ANÁLISE COMPARATIVA ENTRE DOIS FRAMEWORKS MVC WEB PARA PLATAFORMA JAVA: SPRING MVC E VRAPTOR

Cezar Toshiaki Nakase
cezar.nk@gmail.com

Prof. MSc. Claudinei Di Nuno
professorclaudinei@uol.com.br

Curso de Pós-Graduação *Lato Sensu* em Desenvolvimento Orientado a Objetos com Java
UNESA–Universidade Estácio de Sá

Resumo
O objetivo deste artigo é de apresentar o comparativo de dois *frameworks Java* voltado ao desenvolvimento web, no qual serão o *Spring MVC* e *VRaptor 4.* Com intuito de auxiliar o desenvolvedor na escolha do *framework*, com base na estrutura e necessidade do seu projeto. Neste artigo será apresentado um estudo de caso, o desenvolvimento de um protótipo utilizando os *frameworks,* podendo observar, o fluxo do processamento da informação, a divisão do código em camadas, a realização de operações de registrar, alterar, excluir e consultar informações junto ao banco de dados, a utilização da injeção de dependências, o tratamento de dados do tipo *json* e a inclusão de validadores. Além disso, serão abordados os pontos fortes e fracos dos *frameworks*, a dificuldade em relação à curva de aprendizagem, análise e continuidade da plataforma e suporte do mantenedor. Com base nas informações apresentadas, o artigo servirá como disseminador dos *frameworks Java* para web e de material de apoio para futuras pesquisas e desenvolvimento.

Excluído: a

Palavras-chave: *Framework*. Comparativo. Padrão MVC. *Vraptor* e *Spring MVC*.

1 Introdução

Quando iniciamos o desenvolvimento de um projeto web na linguagem *Java*, uma das questões que muitos programadores têm de defini-la, é com qual ferramenta será utilizada para o desenvolvimento do projeto?

E desenvolver sistemas em *Java* para web, antes das criações dos *frameworks*, que em resumo é um conjunto de bibliotecas que reúne inúmeras funcionalidades que ficam à disposição do programador, aumentando sua produtividade; era uma tarefa trabalhosa, pois mesmo o *Java* possuindo os *Servlets* e o JSP para auxiliar nos serviços específicos web, as requisições e tratamentos eram feitos de forma manual, onde cada programador criava sua metodologia de criação, bibliotecas e ferramentas com códigos massivos para sanar suas necessidades, assim não havendo uma padronização no desenvolvimento web com *Java*.

Atualmente, temos a disposição vários *frameworks* para facilitar a produtividade e a padronização do código. Porém, devemos analisar se vale à pena utilizar? Como escolher? Qual ganho e vantagem terão ao escolher uma?

Baseado nesses questionamentos, por boa prática, devemos seguir alguns critérios para a escolha do *framework*. Por exemplo, a sua estrutura, pois verificamos se ele poderá atender as necessidades; o quão reconhecido ele é, pois assim podemos ter mais informações através de comunidades e fóruns a respeito de novas ideias, nova funcionalidade e a qualidade do *framework*; segurança, verificar a capacidade de garantir o funcionamento do sistema, o

gerenciamento de riscos e redução de vulnerabilidade; e a documentação, pois sendo bem formulada serão mais fáceis o seu entendimento e a sua utilização.

Seguindo as boas práticas citado anteriormente, este artigo vai apresentar os *frameworks Spring MVC*e o *VRaptor4*.

E a relevância da escolha dos frameworks encontra-se da seguinte forma: *Spring MVC* – atualmente, um dos *frameworks* de *Java* para web mais utilizada, além de ser mais completo em questões de bibliotecas e ferramentas e com muitos módulos que facilitam na configuração do projeto. *VRaptor* 4 – um *framework* desenvolvido por brasileiros, levando o conceito de praticidade na criação do projeto por seguir a convenção de *Convention over Configuration,* conceito de redução de arquivos adicionais de configuração, facilitando e agilizando o desenvolvimento do projeto e a manutenção do código; e por possuir total integração com *Java EE*.

Para análise comparativa, será apresentado um estudo de caso, no qual foi desenvolvido um protótipo de um sistema Web para auxiliar na gestão de controle de pagamentos dos ocupantes de um prédio e no gerenciamento de registro dos moradores e condômino. A aplicação foi escolhida por conter telas de cadastro, edição, exclusão, consulta e controle de acesso aos usuários do sistema. Dessa forma foi possível avaliar o desempenho dos *frameworks* tanto nas tarefas, de simplesmente manter uma tabela num banco de dados, quanto para tarefas mais complexas utilizando tabelas relacionais e gerenciamento de sessão.

Por conseguinte, o objetivo final em servir como um material de pesquisa para desenvolvedores que buscam alternativas de frameworks *Java* para o desenvolvimento do seu projeto web, apresentando comparativo de ambos os frameworks, visando os prós e contras. Tornando mais claro a diferença em seus pontos específicos, facilitando a visualização para escolha do framework de acordo com o projeto.

2 Fundamentação Teórica

2.1 Padrões de Projeto para Ajudar o Desenvolvedor

Excluído: <#>Um Pouco da História do Java no Mundo Web¶
<#>Desde o avanço da internet, em meados de 1996, quando o Java iniciou seus primeiros passos ao mundo da web através dos recursos do *Servlets*: Oracle (2013, p. 1) "uma classe de linguagem de programação Java usada para estender os recursos de servidores que hospedam aplicativos acessados por meio de um modelo de programação de solicitação-resposta"; foi um grande avanço, dando vida à comunicação da linguagem Java, Banco de Dados e página web. Porém, no seu início, não havia uma organização definida entre os códigos, se misturavam códigos Java com os de apresentação (*HTML, CSS e JavaScript*) em um único arquivo. Com o surgimento do JSP (*Java Server Page*), onde o intuito era de organizar e definir as especificações que uma página web deveria seguir. ¶
<#>O JSP combina HTML e XML com *servlet* Java ™ (extensão de aplicativo de servidor) e tecnologias *JavaBeans* para criar um ambiente altamente produtivo para desenvolvimento e implantação de sites confiáveis, interativos e independentes de plataforma de alto desempenho (BERGMAN; CHOPRA, 2001, p. 1).¶
<#>Resultando em melhora na questão da separação dos códigos.¶
<#>Porém, mesmo com essas tecnologias criadas, o desenvolvimento web com Java, ainda não seguia uma estrutura organizada, e a necessidade de padronização ou uma arquitetura eficiente era inevitável.¶

Visando solucionar conflitos referentes à padronização no desenvolvimento de um projeto, foram criados conceitos onde reunia as melhores práticas formalizadas por programadores para solução de um problema em comum, técnicas adotadas dentro de um contexto para uma melhor organização, no qual segundo Alexander (1979 *apud* DOUG, 1993, p. 1) devia ter as seguintes características: "Encapsulamento, Generalidade, Equilíbrio, Abstração, Abertura e Combinatoriedade". Padrões de projetos não são específicos a uma empresa ou tecnologia, são soluções reutilizáveis aplicados pela comunidade de desenvolvimento como um todo. Onde define segundo Alexander (1979 *apud* DOUG, 1993, p. 1), "cada padrão descreve um problema no nosso ambiente e o núcleo da sua solução, de tal forma que você possa usar esta solução mais de um milhão de vezes, sem nunca fazê-lo da mesma maneira".

Segundo Laborde (2013), a utilização de padrões possibilita várias vantagens no desenvolvimento de software, dentre elas, pode ser citada à diminuição do processo de aprendizagem de um novo engenheiro de software dentro de um projeto, a reutilização e customização em projetos de desenvolvimento.

O uso de padrões auxilia no desenvolvimento de um projeto com bom nível de coesão e reusabilidade, o que facilita o processo de manutenção do software. (FERREIRA, 2013).

Resumindo a sua importância na utilização de um padrão, o ganho na qualidade do código, a reutilização de soluções para um determinado problema e contribuição de

especialistas e desenvolvedores para auxilio de manutenção, pois o padrão representa o conhecimento entre os envolvidos.

Diante desse cenário, o padrão de arquitetura MVC (*Model-View-Controller*) se destaca nas mais diversas aplicações, e principalmente em aplicações web, por sua organização, onde são divididas em camadas bem estruturadas em relação aos dados, regras de negócios e interface, além de ser um modelo utilizado por muitos *frameworks*.

O MVC criado em 1979 por Trygve Reenskaug, no qual ele justifica sua criação da seguinte forma: "Eu criei o padrão *Model-View-Controller* como uma solução óbvia para o problema geral de dar aos usuários controle sobre suas informações como visto de múltiplas perspectivas" (REENSKAUG, 1979, p. 1), transcrevendo o modelo mental humano e o modelo digital do computador.

- *Model* (Modelo): é a representação da estrutura de dados, camada responsável pela manipulação, consulta e persistência das informações no banco de dados;
- *View* (Visão): é a camada responsável pela interface apresentado ao usuário, no qual deve gerir requisições e respostas, através de componentes visuais, por exemplo, formulários, tabelas, menus e botões para entrada e saída de dados e que a visão reflita o estado do modelo;
- *Controller* (Controladora): responsável pela comunicação ou fluxo de informação entre a camada do modelo e a camada de visão, é nele onde dizemos quais regras de negócio o sistema deve seguir, quais operações devem ser executadas.

Destacando a camada Controllers, por ser a camada onde atua os *frameworks,* para Ladd (2006, p. 52), *Controllers* são responsáveis pelo processamento das requisições HTTP, pelo execução das regras de negócio, pela composição dos objetos de resposta e por passar o controle de volta ao fluxo de tratamento principal. O *Controller* não trata a rederização da visão, focando no tratamento de requisições e respostas e delegando ações à camada de serviço.

Assim concluindo, o padrão MVC nos proporciona as seguintes vantagens:

- O MVC por trabalhar em multi-camadas, tende a facilitar o gerenciamento do projeto e código de uma forma mais clara, pois mantém explícito o que cada camada deve executar, tornando o código mais limpo;
- É possível o desenvolvimento das camadas do projeto em paralelo ou escalável, além de maior integração da equipe em questão da divisão de tarefas;
- Possibilidade do reaproveitamento ou re-usabilidade do código de uma forma mais fácil, podendo incluir bibliotecas ou adicionando interfaces no projeto;
- Redução na dificuldade na manutenção do software, pois são realizadas as correções e alterações separadamente, não afetando outras camadas do sistema;
- Diversidade de frameworks ou tecnologias que estão utilizando essa metodologia de padrão de projeto.

2.2 *Spring* MVC, popular e com muitos recursos

Spring é organizado de forma modular, permitindo que você se preocupe apenas com *Spring* é organizado de forma modular, permitindo que você se preocupe apenas com os *Spring* é organizado de forma modular, permitindo que você se preocupe apenas com os

Spring é baseado na estrutura de POJOS (*Plain Old Java Object*), um objeto Java não limitado por nenhuma restrição, sem a necessidade de implementar ou estender classes pré-especificadas na estrutura. Segue o padrão de IOC (*Inversion of Control* – Inversão de Controle), um fluxo do controle do sistema é invertido, permitindo que indique outro

elemento o controle do método dizendo quando deve ser executado. Para Kayal (2008, p. 23) "O contêiner de Inversão de Controle de Mola (IOC) é o coração de todo o framework. Isso ajuda a unir as diferentes partes do aplicativo,formando assim uma arquitetura coerente". Um dos métodos utilizados pelo IOC é a injeção de dependência, no qual separa um objeto de suas dependências, deixando o foco da classe apenas nos recursos para realizar as tarefas que precisa, e possamos decidir quais dependências serão injetadas durante o tempo de execução. Uma das características mais conhecidas quando programamos com *Spring*.

Descrito por Hemrajani (2006, p. xxx) da seguinte forma:

> O Spring Web MVC Framework é uma estrutura robusta, flexível e bem projetada para aplicativos da web em rápido desenvolvimento usando o padrão de design MVC, usando este módulo Spring são semelhantes àqueles que você obtém do resto do Spring Framework.

Além de possuir uma documentação robusta, uma comunidade ativa onde possui uma página exclusiva no site do *StackOverflow,* apenas com perguntas relativas ao *Spring* MVC e um amplo pacote de ferramentas abrangente para qualquer tipo de projeto que você possa ter.

2.3 *VRaptor* para Agilizar o Desenvolvimento

Diferente de outros frameworks, *VRaptor*, criado em 2004, pelos brasileiros Paulo Silveira e Guilherme Silveira, apresenta-se como uma alternativa eficiente e com a proposta de sua simplicidade e por trazer a ideologia de ser rápido e de fácil aplicação. E desde esse período, houve novas versões e melhorias, acompanhando as melhores práticas de desenvolvimento no mercado e desempenho; atualmente o *framework* encontra-se na versão 4, trazendo total integração com *Java* EE 7, mas sempre mantendo sua proposta. "O *VRaptor* 4 traz alta produtividade para um desenvolvimento Java Web rápido e fácil com CDI[1]" (VRAPTOR, 2018, p. 1).

Além disso, o *VRaptor,* trabalha com o conceito de estrutura MVC e integra com as arquiteturas atuais como o REST – *Representational State Transfer* (Transferência de Estado Representacional) e *ActionBased,* trazendo consigo mais benefícios em sua utilização.

O *framework VRaptor* possui a característica de flexibilidade, permitindo a possibilidade de sobrescrever praticamente quase todos os seus comportamentos, sem a necessidade das configurações em XML. Assim, o desenvolvedor ganha autonomia para fazer ajustes e configurações específicas de acordo com seu projeto. Essa facilidade é descrita por Cavalcanti (2014, p. 3), mostrando a importância da seguinte forma: "Mesmo os problemas mais complexos e necessidades mais específicas dos projetos conseguiram ser resolvidos sobrescrevendo o comportamento do *VRaptor* usando os meios normais da sua API, ou sobrescrevendo um de seus componentes".

Finalizando, o *VRaptor* tem se empenhado em atrair e conquistar desenvolvedores, através dos conceitos citado acima, mantendo sempre seu foco de extensibilidade e desenvolvimento de aplicações o mais fácil e produtivo.

[1] CDI - *Contexts and Dependency Injection* (Injeção de Dependência e Contextos): possui um conjunto de serviços com intuito de melhorar a estrutura do código visando a produtividade, fornece uma arquitetura uniforme para injeção de dependência e o gerenciamento do ciclo de vida de beans (CDI, 2018).

Excluído: Por ser um dos mais completos frameworks com diversos componentes de auxílio, além de ser um dos mais utilizado para desenvolvimento web com Java. É verdadeiramente abrangente e expansivo que é mais do que capaz de assumir qualquer tarefa ou projeto em potencial no qual você deseja trabalhar.¶
Devido à modularidade da própria ferramenta, isso permite que você escreva códigos muito limpos e acessíveis. Há uma enorme quantidade de documentação excelente e uma comunidade próspera que o ajudará se você tiver dúvidas ou preocupações sobre como fazer certas coisas ou como certas coisas funcionam ou qualquer coisa desse tipo.¶
Possui as funcionalidades necessárias para processar as requisições HTTP, gerenciar os componentes para o processamento de dados além de processar e apresentar a resposta da requisição, seguindo o padrão MVC.¶

Excluído: O que seria o CDI e qual vantagem da utilização desse componente ao *VRaptor*?¶
CDI - *Contexts and Dependency Injection* (Injeção de Dependência e Contextos), em sua versão 1.0 introduzida junto a plataforma *Java* EE 6 e atualmente na versão 2.0 continuada no *Java* EE 8; possui um conjunto de serviços com intuito de melhorar a estrutura do código visando a produtividade, fornece uma arquitetura uniforme para injeção de dependência e o gerenciamento do ciclo de vida de beans. De acordo com o site oficial do CDI fica claro o objetivo e ganhos no caso de sua utilização (CDI, 2018):¶
Um ciclo de vida bem definido para objetos com estado ligados a contextos de ciclo de vida, em que o conjunto de contextos é extensível;¶
Um mecanismo de injeção de dependência sofisticado e seguro de tipos, incluindo a capacidade de selecionar dependências no tempo de desenvolvimento ou de implantação, sem configuração detalhada;¶
Suporte para modularidade *Java* EE e a arquitetura de componente *Java* EE - a estrutura modular de um aplicativo *Java* EE é levada em conta ao resolver dependências entre componente Java EE (CDI, 2018).¶

Excluído: ; Um exemplo, ao utilizar a arquitetura REST, Rotem-Gal-Oz (2012, p. 234), faz uma observação positiva em sua utilização: ¶
O termo implica integração fácil e rápida, frequentemente usando APIs e fontes de dados para produzir resultados enriquecidos que não era necessariamente a razão original para produzir os dados da fonte bruta. As principais características do *mashup* são combinação, visualização e agregação. É importante tornar os dados existentes mais úteis, além disso, para uso pessoal e profissional. ¶
Em questão da arquitetura *ActionBased*, que suportam as requisições de entrada através de controladores de ações. Silveira, Silveira, Lopes, Moreira, Stepat e Kung (2011, p. 152-153) orientam da seguinte forma: ¶
Podemos preferir trabalhar orientados a requisições e respostas com características stateless para melhor aproveitar a Web e outras ferramentas que giram em torno do HTTP, além de garantir escalabilidade e disponibilidade mais facilmente. Nesse tipo de situação, usar um framework action-based costuma se encaixar melhor como solução....

3 Materiais e Métodos

Para avaliação e levantamento de critérios de comparação entre os *frameworks* foi desenvolvido um protótipo com cada ferramenta estudada. A camada de negócio da aplicação foi desenvolvida e usada da mesma forma para os dois *frameworks*.

3.1 Ferramentas Utilizadas no Desenvolvimento do Sistema

Para o desenvolvimento do sistema, foi utilizado o sistema operacional Linux na distribuição Debian na versão 3.16.51 (8 jessie), a IDE JAVA (*Integrated Development Environment)* ou Ambiente de Desenvolvimento Integrado utilizado foi o *Eclipse* versão *Oxygen*.3a, para o servidor de banco de dados o MySql 5.5.62-0 e para o servidor de aplicação o *Apache Tomcat* 7.0.90.

3.2 Tecnologias Utilizadas no Desenvolvimento

O sistema desenvolvido foi baseado na arquitetura MVC, e os *frameworks* referente ao estudo trabalham na camada *Controller*, para as camadas de *Models* e *Views* foram utilizados as mesmas tecnologias.

A camada *Models,* responsável pelo gerenciamento e persistência das informações no banco de dados e execução das regras de negócios, foi utilizado o *framework* Hibernate na versão 5.3.7.

A camada Views, responsável pela apresentação com o usuário, e a interface que proporcionará à entrada de dados e a visualização de respostas geradas, nas aplicações web. Foram utilizados as seguintes tecnologias: JSP (Java Server Pages); JSTL, estendendo a especificação JSP adicionando uma biblioteca de tags para tarefas comuns; JQuery (1.12-4) e Bootstrap (3.3.7).

3.3 Descrição do Desenvolvimento das Funcionalidades do Sistema

Excluído: <#>JSP (Java Server Pages) no qual serve para criar páginas web com programação em Java;¶
<#>JSTL no qual estende a especificação JSP adicionando uma biblioteca de tags para tarefas comuns, tais como processamento de dados XML, execução condicional, loops e internacionalização;¶
<#>JQuery versão 1.12-4, biblioteca de funções Javascript, para facilitar a manipulação de elementos da interface, a troca de informações com a camada *Controller* além de facilitar a criação de animações;¶
<#>Bootstrap versão 3.3.7, framework para facilitar a criação de elementos de interface web, utilizando a combinação de HTML, CSS e *Javascript*, melhorando a experiência do usuário em um site amigável e responsivo.¶

Neste capitulo é apresentado a implementação do sistema proposto, um sistema de gestão predial para gestão de pagamentos dos moradores de um prédio e o controle de cadastro de todos residentes e condômino, suas funcionalidades e estrutura a ser desenvolvido, suas telas e principais códigos utilizando os *frameworks* referente ao estudo, *Spring MVC* e *Vraptor,* no qual, mencionado anteriormente, trabalham na camada *Controller*, que funciona de intermediário entre a camada de apresentação e a camada de negócios, sua função como já diz é controlar e coordenar o envio de requisições feitas entre a *View* e o *Model*.

3.4 Telas de Cadastro, Listagem e Edição

Neste item é demostrada a implementação de uma tela de cadastro e consulta do protótipo para criação e edição do perfil de um usuário, especificando detalhadamente cada particularidade de cada *framework*.

Figura 1 – Criação da Classe PerfilController

VRaptor

```
@Controller
public class PerfilController {

    private PerfilDao perfilDao;

    @Inject
    public PerfilController(PerfilDao perfilDao){
        this.perfilDao = perfilDao;
    }

    @Deprecated
    public PerfilController() {}
```

Spring MVC

```
@Controller
public class PerfilController {

    @Autowired
    private PerfilDAO perfilDao;

```

Fonte: Nakase (2019)

Na figura 1 mostra o início da criação da classe responsável pelas ações que serão executadas, nota-se que ambas *frameworks* utilizam a anotação *@Controller* para dizer que a classe *PerfilController* e seus métodos públicos, ficarão disponíveis para receber requisições web.

Após a criação da classe, o Vraptor utiliza a anotação @Inject para poder injetar as dependências em seu construtor, sendo necessário adicionar também um construtor *default*, já para o *Spring MVC* utilizamos a anotação *@Autowired* para que o *framework* identifique os pontos no qual será injetada.

Figura 2 – Funções Adiciona, Remove e Edita

VRaptor

```
@Post("perfil/adiciona")
@IncludeParameters
public void adiciona(@Valid Perfil perfil) {
    validator.onErrorForwardTo(this).form();
    perfilDao.adiciona(perfil);
    result.include("mensagem","Cadastrado com sucesso!");
    result.redirectTo(this).lista();
}

@Delete("/perfil/remove")
public void remove(Perfil perfil) {
    perfilDao.remove(perfil);
    result.include("mensagem","Removido com sucesso!");
    result.redirectTo(this).lista();
}

@Put("/perfil/edita")
public void edita(Perfil perfil){
    result.include(perfilDao.busca(perfil));
    result.redirectTo(this).form();
}

@Get("perfil/lista")
public void lista() {
    result.include("perfil",perfilDao.lista());
}
```

Spring MVC

```
@RequestMapping(value = "/perfil", method = RequestMethod.POST)
public String adiciona(@ModelAttribute("Perfil") Perfil perfil) {
    perfilDao.persist(perfil);
    return "redirect:/perfil";
}

@RequestMapping(value = "/perfil/{id}",method = RequestMethod.DELETE)
public String remove(@PathVariable("id") int id) {
    perfilDao.remove(perfilDao.find(id));
    return "redirect:/perfil";
}

@RequestMapping(method = RequestMethod.PUT)
public String edita(@ModelAttribute("Perfil") Perfil perfil) {
    perfilDao.merge(perfil);
    return "redirect:/perfil";
}

@RequestMapping(value = "/perfil", method = RequestMethod.GET)
public String lista(ModelMap modelMap) {
    modelMap.addAttribute("Perfil", perfilDao.findAll());
    return "perfil/lista";
}
```

Fonte: Nakase (2019)

Por seguinte, a figura 2 mostra como são criados os métodos controladores, responsáveis pelas ações de cadastrar, editar, remover e listar. Inicialmente, um ponto positivo verificado no *Vraptor*, é a possibilidade de dizermos de forma explícita, através do conceito de anotação, qual o tipo de requisição que será executado. Foram utilizados as anotações *@Post* para o método adiciona, *@Delete* para o método remove, *@Put* para o método edita e o *@Get* para o método lista. Importante observar que junto as anotações, foram declaradas as URL que estarão acessíveis tanto para camada *View* ou para o navegador.

Já ao *Spring MVC,* o mapeamento da URL é realizado através também do conceito de anotação, especificamente o *@RequestMapping*, onde dizemos também o tipo de método que será realizado através do parâmetro *RequestMethod.* Outra diferença entre os *frameworks*, é na questão da criação do método, onde no *Spring MVC,* para dizermos qual

View estamos referenciando, declaramos a tipologia do método em *String* e seu *return* (retorno) o caminho para onde queremos enviar a informação. Ao *VRaptor*, possui um método próprio para fazer esse serviço, não sendo necessário fazer a tipagem, e sim utilizando o método do framework *Result,* através dele podemos dizer qual View estamos referenciando, além de ter outros recursos disponíveis.

Figura 3 – Página JSP da tela de cadastro do perfil do usuário

VRaptor

```
<form action="${linkTo[PerfilController].adiciona(null)}" method="POST">

<div class="row">
<div class="col-md-7">
    <div class="form-group">
        <label for="nome">Nome:</label>
        <input type="text" placeholder="Digite o nome do usuario *" name="perfil.nome"
            class="form-control" value="${perfil.nome}" />
    </div>
</div>
```

Spring MVC

```
<form:form action="/perfil" method="POST" modelAttribute="Perfil">

<div class="row">
<div class="col-md-7">
    <div class="form-group">
        <label for="nome">Nome:</label>
        <input type="text" placeholder="Digite o nome do usuario *" name="nome"
        class="form-control" value="${perfil.nome}" />
    </div>
</div>
```

Fonte: Nakase (2019)

Na tela de cadastro (figura 3), ambos construídos na estrutura de *forms,* podemos notar a diferença na declaração da ação que será executada, ao form do *Spring MVC* (como podemos ver na linha), dizemos a *URL ('/perfil')* disponibilizada pelo *Controller*, já ao *form* do *Vraptor* podemos referenciar a classe e o método que executará a ação (${linkTo[PerfilController].adiciona(null)}).

Outro ponto a observar, é forma de referenciar os campos no formulário que serão enviados à camada *Controller* para serem gravados no banco de dados. Como podemos ver em um exemplo na (linha) do *form* do *Vraptor*, o campo nome para ser identificado na camada *Models*, é necessário dizer ao atributo *name, o* nome do *Controller* e o nome que está referenciado no banco de dados (perfil.nome), já ao formulário na (linha) do Spring MVC, apenas referencio ao atributo *name* o nome como está no banco de dados (nome).

Figura 4 – Método do *Controller* responsável por listar todos perfis cadastrados

VRaptor

```
1  @Get("perfil/lista")
2  public void lista() {
3      result.include("perfil",perfilDao.lista());
4  }
```

Spring MVC

```
1  @RequestMapping(value = "/perfil", method = RequestMethod.GET)
2  public String lista(ModelMap modelMap) {
3      modelMap.addAttribute("perfil", perfilDao.findAll());
4      return "perfil/lista";
5  }
```

Fonte: Nakase (2019)

Na figura 4, vemos como o *Spring MVC* e *Vraptor* atuando de forma parecida na camada *Controller*, enviando informações à camada *View* para carregar em uma página JSP, a listagem dos perfis cadastrados, em uma tabela.

No *Controller* do *VRaptor* conforme a (linha), vemos a utilização do serviço *include* do método *Result,* para enviar o objeto 'perfil' a camada *View*, no qual é uma lista retornada do banco de dados através da função 'perfilDao.lista()'.

No *Controller* do *Spring MVC* na *(linha),* temos o mesmo procedimento, apenas utilizando a nomenclatura específica do *framework*, no qual utiliza o serviço *addAtribute* do método *modelMap* para enviar o objeto 'perfil' a camada *View.*

E no código da página JSP, na figura 5 na linha 2, recebendo o objeto 'perfil' e carregando as informações através de um *loop* em *JSTL* as informações na página.

Figura 5 – Página JSP listando em uma tabela todos os perfis cadastrados

```
1  <tbody>
2  <c:forEach items="${perfil}" var="perfil">
3      <tr>
4          <td>${perfil.id}</td>
5          <td>${perfil.nome}</td>
6          <td>${perfil.cpf}</td>
7          <td>${perfil.data_nascimento}</td>
8          <td>${perfil.estado_civil}</td>
9          <td>${perfil.andar_ocupado}</td>
10     </tr>
11 </c:forEach>
12 </tbody>
```

Fonte: Nakase (2019)

3.5 Validação das Informações

A validação é a certificação de que o sistema atenda as necessidades e expectativas do cliente. Mantendo a integridade da informação e respeitando as regras de negócio.

Seguindo esse conceito, tanto o *Spring MVC* e o *VRaptor*, utilizam um dos recursos disponíveis pelo *JAVA EE 7* para fazer a validação, que é através do Bean Validation.

Figura 6 – Validação do formulário do método adicionar perfil no *Controller*

VRaptor

```
1  import javax.validation.Valid;
2
3  @Controller
4  public class PerfilController {
5
6      private PerfilDao perfilDao;
7      private Validator validator;
8
9      @Inject
10     public PerfilController(PerfilDao perfilDao,Validator validator){
11         this.perfilDao = perfilDao;
12         this.validator = validator;
13     }
14
15     @Post("perfil/adiciona")
16     public void adiciona(@Valid Perfil perfil) {
17         validator.onErrorForwardTo(this).form();
18         perfilDao.adiciona(perfil);
19         result.redirectTo(this).lista();
20     }
```

Spring MVC

```
1  import javax.validation.Valid;
2
3  @Controller
4  @RequestMapping("/perfil/**")
5  public class PerfilController {
6
7      @RequestMapping(value = "/perfil", method = RequestMethod.POST)
8      public String adiciona(@Valid @ModelAttribute
9
10         ("Perfil") Perfil perfil) {
11         if(result.hasErrors()) {
12           return "perfil/formulario";
13         }
14
15         perfilDao.persist(perfil);
16         return "redirect:/perfil";
17     }
```

Fonte: Nakase (2019)

Na figura 6, como é feito a implementação da validação na classe *Controller*, na linha 1 a importação da biblioteca com os recursos do BeanValidation e na linha 16 do *VRaptor* e linha 8 do *Spring MVC* a utilização através da anotação *@Valid* como parâmetro do método a ser validado.

3.6 Tela de Consulta com Utilização de Arquivo *JSON*

Neste item, será apresentado a tela onde um usuário pode consultar a lista de pagamentos de moradores registrados no sistema. Nesta tela foi necessário utilizar tabelas relacionais para retornar a lista de pagamentos, pois a lógica criada diz que, para um morador pode existir um ou vários pagamentos, nessa estrutura, fiz o retorno da lista utilizando arquivo *Json (Javascript Object Notation),* formato de transmissão de informações no formato texto, atualmente muito usado em *web services* e aplicações Ajax. Assim podendo avaliar como os *frameworks* em estudo trabalham com esse tipo de informação.

Figura 7 – Classe Pagamento da camada *Models* e os *Controllers* gerando arquivo *Json*

Pagamento - Models

```
1  import java.io.Serializable;
2
3  @Entity
4  public class Pagamento implements Serializable{
5
6      @Id
7      @GeneratedValue(strategy=GenerationType.IDENTITY)
8      private int id;
9      private BigDecimal valor;
10     private BigDecimal multa;
11
```

VRaptor

```
1  @Get("pagamento/listaPagamento")
2  public void listaPagamento() {
3      List<Object[]> lista = pagamentoDao.listaJson();
4      result.use(Results.json()).from(lista).serialize();
5  }
6
```

Spring MVC

```
1  import org.springframework.web.bind.annotation.ResponseBody;
2
3  @RequestMapping(value="/pagamento/listajson", method = RequestMethod.GET)
4  @ResponseBody
5  public List<Object[]> listJson(ModelMap modelMap){
6      return pagamentoDao.listaJson();
7  }
8
```

Fonte: Nakase (2019)

A figura 7 mostra que o *VRaptor* e o *Spring MVC* possa realizar operações com arquivos

Excluído:

json, é necessário implementar a classe *Serializable* na classe *Models* (Pagamento – *Models* linha 4).

Após isso, para o *VRaptor*, basta implementar na camada *Controller* no método listaPagamento o recurso *.json*() e *.serialize*() do método *Results*, como vemos na (*VRaptor* linha 4), assim o próprio *VRaptor* trata de transformar o resultado vindo do *Models* em objeto *json*.

Já no *Spring MVC*, foi necessário a instalação do *plugin Jackson* para fazer a serialização do objeto *json*. Porém, após a instalação, é só fazer a anotação com @ResponseBody ao método, que transforma o retorno em objeto *json,* (*Spring MVC* linha 4).

3.7 Tela Login e Utilização de *Interceptor*

É comum em sistemas possuir a regra de negócio onde apenas os usuários autorizados e que estejam logados possam fazer qualquer consulta ou modificação em nosso sistema.

Ao protótipo foi criado o sistema de login no qual possui o seguinte fluxo: Quando uma requisição chega, ele cai automaticamente ao *interceptor* de autenticação. O *interceptor* descobre se o usuário está logado; se ele está, ele deixa a requisição continuar como se nada tivesse acontecido. Mas, caso o usuário não esteja logado (ou seja, não existe nada na Sessão), ele redireciona para a página de login.

A seguir será mostrado como cada *framework* trabalham nesse gerenciamento de acesso, começando com o *VRaptor,* de início, criando-se uma classe chamado *LoginController* e o método *autentica,* no qual executará a regra de autenticação, recebendo os parâmetros *login* e senha e verificando se ambas informações é de um usuário válido e se o usuário está logado, caso positivo direciona a página *index*(), caso contrário emite a mensagem de usuário inválido e redireciona para página de *login* novamente (Figura 8).

Excluído: ¶

Excluído: .¶
C

Excluído: foi criado

Figura 8 – Método autentica() da classe LoginController

VRaptor

```
1  @Open
2      public void autentica(String login, String senha){
3          Usuario usuario = usuarioDao.busca(login,senha);
4          System.out.println(usuario);
5          if(usuario != null){
6              usuarioLogado.fazLogin(usuario);
7              System.out.println(usuario.getNome());
8              result.redirectTo(IndexController.class).index();
9          } else {
10             validator.add(new SimpleMessage("Login_invalido","Login ou senha incorretos"));
11             validator.onErrorRedirectTo(this).form();
12             System.out.println("não encontrado");
13         }
14     }
```

Fonte: Nakase (2019)

E para o sistema saber que o usuário está em uma mesma sessão, não sendo necessário fazer o *login* a cada requisição. Foi criado uma classe UsuarioLogado, onde foi determinado o escopo dela com a anotação *@SessionScoped*, assim, cada sessão de usuário terá uma classe mantendo suas informações em memória.

Figura 9 – Classe UsuarioLogado, responsável por armazenar o usuário logado no sistema

VRaptor

```
1  import java.io.Serializable;
2
3  @Named
4  @SessionScoped
5  public class UsuarioLogado implements Serializable{
6      private Usuario usuario;
7
8      public void fazLogin(Usuario usuario){
9          this.usuario = usuario;
10     }
11
12     public boolean isLogado(){
13         return this.usuario != null;
14     }
15 }
```

Fonte: Nakase (2019)

Após possuirmos o usuário da sessão, precisamos determinar quais métodos serão interceptados. Então, foi criado a classe AutorizaçãoInterceptor, para permitir a execução da lógica do nosso *Controller* apenas caso o usuário esteja logado. E para interceptar uma requisição é apenas anotarmos o método com *@AroundCall* (linha 8). Outra regra é que esse método precisa receber como parâmetro a classe *SimpleInterceptorStack* (linha 9), cujo método *next*() (linha 11) vai indicar o ponto em que o código será executado. E para finalizar, é preciso ensinar ao *interceptor* do *VRaptor* que esses métodos não devem ser interceptados. Para fazer isso, é necessário apenas adicionar um método anotado com *@Accepts* (linha 3) em nosso *interceptor* como vemos na figura 10.

Figura 10 – Classe AutorizaçãoInterceptor, responsável por gerenciar as requisições na sessão

VRaptor

```
1  public AutorizacaoInterceptor() {}
2
3  @Accepts
4  public boolean accept(){
5      return !method.containsAnnotation(Open.class);
6  }
7
8  @AroundCall
9  public void intercept(SimpleInterceptorStack stack){
10     if(usuarioLogado.isLogado()){
11         stack.next();
12     } else {
13         result.redirectTo(LoginController.class).form();
14     }
15 }
```

Fonte: Nakase (2019)

Vejamos agora na figura 11 o procedimento com o *Spring MVC*. Primeiramente foi criado a classe *LoginController* com dois métodos, efetuaLogin() (linha 2 ao 15) para validar o usuário e o autentica() (linha 18 ao 29) no qual testa a condição de caso o usuário seja válido, inclui na sessão, senão retorna a página de *login*.

Para lidar com a sessão, foi recebido na *action* do método a classe *HttpSession* (linha 18). Essa classe possui um método *setAttribute()*(linha 22) que permite guardar um objeto na sessão.

Excluído: ¶

Figura 11 – Métodos efetuaLogin() e autentica(), responsáveis por autenticar e incluir usuário a sessão

Spring MVC

```
1   @RequestMapping("/login/efetuaLogin")
2     public String efetuaLogin(Usuario usuario, HttpSession session) {
3
4       if(usuarioDao.busca(usuario)) {
5         // usuario existe, guardaremos ele na session
6         session.setAttribute("usuarioLogado", usuario);
7         session.setAttribute("usuarioNome",usuario.getNome_guerra());
8         return "redirect:/index";
9       }
10      // ele errou a senha, voltou para o formulario
11      return "redirect:/";
12    }
13          result.redirectTo(LoginController.class).form();
14      }
15  }

17  @RequestMapping("/login/loginEfetua")
18  public String autentica(Usuario usuario,HttpSession session,Model model){
19      Usuario us = usuarioDao.procura(usuario);
20
21      if(us != null){
22          session.setAttribute("usuarioLogado", us);
23          return "redirect:/index";
24      } else {
25          System.out.println("não encontrado");
26          model.addAttribute("mensagem","Login ou senha invalido!");
27          return "login/form";
28      }
29  }
```

Fonte: Nakase (2019)

Utilizando o *Spring MVC*, também temos o conceito de Interceptadores, que funcionam como Filtros, ou seja, toda requisição antes de ser executada passará por ele. Nele, podemos por exemplo, impedir que a requisição continue se o usuário não estiver logado.

Foi criado a classe *AutorizadorInterceptor* conforme figura 12, no qual foi utilizado o método *preHandle* da interface *HandlerInterceptorAdapter* (linha 5), classe responsável por interceptar e executar antes da ação.

Nesse interceptor, que verificaremos se existe a variável usuário logado na sessão. Caso positivo, deixa a requisição continuar, caso contrário devolvemos a tela de *login* (linhas 13 ao 18).

Figura 12 – Classe AutorizadorInterceptor, responsável por gerenciar as requisições na sessão

Spring MVC

```
1   import org.springframework.web.servlet.handler.HandlerInterceptorAdapter;
2
3   public class AutorizadorInterceptor extends HandlerInterceptorAdapter {
4       @Override
5       public boolean preHandle(HttpServletRequest request, HttpServletResponse response, Object
        controller)
6       throws Exception {
7
8           String uri = request.getRequestURI();
9           if(uri.endsWith("/") || uri.endsWith("loginEfetua") || uri.contains("resources")) {
10              return true;
11          }
12
13          if(request.getSession().getAttribute("usuarioLogado")!=null) {
14              return true;
15          } else {
16              response.sendRedirect("redirect:/");
17              return false;
18          }
19
20      }
21  }
```

Fonte: Nakase (2019)

Excluído:

Spring MVC

```
1   @RequestMapping("/login/efetuaLogin")
2     public String efetuaLogin(Usuario u
3
4       if(usuarioDao.busca(usuario)) {
5         // usuario existe, guardaremos
6         session.setAttribute("usuarioLo
7         session.setAttribute("usuarioNo
8         return "redirect:/index";
9       }
10      // ele errou a senha, voltou para
11      return "redirect:/";
12    }
13          result.redirectTo(LoginContro
14      }
15  }

17  @RequestMapping("/login/loginEfetua")
18  public String autentica(Usuario usuar
19      Usuario us = usuarioDao.procura(u
20
21      if(us != null){
22          session.setAttribute("usuario
23          return "redirect:/index";
24      } else {
25          System.out.println("não encor
26          model.addAttribute("mensagem"
27          return "login/form";
28      }
29  }
```

4 Discussão

4.1 Spring MVC

O *Spring MVC* ajuda a construir aplicações web flexíveis e com baixo acoplamento. O padrão de design Modelo-visão-controlador ajuda na separação da lógica de negócio, lógica de apresentação e lógica de navegação. Uma estrutura completa para a criação de aplicativos da web com enorme estabilidade, amplo alcance e usada por muitas pessoas, a simplicidade de configurar, pouca intrusão do framework, diminuição do acoplamento ao utilizá-lo, a modularização dos projetos, a ótima integração entre os projetos Spring e a fácil customização do *framework*. *Spring MVC* é um dos módulos que compõem o *Spring Framework* utilizado para construir aplicações web. Ele conta com as boas práticas de projeto para desenvolvimento de software web utilizando a plataforma Java EE. Pontos negativos a serem observados no *Spring MVC*, no qual embora simples, possui muitas camadas e abstrações que podem ser difíceis de depurar se surgirem problemas. Também é altamente dependente do núcleo do *Spring*. É uma estrutura antiga e madura que possui inúmeras maneiras de estendê-la e configurá-la - e isso, na verdade, tornando-a bastante complexa, ele ainda não fornece nenhuma estrutura rica para construir boas Interfaces. Curva de aprendizado íngreme, mas quando você trabalha com o produto algumas vezes, é muito fácil se adaptar e melhorar, porém se deseja incluir outros módulos do Spring, pode exigir mais tempo de aprendizagem, pois requer mais tempo para adquiri conhecimento para customizar novos componentes. A documentação oficial cobre praticamente tudo. O site oficial também tem uma série de ótimos tutoriais em formatos de vídeo e texto. Há links para os repositórios do *Github* para aplicativos de amostra do *Spring* e também há muitos tutoriais de terceiros para o fato de que o *Spring MVC* é amplamente utilizado por muitos desenvolvedores experientes. Entretanto, como o *framework MVC* é apenas uma parte do Spring, ele acaba tendo uma documentação menos detalhada tanto nos livros quanto na documentação oficial. *Spring MVC* tem uma comunidade massiva de seguidores que são muito úteis e forneceram vários tutoriais e respostas sobre o SO. A Spring até realiza uma conferência anual chamada *SpringOne*. Os fóruns do *Spring* e SO são ótimos lugares para perguntar e obter ajuda sobre qualquer coisa relacionada à Primavera. O blog e o boletim informativo do site mantêm os desenvolvedores informados sobre todas as notícias relacionadas à estrutura. Um levantamento em um dos maiores fóruns internacional o *StackOverflow*, o *framework* possui 49.722 interações relacionado ao *Spring MVC*, em uma matéria do site *JavaPipe* e *DailyRazor*, o *Spring MVC* aparece entre os dez *frameworks Java* mais utilizado, e um levantamento que foi realizado pelo *Rebellabs* em 2016 a *framework* apareceu em primeiro lugar em uso. O *Spring MVC* é bem reconhecido no mercado trabalho tanto nacional como internacional, pois junto com o *Spring*, está constantemente mudando e melhorando. A questão é que seus desenvolvedores terão que acompanhar constantemente a tecnologia para melhorar o aplicativo à medida que o Java cresce, os navegadores da Web mudam e outras melhorias acontecem no espaço da Web.

4.2 VRaptor

Se você precisa de uma estrutura fácil e funcional para criar programas de web com Java, o *VRaptor* é o caminho certo. De *e-commerces* a aplicações de grande escala. Sem dúvida, é fácil de usar e agradável criar um utilitário com este *framework*.

O *VRaptor* é um *Framework MVC* para desenvolvimento rápido de aplicações WEB que

faz uso das anotações e conceitos de inversão de controles e injeção de dependência. Outros conceitos como o de Convenção do Invés de Configuração tornam o desenvolvimento bastante produtivo sem perder flexibilidade tornando a curva de aprendizado muito pequena.

Um ponto negativo ao *VRaptor*, talvez seja por não possuir bibliotecas ou componentes voltado a camada da visão, exigindo ao desenvolvedor o conhecimento voltado ao front-end como por exemplo de linguagens como CSS, HTML e JavaScript.

Em questão da documentação o *VRaptor* possui a documentação oficial centralizada em seu site oficial, no qual é possível verificar instruções de uso, exemplos de aplicações e tutoriais, além de possuir a documentação traduzida em português como um diferencial, porém poderia ser ainda melhor se tivesse mais explicações mais detalhadas sobre seu funcionamento do fluxo interno e sua estrutura, além de como resolver algumas exceções a serem tratadas, o *VRaptor* também possui documentações não oficiais através de blogs, fóruns, livros e artigos. A comunidade do *VRaptor*, por sua vez, é um projeto brasileiro e não possui grande expressão no mercado exterior. Em consulta ao *StackOverflow*, possui somente cerca de 123 ocorrências de postagens que referenciam o *VRaptor*.

Excluído: ¶

No mercado de trabalho o *VRaptor* ainda possui pouca representatividade, porém existe grandes empresas que utilizam o *VRaptor*, como o *Mamute*, *GUJ*, *Wine* e *Locaweb*.

5 Conclusão

Na escolha do *framework*, há relevantes considerações que devem ser analisadas para sua escolha, como técnica, segurança, documentação, licença, popularidade, filosofia, sustentabilidade e recurso no mercado.

Há uma grande variedade de *frameworks* para o desenvolvimento Web em Java, o que torna muito difícil a sua avaliação. O levantamento de critérios auxilia a escolha de um framework para uma determinada situação, pois permite a tabulação das características de cada artefato estudado, facilitando assim a análise.

No artigo foi possível visualizar a arquitetura MVC no qual ambos os *frameworks* trabalham, além do desacoplamento da camada visão e utilização de injeção de dependências em que são semelhantes, e a estrutura particular de cada um, podendo visualizar as vantagens e desvantagens.

Porém, é difícil levantar critérios objetivos na comparação de tecnologias. Critérios como velocidade de desenvolvimento ou linhas de código necessárias para desenvolver uma aplicação não seriam avaliadas adequadamente apenas com a construção de protótipos.

O estabelecimento de critérios, embora subjetivos, deve auxiliar futuras avaliações de frameworks, permitindo que o analista investigue diretamente a classificação do artefato nos critérios pré-estabelecidos.

No caso dos frameworks avaliado, não é possível apontar qual seria o ideal para qualquer situação. No entanto, o estudo feito deve auxiliar na análise numa situação específica, uma vez que os dados estão tabulados e seu embasamento está contido no trabalho.

Referências Bibliográficas

CAVALCANTI, Lucas. *Vraptor*: Desenvolvimento ágil para web com Java. São Paulo: Casa do Código, 2014.

CDI. *O que é CDI?* Disponível em: http://cdi-spec.org/. Acesso em: 27 ago. 2018.

Excluído: BERGMAN, Noel J.; CHOPRA, Abhinav. *Introdução às páginas do JavaServer.* 28 ago. 2001. Disponível em: https://www.ibm.com/developerworks/java/tutorials/j-introjsp/j-introjsp.html. Acesso em: 02 Set. 2018.¶

DAILYRAZOR. *The 10 best Java web frameworks for 2018*. Disponível em: www.dailyrazor.com/blog/best-java-web-frameworks/. Acesso em: 20 nov. 2018.

DOUG, Lea. *Christopher,* Alexander: An Introduction for Object-Oriented Designers. 11 dez. 1993. Disponível em: http://g.oswego.edu/dl/ca/ca/ca.html. Acesso em: 07 set. 2018.

FERREIRA, Alex. *Padrões de projeto*: O que são e por que utiliza-los? 2013. Disponível em: http://www.iotecnologia.com.br/padroes-de-projeto-o-que-sao-porque-usar. Acesso em: 18 jun. 2018.

FRANZINI, Fernando. *O que aprendi com livro Vraptor*: Desenvovlimento Ágil para Web com Java. 11 dez. 2013. Disponível em: https://imasters.com.br/back-end/o-que-aprendi-com-o-livro-vraptor-desenvolvimento-agil-para-web-com-java. Acesso em: 18 jun. 2018.

GITHUB. *Caelum, Vraptor 4*: Repositório de download e instruções de instalação da framework. Disponível em: https://github.com/caelum/vraptor4. Acesso em: 27 ago. 2018.

GUERRA, Eduardo. *Design Patterns com Java*: Projeto orientado a objetos guiado por padrões. São Paulo: Casa do Código, 2012.

HEMRAJANI, Anil. *Agile JAVA Development with Spring, Hibernate and Eclipse*. (s.l.): Paperback, 2006.

JAVAPIPE. *10 Best Java web frameworks to use in 2018 (100% Future-Proof)*. Disponível em: https://javapipe.com/hosting/blog/best-java-web-frameworks. Acesso em: 10 nov. 2018.

KAYAL, Dhrubojyoti. *Pro JAVA spring patterns*: Best Practices and Design Strategies Implementing JAVA EE Patterns with the Sprign Framework. Nova York: Apress, 2008.

LABORDE, Gregory. *Design patterns o que é e como implantar*. 30 set. 2011. Disponível em: http://www.oficinadanet.com.br/artigo/desenvolvimento/design-patterns-o-que-e-e-como-implantar. Acesso em: 14 maio 2013.

LADD, Seth *et al*. *Expert spring MVC and web flows*. Nova Iorque: Apress, 2006.

MAPLE, Simon. *Java Tools and Technologies Landscape Report 2016*. 14 jul. 2016. Disponível em: https://zeroturnaround.com/rebellabs/java-tools-and-technologies-landscape-2016/#web-frameworks. Acesso em: 18 nov. 2018.

MATTEI, Marcelo. *Boas práticas no desenvolvimento de websites*. 19 nov. 2007. Disponível em: https://webinsider.com.br/boas-praticas-no-desenvolvimento-de-websites/. Acesso em: 27 ago. 2018.

REENSKAUG, Trygve. *Models - Views - Controllers*. 10 dec. 1979. Disponível em: https://heim.ifi.uio.no/~trygver/1979/mvc-2/1979-12-MVC.pdf. Acesso em: 07 set. 2018.

SPRING. *Documentação oficial do framework*. Disponível em: https://docs.spring.io/spring/docs/current/spring-framework-reference/index.html#pivotal-software. Acesso em: 27 ago. 2018.

SPRING-PROJECTS. *Repositório de projetos, instruções e downloads referente a framework*. Spring Framework. Disponível em: https://github.com/spring-projects/spring-framework. Acesso em: 27 ago. 2018.

VRAPTOR. *Documentação oficial do framework*. Disponível em: http://www.vraptor.org/pt/. Acesso em: 27 ago. 2018.

WEISSMANN, Henrique Lobo. *Vire o jogo com spring framework*. São Paulo: Casa do Código, 2014.

Excluído: ORACLE, *O que é um servlet?* 2013. Disponível em: https://docs.oracle.com/javaee/6/tutorial/doc/bnafe.html. Acesso em: 02 set. 2018.¶

Excluído: ROTEM-GAL-OZ, Arnon. *Soa patterns*. New York: Manning, 2012.¶
SILVEIRA, Paulo *et al. Introdução à arquitetura e design de software*: Uma visão sobre a plataforma Java. Rio de Janeiro: Elsevier, 2011.¶