

LIGA DE ENSINO DO RIO GRANDE DO NORTE
CENTRO UNIVERSITÁRIO DO RIO GRANDE DO NORTE
CURSO DE BACHARELADO DE SISTEMAS DE INFORMAÇÃO

RODRIGO EMERENCIANO DE OLIVEIRA

DOCKERWEB: SOLUÇÃO DE GERENCIAMENTO DE CONTÊINERS DOCKER

NATAL/RN

2016

RODRIGO EMERENCIANO DE OLIVEIRA

DOCKERWEB: SOLUÇÃO DE GERENCIAMENTO DE CONTÊINERS DOCKER

Trabalho de Conclusão de Curso
apresentado ao Centro Universitário do
Rio Grande do Norte (UNI-RN) como
requisito final para obtenção do título de
Bacharel em Sistemas de Informação.

Orientador: Prof. Romulo Fagundes
Cantanhede

NATAL/RN

2016

Catálogo na Publicação – Biblioteca do UNI-RN
Setor de Processos Técnicos

Oliveira, Rodrigo Emerenciano de.

Dockerweb: solução de gerenciamento de contêineres docker. / Rodrigo Emerenciano de Oliveira. – Natal, 2016. 1
55 f.

Orientador: Prof. Rômulo Fagundes Cantanhede.

Monografia (Bacharelado em Sistemas de Informação) – Centro
Universitário do Rio Grande do Norte.

1. Virtualização – Monografia. 2. Paravirtualização – Monografia. 3.
Framework Web - Monografia. I. Cantanhede, Rômulo Fagundes. II. Título.

RN/UNI-RN/BC

CDU 004

RODRIGO EMERENCIANO DE OLIVEIRA

DOCKERWEB: SOLUÇÃO DE GERENCIAMENTO DE CONTÊINERS DOCKER

Trabalho de Conclusão de Curso apresentado ao
Centro Universitário do Rio Grande do Norte (UNI-RN)
como requisito final para obtenção do título de
Bacharel em Sistemas de Informação.

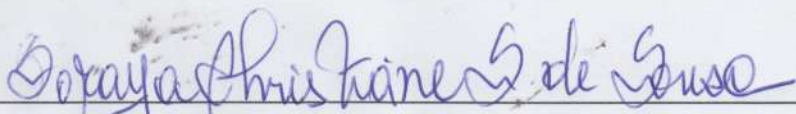
Aprovado em: ____ / ____ / ____

BANCA EXAMINADORA



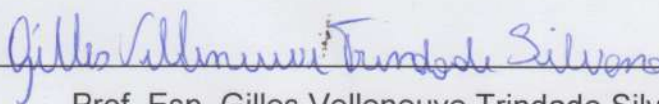
Prof. Msc. Romulo Fagundes Cantanhede

Orientador



Prof.ª Esp. Soraya Christiane Silva de Souza

Membro



Prof. Esp. Gilles Velleneuve Trindade Silvano

Membro

AGRADECIMENTOS

Agradeço aos meus pais por sempre acreditarem e incentivarem no meu processo de profissionalização, fornecendo todas as condições necessárias para que fosse possível alcançar mais essa conquista.

Merecem também meus agradecimentos os prezados mestres, que permitiram a obtenção de conhecimentos antes não atingidos, nos disponibilizando do mesmo modo orientações pessoais e profissionais, no contexto do mercado de trabalho voltado para a área de Tecnologia da Informação.

Por fim agradeço a minha companheira, por me encorajar constantemente, na elaboração deste trabalho, com muita paciência, e compreendendo a minha maior parcela de tempo destinada ao mesmo.

RESUMO

A fim de demonstrar que é possível a adoção de novas técnicas para implantação da solução *DockerWeb* em sua infraestrutura de TI, com a proposta de um melhor gerenciamento, através do uso do sistema em desenvolvimento totalmente web, diminuindo o tempo de acesso em cada *container* para a aplicação de instruções principais e recebimento de dados importantes, evitando o tempo gasto no detalhamento de *containers* de cada servidor *Docker* instalado, oferecendo uma análise de *containers* atuantes em produção ou disponíveis para execução, permitindo a possibilidade de acesso em qualquer navegador, devido a ser um sistema web responsivo disponibilizando a administração do mesmo por dispositivos móveis.

O sistema *DockerWeb* possui um mecanismo de comunicação próprio, realizado por meio da linguagem de programação *Groovy* e dos frameworks *Grails* e *Vert.X* o que permitem a interação dos servidores com tecnologia *Docker* instalada portando o agente *Docker CLI*, e o sistema centralizador o *Docker GUI*.

Palavras-chave: Virtualização, Paravirtualização, *Framework Web*.

ABSTRACT

In order to demonstrate that it is possible to adopt new techniques for deploying the DockerWeb solution in its IT infrastructure, with a proposal for better management, through the use of the fully web development system, reducing the access time in each container to The application of main instructions and the receipt of important data, avoiding the time spent in detailing the containers of each installed Docker server, offering an analysis of containers that are active in production or available for execution, allowing the possibility of access in any browser, due to being A responsive web-based system making it available for mobile devices.

The DockerWeb system has its own communication mechanism, implemented through the Groovy programming language and the Grails and Vert.X frameworks, which allow the interaction of servers with Docker technology installed with the Docker CLI agent, and centralizing the Docker GUI .

Keywords: Virtualization, Paravirtualization, Web Framework.

SUMÁRIO

1.INTRODUÇÃO	9
1.1. OBJETIVOS	10
1.1.2 GERAL	10
1.1.3 ESPECÍFICO	10
2.REFERENCIAL TEÓRICO	11
2.1. História da Virtualização	11
2.2.Tecnologias Utilizadas	14
2.2.1. Virtualização	14
2.2.2. Paravirtualização	19
2.2.3. Docker	20
2.3. Ambiente de Desenvolvimento de Software	23
2.3.1. Java	23
2.3.2. Groovy	24
2.3.3. Vert.X	25
2.3.4. Grails	26
2.4. Banco de Dados	26
2.4.1. HSQLDB	27
2.5. Serviços Web	27
2.5.1. REST	27
3.METODOLOGIA	28
4. ESTUDO DE CASO	29
4.1 ANTES DO SISTEMA DOCKERWEB	29
4.2 DOCKERWEB	33
4.2.1 Visão Geral do Sistema	33

4.2.2 DOCKERWEB CLI	34
4.2.3 DOCKERWEB GUI.....	43
5. CONSIDERAÇÕES FINAIS.....	53
REFERÊNCIAS	54

1.1. OBJETIVO

A proposta inicial do sistema possui seu objetivo no desenvolvimento, implantação e utilização do mesmo, por administradores de rede ou usuários encarregados para gerenciamento da infraestrutura de TI que possuam a tecnologia *Docker* instalada, e outros que tem o desejo de implantar a solução tecnológica.

1.1.2 GERAL

No princípio foi apontado a automatização do processo comum para o acesso remoto ao servidor instalado com *docker*, com o intuito da realização de comandos básicos, surgindo a ideia da criação de um sistema de informação *web*, centralizando todos os servidores que portam o *Docker*, e seus respectivos *docker containers*, apresentando informações essenciais de seu funcionamento.

Dessa forma, o sistema opera de forma a diminuir o trabalho que o usuário *docker* possui atualmente, tanto em digitar cada comando em terminal, quanto no auxílio detalhado de dados indispensáveis para o administrador de infraestrutura de TI, com praticidade para o clone de imagens disponibilizadas no repositório oficial do Docker através de integração com o sistema.

1.1.3 ESPECÍFICO

- Desenvolver e implantar uma aplicação cliente em *Groovy*, executada em servidores físicos *Docker*, que permita o recebimento e envio de informações, utilizando a arquitetura REST, por meio dos protocolo HTTP para ser acionado comandos e gerar seu retorno em formato *JSON*, tornando-se possível trabalhar com tais dados enviados, em uma plataforma central, por meio de seu endereço de rede.
- Desenvolver e implantar uma aplicação *web* centralizadora em *Groovy* e *Grails*, disparando comandos pelo protocolo HTTP e manipulando os dados recebidos em formato *JSON*, oferecendo uma opção de acesso a informações dos *containers*, cadastro de servidores, e controle de atividade do *container*.

- Executar a emprego de técnicas que viabilizam a comunicação entre a aplicação centralizadora e as demais clientes.

2. REFERENCIAL TEÓRICO

2.1 História da Virtualização

Segundo Pollon (2008), a formulação da definição de virtualização começou a ser discutida já na década de 1950, apesar de parecer um tema apresentado a área de Tecnologia como algo recente. No período em questão, a utilização de *mainframes*, não oferecia a funcionalidade de multitarefas, dessa maneira não possibilitava a execução de programas de forma simultânea. Tendo em vista que os *mainframes* possuíam poucos recursos e a compra de componentes de *hardware* comparado com os dias atuais necessitava de um alto grau de investimento para aquisição. No início da década de 1960 a IBM lançou um novo conceito da *Time Sharing* que consiste em executar alternadamente os processos de forma que o usuário, tivesse a percepção de que os eles fossem executados de forma simultânea.

O termo *Time Sharing* originou-se com base em um estudo realizado pelo professor Christopher Strachey da Universidade de Oxford, no ano de 1959, com o seguinte título *Time Sharing in Large Fast Computers*, que foi publicado em Nova Iorque, na Conferência Internacional. Trabalho este que se tornou útil para que outros cientistas pudessem prosseguir mais além do que era proposto a eles no momento. O centro de computação do MIT, 1961, criou um dos primeiros sistemas de tempo compartilhado, *Compatible Time-Sharing System (CTSS)*, que mostrou-se muito importante para a exibição e entendimento de sistemas de tempo compartilhado, além de contribuir para a geração de bases seguintes.

O *Atlas Computer*, um dos primeiros supercomputadores concebidos em 1962, pela Universidade de Manchester, atuava com os termos de *time sharing*, *multiprogramming*, *virtual memory* e controle compartilhado de periféricos, assim

como, apresentava a funcionalidade de administrar o tempo de processamento, onde poderia ser comparado a um ordenador de tarefas.(POLLON, 2008).

Em 1964 foi iniciado o desenvolvimento do CP-40 e do *CMS - Cambridge Monitor System*, conduzido por Robert Creasy, no Centro Científico de *Cambridge*, da IBM. Em que foi o sistema operacional primitivo a ter verdadeiramente o conceito de *full virtualization*, e oferecia a possibilidade de simular ao mesmo tempo até 14 máquinas emuladas que posteriormente, foram nomeadas de máquinas virtuais.

Através do Centro de investigação Thomas J. Watson da IBM, no ano de 1965, foram aprimorados o conhecimento sobre virtualização, após a conclusão do sistema em fase de experiência com nome de M44/44X, usando como alicerce o IBM 7044, desta forma herda o M44, pois apresentava uma forma de emular diversos IBM 7044 em modo de *virtual machines*, além disso recebeu o nome também de 44X. A prática que o M44 propiciava para divisão lógica dos *mainframes*, com o uso, tanto de *software* como de *hardware*, possibilitava que executassem variadas aplicações e processos simultaneamente, como se correspondesse a inúmeros *mainframes*. Com a aplicação dessa funcionalidade oferecida pelo sistema experimental M44, pode-se verificar que o *mainframe* era capaz de executar diversas tarefas ao mesmo tempo. O M44/44X fazia o uso de concepções de paginação, memória virtual, e multiprogramação. A emulação de *hardware* da máquina hospedeira, não era realizada de forma completa, sendo assim considerado como paravirtualização.

A preocupação com o maior uso dos recursos dos *mainframes*, tendo como exemplo principal a memória, e o crescente sucesso da IBM em acreditar fortemente neste seguimento, foram precursores da inserção de testes de viabilidade técnica, para uma proposta de um novo *mainframe*. Esses testes eram realizados por intermédio de máquinas virtuais, já acessíveis no mercado.

Dificuldades foram surgindo, à medida em que a ordem da tecnologia de virtualização tinha seu desenvolvimento sazonado. Teve como fracasso o projeto *System/360 Modelo 67* com engenharia de tempo compartilhado, que efetuava a memória e *hardware* virtuais, porém devido a implicações com o sistema operacional OS/360, foi cancelado em 1971.

Em 1966, a IBM deu início a um projeto simultâneo intitulado de CP-67: onde foi feita uma modificação do CP-40 para que fosse possível a execução no

System/360 modelo 67. Esse tipo de mudança do CP-40 foi a primeira elaboração de uma estrutura de máquina virtual, onde foi extensivamente distribuída. (POLLON, 2008).

O código fonte da primeira versão de CP/CMS, foi disponibilizado pela IBM em 1968, com o código disponível a todos, a *National CSS* lança o VP/CSS, embasado no CP/CMS com o objetivo de uma maior eficiência, para utilizar em seus planejamentos rentáveis.

A IBM deu partida em 1970 a elaboração do CP-370/CMS, que era constituído por uma reformulação do CP-67/CMS no seguimento da série *System/370(S/370)*, porém só em 1972 foi criado o sistema operacional VM/370, o primogênito da família de máquina virtual VM/CMS. Desenvolvido com base no CP-370/CMS e indicado para o *System/370*, com *hardware* de memória virtual. O *hypervisor*, lançado na década de 1970, com o intuito de redução de custos, caracteriza-se por ser uma camada de *software*, localizada entre o *hardware* e o sistema operacional, e é por meio dele que é possível a realização de todo o gerenciamento de recursos físicos, tais como: memória, processador, interface de rede, disco, etc. Foi inicialmente aplicado em *mainframes* para servir outros setores, contendo múltiplos sistemas operacionais. Como principal funcionalidade tinha a possibilidade de rodar uma VM dentro de outra VM, assim como, no mesmo período a *National CSS*, implementa uma melhora na performance do VP/CSS indicado para o *System/370*, através da utilização de paravirtualização, onde era possível através dessa técnica solicitações diretas no *hypervisor*, porém sem o uso de virtualização, e em contrapartida não emular operações de baixo nível.

Com o surgimento dos computadores pessoais, tais como por exemplo: *Apple II*, *Atari 400/800*, *Commodore VIC-20*, *IBM PC*, *ZX Spectrum*, a virtualização teve uma redução na sua procura comparado ao estado inicial, durante o período de 1973 e 1987. A IBM ainda sim não desanimou, e continuou investindo na família de VM, mesmo com a diminuição do interesse na produção de *mainframes* mais aprimorados (POLLON, 2008).

2.2 Tecnologias Utilizadas

Com o objetivo estratégico e o seguimento voltado para a implantação de tecnologia sustentável, a análise dos benefícios e malefícios que serão conquistados, e a escolha da tecnologia com o objetivo de diminuir a quantidade de gastos financeiros com *hardware*, suporte, e refrigeração, bem como seu retorno em termos de desempenho ao negócio, tornam-se pontos marcantes dessa observação.

A adesão de medidas sustentáveis pela TI, visam o preenchimento de requisitos através do planejamento e aplicação destes métodos, que tem como principal preocupação a orientação da sustentabilidade empresarial. A redução de custos e a preservação de recursos organizacionais, é um dos motivos que torna interessante a escolha do “movimento verde”, além da importância pela responsabilidade social corporativa (SALLES; ALVES; DOLCI; LERCH, 2013 apud CHEN et al., 2008; SCHIMIDT et al, 2010).

Projetar, fabricar, consumir e desfazer-se de servidores, computadores e seus periféricos, de modo proveitoso e efetivo com a menor repercussão possível ao meio ambiente, é de constante importância para a preservação energética, com atenção para o custo de descarte e reciclagem, caracteriza o cumprimento e conhecimento da TI Verde (SALLES; ALVES; DOLCI; LERCH, 2013 apud ELLIOT, 2007).

A diversidade de opções de ferramentas e tecnologias nos fornece maneiras possíveis de se trabalhar com responsabilidade ambiental. Assim é exposto as mais empregadas.

2.2.1 Virtualização

O método de elaborar formas de atuação com base em soluções de *software*, em alternativa ao uso do seguimento físico, caracteriza a virtualização. A mesma oferece o benefício de redução de gastos da Tecnologia da Informação, o crescimento da agilidade e produtividade, em qualquer tamanho de empresa. Pode ser implementada em servidores, armazenamento, aplicativos e redes (VMWARE, 2016).

O modo de operação de servidores físicos que não utilizam a virtualização é reduzido em até 15% devido a não utilizar por completo todos os seus recursos. Com servidores virtualizados é possível usufruir em sua totalidade, e verificar um ganho de desempenho, disponibilidade, e diminuição de despesas.

A rede virtualizada exibe a mesma apresentação de uma rede física realizando a emulação de portas lógicas, *switches*, roteadores, *firewalls*, *VPNs*, balanceamento de cargas e outros. Viabiliza a execução também de testes, dispondo de um ambiente completo para simulação do que se deseja alcançar em um serviço específico (VMWARE, 2016).

Segundo Parkuts, Dufech, Orlovski(2013) citado por Popek e Goldberg(1970), máquina virtual é uma réplica de forma isolada e efetiva de uma máquina física, que opera sobre uma máquina real, com permissão de entrada para o *hardware*.

Nos anos de 1970, segundo Oliveira, Parkuts, Dufech, Orlovski(2013) citado por Carissimi(2008) e Toscano(2010), foi encontrado a virtualização como solução para a fabricação de computadores não importando qual fosse a marca do fabricante, onde somente possuíam um único sistema operacional, e por esse motivo existia problema de incompatibilidade com alguns serviços dedicados a um outro sistema operacional com a virtualização este é executado em cima do sistema nativo, proporcionado a execução de serviços antes conflitante.

Com o surgimento da linguagem Java e a exigência de aperfeiçoamento da segurança na década de 90, fez com que o interesse pela no uso da virtualização aumentasse (PARKUTS, DUFECH, ORLOVSKI(2013) apud MAZIERO, 2013). A empresa *VMware* é uma das empresas que emerge com uma das soluções de *software* para suprir a necessidade da deficiência de suporte para virtualização, característico nos primeiros processadores da *Intel*.

Hoje é possível através da virtualização e com uma máquina robusta, executar o serviço de mil servidores, de menor potência, com a performance de carga de maneira equilibrada (PARKUTS, DUFECH, ORLOVSKI(2013) apud MAZIERO, 2013).

Segundo Parkuts, Dufech, Orlovski(2013) citado por Tanenbaum(2009), os benefícios de se utilizar máquinas virtuais são diversos, tais como: a redução financeira de gastos com energia usada na refrigeração de servidores e diminuição

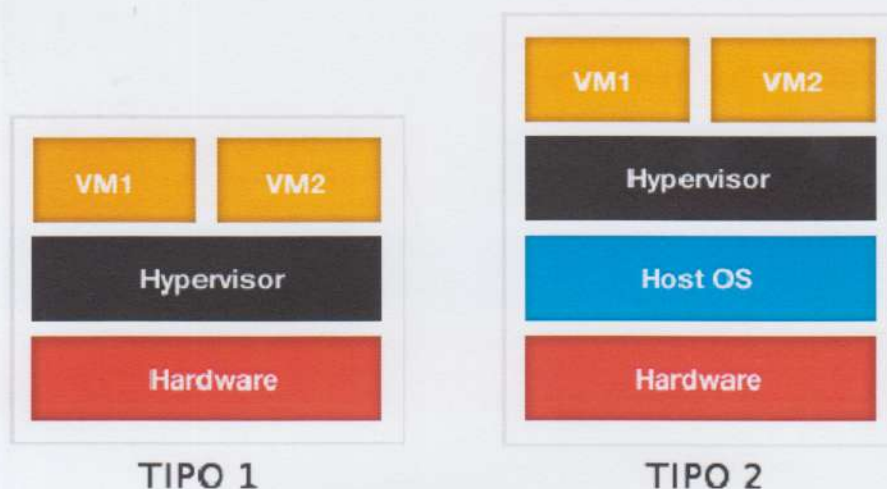
de *hardware* adquirido, que possibilita também o decréscimo de espaço físico em organizações que exigem a reprodução de vários serviços.

Para solucionar erros com ciclos de processamento e a nível de investimento financeiro, a virtualização é obtida como solução, e a oportunidade de abandonar o teoria de “um servidor por serviço” (CARISSIMI,2008). Através desse meio é possível que um *hardware* específico, possa executar uma variedade de aplicações e suas bibliotecas, emulando diversos sistemas operacionais, em ambientes que exijam uma demanda alta, minimizando a quantidade de máquinas físicas. Um dos pontos que definem a escolha do uso da virtualização é a redução de custos com infraestrutura física e sua portabilidade(CARISSIMI,2008).

Em *desktops*, a virtualização serve como alternativa para elaboração de ambientes experimentais preservando o sistema operacional hospedeiro. Nas interfaces de rede físicas, também é possível a virtualização da infraestrutura de rede, com a simulação de *switches* e roteadores, por meio da criação de processos(CARISSIMI,2008).

A virtualização tem sua estrutura constituída a partir de uma camada fina chamada *Hypervisor* ou MMV - Monitor de máquina virtual conforme a figura 1, capaz de realizar o gerenciamento de instruções capazes de manter o tempo de resposta de forma eficiente e ágil. É desejável também todo o controle de recurso do *hardware* como por exemplo, memória, processamento, e igualdade em desempenho de sua reprodução equivalente ao de uma máquina real (PARKUTS, DUFECH, ORLOVSKI(2013) apud POPEK E GOLDBERG, 1974).

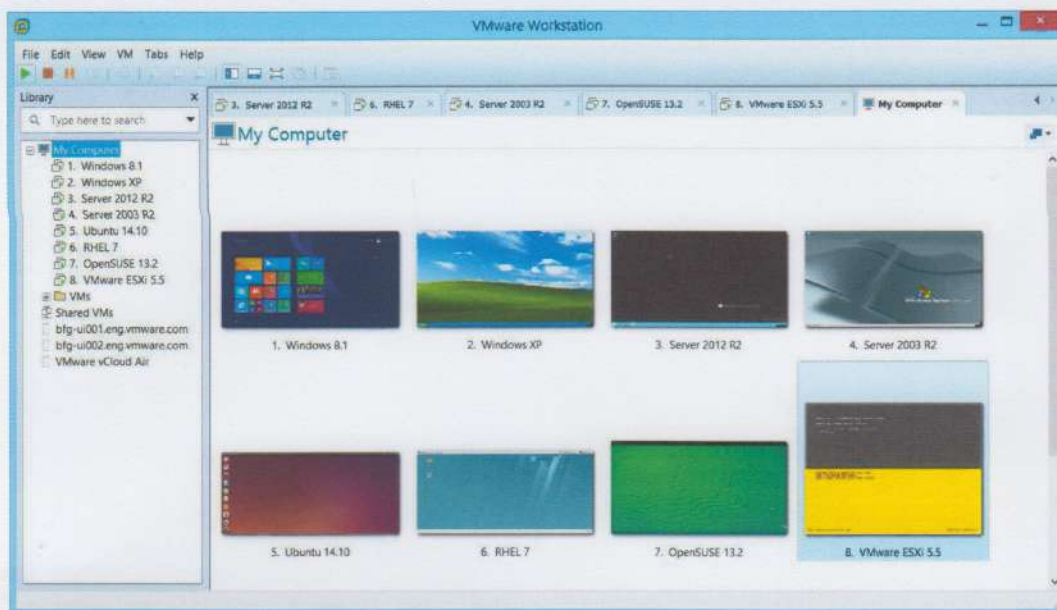
Figura 1 - Tipos de Hypervisor



Fonte: (LEMAF, 2016).

Ocorre uma simulação do *hardware* real, que proporciona o compartilhamento do leitor de CD-ROM, entradas USB, áudio, rede, memória e processador, disponibilizando todos esses recursos de forma virtualizada e equilibrada. A reprodução de um sistema operacional através de *software* de virtualização, é chamado de convidado ou hóspede. O sistema nativo onde é instalado a plataforma de virtualização tem o nome de hospedeiro (PARKUTS, DUFECH, ORLOVSKI(2013) apud SALGADO, 2011).

Há uma solução *desktop* de Monitor de Máquina Virtual disponível sob forma de licença gratuita no mercado, da empresa *VMware* com nome de *VMware Workstation Player* conforme a figura 2 atualmente encontra-se na versão 12.5, com interface intuitiva, que proporciona a reprodução de máquina virtuais de maneira simples. O *VMware Workstation Player* pode ser instalado nos sistemas operacionais Windows e Linux. O mesmo utiliza a estrutura do tipo 2 apresentado na figura de 1 para o seu funcionamento. O objetivo da organização é permitir o acesso de uso para estudantes e instituições sem fins lucrativos (VMWARE, 2016).

Figura 2 - VMware Workstation Player

Fonte: (VMware, 2014).

O *VMware vSphere Hypervisor* é categorizado para a virtualização de servidores, onde faz a utilização da virtualização parcial aplicando a camada *hypervisor* diretamente no *hardware*. A solução oferece melhor gerenciamento, possibilitando a migração de máquinas virtuais, aumento de escalabilidade e menor tempo de indisponibilidade de serviços, conforme a figura 3. A versão atual para *download* gratuito é a 6.0 U2.

Figura 3 - VMware vSphere Hypervisor

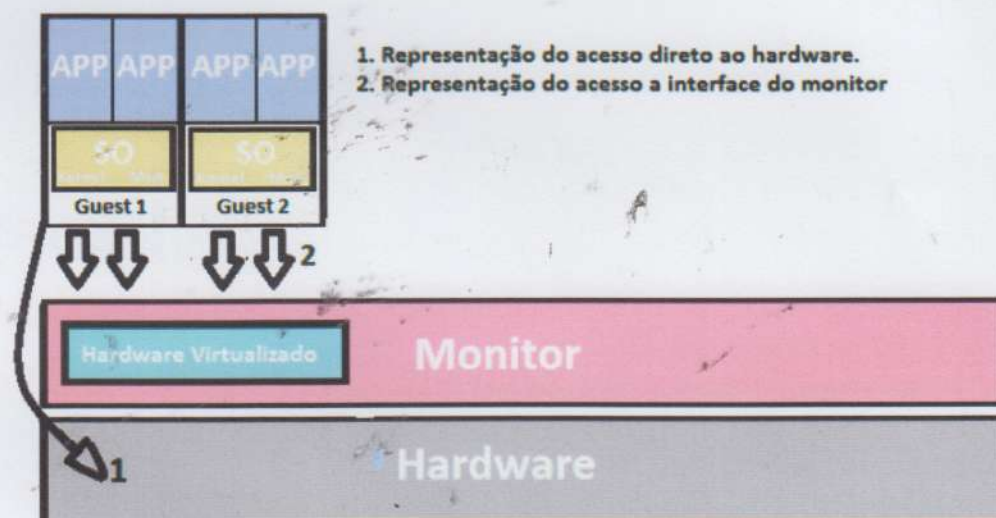
Fonte: (VMware, 2016).

2.2.2 Paravirtualização

No ano 2000, a paravirtualização foi incorporada na Universidade de Washington em seu monitor de máquina virtual com nome de Denali. Conceito esse que promoveu o uso posteriormente em projetos, como o VMware e Xen. O principal motivador para a aplicação da paravirtualização foi a ocorrência de incontáveis erros e incompatibilidades, no momento em que era implementada a virtualização total em processadores de arquiteturas x86 (FRANCO, 2010).

A interface entre o *hardware* e máquina convidada é otimizada, com o emprego da paravirtualização conforme a figura 4, pois seu funcionamento somente realiza a emulação do *hardware* em situações de instruções privilegiadas, quando não necessário e permitido o acesso do sistema virtualizado sem desvios ao *hardware*, tendo como consequência o aumento eficiente da sua performance (FRANCO, 2010).

Figura 4 - Funcionamento da Paravirtualização



Fonte: (UNIOESTE, 2010).

Um exemplo do controle de acesso ao *hardware* existe quando há a necessidade de modificar a memória a qual a máquina virtual convidada realiza a requisição, é então inevitável a intermediação do monitor de máquina virtual para

realizar a definição das posições em que a máquina convidada pode se conectar, conforme a figura 5.

A melhora no desempenho com uso da paravirtualização, exige a modificação do sistema operacional convidado a fim de possibilitar o encaminhamento de instruções diretas ao *hardware*, definindo em qual momento é preciso a interação com o monitor de máquina virtual (AGARD, 2009).

Figura 5 - Paravirtualização 02



Fonte: (Universidade São Francisco, 2008).

2.2.3 Docker

O alto período gasto na preparação e ordenação da instalação e configuração de servidores, *web servers*, conexão com banco de dados em uma infraestrutura, e o longo procedimento de desenvolvimento de aplicações bem como o seu processo de distribuição é um dos motivadores para a adoção de técnicas que proporcionem a automatização dos fluxos (ROMERO, 2015).

A problemática de possibilitar as aplicações serem reproduzidas em ambientes distintos ou em até servidores personalizados, torna o trabalho do desenvolvedor um pouco mais complexo, devido ao fato de estar fora de seu domínio a implantação de seu sistema em diferentes contextos. A fim de oferta-se como solução, o Docker surge apresentando-se em março de 2013, ao longo do evento PyCon US em Santa Clara, lançada pela DotCloud e desenvolvido na linguagem Go por Solomon Hykes.

Com a política de código aberto(*open source*) e a proposta de isolamento, permanecendo com a mesma organização de todos os serviços que compõem a aplicação, em qualquer o meio (ROMERO,2015).

O *Docker* tem como ilustração um navio cargueiro, contendo inúmeros *containers* que em seu conteúdo estão localizadas os serviços e aplicações. O *back-end* do *Docker* tem sua base em *Linux Containers(LXC)*. O *LXC* é uma técnica de virtualização em grau de sistema operacional, que permite a execução de sistemas Linux de maneira isolada conforme a figura 6, em forma de *containers*, tudo em apenas um único hospedeiro, ele não provê a máquina virtual porém realiza o compartilhamento de seus recursos tais como: CPU, memória, rede, I/O (ARCHLINUX, 2015).

O *LXC* é similar ao *Chroot*. Esta forma uma espécie de jaula pois, quando praticado, modifica diretórios do ambiente em execução comum, gerando um controle de permissões, com o acesso disponível somente para o processo atual e seus filhos(ARCHLINUX, 2016).

O uso do mesmo Linux *Kernel* do servidor hospedeiro, através de Linux Containers, apresenta um aumento significativo no tempo de resposta de cada processo percebendo-se assim que o *Docker* é visto como opção a virtualização completa e leve, quando é se defrontado com *Hypervisors KVM, VMware ESXi e XEN* (ROMERO, 2015).

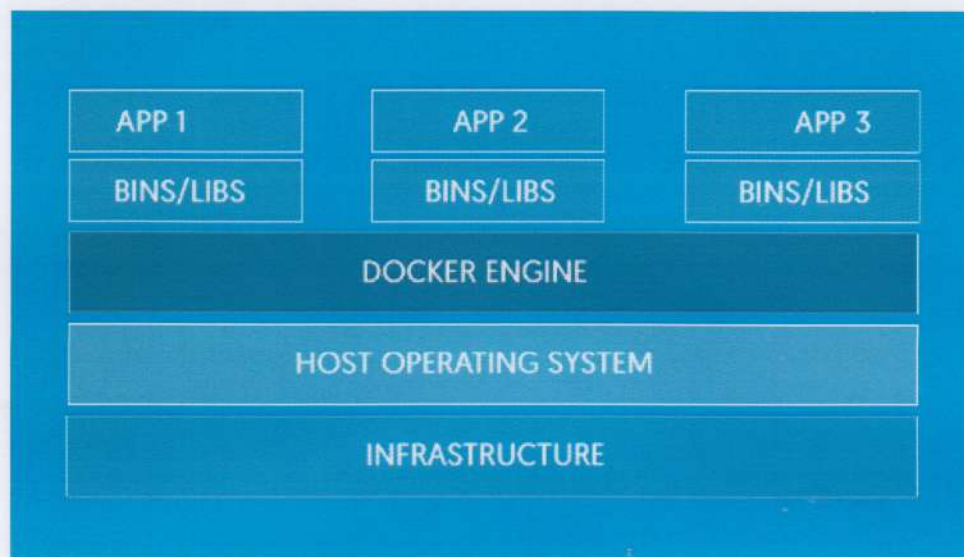
O tempo em que o processo de inicialização de *containers* exigem são poucos segundos, justamente por realizar a utilização do Linux Kernel, que tem como recurso o *Cgroups(control groups)*, responsável pelo gerenciamento de *hardware* como a limitação e isolamento de CPU, memória, disco, rede etc. O controle de separação de grupos de processos é feito pelo *Namespaces*, com objetivo de limitar alcance de processos de diferentes grupos (ROMERO,2015).

O *Namespaces* é fortemente utilizado pelo *Docker*, suas camadas, são divididas da seguintes maneira:

- **pid** - isolamento de processos(PID);
- **net** - controle de interfaces de rede;
- **ipc** - controle de recursos de IPC(*InterProcess Communication*);

- **mnt** - gestão de pontos de montagem;
- **uts** - UTS (*Unix Timesharing System*) isolar recursos do kernel;

Figura 6 - Docker



Fonte: (Docker, 2016).

O *Docker* desfruta do *Union file Systems*, um tipo de sistema de arquivos em camadas, leves e ágeis, principalmente na construção de imagens no momento de elaboração de containers.

Em sua organização o *Docker* oferece um repositório oficial com registro público cujo o domínio é: <http://hub.docker.com> conforme a figura 7, para o armazenamento de imagens, a qual viabilizam a criação de *containers*. O controle de acesso é feito através de usuário e senha que podem ser cadastrados gratuitamente. Os repositórios criados podem ser do tipo público ou privados.

Figura 7 - Docker Hub



Fonte: (Docker HUB, 2016).

2.3. Ambiente de Desenvolvimento de Software

A necessidade de produção e melhoria constante na elaboração de softwares, gerou a necessidade da criação de um sistema computadorizado, utilizado para preencher estas demandas com as funcionalidades de dar suporte a fabricação, manutenção e possibilitar melhor gestão e controle do *software*, caracterizando o ambiente de desenvolvimento de *software* (VILELLA, TRAVASSOS, ROCHA, 2004).

2.3.1 Java

Criada em 1992 pela empresa Sun Microsystems, a linguagem de programação Java tinha como objetivo inicial atender a um nicho de mercado específico, sua implementação estava voltada para os aparelhos eletrônicos, tais como: televisão, forno micro-ondas, aparelhos de TV a cabo, agendas eletrônicas, etc. Porém, não obteve o sucesso comercial. Dessa forma a Sun observou com o surgimento da *WEB* que o Java poderia ser utilizado em pequenas aplicações dentro do *web browser*, através dos chamados *applets*, permitindo o envio e recebimento de dados através do tráfego da Internet. Dessa forma, o Java conseguiu atingir uma enorme popularização.

A linguagem Java é baseada em Orientação a Objetos dispondo de um grande conjunto de bibliotecas, o que facilita no desenvolvimento e permite o aproveitamento e reparos do código.

A portabilidade do Java é um dos fatores motivadores na escolha da linguagem visto que, a mesma faz o uso em seu funcionamento da *Java Virtual Machine (JVM)*, que proporciona a execução do *software* desenvolvido em Java, em diversos sistemas operacionais, sem a necessidade de ter que ser elaborado um código para cada tipo de sistema por exemplo, para o Linux e outro para Windows. A compatibilidade do Java com vários sistemas, se dá pelo fato de existir um compilador com nome de *javac*, o qual gera o código de máquina conhecido com *bytecode* e verifica a ocorrência de possíveis erros, evitando a análise que seria feita somente em tempo de execução, definindo o Java como a uma escolha segura (INDRUSIAK,1996).

2.3.2 Groovy

O Groovy é uma linguagem de programação, que possui sua sintaxe semelhante ao do Java, seus benefícios similares, possibilitam maior ascensão de programadores Java pois, diminui a curva de aprendizagem da mesma. Criada em 2003 por Guillaume Laforge, o Groovy tem uma tipagem dinâmica, permitindo a inclusão e modificação de métodos, funções e classes de códigos já existentes, tudo em tempo de execução, em linguagens estáticas esse tipo de funcionalidade somente é possível em tempo de compilação ou através de técnicas de padrões de projetos considerados complexos.

A redução de código usada na melhoria de sintaxe, contextualiza o Groovy como uma linguagem simplificada, tornando fáceis suas tarefas antes consideradas de grande dificuldade. Além disso foi desenvolvido para executar em JVM. Podemos verificar na figura 8, o exemplo de implementação *bigdecimal* de um delta de uma equação de segundo grau, e notar a significativa redução de código (WEISSMANN,2015).

Figura 8 - Groovy BigDecimal

```
BigDecimal delta(BigDecimal a, BigDecimal b, BigDecimal c) {
    return b.multiply(b).subtract(new BigDecimal(4).multiply(a).multiply(c))
}
```

E em Groovy?

```
BigDecimal delta(BigDecimal a, BigDecimal b, BigDecimal c) {
    (b*b) - (4*a*c)
}
```

Fonte: Falando de Grails - Altíssima produtividade no desenvolvimento web.

Diferentemente do Java, o *Groovy* não necessita de uma classe para realizar sua execução, a mesma pode ser reproduzida de forma simples, observa-se na figura 9, onde é ensinado na primeira aula ao se iniciar o estudo de uma linguagem de programação, como seria o *Hello World em Groovy*.

Figura 9 - Hello World com Groovy

```
public class OlaMundo {
    public static void main(String args[]) {
        System.out.println("Oi Mundo!");
    }
}
```

E em Groovy?

```
println "Oi Mundo!"
```

Fonte: Falando de Grails - Altíssima produtividade no desenvolvimento web.

2.3.3 Vert.X

O *framework web* Vert.x foi desenvolvido orientado a eventos, o que o torna a aplicação escalável, de maneira a gerenciar muitas requisições com o mínimo de *hardware* disponível. Sua diversidade de opções, de ser implementado em várias linguagens tais como Java, Javascript, Groovy, Ruby e Ceylon, o caracteriza como um *framework* poliglota (VERT.X, 2016).

Sua proposta faz o uso da arquitetura de microsserviços aplicando-se de forma leve, rápida e simplificada pois, ao contrário de aplicações monolíticas feitas

em unidade, realiza a divisão de funcionalidades em serviços menores e ágil. A sua diminuição não pode ser subestimada visto que é capaz do processamento de altos volumes de dados, ótimo desempenho e de acordo com a necessidade, pode fazer uso de requisições HTTP com retorno no formato *JSON*, algo muito utilizado por desenvolvedores.

2.3.4 Grails

Com proposta de alta produtividade através do uso de paradigma por convenção, foi criado no ano de 2005, o *framework Grails* sua linguagem base é o *Groovy*, apresentando-se como alternativa ao custoso e dramático desenvolvimento com Java Web.

O *Grails* destaca-se por sua fácil utilização do mapeamento objeto relacional com *hibernate*, sua arquitetura é desenvolvida nos padrões MVC (modelo-visão-controlador), o que garante sua estrutura inserida nos principais padrões de projeto de *software* praticados no mercado. Possui um ambiente de linha de comando interativo e sistema de construção baseado no *Gradle*. O *Tomcat* já é incorporado ao projeto *Grails*, diminuindo o tempo empregado em instalação e configuração do mesmo (GRAILS, 2016).

2.4 Banco de Dados

É considerado banco de dados uma coleção de dados estruturados, que suprem as necessidades de diversos usuários, contidos em uma ou mais organizações, podendo ser uma lista de compras, galeria de imagens e até mesmo uma imensa quantidade de informação corporativa. Como resultado de sua utilização, obtém-se uma maior disponibilidade, aumento da facilidade de acesso e atualização de transações complicadas além da diminuição de redundância de dados (ALBUQUERQUE, 2011).

2.4.1 HSQLDB

O HSQLDB é um dos principais softwares de banco de dados relacional escrito em Java. O mesmo oferece mecanismos simples, de forma reduzida em comparação com outros SGBD's pois, ocupa pouco espaço em disco. Apresenta também performance rápida pois, tem poucos recursos porém, mesmo assim, ainda é bastante eficiente.

2.5 Serviços Web

Com o surgimento da Internet e tecnologias correspondentes, aliado a facilidade de acesso a redes sociais em qualquer parte do mundo, apresenta-se uma camada de comunicação, como alternativa para ultrapassar a limitação empresarial. Tal evolução impulsionou o desenvolvimento de novas arquiteturas definidas como Serviços Web.

O Serviço Web é ser subdivido em quatro fases para identificar o seu funcionamento: primeiramente, o cliente e servidor tomam conhecimento um do outro logo após, definem qual o objetivo do serviço específico que irá permitir a interação entre os agentes. De acordo com as necessidades encontradas, é realizada a implementação de métodos para atender a essa demanda. Por fim, então ocorrendo a comutação de mensagens entre cliente e servidor, contextualizando a comunicação entre os mesmos (NUNES, DAVID, 2005).

2.5.1 REST

Diferentemente de práticas já conhecidas de desenvolvimento web, mediante especificações com o uso de IDE's, o modelo REST oferta-se como opção para implementação de Serviços Web. Em uma arquitetura REST, é obrigatório a apresentação de recursos e disponibilização dos mesmos com identificação de forma global. Conforme a figura 10, no exemplo pode-se obter todos os dados de um aluno, utilizando o recurso que representa o documento XML, contido nele todas as informações (NUNES, DAVID, 2005).

Figura 10 - XML REST

```
GET http://univ.example.com/studentInfo/123456789

<?xml version="1.0" ?>
<u:StudentInfo xmlns:u="http://univ.example.com">
  <u:code>123456789</u:code>
  <u:name>João Silva</u:name>
  <u:age>19</u:age>
</u:StudentInfo>
```

Fonte: (FEUP, 2004).

Em teoria o modelo REST não possui limitação de protocolos, porém na prática apresenta maior compatibilidade com o protocolo HTTP, essa realidade também existe para o HTTPS, através da adição de uma camada extra. Além do HTTP, podem ser destacados do uso das tecnologias XML(*eXtensible Markup Language*), JSON(*Javascript Object Notation*) e URI(*Uniform Resource Identifier*).

3. METODOLOGIA

A metodologia adotada neste trabalho é baseada em uma pesquisa bibliográfica para construção de uma base sólida, com o objetivo de desenvolver uma aplicação de gerenciamento *web* nomeada de *DockerWeb*, seguindo os padrões de projeto de *software* a fim de proporcionar um melhor controle, administração e aumento de produtividade com o auxílio da automatização de *docker containers*, usuários que adotaram ou que desejam aderir a tecnologia *Docker* em seu parque de TI podem obtê-la como alternativa para disponibilização de serviços.

Por meio do uso do conjunto de tecnologias já citadas, a implantação do *DockerWeb* tem sua proposta de facilitar o dia-dia do profissional de TI, contribuindo com o meio ambiente, uma vez que faz uso da tecnologia *docker* através de técnicas de paravirtualização em concordância com as políticas da TI Verde, diminuindo o número de máquina servidoras necessárias.

4. ESTUDO DE CASO

Nesta seção será exposto como era o funcionamento do uso da tecnologia *Docker* antes da implantação do sistema *DockerWeb* e como o mesmo foi organizado e implementado, descrevendo sua proposta e resultados do ganhos obtidos.

4.1 ANTES DO SISTEMA DOCKERWEB

O *Docker* são *containers* que permitem empacotar aplicativos ou trechos de códigos, em um sistema de arquivos completo, onde contém tudo que é necessário para execução de aplicativos e serviços. Com características de ser leve, e seguro, pois trabalha com o compartilhamento do *kernel* do sistema hospedeiro, o que garante o uso eficiente da memória RAM, não sendo preciso uma nova virtualização completa de todo o *hardware*. Além disso o *Docker* fornece um ambiente seguro, através de uma camada adicional de proteção dos *containers*, chamada *namespaces* que proporciona o isolamento dos processos (ROMERO, 2015).

A adoção da tecnologia *Docker*, traz benefícios na redução de custos de infraestrutura, através de um menor número de máquinas utilizadas. Ao utilizar a funcionalidade de criação de imagem por meio do arquivo *Dockerfile*, pode-se desenvolver um ambiente preciso do que se deseja e realizar a reprodução do container em qualquer local cliente, agilizando e diminuindo possíveis erros de implantação de serviços, sem a preocupação com ajustes, somente executando a imagem armazenada em seu repositório oficial.

Atualmente o *Docker* não possui uma interface gráfica dessa forma, toda sua execução acontece em linha de comando, conforme a tabela 1, figuras 11, 12 e 13, seguindo a exemplificação de comandos básicos.

Tabela 1 - Comandos Docker.

Comando	Função
docker ps	Listar containers ativos.
docker ps -a	Listar containers ativos e inativos.
docker images	Listar imagens já salvas na máquina local.
docker rm <ID>	Comando para remove container.
docker rmi <ID>	Comando para remover imagem.
docker pull repositorio/imagem:tag	Instrução para baixar uma imagem oficial do Docker HUB.
docker push usuario/repositório	Instrução para realizar o compartilhamento de imagens para o Docker Hub.
docker run <modo de exibição> <imagem:tag>	Comando para executar um container a partir de uma imagem.
docker search <imagem>	Comando para realizar a pesquisa de imagens no repositório oficial do Docker.
docker stop <ID>	Parar o container em uso.
docker start <ID>	Iniciar o container parado.
docker diff <ID>	Para exibir modificações aplicadas ao container.
docker inspect <ID>	Para acessar informações de baixo nível de containers e imagens.

Figura 11 - Exibindo dockers containers em uso.

```

root@srvdocker:~# docker ps
CONTAINER ID   IMAGE          COMMAND                  CREATED        STATUS        PORTS
77cb03081d14   rancher/server "/usr/bin/s6-svscan /"   About an hour ago   Up About an hour   3306/tcp, 0.0.0.0:8383->8080/tcp
bc9dbcafaab5   happy_bartik   jpenren/alpine-dcn4chee "sh -c /opt/dcn4chee."   About an hour ago   Up About an hour   0.0.0.0:11112->11112/tcp, 0.0.0.0:8282->8080/tcp
pacshsqldb
root@srvdocker:~#

```

Fonte: (Imagem do próprio autor, 2016).

Figura 12 - Exibindo imagens armazenadas no servidor.

```
root@srvDocker:~# docker images
```

REPOSITORY	TAG	IMAGE ID	CREATED	SIZE
rodrigoemerenciano/dcm4chee	1.3	9e7d57fae856	2 weeks ago	974.8 MB
postgres	1.0	dab8b36687ee	2 weeks ago	634 MB
dcm4cheehsql	1.0	0bac33bdaacb	3 weeks ago	248.8 MB
<none>	<none>	099d2f29f17d	3 weeks ago	210.2 MB
mysql	latest	eda6a4884645	3 weeks ago	383.4 MB
<none>	<none>	2ba9b29d52ca	3 weeks ago	187.9 MB
mysql/mysql-server	latest	43d4a6c26521	4 weeks ago	369.1 MB
webdevops/nginx	latest	42c51e693bfe	5 weeks ago	330.4 MB
rodrigoemerenciano/apache	1.0	a9d0ccbccc4f	5 weeks ago	224.5 MB
rodrigoemerenciano/apache	1.1	a9d0ccbccc4f	5 weeks ago	224.5 MB
postgres	9.5	239612485614	5 weeks ago	263.9 MB
ubuntu	14.04	f2d8ce9fa988	5 weeks ago	187.9 MB
java	8	96cddf5ae9f1	6 weeks ago	640.9 MB
ubuntu	14.04.5	b1719e1db756	6 weeks ago	188 MB
zarra/dcm4chee_base	latest	f93b4cce27c7	9 weeks ago	775.6 MB
swarm	latest	942fd5fd357e	11 weeks ago	19.47 MB
sameersbn/mysql	latest	ffd5a3cd7890	3 months ago	294.1 MB
jpenren/alpine-dcm4chee	latest	f9456c72a416	6 months ago	248.8 MB
dcm4chee/slapd-dcm4chee	5.1.0	5379729cf86c	7 months ago	177.4 MB
dcm4chee/dcm4chee-arc-psql	5.1.0-secure-ui	d885d0770e0a	8 months ago	524.3 MB
dcm4chee/postgres-dcm4chee	5.1.0	7a8f9b39bd9d	8 months ago	263.6 MB

Fonte: (Imagem do próprio autor, 2016).

Figura 13 - Exibindo docker containers ativos e inativos.

```
root@srvDocker:~# docker ps -a
```

CONTAINER ID	IMAGE	COMMAND	CREATED	STATUS	PORTS
bcd0bcafaab5	jpenren/alpine-dcm4chee	"sh -c /opt/dcm4chee-"	2 minutes ago	Up 2 minutes	0.0.0.0:11112->11112/tcp, 0.0.0.0
1.0:8282->8080/tcp	pacshsqldb				
d41fc0e1c1db	swarm	"/swarm manage token:"	5 days ago	Exited (0) 5 days ago	
550979ecbeae	swarm	"/swarm join --addr="	5 days ago	Exited (0) 5 days ago	
7106efc840b6	swarm	"/swarm join --addr="	5 days ago	Exited (0) 5 days ago	
79c38ce62362	swarm	"/swarm join --addr="	5 days ago	Exited (0) 5 days ago	
83962b996cbe	mysql/mysql-server:latest	"entrypoint.sh mysql"	2 weeks ago	Exited (0) 2 weeks ago	
88066b43b6e3	dcm4cheehsql:1.0	"sh -c /opt/dcm4chee-"	3 weeks ago	Exited (0) 3 weeks ago	
94decf960519	pacAlpina	"/stage/start.bash"	3 weeks ago	Created	0.0.0.0:8080->8080/tcp, 0.0.0.0
111112->11112/tcp	pacsl				
62e5f5d31dc3	099d2f29f17d	"/bin/sh -c 'apt-get "	3 weeks ago	Exited (0) 3 weeks ago	
4110eaf5987a	ecstatic_ptolony	"/bin/sh -c 'apt-get "	3 weeks ago	Exited (0) 3 weeks ago	
0a310b5f4ca8	happy_panini	"/bin/sh -c 'apt-get "	3 weeks ago	Exited (0) 3 weeks ago	
80bebea42131	8a3c34968c4f	"/bin/sh -c 'apt-get "	3 weeks ago	Exited (0) 3 weeks ago	
b4de4076ff4e	pedantic_wright	"/bin/sh -c 'apt-get "	3 weeks ago	Exited (0) 3 weeks ago	
c312e71ee628	sharp_engelbart	"/bin/sh -c 'apt-get "	3 weeks ago	Exited (0) 3 weeks ago	
074a3d98d2ed	0f06bec5060	"/bin/sh -c 'apt-get "	3 weeks ago	Exited (0) 3 weeks ago	
70dc485bdaa9	771483afccf	"/bin/sh -c 'apt-get "	3 weeks ago	Exited (0) 3 weeks ago	
be3cae277c39	zen_dijkstra	"/bin/sh -c 'apt-get "	3 weeks ago	Exited (0) 3 weeks ago	
65d961791896	8a3c34968c4f	"/bin/sh -c 'apt-get "	3 weeks ago	Exited (0) 3 weeks ago	
7066333f6624	determined_euler	"/bin/sh -c 'apt-get "	3 weeks ago	Exited (0) 3 weeks ago	
f4e7752de1eb	8a3c34968c4f	"/bin/sh -c 'apt-get "	3 weeks ago	Exited (0) 3 weeks ago	
	peaceful_goodall	"/bin/sh -c 'unzip /s"	3 weeks ago	Exited (0) 3 weeks ago	
	fca89bcdec22				

Fonte: (Imagem do próprio autor, 2016).

Todos os comandos a acima apresentados são realizados pelo usuário de forma manual acessando o terminal. Para obtenção de informações e operações iniciais, tais como: a exibição de *containers* que estão sendo executados apontando portas em uso, volumes montados, nome e tempo que o *container* se encontra em atividade, sua quantidade de imagens já salvas no servidor em uso, quais

containers são inativos e ativos, como também, o emprego de instruções para inicialização, parada e remoção dos containers.

Pode ser observado nas figuras 14, 15, 16 e 17, como são executadas algumas das funcionalidades da tecnologia *Docker* já citadas.

Figura 14 - Rodando container em modo daemon, com especificação de portas e nome.

```
root@srvDocker:~# docker ps
CONTAINER ID        IMAGE               COMMAND             CREATED             STATUS              PORTS
77cb03081d14       rancher/server     "/usr/bin/s6-svscan /" 4 hours ago        Up About a minute   3306/tcp
p, 0.0.0.0:8383->8080/tcp
happy_bartik
root@srvDocker:~# docker run --name pacshsqldb --hostname pacDocker -d -p 8282:8080 -p 11112:11112 jpenren/alpi
ne-dcm4chee
```

Fonte: (Imagem do próprio autor, 2016).

Figura 15 - Iniciar container após ter sido parado.

```
ago
b5244b8644d6       fbe31b56f7db      "/bin/sh -c '/etc/init" 5 weeks ago        Created
teste12
12a50eae5c5        fbe31b56f7db      "-p 80:80"           5 weeks ago        Created
teste18
80/tcp
ddeb70ccc96        7cc1e7a25d9a      "/bin/sh -c 'sed \"Ser" 5 weeks ago        Exited (0) 5 weeks
tender_saha
ago
78b59940a829       0781999ba39a      "/bin/bash"          5 weeks ago        Exited (0) 5 weeks
ago
da2c25bd0184       0781999ba39a      "/bin/sh -c '/etc/init" 5 weeks ago        Exited (0) 5 weeks
teste11
ago
root@srvDocker:~# docker start bc0dbcafaab5
teste10
```

Fonte: (Imagem do próprio autor, 2016).

Figura 16 - Parar container após ter sido iniciado.

```
12a50eae5c5        fbe31b56f7db      "-p 80:80"           5 weeks ago        Created
teste10
80/tcp
ddeb70ccc96        7cc1e7a25d9a      "/bin/sh -c 'sed \"Ser" 5 weeks ago        Exited (0) 5 weeks
tender_saha
ago
78b59940a829       0781999ba39a      "/bin/bash"          5 weeks ago        Exited (0) 5 weeks
ago
da2c25bd0184       0781999ba39a      "/bin/sh -c '/etc/init" 5 weeks ago        Exited (0) 5 weeks
teste11
ago
root@srvDocker:~# docker stop bc0dbcafaab5
teste10
```

Fonte: (Imagem do próprio autor, 2016).

Figura 17 - Remover container após ter sido parado.

```

414febff4a17'      2dde0ff8b71b      "/usr/sbin/apache2 -D"      7 weeks ago      Exited (0) 7 weeks
ago
c9a10eb8e1bf      2dde0ff8b71b      "-p 80:80"      7 weeks ago      Created
6f0afdb6384d      2dde0ff8b71b      "/usr/sbin/apache2 -D"      7 weeks ago      Exited (0) 7 weeks
ago
8294723b7f61      fbe31b56f7db      "/bin/sh -c '/etc/ini"      7 weeks ago      Exited (0) 7 weeks
ago
d5cf13b6bc8c      fbe31b56f7db      "/bin/sh -c '/etc/ini"      7 weeks ago      Exited (0) 7 weeks
ago
2b16881e32ce      webdevops/nginx      "/opt/docker/bin/entr"      7 weeks ago      Exited (0) 7 weeks
ago
64c3ed4c4909      fbe31b56f7db      "/bin/bash"      7 weeks ago      Exited (0) 7 weeks
ago
73a64703c621      fbe31b56f7db      "/bin/sh -c '/etc/ini"      7 weeks ago      Exited (0) 7 weeks
ago
e7038ff3cf3b      fbe31b56f7db      "/bin/sh -c '/etc/ini"      7 weeks ago      Exited (0) 7 weeks
ago
b5244b8644d6      fbe31b56f7db      "/bin/sh -c '/etc/ini"      7 weeks ago      Created
12a50eae5c5      fbe31b56f7db      "-p 80:80"      7 weeks ago      Created
80/tcp
ddeb7c70ccc96      7cc1e7a25d9a      "/bin/sh -c 'sed \"Ser"      7 weeks ago      Exited (0) 7 weeks
ago
78b59940a829      0781999ba39a      "/bin/bash"      7 weeks ago      Exited (0) 7 weeks
ago
da2c25bd0184      0781999ba39a      "/bin/sh -c '/etc/ini"      7 weeks ago      Exited (0) 7 weeks
ago
root@srvDocker:~# docker rm -f da2c25bd0184
da2c25bd0184
root@srvDocker:~#

```

Fonte: (Imagem do próprio autor, 2016).

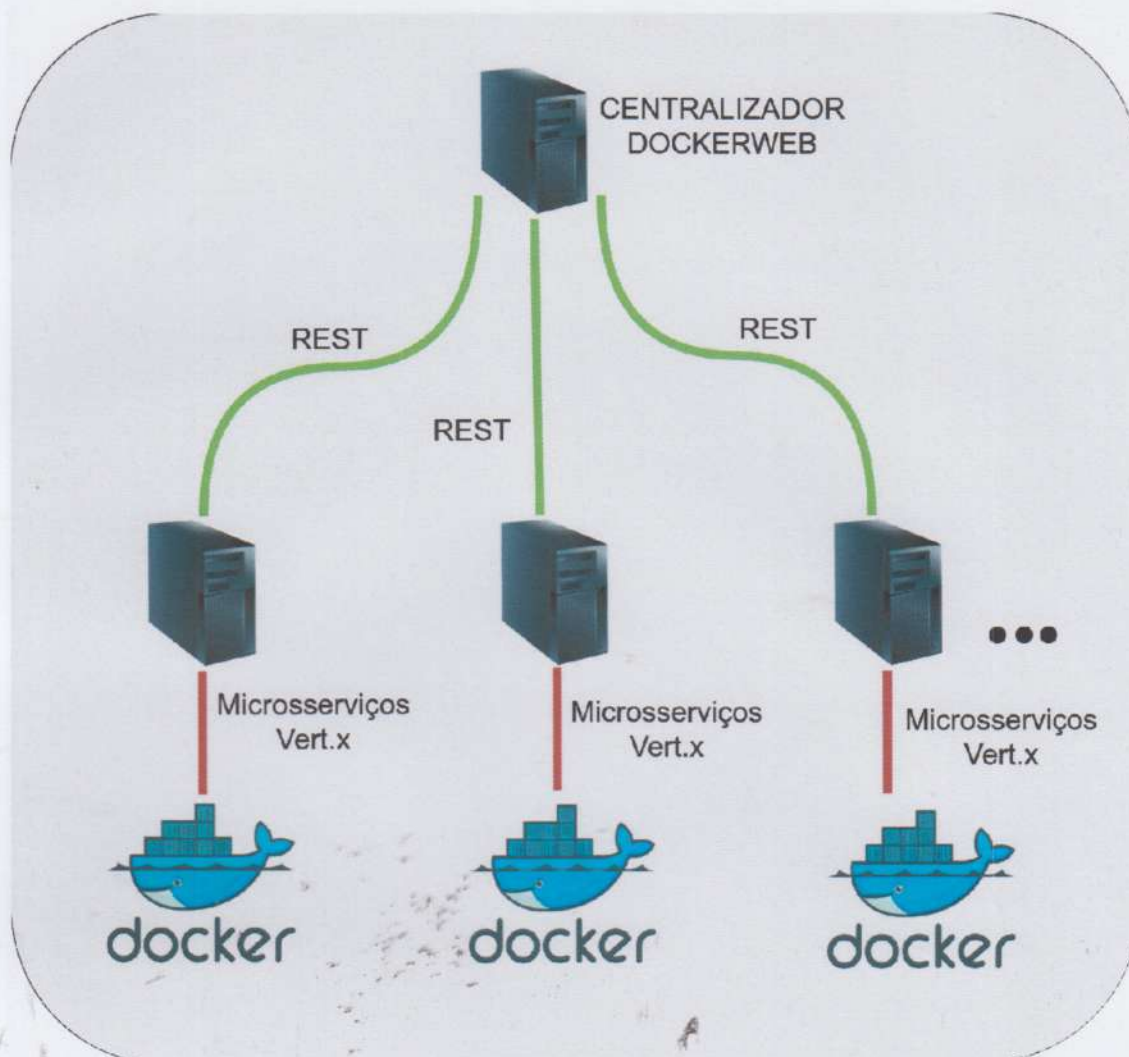
4.2 DOCKERWEB

4.2.1 Visão Geral do Sistema

O sistema *DockerWeb* foi desenvolvido com visão de sistemas em rede, utilizando arquitetura REST e proposta de microsserviços instalado no *Docker CLI*, juntamente com a tecnologia *Docker* integrada no mesmo, conforme a figura 18. O *Docker GUI* caracteriza-se por ser um manager, apresentando uma interface gráfica amigável, desenvolvida para executar em versão *web*. Seu projeto para administração e gestão de *docker containers* e imagens, o torna uma solução bastante atrativa para o usuário que busca a automação de processos, quando depara-se em um ambiente com vários servidores *Docker* implantados, onde pode ser notado de certa forma no fluxo comum, que demandaria uma boa parcela de tempo para seu manejo. Dessa forma a interação entre os componentes *DockerWebGUI* e o *DockerWeb CLI*, compõem o *DockerWeb* com proposta para uma aplicação web que venha a modificar o modo de usabilidade do *Docker*, instaurado atualmente, a fim de facilitar o emprego de comandos e seu tratamento de dados importantes a serem exibidos.

Figura 18 - Esquema de Funcionamento do DockerWeb.

ESQUEMA DE FUNCIONAMENTO - DOCKERWEB



Fonte: (Imagem do próprio autor, 2016).

4.2.2 DOCKERWEB CLI

O DockerWeb CLI tem sua proposta constituída no desenvolvimento de uma ferramenta de código aberto, a qual utiliza microsserviços, com o objetivo de tornar a aplicação com mecanismos mais leves e que possibilite a interação com o servidor Docker instalado em sua infraestrutura de TI. O *framework* Vert.x foi a escolha definida pois, que permite o uso de diversas linguagens de programação, dentre elas o Groovy, linguagem principal utilizada no projeto. Através do Vert.x pode-se alcançar uma escolha ideal para se trabalhar com aplicações baseadas em microsserviços,

por apresentar um desempenho alto e leve. O mesmo trabalha com programação reativa, o que em linhas gerais se caracteriza por ser uma aplicação orientada a eventos e estímulos que podem ser executados por requisições HTTP ou cliques de botões.

O mecanismo inicial que impulsiona a lógica do *DockerWeb CLI* é a implementação de código voltada para a funcionalidade de roteamento do Vert.x, disponível em seu serviço web, apresentando-se em uma porta específica previamente cadastrada. Nota-se a necessidade de proteção do sistema oferecido, devido a possíveis ataques através de réplicas de requisições enviadas ao servidor, com o objetivo de obter informações. A segurança do sistema tem sua proposta na elaboração de um algoritmo, que gere uma identificação única da requisição e o servidor realize sua validação a recebendo como parâmetro, evitando as cópias requisições.

Foi realizado a criação de uma classe com nome *DockerCommands.groovy* conforme a figura 19, onde foi implementado variáveis contendo em seus valores o emprego de comando *dockers* já no modo de super usuário, o que é pré-requisito para execução do *Docker*.

Figura 19 - Variáveis Classe *DockerCommands*.

```
class DockerCommands {
    private static final DOCKER_PS = ["sudo","docker","ps","-a","--format","{{.ID}}|{{.Image}}|{{.Command}}|{{.C...
    private static final DOCKER_STOP = ["sudo","docker","stop"]
    private static final DOCKER_START = ["sudo","docker","start"]
    private static final DOCKER_INSPECT = ["sudo","docker","inspect"]
    private static final DOCKER_RM = ["sudo","docker","rm"]
    private static final DOCKER_RMI = ["sudo","docker","rmi"]
    private static final DOCKER_IMAGES = ["sudo","docker","images","--format","{{.ID}}|{{.Repository}}|{{.Tag}}"]
    private static final DOCKER_PORTS = ["sudo","docker","ps","--format","{{.ID}}|{{.Image}}|{{.Ports}}"]
}
```

Fonte: (Imagem do próprio autor, 2016).

A interação do *DockerWeb CLI* com o *Docker*, também encontra-se nessa mesma classe, aplicando variáveis inseridas em sua lógica, quando se necessita das principais funções, como exibido nas figuras 20, 21, 22, 23, 24, 25 e 26 :

- Parar container:

Figura 20 - Função Parar Container.

```
public static stop(containerID){  
    def listaComando = []  
    listaComando.addAll(DockerCommands.DOCKER_STOP)  
    listaComando.add(containerID)  
    Process p = DockerCommands.createProcess(listaComando)  
    def error = p?.errorStream?.text  
    def success = p?.text  
    p.waitFor()  
    p.destroy()  
    def exitValue = p.exitValue()  
    return [exitValue:exitValue,error:error,success:success]  
}
```

Fonte: (Imagem do próprio autor, 2016).

- Iniciar container:

Figura 21 - Função Iniciar Container.

```
public static start(containerID){  
    def listaComando = []  
    listaComando.addAll(DockerCommands.DOCKER_START)  
    listaComando.add(containerID)  
    Process p = DockerCommands.createProcess(listaComando)  
    def error = p?.errorStream?.text  
    def success = p?.text  
    p.waitFor()  
    p.destroy()  
    def exitValue = p.exitValue()  
    return [exitValue:exitValue,error:error,success:success]  
}
```

Fonte: (Imagem do próprio autor, 2016).

- Listar containers:

Figura 22 - Função Listar Containers.

```
public static dockerPS() {
    Process p = DockerCommands.createProcess(DockerCommands.DOCKER_PS)
    def listaResult = []

    final BufferedReader r = new BufferedReader(new InputStreamReader(p.getInputStream()))
    String line = r.readLine()
    while (line != null) {
        def tratado = line.split("\\|")
        listaResult.add([id:tratado[0].replaceAll("\\n",""),image:tratado[1].replaceAll("\\n",""),command:
        line = r.readLine()
    }
    r.close()

    p.waitFor()
    p.destroy()
    return listaResult
}
```

Fonte: (Imagem do próprio autor, 2016).

- Remover container:

Figura 23 - Função Remover Container.

```
public static dockerRm(containerID,force=false){
    def listaComando = []
    listaComando.addAll(DockerCommands.DOCKER_RM)
    if(force)
        listaComando.add("-f")
    listaComando.add(containerID)
    Process p = DockerCommands.createProcess(listaComando)
    def error = p?.errorStream?.text
    def success = p?.text
    p.waitFor()
    p.destroy()
    def exitValue = p.exitValue()
    return [exitValue:exitValue,error:error,success:success]
}
```

Fonte: (Imagem do próprio autor, 2016).

- Listar Imagens:

Figura 24 - Função Listar Imagens.

```
public static dockerImages() {
    Process p = DockerCommands.createProcess(DockerCommands.DOCKER_IMAGES)
    def listaResult = []

    final BufferedReader r = new BufferedReader(new InputStreamReader(p.getInputStream()))
    String line = r.readLine()
    while (line != null) {
        def tratado = line.split("\\|")
        listaResult.add([id: tratado[0].replaceAll("\\s", ""), repository: tratado[1].replaceAll("\\s", ""), tag: tratado[2].replaceAll("\\s", "")])
        line = r.readLine()
    }
    r.close()

    p.waitFor()
    p.destroy()
    return listaResult
}
```

Fonte: (Imagem do próprio autor, 2016).

- Remover imagem:

Figura 25 - Função Remover Imagem.

```
public static dockerRmi(containerID, force=false) {
    def listaComando = []
    listaComando.addAll(DockerCommands.DOCKER_RMI)
    if(force)
        listaComando.add("-f")
    listaComando.add(containerID)
    Process p = DockerCommands.createProcess(listaComando)
    def error = p?.errorStream?.text
    def success = p?.text
    p.waitFor()
    p.destroy()
    def exitValue = p.exitValue()
    return [exitValue: exitValue, error: error, success: success]
}
```

Fonte: (Imagem do próprio autor, 2016).

- Inspecionar container:

Figura 26 - Função Inspecionar Container.

```

public static inspect(containerID){
    def listaComando = []
    listaComando.addAll(DockerCommands.DOCKER_INSPECT)
    listaComando.add(containerID)
    Process p = DockerCommands.createProcess(listaComando)
    def result = p.text
    p.waitFor()
    p.destroy()
    return result
}

```

Fonte: (Imagem do próprio autor, 2016).

O esquema de rotas oferecido pelo Vert.X viabiliza a sua proposta à implementação no recebimento de requisições do sistema centralizador *DockerWeb GUI* por meio da chamada da função no navegador, através do registro de comandos *Docker* associado a cada função inserida na classe *DockerCommands* como aplicação cliente, apresentando os serviços e o seu resultado a geração de objetos, no caso os containers e imagens em formato JSON, caracterizando o uso da arquitetura REST. Abaixo segue como efetuou-se a implementação desse modelo, na classe principal do DockerWeb CLI.

Ao usar o route, inicialmente foi criado na classe Principal um servidor HTTP, e definido a porta para que assim o mesmo possa ser acessado pelo DockerWeb GUI, conforme a figura 27 e 28.

Figura 27 - Servidor HTTP Vert.X.

```

public class Principal extends AbstractVerticle{

    private final Logger logger = LoggerFactory.getLogger(this.getClass());

    @Override
    void start(){
        def server = vertx.createHttpServer()
        def router = Router.router(vertx)
    }
}

```

Fonte: (Imagem do próprio autor, 2016).

- Porta definida:

Figura 28 - Porta do Servidor HTTP.

```
server.requestHandler(router.&accept).listen(8081)
}
```

Fonte: (Imagem do próprio autor, 2016).

- Disponibiliza a listagem de containers a partir da chamada na URI - <http://ipdoservidor:porta/ps>, conforme a figura 29.

Figura 29 - Rota Listagem Containers.

```
router.route("/ps").handler({ routingContext ->
  routingContext.response()
    .putHeader("content-type", "application/json; charset=utf-8")
    .end(Json.encodePrettily(DockerCommands.dockerPS()))
})
```

Fonte: (Imagem do próprio autor, 2016).

- Disponibiliza a listagem de imagens a partir da chamada na URI - <http://ipdoservidor:porta/images>, conforme a figura 30.

Figura 30 - Rota Listagem Imagens.

```
router.route("/ps").handler({ routingContext ->
  routingContext.response()
    .putHeader("content-type", "application/json; charset=utf-8")
    .end(Json.encodePrettily(DockerCommands.dockerPS()))
})
```

Fonte: (Imagem do próprio autor, 2016).

- Remover imagem a partir da chamada na URI - <http://ipdoservidor:porta/images/rmi/:id>, conforme a figura 31.

Figura 31 - Rota Para Remover Imagens.

```
router.route("/images/rmi/:id").handler({ routingContext ->
  def id = routingContext.request().getParam("id")
  routingContext.response()
    .putHeader("content-type", "application/json; charset=utf-8")
    .end(Json.encodePrettily(DockerCommands.dockerRmi(id)))
})
```

Fonte: (Imagem do próprio autor, 2016).

- Remover container a partir da chamada na URI - <http://ipdoservidor:porta/images/rm/f/:id>, conforme a figura 32.

Figura 32 - Rota Para Remover Containers.

```
router.route("/rm/f/:id").handler({ routingContext ->
  def id = routingContext.request().getParam("id")
  routingContext.response()
    .putHeader("content-type", "application/json; charset=utf-8")
    .end(Json.encodePrettily(DockerCommands.dockerRm(id, true)))
})
```

Fonte: (Imagem do próprio autor, 2016).

- Parar container a partir da chamada na URI - <http://ipdoservidor:porta/stop/:id>, conforme a figura 33.

Figura 33 - Rota Parar Container.

```
public static stop(containerID){
  def listaComando = []
  listaComando.addAll(DockerCommands.DOCKER_STOP)
  listaComando.add(containerID)
  Process p = DockerCommands.createProcess(listaComando)
  def error = p?.errorStream?.text
  def success = p?.text
  p.waitFor()
  p.destroy()
  def exitValue = p.exitValue()
  return [exitValue:exitValue,error:error,success:success]
}
```

Fonte: (Imagem do próprio autor, 2016).

Iniciar container a partir da chamada na URI - <http://ipdoservidor:porta/start/:id>, conforme a figura 34.

Figura 34 - Rota Iniciar Container.

```
router.route("/start/:id").handler({ routingContext ->
  def id = routingContext.request().getParam("id")
  routingContext.response()
    .putHeader("content-type", "application/json; charset=utf-8")
    .end(Json.encodePrettily(DockerCommands.start(id)))
})
```

Fonte: (Imagem do próprio autor, 2016).

- Exibir informações de baixo nível do container a partir da chamada na URI - <http://ipdoservidor:porta/inspect/:id>, conforme a figura 35.

Figura 35 - Rota Inspeccionar Container

```
router.route("/inspect/:id").handler({ routingContext ->
  def id = routingContext.request().getParam("id")
  routingContext.response()
    .putHeader("content-type", "application/json; charset=utf-8")
    .end(DockerCommands.inspect(id))
})
```

Fonte: Imagem do próprio autor.

4.2.3 DOCKERWEB GUI

O *DockerWeb GUI* tem sua proposta em apresentar-se de forma a ser uma aplicação centralizadora que exibe e administra servidores instalados na infraestrutura de TI que portam o *DockerWeb CLI* e a tecnologia *Docker* dessa forma, proporcionando a comunicação entre o ambos para que o sistema *DockerWeb* funcione por completo. A implementação do *DockerWeb GUI* foi realizada por meio da linguagem *Groovy* versão: 2.4.5 e *framework Grails* na versão: 3.0.15, visando o desenvolvimento web, de forma ágil e com a menor curva de aprendizado para o programador Java.

Inicialmente implementou-se a classe de domínio para o objeto servidor, inserindo os atributos: IP e Porta, conforme a figura 36.

Figura 36 - Classe Domínio Servidor.

```
1 package br.com.servers

   class Servidor {

       String IP
       String porta

       static constraints = {
       }
   }
```

Fonte: (Imagem do próprio autor, 2016).

O *Grails* indica fortemente a aplicação da lógica em uma camada de serviço fornecida, pois quando se emprega todos os mecanismos na classe *controller* do projeto, não permite o seu reuso. Dessa forma foi criada um service com o nome de *ServidorService.groovy* conforme a figura 37 e 38, contendo as principais funcionalidades do sistema *DockerWeb*, importando os pacotes *grails.plugins.rest.client.RestBuilder*, *grails.plugins.rest.client.RestResponse*, possibilitando o desenvolvimento de um cliente em modelo Rest via http com o retorno em formato JSON. Em nível de serviço, foi implementada a listagem de

containers, listagem de imagens, exibição de dados de baixo nível de *containers*, parar *container*, iniciar container e remoção de *containers* e imagens, correspondentes a funções construídas no *DockerWeb CLI*.

Figura 37 - Camada Serviço Servidor.

```
package br.com.servers

import grails.transaction.Transactional
import grails.plugins.rest.client.RestBuilder
import grails.plugins.rest.client.RestResponse

@Transactional
class ServidorService {

    def listContainer(Servidor servidor) {
        RestBuilder rest = new RestBuilder()
        RestResponse resp = rest.get("http://${servidor.IP}:${servidor.porta}/ps")
        return resp.json
    }

    def listImage(Servidor servidor) {
        RestBuilder rest = new RestBuilder()
        RestResponse resp = rest.get("http://${servidor.IP}:${servidor.porta}/images")
        return resp.json
    }
}
```

Fonte: (Imagem do próprio autor, 2016).

Figura 38 - Camada Serviço Servidor 02.

```
def inspect(Servidor servidor, String containerID){
    RestBuilder rest = new RestBuilder()
    RestResponse resp = rest.get("http://${servidor.IP}:${servidor.porta}/inspect/${containerID}")
    return resp.json
}

def stop(Servidor servidor, String containerID){
    RestBuilder rest = new RestBuilder()
    RestResponse resp = rest.get("http://${servidor.IP}:${servidor.porta}/stop/${containerID}")
    return resp.json
}

def start(Servidor servidor, String containerID){
    RestBuilder rest = new RestBuilder()
    RestResponse resp = rest.get("http://${servidor.IP}:${servidor.porta}/start/${containerID}")
    return resp.json
}

def rm(Servidor servidor, String containerID){
    RestBuilder rest = new RestBuilder()
    RestResponse resp = rest.get("http://${servidor.IP}:${servidor.porta}/rm/${containerID}")
    return resp.json
}
}
```

Fonte: (Imagem do próprio autor, 2016).

A partir da camada de serviço foi implementada o *controller* *ServidorController.groovy* conforme a figura 39 e 40 a fim de proporcionar a sua execução e aplicar as funções já citadas acima, chamando as lógicas correspondentes já desenvolvidas no *ServidorService.groovy*.

Figura 39 -Controller Servidor.

```
package br.com.servers

import grails.converters.JSON

class ServidorController {

    static scaffold = Servidor
    def servidorService

    def ps(Servidor servidorInstance){
        def retorno = servidorService.listContainer(servidorInstance)
        render retorno as JSON
    }
    def images(Servidor servidorInstance){
        def retorno = servidorService.listImage(servidorInstance)
        render retorno as JSON
    }

    def inspect(){
        def servidor = Servidor.get(params.id)
        def containerID = params.containerID
        [result:servidorService.inspect(servidor,containerID),servidor:servidor]
    }

    def stop(){
        def servidor = Servidor.get(params.id)
        def containerID = params.containerID
        render servidorService.stop(servidor,containerID) as JSON
    }
}
```

Fonte: (Imagem do próprio autor, 2016).

Figura 40 -Controller Servidor 02.

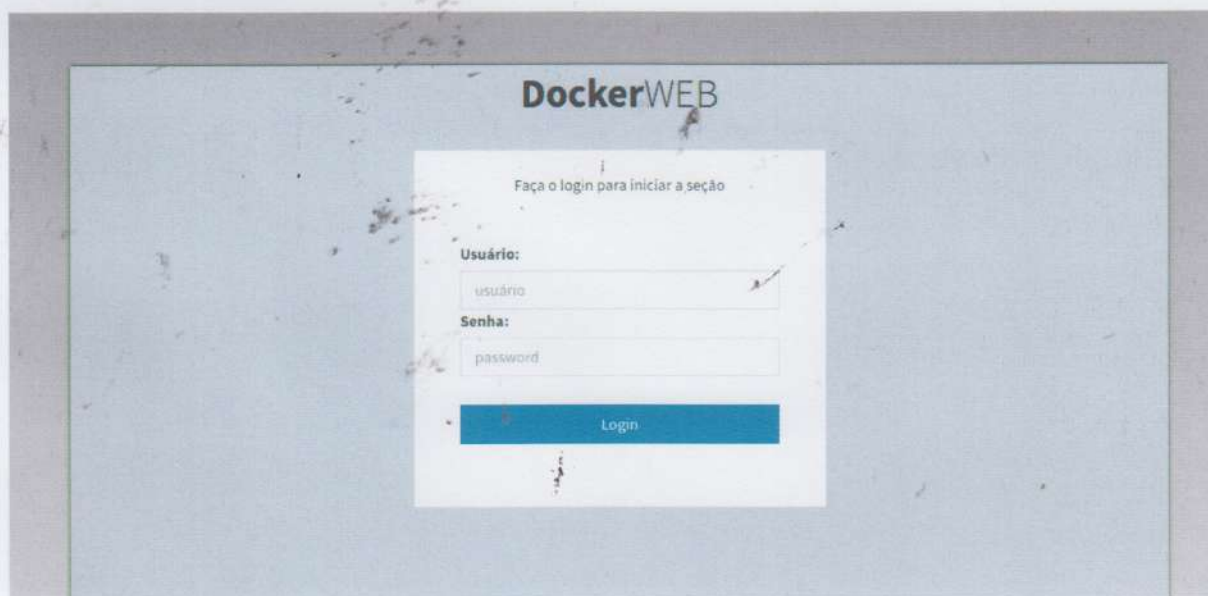
```
def start(){
  def servidor = Servidor.get(params.id)
  def containerID = params.containerID
  render servidorService.start(servidor,containerID) as JSON
}

def rm(){
  def servidor = Servidor.get(params.id)
  def containerID = params.containerID
  render servidorService.rm(servidor,containerID) as JSON
}
}
```

Fonte: (Imagem do próprio autor, 2016).

O acesso ao sistema *DockerWeb* é permitido através de um usuário pré-cadastrado com nome de usuário *admin* e senha *pass*, conforme a figura 41. O sistema de login tem em sua proposta o uso do banco de dados HSQLDB para o armazenamento de registros.

Figura 41 - Tela de Login.



DockerWEB

Faça o login para iniciar a seção

Usuário:
usuário

Senha:
password

Login

Fonte: (Imagem do próprio autor, 2016).

O tratamento e interação das funções implementadas nas camadas de serviço e *controller* do sistema, com o auxílio da tecnologia Vue.js, um *framework* Javascript reativo, facilitou a proposta do projeto a fim de alcançar resultados rapidamente, devido ao reaproveitamento de código e agilidade na resposta de ações do sistema, com a renderização somente de componentes modificados, tendo em vista o estado atual de uma última renderização realizada.

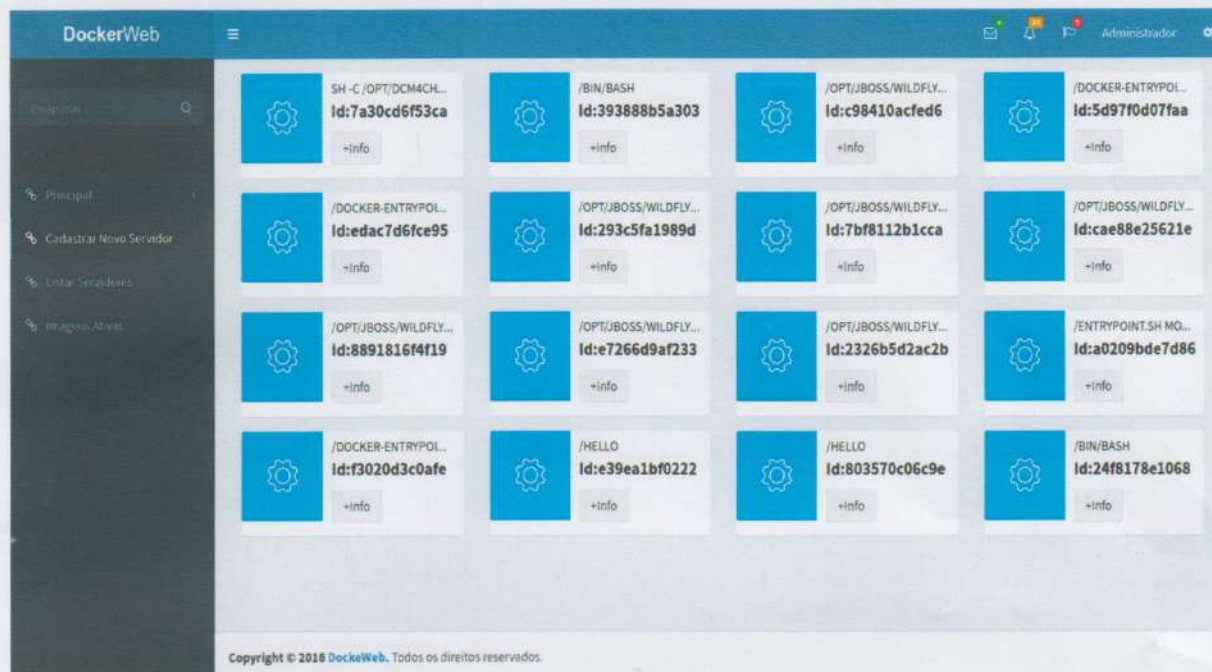
Foram então implementadas as principais funções também com o Vue.js, na pasta *assets* do projeto *grails*, e inserido o código no arquivo *application.js*. Conforme a tabela 2.

Tabela 2 - Tabela REST

Função	HTTP Verb	URL
Listar todos containers ativos e inativos.	GET	/servidor/ps
Listar Imagens.	GET	/servidor/images
Iniciar Container.	GET	/servidor/start/{containerID}
Parar Container.	GET	/servidor/stop/{containerID}
Remover Container.	GET	/servidor/rm/{containerID}
Remover Imagem.	GET	/servidor/rmi/{containerID}
Inspecionar Container.	GET	/servidor/inspect/{containerID}

Na construção da interface gráfica do *DockerWeb GUI*, foi definido o uso do template que faz uso do *bootstrap*, devido a ser um *framework* apropriado para criação de sites e aplicações web, utilizando HTML e CSS, com a oferta de um template responsivo, facilitando o acesso em dispositivos móveis de forma agradável. O template *bootstrap AdminLTE* versão 2, fornecido pela Almasaeed Studio, foi o escolhido para o *DockerWeb GUI*. No template os containers foram implementados na forma de boxes visto na figura 43, contendo a descrição da aplicação que está rodando e seu respectivo Id, contidos na tela do menu principal do sistema, conforme a figura 42.

Figura 42 - Menu Principal.



Fonte: (Imagem do próprio autor, 2016).

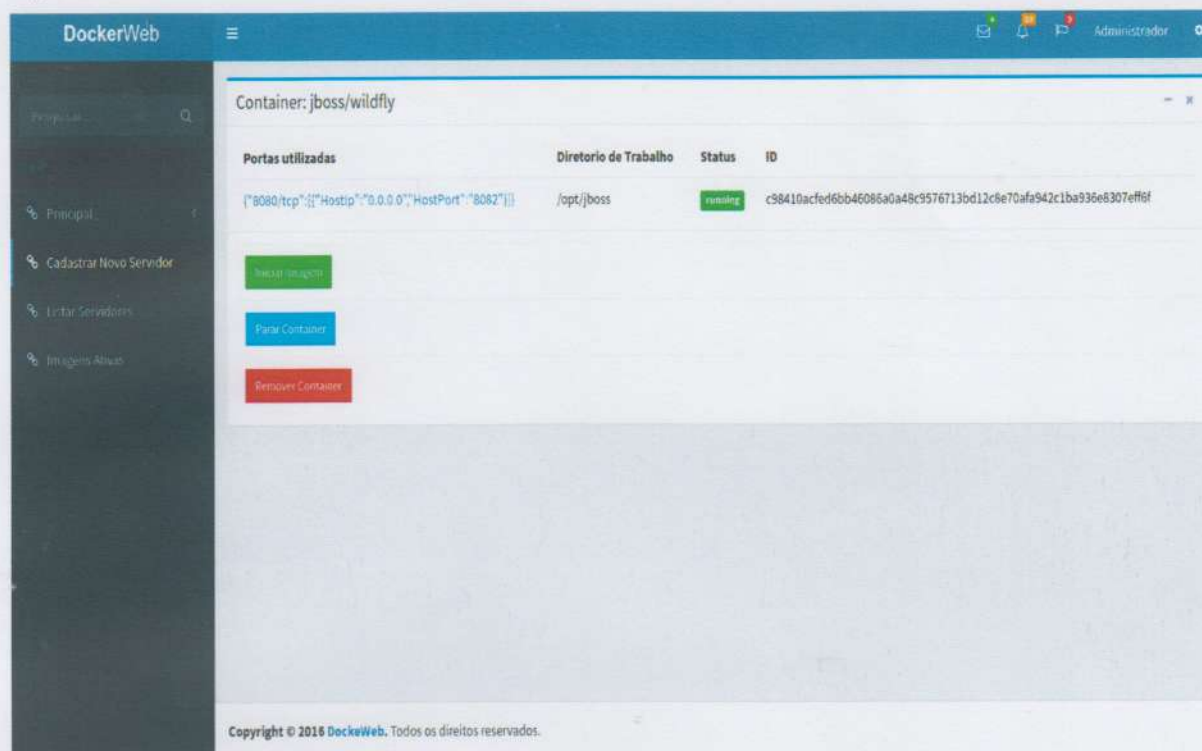
Figura 43 - Box Container.



Fonte: (Imagem do próprio autor, 2016).

Para realizar o acesso de informações do container e administrá-lo, é necessário clicar no botão info de cada box, onde terá como resposta a visão de geral do container com a apresentação de portas utilizadas, diretório de trabalho, status do container, e seu Id, conforme a figura 44 e 45.

Figura 44 - Info Container.



Fonte: (Imagem do próprio autor, 2016).

Figura 45 - Info Container 02.

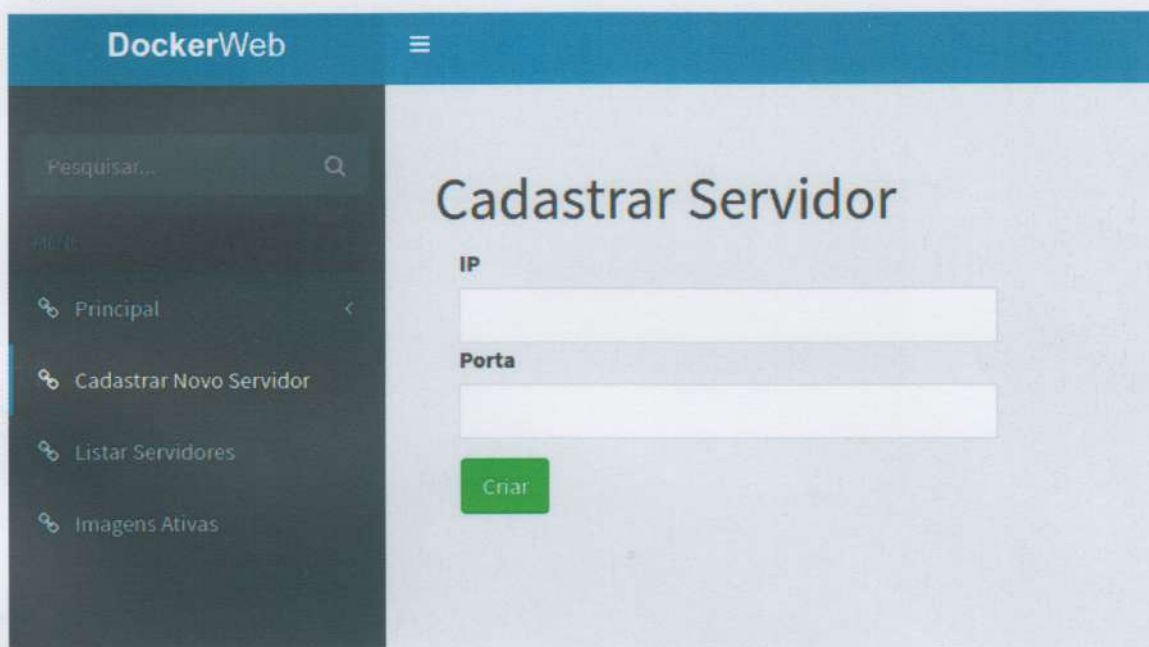


Fonte: (Imagem do próprio autor, 2016).

O cadastramento de servidores é executado clicando em Cadastrar Novo Servidor no menu da barra lateral esquerda, após a escolha da opção, somente é preciso o preenchimento do endereço IP e porta correspondente ao servidor,

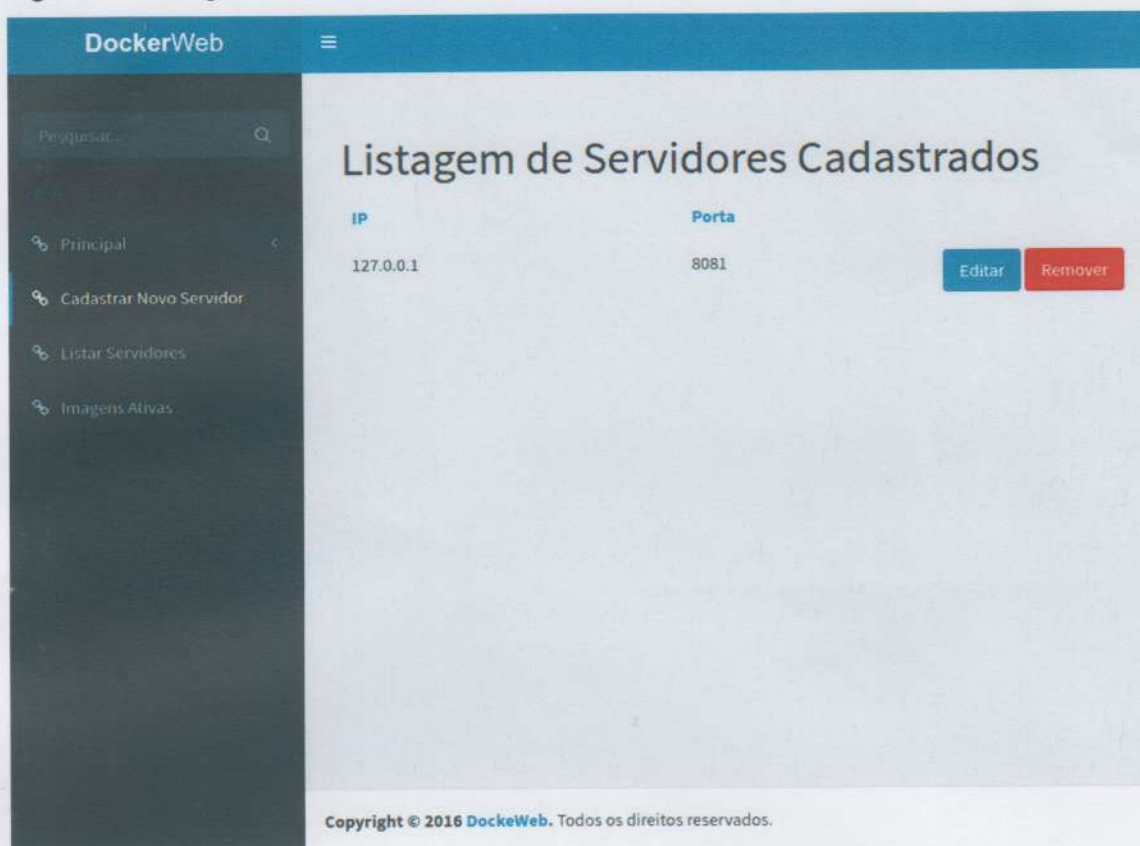
permitindo o DockerWeb GUI administrar os containers instalados no mesmo registrado, conforme a figura 46.

Figura 46 - Cadastro de Servidor.

The image shows a web interface for DockerWeb. At the top, there is a blue header with the text 'DockerWeb' and a hamburger menu icon. Below the header, on the left, is a dark sidebar with a search bar labeled 'Pesquisar...' and a list of menu items: 'Principal', 'Cadastrar Novo Servidor' (highlighted), 'Listar Servidores', and 'Imagens Ativas'. The main content area has a light gray background and is titled 'Cadastrar Servidor'. It contains two input fields: 'IP' and 'Porta'. Below these fields is a green button labeled 'Criar'.

Fonte: (Imagem do próprio autor, 2016).

Para a verificação de possíveis mudanças de valores ou remoção de servidores já cadastrados o usuário necessita apenas acessar o menu lateral esquerdo e clicar em Listar Servidores, e então será apresentada uma listagem de servidores, com os botões editar para alteração de endereço IP ou porta, e botão remover, para deletar o servidor do *DockerWeb GUI*, conforme a figura 47 e 48.

Figura 47 - Listagem de Servidores.

Fonte: (Imagem do próprio autor, 2016).

Figura 48 - Editar Servidor.

Fonte: (Imagem do próprio autor, 2016).

5. CONSIDERAÇÕES FINAIS

Por meio do processo de desenvolvimento do *software DockerWeb*, foi observado os desafios em construir uma aplicação distribuída em rede, com uso do modelo REST, e oferecer o serviço da tecnologia *Docker* em formato Web foi um dos principais motivadores, além da percepção no aumento do aprendizado em *frameworks Grails* e *Vert.X*, que viabilizam e agilizam a construção da solução.

Sugerindo-se como alternativa aos desafios propostos para instituições, quanto a adoção de meios que focam na diminuição dos gastos com hardware e energia, devido ao menor número de máquinas servidoras adquiridas no parque de TI atendendo a demanda requisitada de forma equivalente ou superior.

Como resultado das etapas de elaboração do *DockerWeb*, nota-se a disponibilização do serviço de gerenciamento de *containers docker*, de forma web de fácil acesso, composto pelas principais funcionalidades *docker*, dispondo de ações rápidas, que comparadas com a situação atual de acesso a *containers* demandam mais tempo e controle, passíveis as falhas devido ao não se ter uma visão ampla dos *containers docker* implantados no ambiente de TI.

REFERÊNCIAS

- AGARD, Hellen. **Estudo das Principais Características e Ferramentas com Vmware ESX Server**. Campinas. 2009. Disponível em <<http://lyceumonline.usf.edu.br/salavirtual/documentos/1677.pdf>>. Consultado em 14 de set. 2016.
- ALBUQUERQUE, Fernando. **Banco de Dados**. UNB, 2000. Disponível em <<http://www.cic.unb.br/~fernando/matdidatico/apostilas/resumo/bdadosp.pdf>>. Consultado em 10 de out. 2016.
- ALECRIM, Emerson. **O que é virtualização e para que serve?**. Infowester. agosto de 2012. Disponível em <<http://www.infowester.com/virtualizacao.php#vantagens>>. Consultado em 11 de set. 2016.
- ARCHLINUX. **Change root**. 2016. Disponível em <https://wiki.archlinux.org/index.php/change_root>. Consultado em 16 de set. 2016.
- ARCHLINUX. **Linux Containers**. 2015. Disponível em <[https://wiki.archlinux.org/index.php/Linux Containers \(Portugu%C3%AAs\)](https://wiki.archlinux.org/index.php/Linux_Containers_(Portugu%C3%AAs))>. Consultado em 16 de set. 2016.
- CACIATO, Luciano Eduardo. **Virtualização e Consolidação dos Servidores do Datacenter**. UNICAMP. 2010. Disponível em <http://www.ccuac.unicamp.br/bitl/download/Artigo_Virtualizacao_Datacenter.pdf>. Consultado em 11 de set. 2016.
- CARISSIMI, Alexandre. **Virtualização: da teoria a soluções**. Porto Alegre. 2008. Disponível em <http://hostel.ufabc.edu.br/~marcelo.nascimento/BC1518Q3/arquivos/virtualizacao_c_ap4-v2.pdf>. Consultado em 11 de set. 2016.
- FRANCO, Alessandro. **Analizando a utilização da virtualização em laboratório de ensino**. Cascavel. 2010. Disponível em <<http://www.inf.unioeste.br/~tcc/2010/TCC-Alessandro%20Franco.pdf>>. Consultado em 14 de set. 2016.
- GRAILS. **QuickReferenceGrails - version: 3.2.3**. Disponível em <<http://docs.grails.org/latest/guide/introduction.html>>. Consultado em 10 de out. 2016.
- HSQldb. **HSQldb - 100% Java Database**. Disponível em <<http://hsqldb.org/>>. Consultado em 11 de out. 2016.
- INDRUSIAK, Leandro. **Linguagem Java**. Rio Grande do Sul - 1996. Disponível em <<http://www.cin.ufpe.br/~arfs/introjava.pdf>>. Consultado em 02 de out. 2016.
- NUNES, Sérgio; DAVID, Gabriel. **Uma Arquitetura Web para Serviços Web**. Universidade do Porto, Porto - Portugal. 2005. Disponível em <<https://repositorio-aberto.up.pt/bitstream/10216/2812/51002.pdf>>. Consultado em 11 de out. 2016.
- PARKUTS, Eliésio; DUFECH, Saulo; ORLOVSKI, Regiane. **Virtualização - Conceitos e Aplicações**. Guarapuava. 2013. Disponível

em <http://semanaacademica.com.br/system/files/artigos/tcc_pronto.pdf>. Consultado em 14 de set. 2016.

POLLON, Vanderlei. **Virtualização de servidores em ambientes heterogêneos e distribuídos - estudo de caso**. Porto Alegre. outubro de 2007. Disponível em <<http://www.lume.ufrgs.br/bitstream/handle/10183/15988/000695318.pdf>>. Consultado em 09 de set. 2016.

SALLES, Ana Carolina; ALVES, Ana Paula; DOLCI, Décio; LUNARDI, Guilherme. **Adoção de Práticas de TI Verde nas Organizações: Um Estudo Baseado em Mini Casos**. Bento Gonçalves. 2013. Disponível em <http://repositorio.furg.br/bitstream/handle/1/5373/2013_EnADI145.pdf?sequence=1>. Consultado em 11 de set. 2016.

SAUDATE, Alexandre. **REST: Construa API's inteligentes de maneiras simples**. São Paulo: Casa do Código, 2014.

VERT.X. **Vert.X is a tool-kit for building reactive applications on the JVM**. Disponível em <<http://vertx.io/>>. Consultado em 10 de out. 2016.

VILLELA, Karina; TRAVASSOS, Guilherme; ROCHA, Ana Regina. **Definição e Construção de Ambientes de Desenvolvimento de Software Orientados a Organização**. Rio de Janeiro. 2004. Disponível em <<http://www.lbd.dcc.ufmg.br/colecoes/sbqs/2004/032.pdf>>. Consultado em 02 de out. 2016.

VMWARE. **O que é virtualização?**. Disponível em <<http://www.vmware.com/br/solutions/virtualization.html>>. Consultado em 12 de set. 2016.

VMWARE. **TryVmware Workstation Player**. Disponível em <<http://www.vmware.com/products/player/playerpro-evaluation.html>>. Consultado em 12 de set. 2016.

WEISSMAN, Henrique. **Falando de Grails - Altíssima produtividade no desenvolvimento web**. São Paulo: Casa do Código, 2015.