Análise Comparativa entre dois Frameworks MVC *WEB* para Plataforma *Java*: *Spring* MVC e *VRaptor*

Cesar Toshiaki Nakase
cezar.nk@gmail.com

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

Curso de Pós-Graduação *Lato Sensu* em Desenvolvimento Orientado a Objetos com Java
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, utilizando os *frameworks* para o desenvolvimento de um sistema *Web,* auxiliando na gestão de controle de pagamentos dos ocupantes de um prédio e no gerenciamento de registros dos moradores e condômino*,* 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.

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

1 Introdução

Cada vez mais a WEB está carregada de empresas que buscam através das facilidades tecnológicas a exposição de seus produtos e serviços, onde muitas vezes conseguem de forma clara e objetiva, alcançar suas metas graças à estas recursividades. É válido dizer que os *frameworks* possuem vários benefícios para o desenvolvimento de software voltado para estas empresas, já que as mesmas investem muito em inovações que buscam a compreensão do usuário, feedback, facilidade de uso, interatividade, entre outros.

Desenvolver sistemas em *Java* para web, antes das criações dos *framework*s, 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 havia uma padronização no desenvolvimento web com *Java*.

Atualmente, tem-se a disposição vários *frameworks* para facilitar a produtividade e a padronização do código. Porém, é preciso analisar se vale à pena utilizar, como escolher e quais as vantagens de se escolher um *framework*.

Baseado nesses questionamentos, por boa prática, busca-se seguir alguns critérios

para a escolha do *framework*. Por exemplo, a sua estrutura, validando se ele poderá atender as necessidades ou não, informações através de comunidades e fóruns a respeito de novas ideias, novas funcionalidades, além da sua qualidade, segurança, capacidade de garantir o funcionamento do sistema, o gerenciamento de riscos e redução de vulnerabilidades e a documentação, pois sendo bem formulada serão mais fáceis o seu entendimento e a sua utilização.

Seguindo as boas práticas citadas anteriormente, este artigo apresenta os *frameworks Spring MVC* e o *VRaptor4*. A relevância da escolha dos frameworks encontra-se da seguinte forma: *Spring MVC* – atualmente, um dos *frameworks* de *Java* para web mais utilizadas, 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, além de por possuir total integração com *Java EE.*

Para análise comparativa, será apresentado um estudo de caso, no qual foi desenvolvido um sistema *Web* para auxiliar na gestão de controle de pagamentos dos ocupantes de um prédio e no gerenciamento de registros 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 *framework*s *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 Arquitetura para serviços WEB

Um dos maiores consórcios de empresas é a *World Wide Web Consortium* (W3C) [W3C, 2018], que busca por definição para um Serviço Web como "um sistema de *softwares* responsáveis por proporcionarem a interação entre duas máquinas diferentes através de uma rede", tendo como interface padrão a WSDL (*Web Services Description Language*). Como padrão a W3C apoia-se em grandes empresas como a Microsoft, IBM, HP, APPLE, entre outras que fornecem subsídios técnicos através de manuais de orientação técnica que ajudam na composição das funções e definições dos termos.

Diante as necessidades emergências tecnológicas, em 2000 a W3C apoiou-se neste grupo com o intuito de desenvolver estudos arquiteturais que permitissem de certa forma a comunicação entre aplicações sistemas baseados em plataformas diferentes, e foi a partir desta, que surgiram padrões de definição como o DCOM (*Distributed Component Object Model*) (DCOM, 2018), CORBA (*Common Object Request Broker Architecture*) (CORBA, 2018), e RMI (*Remote Method Invocation*) (RMI, 2018).

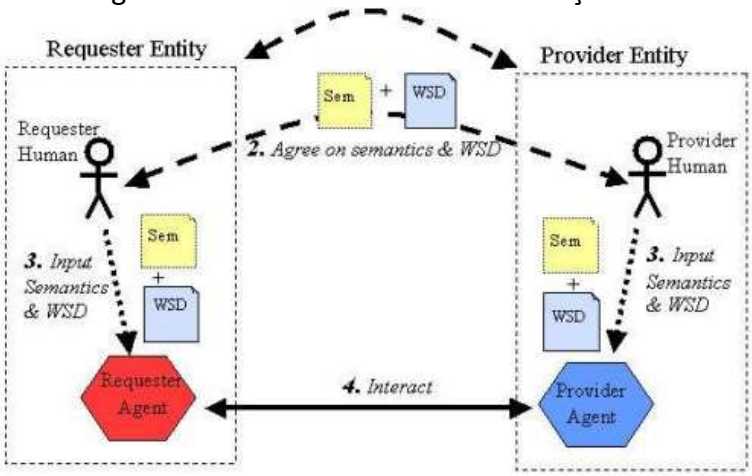
Um dos fatores das novas abordagens se deu devido as dificuldades quando houve a necessidade da empregabilidade das tecnologias voltadas a grande rede de computadores, ou seja, rodando sobre a WEB, estas tecnologias por encontrarem sistemas independentes, deixou a desejar, precisando de reformulação e adaptação para as novas tecnologias. De

acordo com estas necessidades de adaptação, foi proposta pela W3C uma arquitetura computacional cuja finalidade fosse baseada justamente nos serviços WEB tornando possível tais comunicações mesmo que em plataformas distintas.

Diante outras concepções, GOMES (2010) menciona que devido a evolução das redes de computadores surgiu as aplicações distribuídas, porém no início estas aplicações eram centralizadas em um único servidos, mais tarde, com a difusão da internet, estas ações tornaram-se impróprias, pois com o surgimento de aplicações independentes, havia a necessidade da criação de um novo serviço de comunicação que fosse capaz de unir estas plataformas, e justamente criou-se os protocolos de comunicação baseados em XML (*Extensive Markup Language*) dando origem assim aos *Web Services*. O XML é o formato de mensagem adotado pelo W3C para troca de informações entre aplicações distribuídas através do protocolo HTTP. (GOMES, 2010).

Deste modo a arquitetura de Serviços Web baseada nas definições da W3C, tem como base o WSA (*Web Service Architecture*), (WSA, 2004), fornece subsídios técnicos onde os "esqueletos" desta arquitetura estão imersos em um contexto inter-relacionai, possibilitando assim a visualização da intenção da estrutura como uma solução para as interoperabilidades entre os meios de comunicação das plataformas distintas encontradas na Web.

Figura 1 - Processo Geral de um Serviço Web



Fonte: (WSA, 2004) - Adaptado

2.2 Padrões de Projeto

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 Prado (2018) faz referência ao arquiteto Christopher Alexander que define as seguintes características: "Encapsulamento, Generalidade, Equilíbrio, Abstração, Abertura e Combinatoriedade".

Padrões de projetos não são específicos a uma empresa ou tecnologia, são soluções reutilizáveis aplicadas pela comunidade de desenvolvimento como um todo. Sabe-se que padrões de projeto nem sempre são aplicados, principalmente se o foco do problema é relativamente simples, porém vale ressaltar que a medida que este problema cresça futuramente, vale a pena investir em um padrão, pois assim ele lhe poupará de horas e horas de trabalho em códigos fonte e refatoração (PRADO, 2018).

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, pela 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 multicamadas, 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 reusabilidade 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.

2.3 *Spring* MVC

Spring é organizado de forma modular, permitindo que se preocupe apenas com os módulos necessários. Atualmente estão disponíveis 21 módulos, fornecendo tudo o que um desenvolvedor pode precisar para o uso no desenvolvimento de um projeto. Weissmann (2014, p. 27) ressalta a importância dos módulos "É importante termos esta visão panorâmica dos módulos que compõem o framework para que fique claro o quão abrangente ele é: basicamente o *Spring* abrange todas as necessidades de uma aplicação corporativa".

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 decidir quais dependências serão injetadas durante o tempo de execução. Uma das características mais conhecidas quando se programa com *Spring*.

Descrito por Hemrajani (2006, p. 171) 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 se 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 se possa ter.

2.4 *VRaptor*

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 (*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*", (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 *Action Based,* 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”.

Citando as camadas para uma melhor explanação sobre o Vraptor e suas funcionalidades, de acordo com Cavalcanti (2014, p. 5) o *framework Vraptor* disponibiliza grandes flexibilidades voltadas a tecnologias da camada de Visão. Isto ocorre devido ao baixo acoplamento através dos acopladores que proporcionam através do modelo MVC baseado em ações, que se utiliza de *templates* gerando páginas totalmente dinâmicas (ex: JSP, *Velocty*, *Freemarker*), porém o autor deixa claro que é preciso que os desenvolvedores criem seus próprios componentes baseados nestes conceitos, utilizando-se de bibliotecas externas como a *JQuery UI*, *Bootstrap*, ExtJS, AngularJS, entre outras.

Já na camada *Controller* do *Vraptor*, é representado segundo Cavalcanti (2014, p. 6), pela classe Java que é totalmente responsável pelo recebimento das requisições HTTP. Responsável por controlar o modelo MVC que é baseado em ações, ela fica responsável pela intermediação entre as camadas Visão e Modelo, dando definições sobre a visão que será utilizada em uma resposta da requisição HTTP.

A camada REST é responsável por fazer integrações entre as aplicações MVC. Sabe-se que mediante as exigências de mercado e as tecnologias dispostas para tais atualizações, é praticamente impossível ter uma aplicação que não faça interação com outras aplicações WEB, e para isto a integração destes sistemas distribuídos sobre a camada REST vem ganhando destaque sobre o protocolo HTTP nos últimos anos (SILVEIRA, et. al, 2012).

Já Saudate (2013, p. 10) menciona que existem boas práticas baseadas em REST que implicam diretamente em usos adequados relacionando-se com o método HTTP como por exemplo (*GET*, *POST, PUT, DELETE*) além de mencionar também a adequação baseada nas URLs, códigos de status padronizados representando o sucesso ou a falha do código.

Na camada de Injeção de Dependências Cavalcanti (2014, p. 7) menciona que o *Vraptor* possui um *framework* responsável por realizar o controle da injeção de dependências dos seus componentes dimensionando assim o ciclo de vida dos mesmos. Após a versão *Vraptor* 4 (que faz a utilização das funcionalidades do CDI), faz com que o *framework* seja totalmente extensível, possibilitando assim uma total integração com todos os recursos nativos do servidor de aplicação.

Caelum (2019) menciona que na camada de conversores as requisições do servidor passar por eles, que ficam responsáveis em transformar os conjuntos de *Strings* que são recebidas através dos tipos de objetos, permitindo assim o perfeito armazenamento. O *Vraptor* já possui através das suas bibliotecas um conjunto de conversores implementados para as requisições mais simplificadas do Java como números, caracteres (*Strings*), datas, *boolean*, entre outros. Já na camada Validadores, destaca a especificação das validações baseadas em *Bean Validation*. Destaca-se também que o Vraptor possui uma API que possibilita as validações de forma simples, determinando as ações de quando uma restrição de validação é violada, retornando assim mensagens de erro.

Outra camada importante são os Interceptadores, funcionando como um “análogo” ao clássico sistema de filtro da *Servlet*, porém no *Vraptor* já integrado no sistema de dependências. Baseando-se nos controles de acesso das aplicações, controle de transação, logs, funcionalidades, onde geralmente este sistema de controle é acionado, a implementação desta camada facilita muito os processos voltados a manutenção, podendo o desenvolvedor restringir as funções de *call-back* ante e depois das funções de execução de cada sistema de *Controller.* Na última camada, Plugin, ao atingir um certo nível de maturidade e após conquistar destaque dentro das comunidades de desenvolvimento, vários desenvolvedores começaram a criar *plugins* baseados na arquitetura de *Vraptor*. Estes *plugins*, facilitam os

processos de criação dos componentes que são totalmente reusáveis prometendo resolver problemas comuns e decorrentes que podem ser totalmente aplicados durante qualquer instanciação dos objetos da aplicação (CAELUM, 2019).

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

3 Materiais e Métodos

Para avaliação e levantamento de critérios de comparação entre os *frameworks* foi desenvolvido um sistema 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* referentes ao estudo trabalham na camada *Controller*, para as camadas de *Models* e *Views* foram utilizadas 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 utilizadas 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

Nesta subseção 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 desenvolvida, 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 quadro é demostrado a implementação de uma tela de cadastro e consulta do sistema para criação e edição do perfil de um usuário, especificando detalhadamente cada particularidade de cada *framework*.

Quadro 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: Autoria Própria

No Quadro 1 apresenta o início da criação da classe responsável pelas ações que serão executadas, nota-se que ambos 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.

Quadro 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","Perfil cadastrado com sucesso!");
   result.redirectTo(this).lista();
}
@Delete("/perfil/remove")
public void remove(Perfil perfil) {
   perfilDao.remove(perfil);
   result.include("mensagem","Perfil 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(@Valid @ModelAttribute("Perfil") Perfil perfil,BindingResult result) {
   if(result.hasErrors()) {
     return "perfil/formulario";
   }
   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: Autoria Própria

O quadro 2 apresenta 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 dizer de forma explícita, através do conceito de anotação, qual o tipo de requisição que será executado.

Foram utilizadas as anotações *@Post* para o método adiciona, *@Delete* para o método remove, *@Put* para o método edita e o *@Get* para o método lista. Vale ressaltar que é importante observar junto as anotações que foram declaradas as URLs que estarão acessíveis tanto para camada *View* quanto para o navegador.

Já ao *Spring MVC,* o mapeamento da URL também é realizado através do conceito de anotação, especificamente o *@RequestMapping*, onde nota-se 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,* denota-se qual *View* estará sendo referenciada e declara-se a tipologia do método em *String* e seu *return* (retorno) o caminho para onde se quer 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 é possível dizer qual *View* estará sendo referenciada, além de ter outros recursos disponíveis.

Quadro 3 – Página ISP 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="${url}" method="POST" modelAttribute="Perfil">
<div class="row">
  <div class="col-md-7">
    <div class="form-group">
      <label for="nome">Nome:</label> <input type="text"
        placeholder="Digite o nome do usuario *" name="nome"
        class="form-control" value="${perfil.nome}" />
    </div>
  </div>
```

Fonte: Autoria Própria

Na tela de cadastro, Quadro 3, ambos construídos na estrutura de *forms,* pode-se notar a diferença na declaração da ação que será executada, ao *form* do *Spring MVC* (como é notado na linha), *assim se diz* a *URL ('/perfil')* disponibilizada pelo *Controller*, já ao *form* do *Vraptor se* pode referenciar a classe e o método que executará a ação, sendo notado através da instrução: (${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 se pode verificar 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).

Quadro 4 – Método de *Controller* responsável por listar todos os perfis cadastrados

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

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

Fonte: Autoria Própria

No Quadro 4, nota-se como o *Spring MVC* e *Vraptor* atua 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), é percebido 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*),* se tem o mesmo procedimento, apenas utilizando a nomenclatura específica do *framework*, no qual se utiliza o serviço *addAtribute* do método *modelMap* para enviar o objeto 'perfil' a camada *View.*

Abaixo, no Quadro 5 linha 2, o código da página JSP recebe o objeto 'perfil' e carrega as informações através de um *loop* em *JSTL* as informações na página.

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

```
/** Página JSP **/
<tbody>
<c:forEach items="${perfil}" var="perfil">
 <tr>
    <td>${perfil.id}</td>
    <td>${perfil.nome}</td>
    <td>${perfil.cpf}</td>
    <td>${perfil.data_nascimento}</td>
    <td>${perfil.estado_civil}</td>
    <td>${perfil.andar_ocupado}</td>
 </tr>
</c:forEach>
</tbody>
```

Fonte: Autoria Própria

3.5 Validação das Informações

A validação é a certificação de que o sistema atenda às 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*.

Quadro 6 – Validação de formulário do método adicional perfil no *Controller*

```
/** VRaptor **/
import javax.validation.Valid;
@Controller
public class PerfilController {
        private PerfilDao perfilDao;
        private Validator validator;
 private Result result;
        @Inject
        public PerfilController(PerfilDao perfilDao,Validator validator, Result result) {
                this.perfilDao = perfilDao;
                this.validator = validator;
  this.result = result;
        }
```

```
 @Post("perfil/adiciona")
        public void adiciona(@Valid Perfil perfil) {
                validator.onErrorForwardTo(this).form();
                perfilDao.adiciona(perfil);
                result.redirectTo(this).lista();
```

```
/** Spring MVC **/
import javax.validation.Valid;
@Controller
@RequestMapping("/perfil/**")
public class PerfilController {
        @Autowired
        private PerfilDAO perfilDao;
 @RequestMapping(value = "/perfil", method = RequestMethod.POST)
        public String adiciona(@Valid @ModelAttribute("Perfil") Perfil perfil,BindingResult
result) {
                if(result.hasErrors()) {
             return "perfil/formulario";
            }
                perfilDao.persist(perfil);
                return "redirect:/perfil";
 }
```

Fonte: Autoria Própria

No Quadro 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, apresenta-se a tela onde um usuário poderá 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, sendo que o retorno da lista ocorre através da utilização do arquivo *Json (Javascript Object Notation),* formato de transmissão de informações no formato texto, atualmente muito usado em *web services* e aplicações Ajax. Assim se pode avaliar como os *frameworks* em estudo trabalham com esse tipo de informação.

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

```
/** Models **/
import java.io.Serializable;
@Entity
public class Pagamento implements Serializable {
        @Id
        @GeneratedValue(strategy=GenerationType.IDENTITY)
        private int id;
        private BigDecimal valor;
        private BigDecimal multa;
```

```
}
/** VRaptor **/
@Get("pagamento/listaPagamento")
public void listaPagamento() {
  List<Object[]> lista = pagamentoDao.listaJson();
    result.use(Results.json()).from(lista).serialize();
}
/** Spring MVC **/
import org.springframework.web.bind.annotation.ResponseBody;
@RequestMapping(value="/pagamento/listajson", method = RequestMethod.GET)
@ResponseBody
public List<Object[]> listJson(ModelMap modelMap) {
  return pagamentoDao.listaJson();
}
```

Fonte: Autoria Própria

O Quadro 7 mostra que o *VRaptor* e o *Spring MVC* possam realizar operações com arquivos *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 é percebido 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, basta fazer a anotação com *@ResponseBody* ao método, que transforma o retorno em objeto *json,* (*Spring MVC* linha 4).

3.7 Tela de *Login* e utilização de Interceptor

É comum em sistemas possuir a regra de negócio onde apenas os usuários autorizados que estejam logados possam fazer qualquer consulta ou modificação no sistema.

No 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, e se ele está, 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* trabalha 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 (Quadro 8).

Quadro 8 – Método autentica() da classe *LoginController*

```
/** VRaptor - LoginController **/
  @Open
public void autentica(String login, String senha){
   Usuario usuario = usuarioDao.busca(login,senha);
   System.out.println(usuario);
```

```
    if(usuario != null){
        usuarioLogado.fazLogin(usuario);
        System.out.println(usuario.getNome());
        result.redirectTo(IndexController.class).index();
    } else {
        validator.add(new SimpleMessage("Login_invalido","Login ou senha incorretos"));
        validator.onErrorRedirectTo(this).form();
        System.out.println("não encontrado");
    }
}
```

Fonte: Autoria Própria

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.

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

```
/** VRaptor - UsuarioLogado **/
import java.io.Serializable;
@Named
@SessionScoped
public class UsuarioLogado implements Serializable{
        private Usuario usuario;
  public void fazLogin(Usuario usuario){
                this.usuario = usuario;
        }
  public boolean isLogado(){
                return this.usuario != null;
        }
}
```

Fonte: Autoria Própria

Após obter o usuário da sessão, é preciso determinar quais métodos serão interceptados. Então, foi criado a classe AutorizaçãoInterceptor, para permitir a execução da lógica do *Controller* apenas caso o usuário esteja logado.

E para interceptar uma requisição denota-se 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) no *interceptor* como e percebido no Quadro 10.

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

```
/** VRaptor - AutorizaçãoInterceptor **/
public AutorizacaoInterceptor() {}
@Accepts
```

```
public boolean accept(){
  return !method.containsAnnotation(Open.class);
}
@AroundCall
public void intercept(SimpleInterceptorStack stack){
  if(usuarioLogado.isLogado()){
    stack.next();
  } else {
    result.redirectTo(LoginController.class).form();
  }
}
```

Fonte: Autoria Própria

É percebido no Quadro 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.

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

```
/** Spring MVC - LoginController **/
@RequestMapping("/login/efetuaLogin")
public String efetuaLogin(Usuario usuario, HttpSession session) {
if(usuarioDao.busca(usuario)) {
   // usuario existe, guardaremos ele na session
   session.setAttribute("usuarioLogado", usuario);
   session.setAttribute("usuarioNome",usuario.getNome_guerra());
   return "redirect:/index";
  }
  // ele errou a senha, voltou para o formulario
  return "redirect:/";
}
@RequestMapping("/login/loginEfetua")
public String autentica(Usuario usuario,HttpSession session,Model model){
  Usuario us = usuarioDao.procura(usuario);
  if(us != null){
    session.setAttribute("usuarioLogado", us);
    return "redirect:/index";
  } else {
    System.out.println("nÃ£o encontrado");
    model.addAttribute("mensagem","Login ou senha invalido!");
    return "login/form";
  }
}
```

Fonte: Autoria Própria

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

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

Nesse interceptor, verifica-se 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).

Quadro 12 – Classe AutorizadoInterceptor, responsável por gerenciar as requisições na sessão

```
/** Spring MVC - AutorizadorInterceptor **/
import org.springframework.web.servlet.handler.HandlerInterceptorAdapter;
public class AutorizadorInterceptor extends HandlerInterceptorAdapter {
        @Override
        public boolean preHandle(HttpServletRequest request, HttpServletResponse
response, Object controller)
        throws Exception {
                String uri = request.getRequestURI();
                if(uri.endsWith("/") || uri.endsWith("loginEfetua") ||
uri.contains("resources")) {
                        return true;
                }
                if(request.getSession().getAttribute("usuarioLogado")!=null) {
                        return true;
                } else {
                        response.sendRedirect("redirect:/");
                        return false;
                }
        }
}
```

Fonte: Autoria Própria

4. Resultados

Através do desenvolvimento do sistema, foi possível perceber vantagens e desvantagens utilizando o *Spring MVC* e o *VRaptor*. Vale ressaltar que o *MVC Spring* é mais conhecido no mercado, porém necessita de um pouco mais de configuração, além do mesmo expor as classes que precisam ser carregadas. Já comparando com o *VRaptor*, o mesmo auxilia nas tarefas que foram adicionadas ao sistema, trabalhando diretamente com os componentes que foram disponibilizados através das instâncias utilizadas.

A lista do que cada um pode contribuir para a aplicação é bem extensa, pois cada um dos frameworks possuem certas "particularidades" que precisam ser levadas em consideração, por exemplo, enquanto o *VRaptor* (*framework* brasileiro) trabalha de forma mais clara as validações, integrações de sistema, *uploads* e *downloads* de arquivos, além de proporcionar conversores e testes, o *Spring* é voltado para um desenvolvimento mais robusto, composto por vários módulos como segurança, *templates*, internacionalização, entre outros.

Abaixo, se descreve os pontos positivos e negativos relacionados ao sistema baseados nas duas plataformas, dimensionando os processos, destacando os planos de negócio, porém a escolha entre um deles acaba sendo muito particular, pois os dois trazem resultados satisfatórios.

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.

Possuindo uma estrutura completa para a criação de aplicativos da web com enorme estabilidade, amplo alcance e sua simplicidade de configuração, por ser notada pela alta usabilidade entre os programadores para WEB. Com pouca intrusão do framework, a diminuição do acoplamento ao utilizá-lo facilita a modularização dos projetos, a 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.

Os 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, a torna bastante complexa, ela ainda não fornece nenhuma estrutura rica para construir boas Interfaces.

Relacionando-a com curva de aprendizado íngreme, quanto mais se trabalha, mas se percebe que é muito fácil se adaptar e melhorar suas bibliotecas, porém se deseja incluir outros módulos do Spring, pode exigir mais tempo de aprendizagem, pois requer mais tempo para adquiri conhecimento e 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 de desenvolvimento 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*

Com uma estrutura fácil e funcional para se 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 Java *Script*.

Em questão da documentação o *VRaptor* possui oficialmente em seu site, no qual é possível verificar instruções de uso, exemplos de aplicações e tutoriais, além de possuir a versão traduzida em português como um diferencial, porém poderia ser ainda melhor se apresentasse mais explicações 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*. 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.

Através do apontamento dos frameworks estudados neste objeto de estudo para o caso deste trabalho o *VRaptor* se mostra mais eficiente, pois trata de maneira mais clara várias instancias empregadas no sistema. Enquanto o MVC Spring é voltado para aplicações maiores, possuindo um número elevado de módulos, etc. No caso deste sistema, foi possível perceber

a diferença entre as duas plataformas, porém vale ressaltar que as duas são ótimas, e deve-se considerar as particularidades de cada uma delas, e escolher como e onde serão aplicadas.

Referências Bibliográficas

CAELUM. **Desenvolvimento Ágil para a Web 2.3 com *Vraptor*, *Hibernate* e AJAX.** Disponível em: https://www.caelum.com.br/download/caelum-java-web-vraptor-hibernate-ajax-fj28.pdf. Acesso em: 16 fev. 2019

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.

CORBA**. *The official CORBA standard from the OMG group***. 2009. Disponível em http://www.omg.org/spec/CORBA/About-CORBA/. Acesso em: 07 Fev. 2019

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

DCOM. ***Distributed Component Object Model (DCOM) Remote Protocol Specification***. 2009. Disponível em http://msdn.microsoft.com/pt-br/library/cc201989(enus).aspx. Acesso em: 07 Fev. 2019

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***: **Desenvolvimento 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 do *framework***. Disponível em: https://github.com/caelum/vraptor4. Acesso em: 27 ago. 2018.

GOMES, D. A. ***Web Services SOAP em Java***: São Paulo, Novatec, 2010.

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: 03 Fev. 2019

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.

PRADO, Kelvin Salton do. **Padrões de Projeto em Python.** Disponível em: https://medium.com/@kelvin_sp/padr%C3%B5es-de-projeto-em-python-4c3a1be9dd50. Acesso em: 03 fev. 2019.

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.

RMI**.** ***Remote Method Invocation Home***. 2009. Disponível em http://www.oracle.com/technetwork/java/javase/tech/index-jsp-136424.html. Acesso em: 07 Fev. 2019

SAUDATE, A., **REST: Construa API´s inteligentes de maneira simples**. Ed. Casa do Código, São Paulo, 2013.

SILVEIRA, P., SILVEIRA, G., LOPES, S., MOREIRA, G., STEPPAT, N., KUNG, E., **Introdução à Arquitetura e *Design* de *Software*: Uma Visão Sobre a Plataforma Java**. Rio de Janeiro: Ed. Elsevier, 2012.

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.

W3C. ***World Wide Web Consortium***, 2018. Disponível em: http://www.w3.org/. Acesso em: 07 Fev. 2019

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

WSA. ***Web Services Architecture***, 2004. Disponível em: https://www.w3.org/TR/ws-arch/wsa.pdf, Acesso em: 07 Fev. 2019