útil e eficiente do que uma busca em que o usuário necessita especificar qual campo ele deseja procurar.

Levando em conta a relevância, pode-se executar múltiplas buscas começando nas tabelas e colunas de maior prioridade, e depois ir para as de menor prioridade, sendo que a busca pode ser interrompida assim que o limite de resultados já tenha sido atingido por um critério prioritário. Ter de realizar múltiplas buscas menores melhora o desempenho quando os resultados estiverem nos campos prioritários, mas o degrada no caso contrário.

Para contemplar o pior dos casos, todas as tabelas e as colunas ainda precisam ser buscadas de forma exaustiva. Por isso a implementação dessa etapa se torna um pouco mais complexa e demorada do que a força-bruta, mas apesar disso essa abordagem elimina quase que totalmente a etapa de pós-processamento, pois os dados já são retornados ordenados como se deseja.

6.1.3 Lógica para identificar campos

Uma alternativa que pode ser empregada para simplificar as buscas é a criação, na aplicação-servidor, de uma lógica anterior a busca, uma espécie de filtro, que permita avaliar o que se deseja buscar e, com isso, restringir a amplitude da busca ao invés de fazer diretamente uma busca exaustiva. Essa técnica não pode ser utilizada sozinha, pois não há como identificar qualquer tipo de informação que o usuário busca apenas com essa análise, mas pode ser combinada com as anteriores para reduzir a busca ou a necessidade de conversão entre os formatos.

Essa abordagem pode ser muito útil especialmente para tratar de buscas que contenham dados numéricos, sejam eles códigos (cpf, telefones etc.) ou valores (comissão, prêmio, franquias etc.), pois esses tipos de dados normalmente têm um padrão que pode ser identificado: uma quantidade de algarismos numéricos, símbolos especiais (como o cifrão), sufixos ou prefixos etc.

6.1.4 Campos específicos para buscas

Outra alternativa é criar estruturas auxiliares no banco de dados para permitir essas buscas completas. Ou seja, não buscar diretamente nos dados reais, mas preparar os dados para poder ser realizada uma busca completa e eficiente em segundo momento.

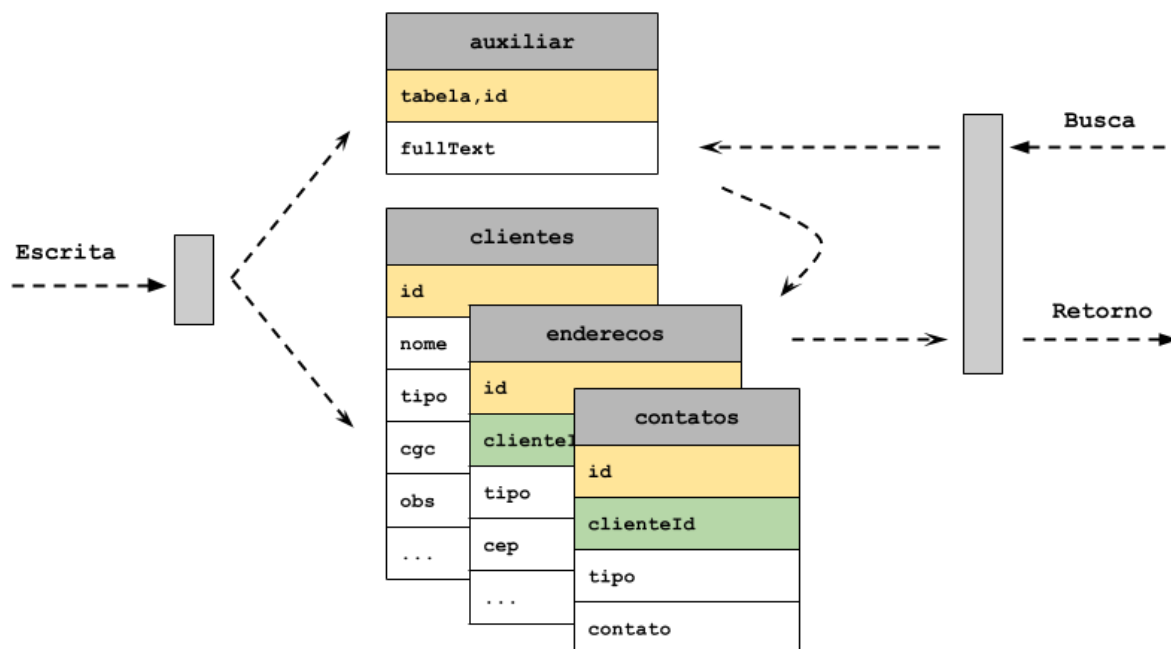
Inicialmente, cria-se uma tabela auxiliar exclusiva para buscas. Essa tabela deve conter pelo menos uma coluna de texto, e outra coluna para armazenar uma

referência para outra tabela e linha qualquer. Cria-se também índices de busca sob essa coluna de texto.

Com o auxílio da aplicação-servidor, ou usando funções criadas no próprio banco SQL, cria-se uma lógica em que toda a operação de escrita sob os dados reais realize uma cópia especial na tabela auxiliar. Essa cópia é uma versão de todos os dados originais juntos, em puro texto, que será armazenada em apenas uma coluna auxiliar. É importante garantir que em cada operação sob os dados reais, inserção, atualização e remoção, os dados das estruturas auxiliares permaneçam atualizados, ou seja, sempre em sincronia.

A busca unificada passa, então, a ser realizada apenas sob uma coluna da tabela auxiliar, que por sua vez mantém uma referência para os dados reais. Observa-se que ainda é necessário, porém, algum tipo de pós-processamento para determinar a relevância de cada correspondência encontrada, para assim ordenar os dados da busca antes de retornar o resultado da pesquisa à aplicação-cliente.

Figura 27 - Diagrama de busca unificada em SQL com tabela auxiliar.



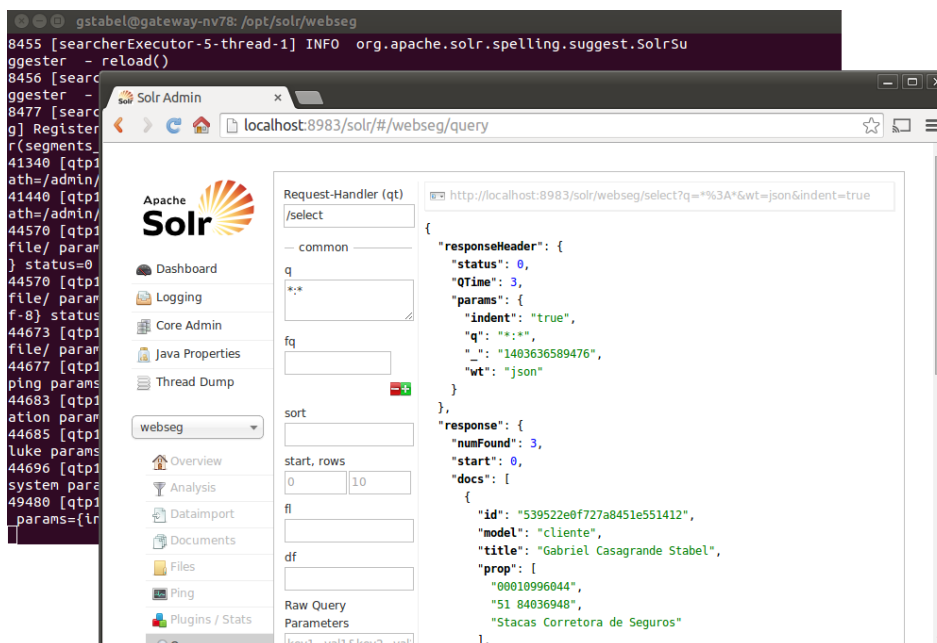
6.2 Busca unificada com SOLR

Solr é uma aplicação servidor, ou seja, roda como uma aplicação independente, de código aberto, escrita em Java, que utiliza como sua base a biblioteca de indexação Apache Lucene. Seu principal objetivo é implementar um serviço independente, ou seja, separado da aplicação, de busca livre em texto, ou busca completa em texto (*full-text search*).

A comunicação entre o Solr e a aplicação servidor em PHP ocorre através do uso local de sockets. É utilizado o protocolo HTTP com método GET para a leitura, isto é, buscas, e POST para escrita. As respostas, assim como a escrita, são realizadas

com arquivos que podem ter diversos formatos estruturados, como XML e JSON. Esses arquivos apresentam uma estrutura com parâmetros e valores definidos pelo Solr para representar suas operações.

Figura 28 - Servidor e interface administrativa do Solr sendo executada.



Além de ser uma aplicação independente, seus dados também são armazenados de forma independente, ou seja, sem conexão direta ao banco de dados real do sistema. Para isso, Solr define um modelo próprio de inserção de dados. Cada registro que se deseja tornar buscável é inserido como um documento com diversos campos e um identificador próprio. Esse identificador deve ser único, e sua unicidade precisa ser garantida pela aplicação-servidor. Quando uma escrita é feita como um identificador já existente, Solr interpreta essa ação como uma atualização do registro antigo. Por sinal, essa é a forma padrão de atualização dos dados.

Coincidentemente, vimos que o formato de documentos usado pelo Solr é muito semelhante à organização dos documentos flexíveis (equivalente às linhas das tabelas) do MongoDB que foi apresentado antes. Solr disponibiliza diversos métodos de filtragem de dados para criar seus índices de busca, diversas opções de armazenamento, e, especialmente, diversas formas e opções para executar as pesquisas nesses dados. Além disso, disponibiliza mecanismos para que tudo isso possa ser customizado especificamente para cada aplicação.

6.2.1 Configuração dos dados

Antes que os dados possam ser inseridos e catalogados no Solr, é necessário que seja especificada a forma como os dados vão ser apresentados. Em outras palavras, é necessário que seja definida uma estrutura, um esquema (*schema*) para os documentos catalogados, com os dados de cada campo que será utilizado. Essa definição se dá através de um arquivo de configuração chamado *schema.xml* com

elementos e atributos específicos, que definem cada campo possível nos documentos, conforme vemos abaixo:

name: nome único que identifica o campo. O valor do campo será sempre o id do registro no banco de dados.

type: o tipo de dado que se espera nesse campo (*string*, *text_pt*, etc). Essa informação também define que tipo de filtro deve ser aplicado naquele campo antes de ser catalogado.

multiValued: indica se pode haver mais de um valor, do mesmo tipo, naquele campo.

indexed: indica se o valor daquele campo deve ser catalogado ou não, se deve constar da busca. No nosso caso, a informação de a qual *model* pertence aquele documento não é catalogada por ser apenas uma referência ao banco de dados original.

stored: indica se o valor original puro do campo, sem filtros, deve ser armazenado e retornado na resposta à pesquisa. Como nós iremos consultar os valores originais no banco real, após a pesquisa, só desejamos armazenar as referências a esse dado, isso é, *model* e id.

required: indica se o campo é obrigatório no documento.

Note-se que não é necessário definir o nome de diversos campos que tenham configuração igual. Pode-se utilizar *wildcards* para definir uma regra de formação para um nome do campo. Por exemplo, todos os campos cujo nome termina em *_txt* pode ser representado pela regra **_txt*.

Figura 29 - Definição de esquema e documentos correspondentes.

```
//Definição dos campos do documento
<field name="id" type="string" indexed="true" stored="true"
required="true" multiValued="false" />
<field name="model" type="string" indexed="false" stored="true"
required="true" multiValued="false"/>
<field name="title" type="text_pt" indexed="true" stored="false"
required="true" multiValued="false"/>
<field name="prop" type="text_pt" indexed="true" stored="false"
multiValued="true"/>
<field name="*_txt" type="text_pt" indexed="true" stored="false"
multiValued="true"/>
```



```
//Exemplos de documentos conforme definição.
{id: "51e866e48737f72b32ae4fbc", model: "clientes",
 title: "Gabriel Casagrande Stabel" },

{id: "69f856e48738e72b32ae485a", model: "clientes",
 title: "Stacas Corretora de Seguros",
 prop: ["51 91055000", "51 32685523", "071134520000149"],
 observacao_txt: ["Fica perto da esquina."],
 desc_txt: ["O é Gabriel sócio"]}
```

Outra opção disponível para manipular os campos é a regra *copytext*. Com ela pode-se definir que determinados campos - ou padrões com nomes de campos usando *wildcards* - devem ser copiados para outro campo a cada recebimento ou atualização de documentos. Dessa maneira, pode-se agrupar campos diferentes em um só ou executar mais de um tratamento em um mesmo campo, copiando para outro campo com tipo diferente, tudo isso sem ter a necessidade de mexer na aplicação- servidor.

6.2.2 Índices de busca

A criação e atualização dos índices é um processo que ocorre a cada vez que um documento é inserido, atualizado ou removido. Usando a definição no esquema, o Solr sabe quais campos do documento devem ser indexados, isto é, catalogados para pesquisa. Baseados no tipo de dado de cada campo são aplicados filtros, que modificam os dados originais. Em seguida, esses dados são quebrados em *tokens* que são armazenados em uma estrutura interna de busca eficiente.

Foram utilizados quatro tipos de filtros, disponíveis nativamente, em nossos campos-padrão de texto:

ASCIIFoldingFilterFactory: converte todos os caracteres com acentos, com um ou mais bytes, para os 127 caracteres básicos de ASCII.

LowerCaseFilterFactory: converte os caracteres maiúsculos para minúsculos.

StopFilterFactory: desconsidera, descartando-as, diversas palavras simples do português a partir de uma relação predefinida. Essas palavras são artigos e preposições que não acrescentam muito significado às buscas, e podem causar falsas correspondências.

ReversedWildcardFilterFactory: normalmente as árvores de pesquisa são construídas utilizando os primeiros caracteres de cada *token*. Esse fato impossibilita a busca de *substrings* sem que se saiba os primeiros caracteres. Para tanto, esse filtro cria um novo *token* com uma ordem invertida dos caracteres. O seu nome,

ReversedWildcard, faz referência a esse filtro possibilitar o uso de *wildcards* na frente das buscas.

Além desses filtros, é utilizado o *StandardTokenizerFactory* para criar os *tokens*, que também aplica algumas regras para remover sequências de símbolos que não são letras ou que não podem representar palavras.

6.2.3 Pesquisas com eDisMax

Existem diversos módulos de pesquisas em Solr, cada um com seus diversos parâmetros de configuração. Há também a possibilidade de criar novas lógicas de buscas para casos mais avançados. Essas pesquisas são realizadas através de requisições HTTP, em que se pode escolher o tipo da pesquisa e passar os parâmetros desejados.

A pesquisa padrão permite decidir quais campos serão pesquisados, e quais critérios serão utilizados em cada campo. Permite também o uso de operadores lógicos entres esses critérios, (como AND e OR). Não há, porém, opções de busca reversa, sem usar as iniciais do *token*, nem opção de pesos entre os campos para ordenar os resultados.

Para atender às necessidades de nosso sistema, foi usado o módulo de busca eDisMax (*The Extended DisMax Query Parser*).

Figura 30 - Exemplo de pesquisa utilizando Solr.

```
//Requisição:
Critério: "gab"
Tipo de busca: edismax
Formato da resposta: json
Pesos campos: title = 3, prop = 2 e text = 1

URL correspondente (HTTP request usando GET method):

http://localhost:8983/solr/webseg/select?q=%2B*gab*&defType=edism
ax&wt=json&qf=title%5E3+prop%5E2+text%5E1&tie=0.1
```

```

//Resposta:
{
  "responseHeader":{
    "status":0, "QTime":28,
    "params":{
      "q":"+*gab*", "qf":"title^3 prop^2 text^1", "tie": "0.1",
      "wt":"json", "defType":"edismax"}},
  "response":{
    "numFound":1,"start":0,"docs":[{"
      "id":"539522e0f727a8451e551412",
      "model":"cliente",
      "_version_":1470400391489257472}
    ]
  }
}

```

Com essa busca, aliada ao filtro *ReversedWildcard*, pode-se usar os *wildcards* para localizar qualquer *substring* em quaisquer campos. Por exemplo, com a busca `"*bri*"`, encontra-se uma correspondência em "Gabriel", em "brigadeiro", e em "calibri".

Usando o operador "+" na frente de cada *substring* garante-se que, se houver mais de uma *substring* na query, todas deverão existir no documento para haver uma correspondência. Isso é o equivalente a usar o operador "AND" entre todos os campos.

Para definir o peso de relevância entre cada campo, que determinará a ordem dos resultados, utilizamos o parâmetro *Query Fields* (qf). Com ele, definimos quais campos serão utilizados na busca. No nosso caso, sempre usaremos todos, pois queremos fazer uma busca completa, e saber qual o valor de peso é atribuído se uma correspondência for encontrada em determinado campo. Definimos também um valor para o parâmetro *Tie breaker* (tie) que atribui um peso extra para valores repetidos, permitindo melhor o desempate entre eles.

7 EXEMPLOS DE USO DAS TECNOLOGIAS NO SISTEMA

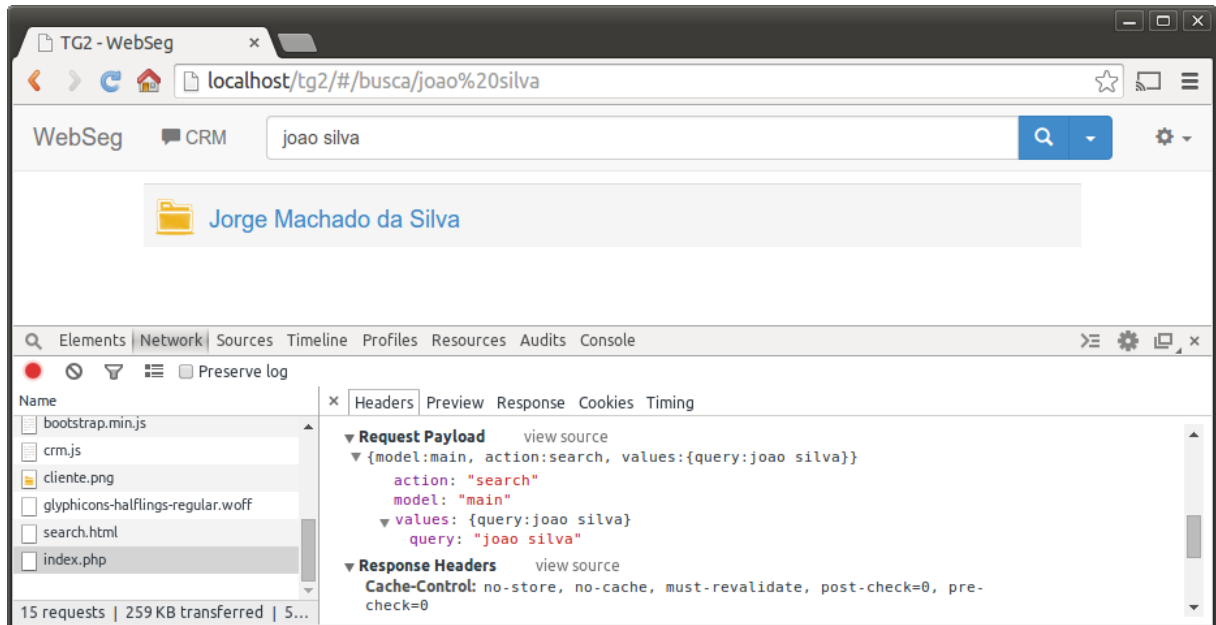
Depois de apresentada a arquitetura geral e as tecnologias específicas utilizadas, vamos apresentar alguns exemplos típicos no nosso sistema do uso dessas tecnologias. Essas implementações que serão apresentadas foram feitas no sistema protótipo que foi desenvolvido, e que será a base para o sistema final.

7.1 Localizando um cliente com a busca unificada

Uma cenário comum é ter de localizar um cliente a partir de informações indiretas. Vamos supor uma situação em que um cliente chamado “João da Silva” entre em contato pois não lembra se pagou ou não seu seguro de automóvel. O primeiro passo do atendente da Corretora, nessa situação, é localizar qual a apólice correspondente ao veículo segurado desse cliente. Imaginemos também que o personagem João não lembre, mas seu seguro não esteja em seu nome, e sim no nome de seu pai “Jorge Machado da Silva”, que é o proprietário legal do veículo.

O atendente acessa o sistema, se autentica, e digita “joao silva” (sem acento) na busca unificada. A aplicação-cliente em AngularJS envia uma requisição para a aplicação-servidor realizar essa busca no formato JSON, definido pela interface de comunicação. Nesse caso, o sistema identifica que o nome pesquisado, “joao silva”, está relacionado ao cliente segurado na apólice “Jorge Machado da Silva”, seu pai:

Figura 31 - Solicitação de busca unificada de “joao silva”, e debug no Chrome.



A aplicação-servidor recebe a solicitação, valida a sessão com o cookie de sessão enviado, e interpreta os dados para descobrir qual a operação desejada. A requisição é repassada para o *controller* responsável pelo *model* indicado, que por sua vez executa a *action* correspondente à busca unificada.

Essa pesquisa será realizada pelo Solr, uma aplicação independente que usa interface HTTP como comunicação. Por isso, a primeira ação será criar essa solicitação HTTP correspondente. A aplicação-servidor, que foi desenvolvida em PHP, utiliza uma classe auxiliar chamada Solr para todas as operações entre eles.

Figura 32 - Requisição HTTP para o servidor Solr de “joao silva”.

```
http://localhost:8983/solr/webseg/select?wt=json&defType=edismax&qf=title%5E3+prop%5E2+text%5E1&rows=50&tie=0.1&q=%2Ajoao%2A%20%2Asilva%2A
```

```
Critério de busca (q): *joao* *silva*
Tipo de busca (defType): edismax
Formato da resposta (wt): json
Pesos campos (qf): title = 3, prop = 2 e text = 1
Limite (rows): 50
Desempate (tie): 0.1
```

Supondo que o nome do filho conste apenas do campo de observação no cadastro de seu pai, da seguinte forma: “Quem dirige esse carro é o filho João da Silva”. Com os filtros que definimos, o Solr removeu o acento de “João” e removeu a preposição “da” ao cadastrar o campo observação nos seus índices de busca. Portanto, ele encontrará o registro do pai com a informação do filho.

A aplicação Solr responde, com um documento JSON, à aplicação-servidor indicando o registro que corresponde ao critério de busca. A aplicação-servidor consulta o banco de dados MongoDB para retornar o registro do pai à aplicação-cliente.

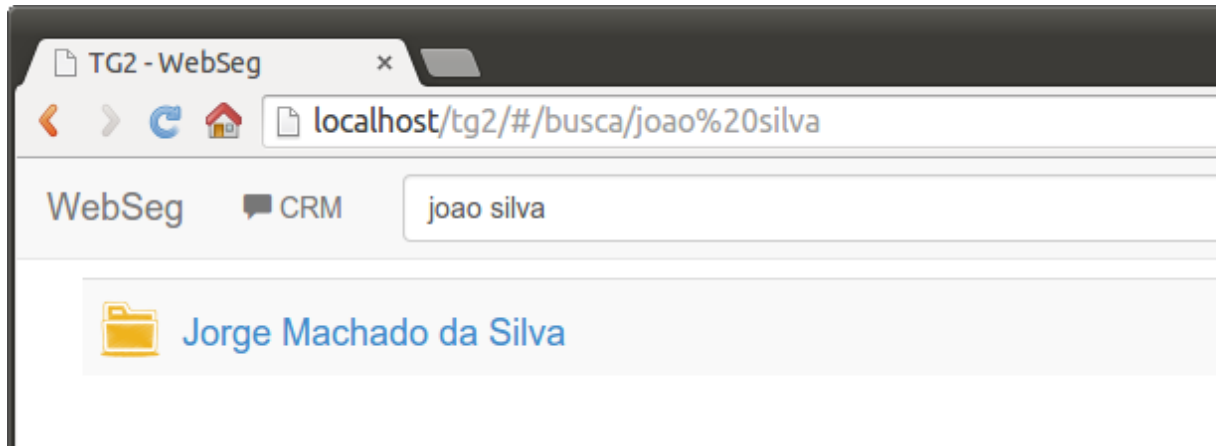
Figura 33 - Resposta do Solr à aplicação-servidor e consulta ao MongoDB.

```
{
  "responseHeader":{
    "status":0,
    "QTime":28,
    "params":{
      //Parâmetros da consulta apresentado na Figura 32
    },
  },
  "response":{"numFound":1,"start":0,"docs":[
    {
      "id":"53b4dd9df727a800128b4567",
      "model":"cliente",
      "title":"Jorge Machado da Silva",
      "prop":[],
      "text":["Quem dirige o carro é o filho João da Silva"],
      "_version_":1472580774070845440}
  ]}

// Consulta ao servidor MongoDB.
db.cliente.findOne(
  {"_id": "ObjectId("53b4dd9df727a800128b4567")" } );

// Documento retornado pelo servidor MongoDB.
{
  "_id" : ObjectId("53b4dd9df727a800128b4567"),
  "clienteId" : 2,
  "nome" : "Jorge Machado da Silva",
  "obs" : "Quem dirige o carro é o filho João da Silva.",
  "relacoes" : [],
  "tipo" : "fisica",
  "updatedAt" : ISODate("2014-07-03T04:41:03.000Z"),
  "updatedBy" : ObjectId("538e316083daee74f9e73715")
}
```

Figura 34 - Resultado da busca na aplicação-cliente.



7.2 Criando um cliente com AngularJS e salvando no MongoDB

Com base no exemplo anterior, o atendente da Corretora deseja agora cadastrar o cliente “João da Silva” e registrar sua filiação no sistema, para facilidade no futuro, conforme definido no procedimento de atendimento da Corretora. O cadastro será feito em formulário de cadastro de novo cliente, conforme demonstrado na figura abaixo:

Figura 35 - Formulário de novo cliente.

The screenshot shows a web browser window with the URL `localhost/tg2/#/cliente/3`. The page title is "WebSeg CRM". The form is titled "Nome do Cliente" and contains the following fields and sections:

- Nome do Cliente:** Input field with the value "João da Silva".
- Tipo de Pessoa:** Dropdown menu with "Física" selected.
- CPF:** Input field with the value "012.456.789-00".
- Observações:** Large text area for notes.
- Relações:** Section with two rows:
 - Row 1: Relationship type "Pai/Mãe", count "2", and name "Jorge Machado da Silva".
 - Row 2: Relationship type "Amigo", ID field "Id", and name field "Nome do cliente".
- Contatos:** Section with a dropdown for "Celular" and a phone number field containing "(00) 0000-0000".

At the bottom right, there is a timestamp: "Última atualização por Administrador em 03/07/2014 01:37:12".

Ao utilizar a vinculação de dados nos dois sentidos (*two-way data binding*) do AngularJS, os dados preenchidos no formulário do novo cliente são atualizados automaticamente na variável `$scope` do *controller* correspondente.

Figura 36 - Código HTML com AngularJS e variáveis correspondentes no *controller*.

```
<div ng-controller="clienteCtrl" class="cliente-form row">
  <div class="col-lg-12">
    <form name="formName" ng-submit="write()" novalidate>
      <div class="row">
        <div class="col-lg-1 text-center">
          <a ng-show="cliente.clienteId" ng-href="#/cliente/{{cliente.clienteId}}">
            <small ng-bind="cliente.clienteId"></small><br>
          </a>
          <span ng-hide="cliente.clienteId">
            <small>Novo</small>
            <br>
            
          </span>
        </div>
        <div class="col-lg-9 form-group">
          <label>Nome do Cliente</label>
          <input ng-model="cliente.nome" class="form-control" type="text" required>
        </div>
      </div>
    </form>
  </div>
</div>
```



```

app.controller('clienteCtrl',
  function($scope , $routeParams , $http , $location)
  {
    $scope.write = function()
    {
      // Lê as variáveis com two-way data binding
      var values = {
        clienteId: $scope.cliente.clienteId,
        nome: $scope.cliente.nome
        // ...
      }

      var writeRequest = { model: 'cliente', action: 'create',
        values: values };

      // Requisição de escrita para o servidor
      $http.post($scope.app.server, writeRequest)
        .success(function(data) {
          //...
        })
    }
  }
);

```

Quando o usuário clica no botão salvar, os dados são enviados ao servidor no formato JSON, definido pela interface de comunicação cliente-servidor.

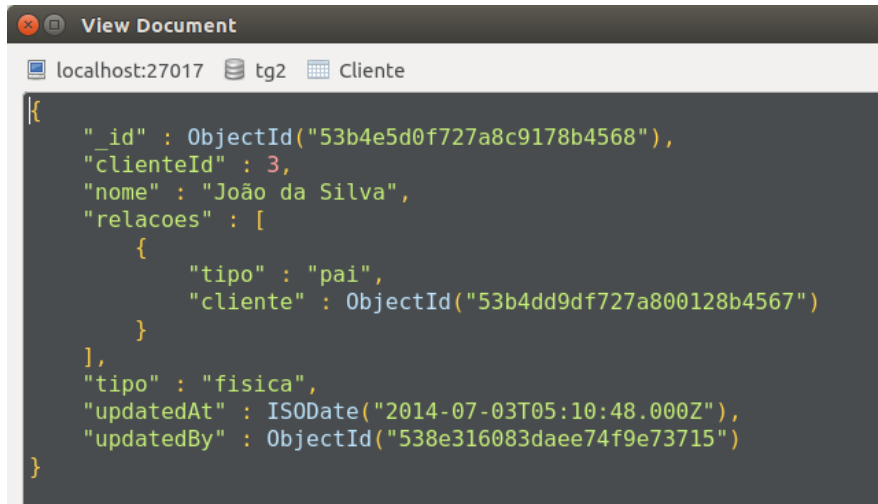
A aplicação-servidor valida a sessão do usuário, os dados recebidos, e os prepara para serem inseridos no MongoDB. Utilizando os documentos embutidos do MongoDB (*embedded document*), a relação entre os dois clientes, pai e filho, é armazenada dentro de um *array* no mesmo registro do cliente.

Figura 37 - Código de inserção e documento armazenado no MongoDB.

```

db.cliente.insert(
{
  "clienteId" : 3,
  "nome" : "João da Silva",
  "relacoes" : [{
    "tipo" : "pai",
    "cliente" : ObjectId("538e316083daee74f9e73715")
  }],
  "tipo" : "fisica",
  "updatedAt" : ISODate("2014-07-03T05:10:48.000Z"),
  "updatedBy" : ObjectId("538e316083daee74f9e73715")
}
)

```



```
{
  "_id" : ObjectId("53b4e5d0f727a8c9178b4568"),
  "clienteId" : 3,
  "nome" : "João da Silva",
  "relacoes" : [
    {
      "tipo" : "pai",
      "cliente" : ObjectId("53b4dd9df727a800128b4567")
    }
  ],
  "tipo" : "fisica",
  "updatedAt" : ISODate("2014-07-03T05:10:48.000Z"),
  "updatedBy" : ObjectId("538e316083daee74f9e73715")
}
```

O banco dados cria automaticamente um objeto do tipo ObjectId na propriedade `_id`, a chave primária padrão de todas as coleções (tabelas).

8 CONCLUSÕES

8.1 Resumo de resultados

Os resultados do trabalho no geral foram bastante satisfatórios, pois as necessidades propostas no projeto foram todas atendidas em menor ou maior grau. Assim, ao final da pesquisa e implementação, foi desenvolvida a base conceitual para a criação de um sistema amplo e completo.

Conseguimos mostrar, no capítulo 4, como o MongoDB é capaz de aumentar a flexibilidade na utilização e desenvolvimento com o banco de dados. Em especial, como suas estruturas flexíveis são capazes de implementar de forma eficiente as relações entre os dados, e também trazer os conceitos de herança entre os objetos para o banco de dados. Vale ressaltar também que MongoDB apresenta uma complexidade adicional mínima para sua utilização se comparado com os banco relacionais comuns.

No capítulo 5, mostramos como a utilização do AngularJS simplifica o desenvolvimento de uma interface ágil, que ainda assim consegue manter a lógica da aplicação separada da interface. De todos os resultados obtidos, consideramos esse o mais surpreendente. AngularJS não só implementa nativamente todas as situações mais comuns para a exibição de um conteúdo dinâmico, mas dispõe de alternativas robustas para a implementação dos casos que ainda não foram tratados pela biblioteca. Tudo isso garantindo um código elegante e organizado para a interface, que certamente garante menos erros e melhor manutenção.

A busca unificada, vista no capítulo 6, foi o último problema a ser tratado. Mostramos que com as abordagens tradicionais em SQL é complexo atingir a todos os requisitos que nos eram necessários: amplitude na busca, todos os campos, e relevância na ordenação da resposta. Isso foi possível de forma simples com a utilização do Solr nas configurações adequadas.

8.2 Limitações do trabalho

Na adoção e pesquisa de todas as tecnologias novas, ficou evidente uma mesma dificuldade: há menos material de documentação, exemplos, bibliotecas adicionais e integração com as IDEs do que as tecnologias tradicionais. Não foi uma surpresa, mas é claramente uma limitação.

Nesse item, o pior exemplo é o Solr. Sua documentação muito pobre faz com que sua curva de aprendizagem seja muito lenta em relação ao seu benefício ao

sistema. Há bastantes detalhes de configurações possíveis. Isso significa que, para configurá-lo como se deseja, é necessário entender tantos detalhes internos da sua implementação que isso torna a complexidade de seu funcionamento comparável à implementação de tecnologias tradicionais de busca. Apesar disso, apresenta vantagens, embora pequenas. De tal forma que, se diminuirmos um pouco os requisitos, seu uso não compensa.

Com relação ao MongoDB, por ser um banco de dados novo, há poucas bibliotecas boas para utilizá-lo. Apesar disso, é uma ferramenta simples e flexível. Sua limitação de não ter chaves estrangeiras precisa ser endereçada pela aplicação. Ademais, a biblioteca Doctrine ODM, que foi utilizada com o PHP, se mostrou muito imatura, ou seja, cheia de pequenos problemas.

Outro ponto importante, negativo desse trabalho, com relação ao MongoDB, é que não foi realizado nenhum teste com uma escala maior de dados. Todos os conceitos foram testados, mas não em uma escala real.

No quesito curva de aprendizagem, AngularJS é o mais complicado dentre os três bancos. Por introduzir muitos conceitos novos e toda uma forma nova de pensar a interface gráfica na web, há um tempo muito grande para seu domínio. Vale ressaltar, entretanto, que o esforço é compensado pela qualidade do resultado produzido com ele.

8.3 Perspectivas futuras

O resultado geral do trabalho atendeu às expectativas, e as perspectivas para a implementação completa do sistema são boas. Uma melhor análise das três abordagens novas será feita à medida que o sistema estiver mais completo. Talvez novas qualidades ou reveses sejam detectados, então.

Quando finalizado, espera-se que possa contribuir para o crescimento da Corretora e, talvez, se transformar em um produto, ou serviço, comercializável.

REFERÊNCIAS

- [1] CHODOROW, K.; DIROLF, M. **MongoDB: The Definitive Guide**. O'Reilly Media; 1 edition (September 24, 2010). ISBN-13: 978-1449381561.
- [2] **Create an Auto-Incrementing Sequence Field**. MongoDB Manual 2.6 <<http://docs.mongodb.org/manual/tutorial/create-an-auto-incrementing-field/>>. Acesso em Junho 2014.
- [3] LYNN, JOHN. **MongoDB Aggregation Framework Principles and Examples**. Amazon Digital Services, Inc.
- [4] **AngularJS API Docs**. Version 1.3.0-beta.14. <<https://docs.angularjs.org/api>>. Acesso em Junho 2014.
- [5] LERNER, ARI. **ng-book: The Complete Book on AngularJS**. 1. ed. FullStack.io, 2003. ISBN 978-0-9913446-0-4
- [6] MURRAY, N. **Form validation with AngularJS**. Publicado em 06 de Julho de 2013. <<http://www.ng-newsletter.com/posts/validations.html>>. Acesso em junho 2014.
- [7] GRAINGER, TYMOTHY; POTTER, T. **Solr in Action**. 1. ed. Manning Publications, 2010.
- [8] **Analyzers, Tokenizers, and Token Filters**. Apache Solr Wiki <<https://wiki.apache.org/solr/AnalyzersTokenizersTokenFilters>>. Acesso em Junho de 2014.
- [9] KUC, RAFAL; SMILEY, DAVID. **Apache Solr 4 Cookbook**. 2. ed. Packt Publishing, 2009.

ANEXO A – TRABALHO DE GRADUAÇÃO 1

Trabalho de Graduação I (TG1)

Engenharia da Computação (ECP)

Instituto de Informática (INF)

Universidade Federal do Rio Grande do Sul (UFRGS)

Aplicação Web para Gerenciamento de Corretora de Seguros

Autor: Gabriel Casagrande Stabel

Orientador: Dr. Marcelo Soares Pimenta

Porto Alegre, 04 de Abril de 2014.

Introdução e Motivação

As tecnologias de software para a internet evoluíram muito, e muito rapidamente, nos últimos anos. Especialmente as tecnologias voltadas para permitir complexas e ricas aplicações baseadas em navegadores que não eram possíveis no passado. Sugiram novas linguagens, bibliotecas e frameworks que abriram muitas possibilidades em termos do que pode ser desenvolvido na web.

Apesar disso, nosso mercado de software no Brasil, na média, ainda está longe de absorver e transformar todos esses novos potenciais em realidade dentro das empresas. No ramo específico de seguros isso não é diferente. Tanto as seguradoras como as corretoras de seguros no geral ainda utilizam tecnologias antigas, e, infelizmente, muitas vezes com implementações de baixa qualidade. Isso impacta diretamente no desempenho do trabalho, na produtividade.

O aluno-autor deste trabalho é sócio-investidor - que não exerce diretamente a atividade fim da empresa - da Stacas Corretora de Seguros Ltda., uma pequena corretora de seguros localizada em Porto Alegre, e possui amplo conhecimento, acadêmico e profissional, em programação. Dessa forma, tem interesse direto em aplicar essas novas tecnologias para desenvolver um aplicação capaz de impulsionar o crescimento da empresa, pois acredita que um sistema informatizado ágil, robusto e bem estruturado é um investimento sólido e direto na organização de processos administrativos. Além disso, o software desenvolvido pode tornar-se um produto ou serviço a ser comercializado futuramente.

Objetivos

Desenvolver uma aplicação web para gerenciar todos o processos internos (exceto controle de caixa, bancário e fiscal) de uma corretora de seguros, desde as negociações em andamento até o arquivamento digital das apólices dos seguros fechados.

Os cadastros básicos que devem ser administrados pela aplicação são quatro:

- Clientes: dados de pessoas físicas e jurídicas, e informações das inter-relações entre os diversos clientes.
- Seguros: dados comuns de todos os seguros e de cada ramo específico (automóvel, residencial, vida, previdência, viagem, etc).
- Casos em Andamento: seguros novos, renovações, sinistros, consultas, etc.
- Arquivos: arquivos digitais e imagens de documentos de clientes e seguros (apólices, contratos, etc).

Do ponto de vista técnico os objetivos são:

- Interface gráfica ágil e dinâmica. Desenvolvida em Javascript e HTML5, no estilo Single-page application. Serão utilizadas as bibliotecas JQuery^[1] e Knockout^[2].

- Separação clara entre interface gráfica (no cliente) e manipulação da dados (no servidor), utilizando um formato de transferência padrão como JSON^[9].
- Criação de uma aplicação *stateless* no servidor, uma aplicação RESTful, apenas mantendo o registro de sessão.
- Sistema de busca unificado (apenas um campo de busca para todos os dados), provavelmente interligando o banco de dados com Solr^[4] ou Sphinx^[5].
- Implementar uma arquitetura no estilo MVC (*model-view-controller*), com influências MVVM (*model view-viewmodel*) da biblioteca Knockout.
- Utilizar um banco de dados *document-oriented* como MongoDB^[3] para aumentar a flexibilidade no desenvolvimento e na manutenção futura.

Banco de Dados

Bancos de dados com interface SQL (*Structured Query Language*) se tornaram muito populares, em especial as implementações open-source MySQL e PostgreSQL. Esses dois bancos são amplamente utilizados, maduros e estáveis. Já trabalhei com ambos em diversos projetos. Conheço as características de uso, diversas peculiaridades, e ferramentas auxiliares para trabalhar com eles. Assim, inicialmente, eu não estava procurando outro banco de dados para esse projeto quando me deparei com MongoDB.

MongoDB alega ser o maior banco de dados open-source com conceito *NoSQL* (sem usar SQL). Essa característica propriamente, de não usar SQL, não me chamou a atenção, pois não considero isso uma vantagem (justo pelo contrário). O que me chamou a atenção foi o conceito de armazenar os registros como documentos flexíveis, sem usar o conceito de tabelas estruturadas. Isso permite a fácil implementação do conceito de herança da programação orientada a objetos, uma tarefa complexa em banco de dados estruturados tradicionais.

No caso concreto dessa aplicação, existem vários ramos de seguros distintos (cerca de uma dezena) que compartilham apenas alguns dados em comum. Isso pode ser facilmente implementado como herança, onde cada seguro de um ramo específico herda características de um registro de seguro genérico.

Outras características que não existem em bancos comuns, ou são de difícil implementação, também despertaram meu interesse, como a possibilidade de incluir *arrays* nos registros (tanto de tipos de dados básicos como de complexos), de criar registros dentro de registros (*embed*), e de não ser necessária a definição prévia do tipo de dados dos campos de cada registro (o que é muito útil durante o desenvolvimento e manutenção).

Abstração da interface com o Banco de Dados

Apesar dos aspectos positivos, há pontos negativos que me deixam apreensivos em adotar MongoDB sem nenhuma alternativa de mudança para uma base mais tradicional.

Não existem restrições (*constraints*) a chaves estrangeiras em MongoDB, logo a consistência entre os registros fica a cargo da aplicação. Também não há um equivalente ao comando *join* do SQL, então buscas complexas entre tabelas também ficam a cargo da aplicação.

Sendo assim, decidi adotar uma abordagem mais conservadora e criar uma camada de abstração para separar o acesso ao banco de dados do resto da aplicação e, desse modo, possibilitar uma mudança mais fácil para um banco de dados tradicional se for necessário.

Arquitetura das tabelas (coleções em MongoDB)

A aplicação terá quatro tabelas principais: clientes, seguros, casos e arquivos. Além disso, cada seguro terá itens e a cada caso haverá registros. A ideia original é de que os itens e os registros usem o sistema de *embed-documents* do MongoDB. Isso será testado melhor ao longo do projeto.

Considero essas tabelas como principais, pois elas terão bastantes operações de escrita, além de leitura. Haverá outras tabelas como de usuários, seguradoras, ramos etc, mas essas tabelas serão secundárias, ou seja, quase somente leitura.

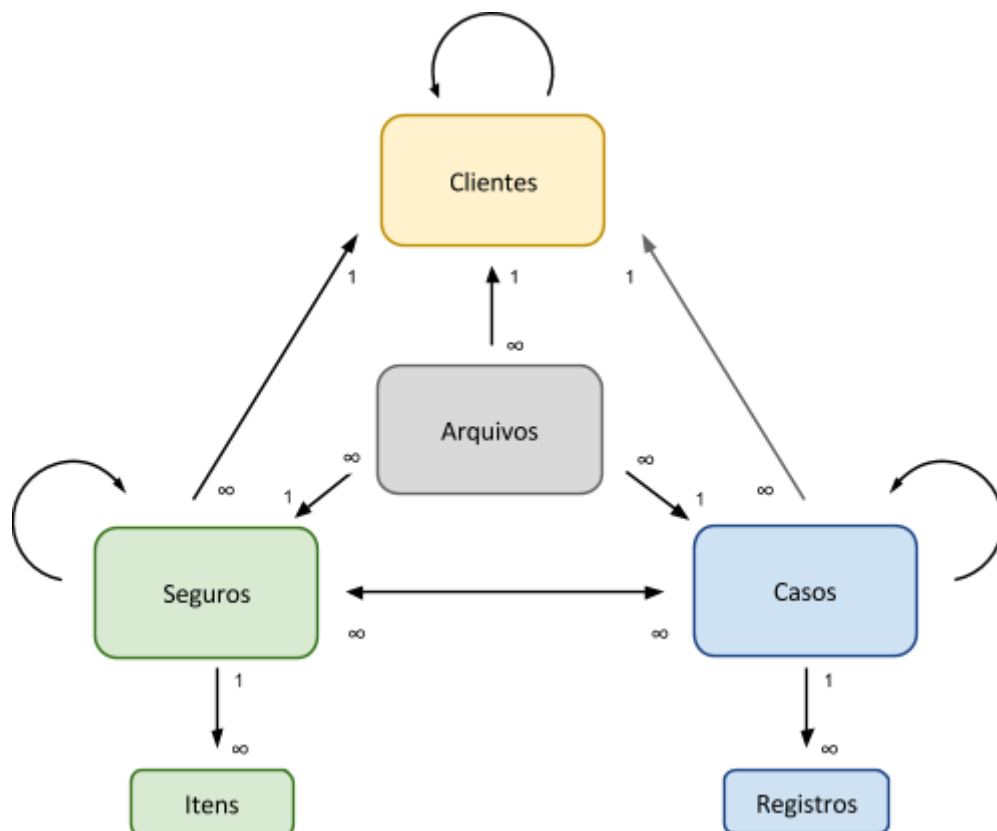


Diagrama simplificado das tabelas principais e suas inter-relações.

As cores, na arquitetura das tabelas acima, fazem referências às cores da interface gráfica que pretendo utilizar para criar uma identificação visual aos usuários com cada tipo de registro: amarelo para clientes, verde para seguros e azul para os casos (os arquivos terão a mesma cor do contexto que forem utilizados).

Modelos de Arquitetura de Software

Existem diversos padrões de projeto de software (*design patterns*) utilizados em aplicações na web. Entre os principais estão o MVC^[10] e suas variações, as vezes chamadas de MV*, já que os conceitos de *model* e *view* estão sempre presentes de uma forma pouco alterada, e o *controller* é que costuma sofrer variações maiores. Duas dessas variações serão as influências nesse projeto.

Me refiro a influências, pois não pretendo seguir estritamente um desses padrões. Considero um padrão mais adequado ao servidor e outro mais adequado ao cliente. Pode-se dizer que estarei usando, como algumas literaturas falam, padrões aninhados, ou seja, padrão dentro de padrão.

No ponto de vista do servidor, será adotado o padrão MVP (*model-view-presenter*), no qual a lógica da aplicação migra do *model* para o *controller*, que passa a ser chamado de *presenter*. No passado também ouvi esse padrão ser chamado de *fat-controller* (ou grande controlador). Já a *view* passa a ser considerada todo o cliente, o navegador, apesar desse último ter seu próprio padrão de programação.

O padrão do cliente será mais parecido com modelo MVVM (*model-view-viewmodel*) com o auxílio da biblioteca Knockout, o que permite uma programação mais declarativa da interface.

Arquitetura do Aplicativo e Tecnologias Associadas

Abaixo, um diagrama da estrutura planejada para o software e a linguagem e biblioteca associada:

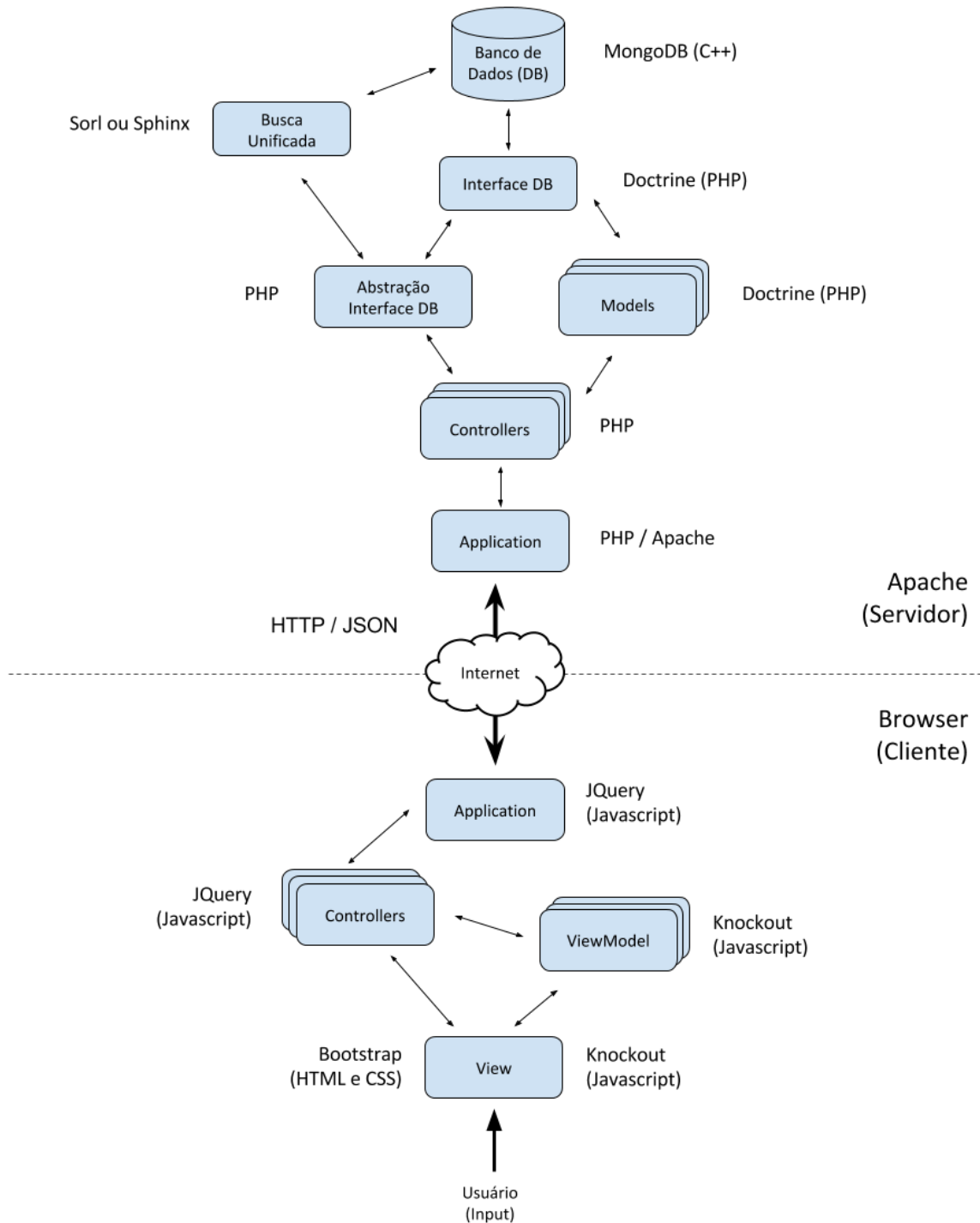


Diagrama das camadas do software (retângulos azuis) e suas respectivas tecnologias (texto ao lado dos retângulos).

- **Banco de Dados (DB):** utilizarei MongoDB, aplicativo servidor e biblioteca. MongoDB é implementado em C e C++.
- **Busca Unificada:** unifica todos os dados do software em um única busca. Utilizarei o Sora ou o Sphinx para gerenciar esse sistema de busca.
- **Interface DB:** conjunto de funções para acesso ao banco de dados pela aplicação. Usará a implementação da biblioteca Doctrine, em PHP, que adiciona uma pequena camada à implementação do MongoDB nativa do PHP5.
- **Abstração de Interface DB:** implementação própria em PHP para isolar a interface DB real do resto da aplicação, como explicado anteriormente.
- **Models:** conjunto de classes que representam os registros do banco de dados para a aplicação. Utilizarei a implementação do ODM (Object Document Mapper) da biblioteca Doctrine.
- **Controllers:** classes que implementam a lógica de cada ação solicitada pelo usuário. Também pode ser visto como o *Presenter* no padrão MVP.
- **Application (Servidor):** classe principal que recebe todas as requisições de usuário, filtra e valida, e passa ao *controller* correspondente. Por segurança o servidor Apache é configurado especialmente para garantir esse comportamento. A camada *Application* também retorna as páginas estáticas de *login* e da aplicação.
- **Comunicação Internet:** a comunicação entre o cliente e o servidor é feita transmitindo conteúdo JSON^[9] por uma conexão HTTP.
- **Application (Cliente):** camada que gerencia todas as requisições de saída e respostas do servidor, assim como exibição de eventuais erros. Controla também o endereço URL mostrado ao usuário. Será implementado usando JQuery.
- **Controllers:** objetos que agrupam funções de manipulação de interface específica de cada módulo da aplicação e preenchem os *ViewModels* com os dados recebidos do servidor. Será implementado usando JQuery.
- **ViewModel:** objetos que contêm o conteúdo a ser exibido pelas *view*. Utilizando o *Knockout* esses dados são automaticamente atualizados quando modificados pelo usuário na *view*.
- **View:** interface gráfica do usuário, utilizando Bootstrap^[8] como biblioteca gráfica (utiliza HTML5 e CSS3). Os dados dinâmicos são apresentados com o auxílio da biblioteca Knockout.

Busca Unificada

Uma das funcionalidades mais importantes que quero adicionar ao sistema é a possibilidade de buscar em diversos registros, de diversas tabelas, ao mesmo tempo, e proporcionar com isso uma busca única e central, na qual o usuário pode encontrar um texto (uma ou mais palavras), ou fragmento de texto (pedaços de palavras), em qualquer lugar do sistema.

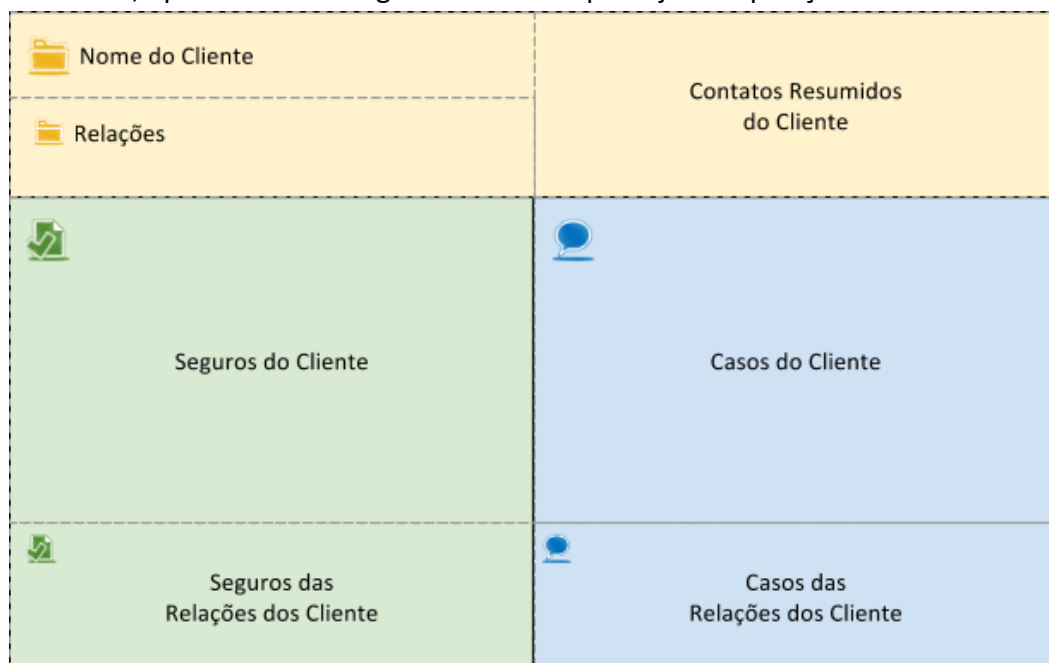
Pretendo utilizar a ferramenta o Sorl ou Sphinx, em conjunto com o banco de dados MongoDB, para implementar esse mecanismo de busca. Ambas ferramentas trabalham paralelamente ao banco de dados e à aplicação. Por essa razão, não necessita de implementação complexa específica para seu uso. Claro que isso ocasionará um *overhead* de recursos (cpu, memória, disco, etc). Esse impacto será medido depois nos testes, durante o projeto.

Para obter esse tipo de independência as duas ferramentas utilizam um sistema de troca de arquivos XML para atualização dos dados. Além disso, possuem sua próprias estruturas internas otimizadas para permitir esse tipo de busca de forma eficiente.

Resumo do Cliente

Além da busca unificada, outra implementação importante será a interface resumida. Essa tela reúne todas as informações mais importantes (contatos, seguros vigentes e casos abertos) de um cliente e de todas as suas relações com outros clientes (família, amigos, funcionários se for um empresa, etc).

Abaixo, apresento um diagrama de como planejo a disposição dessas informações:



Cronograma Planejado

1. (Concluído) Estudo e análises das tecnologias atuais (consolidadas e modernas) de aplicação web.
2. (Em andamento) Projeto da aplicação.
3. Desenvolvimento da aplicação.
4. Testes e análises da aplicação.
5. Redação, revisão, e conclusão do trabalho de graduação.

Ano: 2014

Etapa	Março	Abril	Maio	Junho	Julho
1	1ª 2ª 3ª 4ª				
2	3ª 4ª	1ª 2ª			
Intervalo		4ª	1ª		
3		1ª 2ª 3ª	2ª 3ª 4ª	1ª 2ª 3ª 4ª	... *
4					1ª 2ª
5				1ª 2ª 3ª 4ª	1ª 2ª 3ª 4ª

* o desenvolvimento prosseguirá, após término do trabalho de graduação, até que a aplicação atinga o nível de maturidade necessário para ser posto em produção, isto é, uso efetivo pela corretora.

Referências

Referências aos sites das tecnologias citadas:

- [1] JQuery: <http://jquery.com/>
- [2] Knockout: <http://knockoutjs.com/>
- [3] MongoDB: <http://www.mongodb.com/>
- [4] Solr: <http://lucene.apache.org/solr/>
- [5] Sphinx: <http://sphinxsearch.com/>
- [6] Apache Server: <http://httpd.apache.org/>
- [7] PHP: <http://www.php.net/>
- [8] Bootstrap: <http://getbootstrap.com/>
- [9] JSON: <http://www.json.org/>
- [10] MVC: <http://en.wikipedia.org/wiki/Model-view-controller>