

SERVIÇO NACIONAL DE APRENDIZAGEM INDUSTRIAL
FACULDADE DE TECNOLOGIA SENAI/SC FLORIANÓPOLIS
CURSO SUPERIOR DE TECNOLOGIA EM ANÁLISE E DESENVOLVIMENTO DE
SISTEMAS

FELIPE MENDONÇA RUHLAND

ESTUDO SOBRE O USO DO DOCKER NA EXECUÇÃO DE APLICAÇÕES WEB

Florianópolis/SC

2016

FELIPE MENDONÇA RUHLAND

ESTUDO SOBRE O USO DO DOCKER NA EXECUÇÃO DE APLICAÇÕES WEB

Trabalho de Conclusão de Curso apresentado à Banca Examinadora do Curso Superior de Tecnologia em Análise e Desenvolvimento de Sistemas da Faculdade de Tecnologia do SENAI Florianópolis como requisito parcial para obtenção do Grau de Tecnólogo em Análise e Desenvolvimento de Sistemas.

Professor Orientador: Paulo Bueno.

Florianópolis/SC

2016

FELIPE MENDONÇA RUHLAND

ESTUDO SOBRE O USO DO DOCKER NA EXECUÇÃO DE APLICAÇÕES WEB

Trabalho de Conclusão de Curso apresentado à Banca Examinadora do Curso Superior de Tecnologia em Análise e Desenvolvimento de Sistemas da Faculdade de Tecnologia do SENAI Florianópolis como requisito parcial para obtenção do Grau de Tecnólogo em Análise e Desenvolvimento de Sistemas.

**APROVADA PELA COMISSÃO EXAMINADORA
EM FLORIANÓPOLIS, ?? DE JULHO DE 2016**

Prof. Bobiquins Estevão de Mello, Me. (SENAI/SC)
Coordenador do Curso

Profa. Jaqueline Voltolini de Almeida, Me. (SENAI/SC)
Coordenador de TCC

Prof. Paulo Bueno, Dr. (SENAI/SC)
Orientador

Prof. Fulado de tal, Me. (SENAI/SC)
Examinador

Dedico este trabalho ao meu avô Huri Gomes Mendonça, *in memoriam*, que sempre me apoiou nos estudos mesmo sem entender o contexto do curso.

AGRADECIMENTOS

À minha família, À minha noiva, Aos meus amigos e Aos colegas de trabalho pela ajuda e pelo apoio de sempre.

“Talk is cheap. Show me the code”
(LINUS TORVALDS)

Ruhland, Felipe. **Estudo sobre container linux para execução de aplicações web**. Florianópolis, 2016. 39f. Trabalho de Conclusão de Curso Superior de Tecnologia em Análise e Desenvolvimento de Sistemas - Curso Análise e Desenvolvimento de Sistemas. Faculdade de Tecnologia do SENAI, Florianópolis, 2016.

RESUMO

Atualmente, milhões de aplicações estão rodando na nuvem, estejam elas em smartphones, tablets ou mesmo na web. Praticamente todo aplicativo tem um motor rodando na nuvem, que demanda de mão de obra especializada para a manutenção, evolução e monitoramento dessas aplicações. A nuvem é praticamente ocupada por máquinas virtuais que são disponibilizadas para terceiros rodarem suas aplicações. É provável que este cenário mude, pois os containers linux estão tomando espaço nos ambientes de produção e devem brigar com as máquinas virtuais pela maioria das execuções na nuvem. O Docker é uma ferramenta que busca facilitar a criação de containers linux para melhorar a forma que as aplicações são desenvolvidas, testadas e distribuídas. Outro fator para adotar o uso do Docker é em razão da economia de recurso computacional que é possível atingir. Com a utilização de uma arquitetura de microserviços, é possível utilizar os recursos da máquina de maneira mais inteligente e apropriada, com a possibilidade de ampliar as processos que precisem de mais recursos. Desta forma, entende-se que este assunto é muito atual e importante para o futuro do software em nuvem e que é provável que seja a maioria dos casos num futuro próximo.

Palavras-chave: Docker. Container. Aplicação web.

Ruhland, Felipe. **Estudo sobre container linux para execução de aplicações web**. Florianópolis, 2016. 39f. Trabalho de Conclusão de Curso Superior de Tecnologia em Análise e Desenvolvimento de Sistemas - Curso Análise e Desenvolvimento de Sistemas. Faculdade de Tecnologia do SENAI, Florianópolis, 2016.

ABSTRACT

Today, millions of applications are running in the cloud, on smartphones, tables or even on the web. Practically every application has an engine running on the cloud that labor demand for specialized to maintenance, development and monitoring of these applications. The cloud is occupied by virtual machines that are available to developers runs their applications. It is likely that this scenario change because the linux containers as taking space in production environments and must fight with virtual machines for most runs in the cloud. The Docker is a tool to seek to facilitate the creation of Linux containers to improve the way applications are developed, tested and distributed. Another factor to adopt the use of Docker is because of the computational resource savings that can be achieved. With the use of a microserviços architecture, it is possible to use resources more intelligently and appropriately machine, with the possibility of expanding the processes that need more resources. Thus, it is understood that this issue is very current and important for the future of cloud software and that is likely to be the most cases in the near future.

Key-words: Docker. Container. Web Application.

LISTA DE ABREVIATURAS E SIGLAS

CGroups	Control Groups
LXC	Linux Container
VPS	Servidores privados virtuais
TI	Tecnologia da Informação

SUMÁRIO

1 INTRODUÇÃO	11
1.1 JUSTIFICATIVA	12
1.2 OBJETIVOS	12
1.2.1 Objetivo geral	12
1.2.2 Objetivos específicos	12
1.3 METODOLOGIA	13
1.4 ESTRUTURA DO TRABALHO	13
2 REVISÃO DA LITERATURA	14
2.1 Introdução ao Docker	14
2.1.1 O que é Docker?	14
2.1.2 O que Docker não é?	14
2.1.3 Quem criou Docker?	14
2.1.4 O que são containers	15
2.1.5 Containers vs máquinas virtuais	16
2.2 Funcionamento do Docker	16
2.2.1 Arquitetura Docker	16
2.2.2 Control groups	17
2.2.3 Namespaces	17
2.2.4 Volumes	18
2.2.4.1 Volume vinculado ao host	18
2.2.4.2 Volume gerenciador pelo Docker	19
2.2.5 Camada de Rede	19
2.2.5.1 Container fechado	19
2.2.5.2 Container em ponte	20
2.2.5.3 Container unido	20
2.2.5.4 Container aberto	20
3 PROCEDIMENTOS METODOLÓGICOS	22
4 RESULTADOS E DISCUSSÕES	23
4.1 Problemas comuns na rotina de desenvolvimento de software	23
4.1.1 Ambiente de Desenvolvimento	23
4.1.2 Processo de integração	23
4.1.3 Ambiente de Homologação	24
4.1.4 Distribuição do software	24
4.1.5 Instalação do Software	24
4.1.6 Escalabilidade	25

4.1.7 Entregabilidade	25
4.1.8 Segurança	25
5 CONCLUSÃO	27
5.1 Vantagens em executar aplicações em containers	27
5.1.1 Imagens Docker	27
5.1.2 Dependências de desenvolvimento	28
5.1.3 Composição de containers	29
5.2 Distribuição	29
5.3 Publicação	29
5.4 Estudo de caso	30
5.4.1 Desenvolvimento	30
5.4.2 Integração Contínua	31
5.4.3 Ambiente de Desenvolvimento	31
5.4.4 Ambiente de Produção	31
5.4.5 Considerações finais	32
REFERÊNCIAS	34
APÊNDICE A Código fonte	36

1 INTRODUÇÃO

Mais de dois terços das aplicações web rodam em ambiente unix, segundo W3Techs (2016). Os sistemas são executados em servidores dedicados ou virtualizados. Os servidores dedicados são a maneira mais natural de servir uma aplicação web. São máquinas físicas, normalmente em datacenters, com um sistema operacional linux e com todo o hardware à sua disposição. Entretanto, executar uma aplicação web em um servidor dedicado acaba por desperdiçar muitos recursos do mesmo. Para combater este desperdício, trabalha-se há décadas para aperfeiçoar um servidor que consiga evitar o desperdício, com a divisão dos recursos.

Nos últimos anos, usou-se muito as máquinas virtuais para aproveitar melhor os recursos dos servidores. Elas funcionam como novas máquinas dentro da máquina física e podem ser incluídas na rede como se fossem uma máquina física. Desta maneira, um servidor dedicado passa a ser múltiplas máquinas, com seus próprios recursos e totalmente isoladas, que traz ainda mais segurança para os administradores de sistemas.

Com esta estratégia, os VPS (Servidores privados virtuais) tornaram muito mais acessíveis ao público em geral, pois era possível contratar um pequeno servidor virtualizado e ter total controle das configurações desde servidor. Isso colaborou com pequenos empreendedores que puderam expor seu trabalho de maneira mais econômica e, com isso, abrir um leque de possibilidades para novas empresas.

A máquina virtual, por padrão, funciona como uma máquina totalmente nova. Ou seja, ela possui sistema operacional próprio, permissões individuais e recursos compartilhados com a máquina anfitriã. É possível instalar diversas versões do kernel, por exemplo, sem que uma interfira na outra. Contudo, observa-se que para cada máquina virtual criada num servidor dedicado existe uma sobrecarga do sistema operacional, pois além do sistema instalado na máquina anfitriã, cada máquina virtual possui seu sistema individual. Sabe-se, também, que o sistema operacional necessita de uma série de recursos para o bom funcionamento, de maneira que os recursos não podem ser muito ínfimo para não ocorrer problemas. Em paralelo às máquinas virtuais, existe uma outra alternativa, mais leve, chamada de container.

Containers linux existem com a finalidade de isolar ambientes dentro de um sistema operacional linux. Eles não são máquinas virtuais, mas também conseguem restringir acesso, definir recursos próprios, mas não sobrepõe o sistema operacional. Ele usa o sistema da máquina anfitriã e, com isso, consegue utilizar menos recursos que a máquina virtual. Em razão dessas vantagens, muitos estudam para deixar a criação e manutenção desses containers uma tarefa mais fácil para os profissionais de ti. O caso mais conhecido nos últimos anos é da ferramenta Docker, que descomplicou a maneira de criar e gerir containers dentro de um sistema operacional. Pode-se dizer que o projeto é um sucesso, pois recebe mais utilizadores a cada dia e já possui

investimentos na casa dos milhões de dólares.

Fala-se muito na arquitetura de microserviço, que traz vantagens para o desenvolvedor, por ter um escopo reduzido, facilita a criação e execução de testes, deploy e inúmeros outros fatores. Essa nova abordagem, segundo Fowler (2016b), traz consigo a ideia de executar o microserviço em um container para simplificar a infraestrutura, de modo a facilitar a escalabilidade da aplicação e a sua manutenção. Com este pensamento, pode-se criar milhares de containers com a mesma aplicação, em ambientes isolados, independentes e seguros.

Este projeto tem por objetivo fazer um estudo sobre a tecnologia de containers linux para a execução de aplicações web, em especial o Docker.

1.1 JUSTIFICATIVA

Em razão do Docker, tem-se discutido muito o assunto de containers para execução de aplicações web, com muitos olhares positivos e muita adesão. Acredita-se que é um assunto muito relevante, pois já chamou a atenção dos gigantes da TI, como Google, Red Hat e Microsoft. Já existem inúmeros serviços que utilizam a ideia de container para execução de aplicações e muitas empresas já confiam neste conceito. Inclusive, a grande justificativa que se traz, a princípio, é a vantagem de ter uma aplicação que roda da mesma forma em desenvolvimento, testes, integração e produção. Não existe mais a desculpa que o ambiente não estava identico, pois agora, o ambiente é reduzido a um container linux.

Conforme descrito, os estudos focam a solução Docker, por ser o grande responsável pelo assunto atualmente e por ter chamado tanta atenção dos profissionais de TI, uma vez que o Docker é bem quisto por todas as etapas do desenvolvimento de software.

1.2 OBJETIVOS

Nesta seção são apresentados os objetivos do presente trabalho.

1.2.1 Objetivo geral

Estudo da solução de container linux para a execução de aplicações web.

1.2.2 Objetivos específicos

1. Introdução ao Docker
2. Estudo do funcionamento do Docker
3. Vantagens em executar aplicações em container

4. Exemplos de utilização

1.3 METODOLOGIA

Pretende-se fazer uso da modalidade de pesquisa experimental, com a análise e aplicações das técnicas propostas e a elaboração de um comparativo com o resultado obtido. A pesquisa será realizada de maneira descritiva, na qual os fatos serão analisados, interpretados e comparados para elaboração de uma conclusão.

Os procedimentos utilizados serão de pesquisa bibliográfica, para o melhor entendimento das técnicas, e de maneira experimental, pois exigirá a realização de vivência prática nos containers apresentados, com a sua respectiva resposta. Planeja-se estudar a ferramenta Docker e seus benefícios em todas as etapas de desenvolvimento de software.

1.4 ESTRUTURA DO TRABALHO

O presente trabalho foi estruturado em três capítulos. O primeiro capítulo explica a ferramenta Docker desde a sua motivação, criação e atual estado. O segundo capítulo apresenta as dificuldades encontradas pelas equipes de desenvolvimento de software e, por fim, o terceiro capítulo apresenta uma conclusão prática após o uso de containers docker para realizar um grande projeto de software e as observações cabíveis.

2 REVISÃO DA LITERATURA

2.1 Introdução ao Docker

2.1.1 O que é Docker?

Docker é uma ferramenta de linha de comando, que é executada em plano de fundo, e promove um servidor remoto para simplificar a experiência de instalar, executar, publicar e remover software, segundo Nickoloff (2016, p.6). Possibilita que um software seja posto em um container, junto com suas dependências, em uma unidade padrão de desenvolvimento de software, conforme Docker (2016d). Desta forma, pode-se garantir que o software sempre vai se comportar da mesma maneira, independente do ambiente que for executado.

Matthias e Kane (2015, p. 6) explicam que é bastante difícil compreender o Docker, mas que facilita o entendimento depois de estudá-lo melhor.

2.1.2 O que Docker não é?

Matthias e Kane (2015, p. 5) trouxeram um subcapítulo muito interessante sobre o que não é Docker. Eles destacam que muitos vão tentar utilizar o Docker para sanar outras deficiências, como gerenciamento de configuração, por exemplo. Entretanto, Docker pode ajudar em muitos aspectos, mas ele nunca será um gerenciador de configurações. Enfim, o que Docker não é? Uma plataforma de virtualização, como VMware, KVM ou virtualbox; uma plataforma para nuvem, como cloudstack ou openstack; Gestor de configuração, como puppet ou chef, apesar do Docker simplificar bastante o trabalho de configuração; um framework para deploy, como capstrano ou fabric; ferramenta para orquestração, como fleet ou mesos; um ambiente de desenvolvimento, como vagrant, apesar do Docker facilitar a composição do ambiente de desenvolvimento.

2.1.3 Quem criou Docker?

Em 2008, Solomon Hykes fundou a empresa dotCloud para construir uma plataforma como serviço que fosse independente de linguagem. Outras plataformas como Heroku e Google App Engine tinham restrições de linguagem para rodar as aplicações, como Java, Python e Ruby.

A dotCloud participou do programa de aceleração do Y Combinator em 2010, época que ficou em contato com novos parceiros e possibilidade de atrair grandes investimentos, conforme Mouat (2015, p.8). Entretanto, a maior virada ocorreu em março de 2013, quando o Docker foi lançado como um projeto de código aberto pela dotCloud.

Em pouco tempo, a ferramenta caiu nos braços da comunidade de desenvolvedores. Grandes empresas de tecnologia como Red Hat, IBM, Google e Cisco, contribuem no desenvolvimento do produto, de acordo com TechTarget (2016).

As primeiras versões do Docker eram um embrulho na biblioteca LXC, porém com grandes resultados de estabilidade e performance. Empresas como Spotify e Red Hat passaram a adotar a tecnologia para seus produtos, de acordo com Mouat (2015, p.9).

O ano de 2014 foi um ano muito especial para o Docker, pois foi o ano que recebeu grandes investimentos da Greylock Partners e Sequoia Capital e passou a ter um valor de mercado em US\$ 400M (quatrocentos milhões de dólares). Em 2015, não foi muito diferente. Mais duas rodadas de investimentos ocorreram, que totalizou US\$ 180M (Cento e oitenta milhões de dólares), conforme CrunchBase (2016).

Entre as características marcantes da ferramenta pode-se considerar a leveza, pois compartilha recursos do sistema operacional; abertura, pois pode ser executado na grande maioria dos servidores linux e com versões instáveis para OSX e Windows; seguro, pois o container promove mais uma segunda camada protetora para o ambiente, segundo Docker (2016d).

2.1.4 O que são containers

Nos sistemas unix era comum a expressão *jail* para descrever um ambiente isolado que previnha o acesso a outros recursos do sistema, segundo Nickoloff (2016, p. 4). Somente em 2001, por meio do Solares da Sun, fez-se referência à palavra container para explicar este ambiente isolado, mesmo objetivo do *jail*. Ainda em 2001, a Parralles Inc lançou uma solução comercial chamada Virtuozzo, que em 2005 foi aberto o código do projeto com o nome OpenVZ. Em 2008 o projeto Linux Container (LXC) trouxe consigo CGroups, namespaces e a tecnologia chroot para promover uma solução completa com container.

Finalmente, em 2013, Docker foi lançado e com ele a grande adoção por parte dos desenvolvedores da tecnologia de containers, como conta Mouat (2015, p.3).

Num estudo preliminar, nota-se que o container é mais leve que uma máquina virtual, pois compartilha o sistema operacional para rodar as aplicações. Para completar, ainda elenca outras diferenças entre as máquinas virtuais como o compartilhamento de recursos com a máquina anfitriã, podem ser ligadas e desligadas numa fração de segundos; a portabilidade de ambientes evita os bugs originados em decorrência dessa diferença de ambiente.

A leveza dos containers possibilitam que sejam executados centenas de containers simultâneos, bem como uma configuração muito próxima da executada em produção e trazem uma gama de vantagens aos desenvolvedores de software que trabalham com equipes diferentes, de modo que para executar projetos dependentes, basta executar o container pretendido, sem que haja necessidade de configuração exaustiva.

Em resumo, containers são aplicações encapsuladas com suas dependências.

2.1.5 Containers vs máquinas virtuais

A documentação oficial exalta que um deploy em uma máquina virtual inclui uma aplicação, as dependências necessárias (binários e bibliotecas) e um sistema operacional, que pode ter um tamanho considerável de alguns Gigabytes. Ainda conforme Docker (2016d), containers inclui a aplicação e suas dependências (binários e bibliotecas), porém compartilha o kernel com outros containers e mantém-se num *userspace* isolado do sistema operacional. Além de ser independente de infraestrutura.

Por definição, a máquina virtual tem um caráter de longevidade. Ela tem por objetivo abstrair os servidores físicos para dar mais flexibilidade na publicação de aplicações. Matthias e Kane (2015, p.15) ensinam que mesmo as máquinas virtuais em serviços de nuvem, tem por natureza um caráter de longa duração, pois o custo de ligar e desligar uma máquina é muito grande.

Em contra partida, um container pode ser usado para uma atividade de segundos de duração e seu custo de vida é muito mais econômico, em comparação com a máquina virtual.

De acordo com Nickoloff (2016, p.5), máquinas virtuais promovem um hardware virtual e pode levar minutos para inicializar e estabelecer os recursos necessários para rodar um cópia de um sistema operacional.

O mesmo Nickoloff (2016, p.5) ensina que diferentemente das máquinas virtuais, os containers não usam hardware virtuais, pois as aplicações que rodam em container utilizam a interface do kernel linux da máquina anfitriã. Por isso, não existe nenhuma camada entre a aplicação e a máquina anfitriã e nenhum recurso é desperdiçado. Deve-se lembrar que Docker não é uma tecnologia de virtualização e sim uma ferramenta para criar, gerenciar e remover containers.

2.2 Funcionamento do Docker

2.2.1 Arquitetura Docker

Ressalta-se a importância de conhecer a estrutura do Docker para obter um entendimento pleno do funcionamento da ferramenta e as vantagens que ela pode trazer. O Docker engine é responsável pelo serviço docker, que atua em background e gerencia a criação, manutenção e remoção dos containers; a construção e armazenamento das imagens, e pelo cliente de linha de comando que se comunica com o serviço por http. O cliente docker é, normalmente, o primeiro passo para aprender a utilizar a ferramenta, como explica Mouat (2015, p.36).

Existe também o Docker Registry, que é responsável por gerenciar e armazenar as imagens oficiais e das organizações. O registry oficial é o Docker Hub e, por padrão, todo serviço docker utiliza este registry para buscar as imagens para criar os containers. Empresas costumam ter seu próprio registry para gerenciar suas imagens e permitir a distribuição das mesmas.

Para a criação dos containers, Docker utiliza a biblioteca runc, que também foi desenvolvida pelo Docker. Para gerenciar os recursos do container, utiliza-se de *cgroups* para controlar o uso do processamento, da memória e restringir o acesso aos periféricos, segundo Nickoloff (2016, p.123). Por outro lado, com a finalidade de garantir o isolamento, utiliza-se *namespaces* para garantir que o sistema de arquivos, rede e processos são completamente separados da máquina anfitriã, como leciona Mouat (2015, p.37).

2.2.2 Control groups

Segundo KernelOrg (2016), control groups promovem um mecanismo para agregar e particionar conjuntos de tarefas e seus futuros filhos em um grupo hierárquico com comportamento especializado. Em outras palavras, control groups permitem a definição de limites em recursos para processos e seus dependentes, que é utilizado pelo Docker para controlar o uso de recursos como processamento, memória, leitura em disco e entrada e saída de rede.

É com o control group que o Docker consegue expor as estatísticas de seus containers. Os control groups são uma feature do kernel linux e não existe a necessidade de utilizar o Docker para implementá-lo, entretanto a ferramenta abstrai o comportamento dos control groups para facilitar seu uso.

CloudSigma (2016) explica que se houverem muitos containers na mesma máquina, deve-se fazer uso dos control groups para gerenciar o uso de recurso entre os containers para que sejam priorizados os serviços mais importantes.

2.2.3 Namespaces

Dentro de cada container, verifica-se um sistema de arquivos, interfaces de rede, discos e outros recursos que parecem ser únicos do container, com exceção do kernel compartilhado. Para um container, a interface de rede parece ser realmente singular e exclusiva, como se fosse de um servidor normal. Este comportamento só é permitido graças aos namespaces, que usa um recurso global e faz parecer que ele é exclusivo de um container, como explicam Matthias e Kane (2015, p.161).

Os containers criados pelo Docker, são isolados em diferentes aspectos, como: PID (Process identifiers and capabilities), UTS (Host and domain name), MNT (File system access and structure), IPC (Process communication over shared memory), NET (Network access and structure) e USR (User names and identifiers).

Segundo Nickoloff (2016, p.6), sem um PID namespace, o processo executado dentro de um container compartilharia o id de espaço com outros processos em execução na máquina anfitriã. Isso arriscaria o isolamento e permitiria que processos executados dentro de um container tomassem controle de outros processos na máquina anfitriã.

Para novos containers, é criado um namespace com um ponto de montagem com intuito de isolar o sistema de arquivos. O namespace IPC previne que processos acessem dados na memória de outro container. Ainda que não tenha sido implementado pelo Docker, o namespace USR tem por objetivo permitir que um usuário na máquina anfitriã possua as mesmas permissões num containers, se assim for feito, conforme Nickoloff (2016, p.112).

2.2.4 Volumes

Por padrão, os containers são criado com caráter imutável, segundo Nickoloff (2016, p.18). Matthias e Kane (2015, p. 66) ensinam que os containers possuem armazenamento de natureza efêmera e o espaço em disco alocado dificilmente é adequado para os projetos. Para solucionar estes problemas, existem os volumes. Estes são definidos na criação da imagem e montados no momento da criação do container.

Ou seja, para containers que necessitem de armazenamento de dados, é possível criar um volume para armazenar estes dados. Um volume funciona como um container com o único intuito de persistir os dados e disponibilizar ao container se for necessário.

Conforme Mouat (2015, p. 53), os volumes ainda podem ser utilizados para compartilhar dados entre o host e os containers, bem como entre dois containers. Isso permite o polimorfismo entre os containers, pois imagens criada de forma genérica podem ser utilizadas em conjunto com volumes para modificar o comportamento do containers conforme os dados do volume, segundo Nickoloff (2016, p. 74).

Uma imagem criada para servir conteúdo estático pode se aproveitar desse padrão de projeto para generalizar o seu uso, uma vez que seu comportamento será alterado conforme for definido o seu volume, por exemplo. Existem dois tipos de volumes no ecossistema Docker: Volume vinculado á maquina anfitriã e o volume gerenciado pelo Docker.

2.2.4.1 Volume vinculado ao host

Conforme Docker (2016b), os volumes vinculados ao host são pontos de montagem do disco da máquina anfitriã no disco do container. Funciona como um diretório compartilhado e é uma ótima forma para compartilhar os arquivos entre a máquina anfitriã e o container. Define-se o diretório (ou arquivo) que será montado no container e o caminho do diretório (ou arquivo) no container.

Se houver arquivos no caminho informado no container, este serão desconsiderados e os arquivos definidos na máquina anfitriã serão utilizados. Qualquer alteração nestes arquivos, serão aplicados diretamente na máquina anfitriã.

Quando é utilizada esta estratégia, é possível, inclusive, utilizar o mesmo ponto de montagem para um segundo container que terá os dados e poderá alterá-lo da mesma forma. Assim, é possível que dois containers utilizem o mesmo ponto de montagem de arquivos e possam usufruir dos mesmos com as eventuais alterações.

Existe a possibilidade de evitar que containers alterem arquivos e prejudiquem o funcionamento de outros, precisa-se definir uma propriedade de *somente leitura* para manter o isolamento e assegurar que não ocorra modificações indesejadas.

2.2.4.2 Volume gerenciador pelo Docker

Ainda, segundo Docker (2016b), pode-se criar um container com objetivo único de armazenar os dados de outro container. Para isso, precisa-se de um container para os dados e um outro para a aplicação. Esta estratégia tem caráter persistente, pois mesmo que o container de aplicação seja corrompido ou removido, o volume será mantido e poderá ser utilizado por um novo container.

Para remover o volume, contudo, é necessário que seja explícito na remoção. Caso contrário, o volume permanecerá na máquina com o espaço alocado. Nickoloff (2016, p. 76) chama este volume de volume órfão, que deve ser removido de maneira manual e cuidadosa para não remover um outro volume por engano.

2.2.5 Camada de Rede

A comunicação entre containers é algo essencial para o bom funcionamento das aplicações, como conexão ao banco de dados, ao cache ou a qualquer outro serviço disponível na rede. Conforme Docker (2016c), deve-se ter o controle da camada de rede a qual a aplicação é executada, de maneira segura, para promover o completo isolamento para o container. Segundo Matthias e Kane (2015, p.13), o Docker acaba agindo como uma ponte virtual entre o dispositivo o qual trafega os dados de um lado para o outro, como uma pequena rede virtual.

Por padrão, o Docker apresenta 4 (quatro) possibilidades de rede para seus containers: container fechado, em ponte, unido e aberto.

2.2.5.1 Container fechado

A maneira mais segura de rede com container, pois não permite nenhum tráfego de rede. O processo que rodar neste container terá acesso apenas a interface de loopback. Apesar de

possui grande segurança, não permite que softwares sejam atualizados e nenhuma comunicação seja efetuada. Nickoloff (2016, p.103) ensina que dificilmente será utilizada esta estratégia de rede, que deixa o container sem acesso à rede externa, entretanto pode ser útil para volumes gerenciados pelo Docker, aqueles em containers, para atividades de backup, processamento offline ou ferramentas de diagnóstico. Deve-se conhecer as formas de utilização de rede para decidir qual forma será utilizada.

Em contra partida, os containers em ponte são a forma mais utilizada pelos desenvolvedores.

2.2.5.2 Container em ponte

Utilizada por padrão, a estratégia de rede de containers em ponte possui um nível normal de isolamento, uma vez que possui uma interface privada e utiliza a rede de maneira natural. De acordo com Nickoloff (2016, p.89), os containers em ponte não são acessíveis pela máquina anfitriã, em razão do sistema de firewall.

A estratégia padrão não permite acesso pela interface de rede externa, o que significa que não existe a possibilidade de descobrir este container fora da máquina anfitriã. A maneira mais natural de acessar serviço de um container pela máquina anfitriã, é expor as portas necessárias e vincular à máquina anfitriã, conforme Docker (2016c).

Uma vez que uma porta é exposta e vinculada à máquina anfitriã, os demais containers terão acesso ao serviço exposto e poderão firmar uma relação entre os serviços.

2.2.5.3 Container unido

De maneira resumida, os containers unidos representam dois containers com estratégias distintas: um fechado e outro em ponte.

O objetivo principal é ter, por opção, as duas maneiras de comunicação em containers isolados. Desta forma, pode-se monitorar uma aplicação, promover backups ou rotinas em batch de maneira mais simples.

A documentação oficial do Docker (2016c), inclusive, não apresenta esta estratégia e a única fonte foi Nickoloff (2016, p.96).

Por último, tem-se a estratégia mais permissiva e insegura de todas: o container aberto.

2.2.5.4 Container aberto

No que diz Nickoloff (2016, p.96), os containers abertos são perigosos e tem total acesso a rede da máquina anfitriã, que incluem acesso aos serviços críticos da máquina. Neste caso,

não se possui nenhuma forma de isolamento de rede e somente deve ser utilizado em casos de extrema necessidade.

3 PROCEDIMENTOS METODOLÓGICOS

Este capítulo aborda os métodos e as técnicas utilizadas neste trabalho, de modo que permitiram o tratamento do tema proposto.

O estudo de caso foi feito de maneira descritiva, no qual a literatura foi analisada e interpretada para a elaboração de um parecer final sobre o assunto. O procedimento utilizado foi a pesquisa bibliográfica, para o melhor entendimento do funcionamento da ferramenta e seus efeitos no ambiente de desenvolvimento de software.

Ainda, buscou-se a utilização do método qualitativo para a elaboração do trabalho. Por se tratar de um tema extremamente novo, utilizou-se os livros mais atuais, para manter a coerência entre o estado atual da ferramenta, artigos de documentação da própria ferramenta, que é de excelente qualidade, e artigos na internet que dissertam sobre o tema com um foco mais prático.

4 RESULTADOS E DISCUSSÕES

Neste capítulo, dissertar-se-á sobre as dificuldades encontradas no processo de desenvolvimento de software, desde o desenvolvimento, distribuição e, por fim, a publicação.

4.1 Problemas comuns na rotina de desenvolvimento de software

O processo de desenvolvimento de software é um trabalho complexo e depende de muitas etapas até a conclusão, conforme DevMedia (2016).

4.1.1 Ambiente de Desenvolvimento

Segundo SoftwareQuality (2016a), no desenvolvimento de softwares o ambiente de desenvolvimento é o conjunto de processos e ferramentas usados para auxiliar a criação do programa ou software. Empresas de tecnologia possuem um checklist ou wiki para orientar o programador na configuração do ambiente de desenvolvimento.

Este processo envolve guias de estilo de codificação, instalação de IDEs e plugins, configurações de banco de dados, entre outros detalhes. Em seguida, deve-se configurar as aplicações para que sejam executadas em modo de desenvolvimento no ambiente do desenvolvedor, que também inclui dependências com versões específicas e configurações ímpares, bem como ajustes de depuração e execução de testes.

Um grande desafio que os desenvolvedores enfrentam é orquestrar configuração do ambiente de desenvolvimento, pois este ambiente acaba por ser simplificado em comparação ao ambiente de produção. É muito comum o software correr de maneira estável em desenvolvimento, mas não obter o mesmo resultado em produção, conforme Jester (2016).

Seja banco de dados em memória, configurações padrão ou uma arquitetura mais modesta, os ambiente de desenvolvimento são mais simples para garantir que o desenvolvedor não perca tempo com configurações e ajustes e passa a focar no objetivo principal que é a implementação de funcionalidades.

4.1.2 Processo de integração

Quando uma equipe de desenvolvimento trabalha em uma nova entrega para o cliente, esta deve paralelizar os esforços para buscar uma entrega mais célere e satisfazer o cliente. Este trabalho em separado deve ser integrado para a realização de testes e garantir que as novas alterações não quebraram comportamentos antigos.

Para isso, submete-se os novos códigos a um processo de integração, o qual tem por objetivo garantir o sucesso das funcionalidades, que nenhuma dependência mudou o seu comportamento e que o programa funciona em sua plenitude, segundo Fowler (2016a).

Em muitos casos, esta integração deve ocorrer em um ambiente controlado para poder reproduzir os dados dos testes e, ainda, muitos deles necessitam de outros serviços para garantir o sucesso, como chamadas de rede, consultas à bancos de dados, entre outros. Desta forma, percebe-se que existe mais um ambiente a ser controlado pela equipe de evolução de funcionalidades.

4.1.3 Ambiente de Homologação

Grandes empresas dispõe de um ambiente de homologação que precede o ambiente de produção. Isto ocorre, de maneira geral, para prevenir erros e desacertos entre ambientes. Este procedimento sempre foi muito bem aceito como destaca ProfissionaisTi (2016), mas será que o ambiente de homologação é realmente necessário?

O ambiente de homologação deve representar o ambiente de produção e exige, também, que seja feita uma instalação completa do software, de suas dependências, alterações de banco de dados e configurações de ambiente. Mesmo que seja utilizada uma ferramenta de automatização para a instalação, um ambiente de homologação exige atenção e manutenção para operar normalmente, conforme SoftwareQuality (2016b).

4.1.4 Distribuição do software

Sabe-se que algumas linguagens necessitam de algum tipo de preparação antes de ser instalada no servidor de produção e a grande maioria necessita de algum tipo de compilação. Seja de seus fontes ou de suas dependências. Após a finalização dos arquivos fontes, dependências e configuração, é hora de distribuir a nova versão do software nos servidores de produção.

Muitos optam por criar um arquivo compactado com todos os arquivos necessários e transferir este arquivo pela rede, que acabam por ser muito pesados e obrigam que arquivos que já foram transmitidos tenham que ser novamente enviados.

Mesmo que este trabalho tenha sido automatizado de alguma forma, por scripts ou outros softwares, ainda se faz necessário que estes arquivos sejam enviados um a um para os novos servidores. Ainda, corre-se o risco de ter algum arquivo corrompido ou que tenha sido gerado de maneira errada, o que pode ocasionar falha na atualização do software.

4.1.5 Instalação do Software

Após a implementação de uma ou mais funcionalidade, uma nova versão é gerada e deve ser lançada no ambiente de homologação ou produção. Normalmente este processo é arduo e

depende de uma equipe especializada para concluir esta etapa. A equipe de desenvolvimento deve passar detalhes da implementação e as peculiaridades da versão.

Qualquer erro ocorrido deve ser reiniciado o processo de instalação e pode provocar consequências irreversíveis, como corrupção de dados. Após a instalação, pode ocorrer a necessidade de ampliar os servidores para servir um tráfego maior de usuários e isso acarreta na instalação em novas máquinas. Mas, é claro, que essa instalação não acontece há tempo de suprir a necessidade de tráfego.

4.1.6 Escalabilidade

O aumento repentino de tráfego em aplicações web demandam muita estratégia da equipe de infraestrutura para que os usuários não sintam lentidão ou não recebam resposta do servidor. Essa estratégia deve conter um rápido ataque para suprir esta necessidade e não pode contar com a instalação de novas máquinas físicas, pois não haveria tempo hábil para tal.

Nos dias de hoje, uma empresa que oferece serviços como produto não pode deixar de prestar o serviço sob pena de ter o descrédito pelos usuários e resultar no término dos negócios. Portanto, esses casos exigem que a equipe de infraestrutura possua condições de ampliar o servidor web.

4.1.7 Entregabilidade

A dificuldade de instalar um software normalmente é medida pela frequência que ela ocorre. Quando ocorre frequentemente, deixa a instalação simples e rápida. Se ocorre com pouca frequência, deixa a instalação complexa e demorada. A forma tradicional de instalação normalmente é mais dolorosa e pouco frequente, em razão de diversos fatores como dependências, scripts de migração de banco de dados e etc.

Assim, a entregabilidade é prejudicada, pois uma nova funcionalidade leva tempo para ser instalada em produção.

4.1.8 Segurança

Nos servidores físicos são comuns a execução de várias aplicações web na mesma máquina para utilizar o máximo de recurso possível. Entretanto, isso pode ser muito perigoso, pois uma vulnerabilidade em uma aplicação pode dar acesso à máquina que rodam as outras. Da mesma forma, um vazamento de memória pode impedir o bom funcionamento das demais aplicações e acabar derrubando as demais.

O isolamento proporcionado pelos namespaces contribuem com mais uma camada de segurança, pois os processos executados em containers não terão acesso a máquina anfitriã.

Ou seja, em caso de vulnerabilidade na aplicação web dentro de um container, o único risco envolvido é do próprio container e não do ambiente por completo.

5 CONCLUSÃO

5.1 Vantagens em executar aplicações em containers

Ante os problemas expostos, verifica-se que muitos deles são grandes candidatos a não existirem, uma vez que seja adotado o uso de docker em todas as etapas do desenvolvimento de software. Deve-se manter uma hierarquia de imagens para facilitar, ainda mais, a estrutura de dependências dos projetos.

5.1.1 Imagens Docker

Ao utilizar Docker é comum, e indicado, que seja feita uma hierarquia de imagens para otimizar o tamanho das imagens e, por consequência, diminuir o tempo de instalação. É sugerido pelo Docker (2016a) algumas práticas para criar as imagens da melhor maneira. A primeira sugestão é pensar que os containers devem possuir natureza efêmera, que deva ser destruído e criado sem prejuízo algum.

Na grande maioria dos casos, diz-se boa prática de criar um arquivo com a lista de diretórios e arquivos que devem ser ignorados na criação das imagens. Arquivos de build, logs e de sistema de controle de versão são bons exemplos de arquivos que devem ficar fora do container.

Outra boa indicação é evitar a instalação de pacotes básicos, como editores de texto, por exemplo, que não será utilizado em produção e acabam ocupando espaço na imagem e aumentando o tempo de construção da mesma. Outra dica valiosa é manter um processo por container e abusar da arquitetura de micro-serviços.

Se for preciso se comunicar com outros serviços, deve-se utilizar as estratégias de rede para manter os serviços comunicáveis e trafegar os dados sempre que preciso.

Recomenda-se, também, a utilização do menor número de layers possível, pois, além de tornar o arquivo de imagem mais legível, ele possui menos etapas e pode manter um cache maior. Procura-se, ainda, manter comando de múltiplas linhas em ordem alfabética para facilitar a compreensão.

Criar imagens reusáveis é uma ótima estratégia e pode poupar tempo e manutenção, uma vez que é possível alterar o comportamento das imagens conforme os volumes montados ao container, segundo Tutum (2016).

5.1.2 Dependências de desenvolvimento

Em princípio, notou-se que as aplicações web atuais estão muito complexas e com muitas dependências. Isso torna o desenvolvimento de software um pouco lento, pois são inúmeras configurações que o profissional deve se atentar para poder executar o seu trabalho de maneira desimpedida, segundo Nickoloff (2016, p.14).

Para isso, tem-se o Docker Compose, uma ferramenta que auxilia a criação de serviços interconectados, configurável por arquivos *yaml* e que são controláveis por um programa de linha de comando, conforme Nickoloff (2016, p.232).

Além de evitar que o desenvolvedor seja obrigado a instalar serviços como banco de dados, gerenciador de fila, cache, entre outros, o Docker Compose se encarrega de orquestrar estes serviços de forma automatizada e interligadas, utilizando links para simplificar a conexão entre elas, de acordo com Matthias e Kane (2015, p111).

Sem sombra de dúvidas, é a melhor maneira de criar um ambiente de desenvolvimento do zero. Isso torna a iniciação de novos desenvolvedores mais rápida e com menos tempo para adaptação, desde que saiba o básico de Docker.

O Docker é a ferramenta perfeita para facilitar o ambiente de desenvolvimento, pois os containers tendem a substituir as dependências como banco de dados, cache, sistema de filas e etc, de maneira simples e leve. Esta solução pode ser implementada na própria máquina do desenvolvedor, uma vez que leva segundos para executar um container e ele tem um impacto menor no sistema, em razão do compartilhamento dos recursos do sistema. Com uma única linha de comando é possível executar um banco de dados pronto para ser utilizado.

Se o projeto deve ser testado em mais de um banco, verifica-se ainda mais benefícios, pois é possível parar e executar containers de maneira muito fácil. Isso deixa a troca de banco mais leve para ser executado no ambiente de desenvolvimento e dá mais flexibilidade para os testes, sem depender de um ambiente de testes.

Outro grande benefício é abstrair a configuração de banco de dados de desenvolvedores júnior, por exemplo. Estes podem simplesmente buscar uma imagem pronta para uso e não ter que perder tempo com configurações para conseguir executar o projeto.

Uma outra dependência que cabe citar, é o backend para o desenvolvimento de interfaces. Normalmente desenvolvedores frontend são dependentes de outros desenvolvedores para preparar um ambiente atualizado, em funcionamento e com dados úteis para os testes. Com o Docker, o próprio desenvolvedor frontend pode escolher a imagem que gostaria de utilizar e pode usar configurações pré definidas, o que reduz o tempo de preparação e aumenta o tempo de desenvolvimento.

Entende-se que toda vez que um desenvolvedor ganha tempo de desenvolvimento ou gasta menos tempo arrumando ambientes, a empresa tem menos prejuízo. Esse é um grande

argumento para adotar a ferramenta e começar a reduzir os gastos desnecessários com preparação e ambientes de desenvolvimento.

Conforme ensina Brikman (2016), o Docker deixou a instalação de softwares de terceiros uma tarefa muito mais fácil, pois basta configurar as variáveis e rodar o container para executar a aplicação. Lembra-se que a grande maioria dos software são passíveis de serem executados em containers. Também comenta que é muito mais limpo testar novos serviços, uma vez que a instalação ocorre num container e não necessita de instalação de dependências na máquina de trabalho. Desta forma, o seu sistema continua enxuto e problemas com muitas dependências.

Ainda, garante uma camada extra de segurança pois, caso sejam executados aplicações maldosas, o container vai manter o isolamento com a máquina anfitriã e vai proteger o sistema de arquivos e os dispositivos conectados.

5.1.3 Composição de containers

É normal um desenvolvedor começar a utilizar o Docker e se deparar com uma grande quantidade de containers ou de configurações. Pra resolver este problema, existe o Docker Compose. Ele é uma ferramenta voltada para composição de múltiplos containers conectados e promover um ambiente de desenvolvimento mais simples. Ele possui um arquivo de configuração no formato yaml e é possível criar diversas configurações.

5.2 Distribuição

Para a distribuição do software, sabe-se que é, também, uma tarefa muito árdua, que na grande maioria dos casos necessita de uma equipe especializada no assunto. Com Docker Registry este trabalho foi facilitado, de modo que a distribuição das imagens são feitas de maneira automática e podem ser construídas de maneira muito rápida, com o uso de imagens base, como explica Mouat (2015, p.44). Lembra-se, contudo, que as boas práticas ensinam que as imagens só devem ser publicadas no registry após os testes unitários e de aceitação.

5.3 Publicação

Por fim, o deploy deve ser automático e não deve apresentar problemas se for utilizado o Docker Swarm, que é uma ferramenta de orquestração para construção de cluster de serviços docker, como ensina Nickoloff (2016, p.255). Ele consegue atualizar o cluster de serviços Docker de forma rápida e segura, pois nossos serviços web devem operar com conexão criptografada, segundo Matthias e Kane (2015, p.129).

Desta forma, conclui-se que a ferramenta Docker traz benefícios em todas as etapas de desenvolvimento de software, com a sua distribuição de imagens, publicação em produção e

manutenção do cluster de serviços.

5.4 Estudo de caso

Atualmente, a Gerência de Tecnologia da Softplan desenvolve software com o auxílio da ferramenta Docker para simplificar o desenvolvimento, distribuição e publicação. O projeto é composto por uma api, um projeto web, um programa para consumir um feed, um banco de dados relacional, um sistema de cache e um broker. Na stack ainda existem um agregador de log, um monitor de exceções, servidor de integração contínua e documentação dos projetos.

São 10 (dez) aplicações diferentes, com configurações singulares e conexões específicas. Criar esta configuração manualmente seria possível, mas tomaria um tempo precioso. Este projeto já começou com o intuito de apresentar um retorno desde o seu início. Ou seja, já deve apresentar aos usuários um valor e justificar o investimento.

Desta forma, optou-se por utilizar Docker e acelerar o processo de distribuição do projeto.

5.4.1 Desenvolvimento

Para o desenvolvimento, utilizou-se Docker Compose para gerenciar os containers e facilitar o uso por novos desenvolvedores e diminuir a curva de aprendizado para começar a desenvolver em qualquer das aplicações. A primeira imagem Docker foi da api. Utilizou-se a imagem base do alpine linux, para manter o tamanho da imagem pequeno, com python 3 instalado e as dependências gerenciadas pelo python-pip. A execução do container é feita pelo supervisor e pelo uWSGI.

Para a aplicação web, decidiu-se utilizar bower e grunt para montar as dependências e criar os arquivos compactados e minificados. Ainda, para servir os arquivos estáticos, escolheu-se o Nginx para servidor web. Para o consumidor de feed, utilizou-se a mesma imagem base da api, porém sem a necessidade dos serviços web, pois não há a necessidade de executar um servidor. O banco de dados escolhido foi postgres, por ser opensource e sem custo para uso comercial, o redis para cache e o rabbitmq para gerenciamento de filas.

A imagem docker do Postgres é mantida pela comunidade e permite com apenas uma linha de comando a criação da mesma. É possível alterar sua configuração e suas permissões antes da criação do container. Para não correr riscos de perder dados dentro dos containers, criou-se um volume para armazenar os dados, mesmo que o container do banco seja destruído. Esta é uma boa prática e deve ser seguida na utilização de containers com dados persistentes.

Situação similar ocorreu com o Redis, que já possui uma imagem mantida pela comunidade com as configurações padrões já estabelecidas, mas com opção de alterá-las na criação do container. O RabbitMQ teve uma execução muito parecida. Um detalhe muito importantes,

antes da criação dos containers, é a criação de uma rede docker para que os containers possam se comunicar entre si. Desta forma, criou-se uma rede local para estabelecer esta comunicação e, ainda, compartilhar a internet da máquina anfitriã e permitir que os softwares possam ser atualizados de maneira correta.

5.4.2 Integração Contínua

É muito importante que o desenvolvimento seja integrado continuamente ao projeto, com a execução dos testes e um feedback da saúde do software. Quando se utiliza a tecnologia de containers, deve-se aplicar os testes automatizados com as futuras imagens docker. Ou seja, deve-se criar uma nova imagem e rodar os testes com um novo container.

Desta forma, aproveita-se para realizar um teste completo na imagem para garantir que sua criação foi realizada com sucesso e, ainda, para rodar os testes automatizados. Após os testes, é importante que a nova imagem seja enviada para o registry (local ou nuvem) e possa ser compartilhada com os demais desenvolvedores.

No caso em tela, utilizou-se o Jenkins para realizar o pipeline de integração contínua. Criou-se um job para rodar os testes de pull requests, que não envia as imagens criadas, um job para merge na branch de desenvolvimento, merge na branch principal e, ao final, um job para criação das imagens e um gatilho para que o docker atualize as imagens dos ambientes.

5.4.3 Ambiente de Desenvolvimento

Para validar as atividades e novas funcionalidades, criou-se um ambiente de desenvolvimento para testar as imagens criadas e possibilitar que a equipe possa testar num ambiente idêntico ao de produção. Inclusive, todas as rotinas utilizadas em produção são, também, utilizadas no ambiente de desenvolvimento.

Para o ambiente de desenvolvimento utilizou-se o Docker Compose para orquestrar os containers de maneiras mais simples e com um arquivo único de configuração. Tomou-se esta decisão para deixar o ambiente mais flexível e que seja mais compreensível pelos demais colegas.

5.4.4 Ambiente de Produção

Por fim, para o ambiente de produção, optou-se por utilizar o Docker Swarm, que disponibiliza utensílios para escalar os serviços necessários na medida que as aplicações precisam. A grande vantagem de utilizar containers é justamente ter a possibilidade de ampliar o número de processos dos serviços que são mais necessários.

No caso do projeto em tela, utilizou-se a biblioteca celery para gerenciar a execução de tarefas assíncronas, bem como as filas de atividades. Desta forma, se alguma fila estiver com

muito atraso, consegue-se de maneira simples e transparente aumentar o número de containers para o consumo daquela fila e manter a aplicação em funcionamento.

Da maneira tradicional, a escalabilidade ocorre de maneira menos inteligente, pois era necessário aumentar o número de máquinas virtuais e duplicar os processos. Isso pode ajudar na realização das atividades atrasadas, mas vai gerar um grande desperdício de recursos, pois serviços que não precisam ser aumentados, vão acabar duplicados pois não existe essa separação tão importante.

É por isso que a arquitetura de microserviços é tão utilizado atualmente. Acredita-se que todas as grandes empresas de tecnologia já adotam o uso de microserviços e containers para simplificar a escalabilidade dos serviços e garantir maior desempenho sem desperdício de recursos.

5.4.5 Considerações finais

Ao final, conclui-se que a ferramenta Docker proporciona muitas vantagens ao desenvolvedor. Em primeiro lugar, automatiza e simplifica o ambiente de desenvolvimento e garante que este será o mesmo utilizado em produção. Disponibiliza uma api para facilitar a instalação e atualização de dependências, bem como scripts de migração de dados.

Ainda, falcita muito a integração de novos desenvolvedores, pois permite que o sistema seja executado sem que haja um estudo prévio do mesmo e das configurações iniciais do sistema.

No que tange a arquitetura de microserviços, o Docker tem mais benefícios, pois além de permitir uma economia de recursos computacionais, garante que os processos estejam isolados, que traz mais segurança ao sistema, e permite que a escalabilidade seja realizada de maneira eficiente para expandir apenas os serviços necessários.

Outrossim, entende-se que a ferramenta é uma ótima aposta por se tratar de uma ferramenta de código aberto, que não tem intensão de privatizar o código e possui uma cultura de desenvolvimento moderna e colaborativa.

A última versão instável do Docker é a 1.12 e traz muitas novidades como o swarm mode, que tem por objetivo incentivar o uso do swarm em pequenas aplicações, sem que seja necessário uma configuração extensa e complexa.

Cabe ressaltar que o Docker possui uma suite de soluções proprietárias que simplifica a criação, hospedagem e distribuição de imagens docker, bem como sua distribuição em ambiente de produção. Estas soluções são oferecidas pela empresa e tem cunho comercial.

Espera-se que o Docker continue ampliando seu leque de aplicações para satisfazer ainda mais seus usuários e que este modelo sirva de exemplo para as demais empresas. A empresa incentiva muitos encontros sobre Docker pelo mundo, de forma que oferece brindes para atividades programadas e também promove um grande encontro sobre docker, chamado de

DockerCon que tem periodicidade anual.

REFERÊNCIAS

BRIKMAN. *A productive development environment with Docker on OS X*. 2016. Disponível em: <<http://www.ybrikman.com/writing/2015/05/19/docker-osx-dev/>>. Acesso em: 6 de junho de 2016. Citado na página 29.

CLOUDSIGMA. *Manage docker resources with cgroups*. 2016. Disponível em: <<https://www.cloudsigma.com/manage-docker-resources-with-cgroups/>>. Acesso em: 2 de junho de 2016. Citado na página 17.

CRUNCHBASE. *Docker*. 2016. Disponível em: <<https://www.crunchbase.com/organization/docker>>. Acesso em: 19 de maio de 2016. Citado na página 15.

DEVMEDIA. *Atividades básicas ao processo de desenvolvimento de software*. 2016. Disponível em: <<http://www.devmedia.com.br/atividades-basicas-ao-processo-de-desenvolvimento-de-software/5413>>. Acesso em: 2 de junho de 2016. Citado na página 23.

DOCKER. *Best practices for writing Dockerfiles*. 2016. Disponível em: <https://docs.docker.com/engine/userguide/eng-image/dockerfile_best-practices/>. Acesso em: 6 de junho de 2016. Citado na página 27.

DOCKER. *Manage data in containers*. 2016. Disponível em: <<https://docs.docker.com/engine/userguide/containers/dockervolumes>>. Acesso em: 2 de junho de 2016. Citado 2 vezes nas páginas 18 e 19.

DOCKER. *Understand Docker container networks*. 2016. Disponível em: <<https://docs.docker.com/engine/userguide/networking/dockernetworks/>>. Acesso em: 4 de junho de 2016. Citado 2 vezes nas páginas 19 e 20.

DOCKER. *What is Docker?* 2016. Disponível em: <<https://www.docker.com/what-docker>>. Acesso em: 19 de maio de 2016. Citado 3 vezes nas páginas 14, 15 e 16.

FOWLER, M. *Continuous Integration*. 2016. Disponível em: <<http://martinfowler.com/articles/continuousIntegration.html>>. Acesso em: 3 de junho de 2016. Citado na página 24.

FOWLER, M. *Microservices a definition of this new architectural term*. 2016. Disponível em: <<http://martinfowler.com/articles/microservices.html>>. Acesso em: 7 de junho de 2016. Citado na página 12.

JESTER, D. L. T. *Traditional Development Practice for Software Development*. 2016. Disponível em: <<http://dljt.org/article/software-development-practice/>>. Acesso em: 13 de junho de 2016. Citado na página 23.

KERNELORG. *Cgroups*. 2016. Disponível em: <<https://www.kernel.org/doc/Documentation/cgroup-v1/cgroups.txt>>. Acesso em: 30 de maio de 2016. Citado na página 17.

MATTHIAS, K.; KANE, S. *Docker up and running*. [S.l.]: O Reilly Media, Inc, 2015. Citado 7 vezes nas páginas 14, 16, 17, 18, 19, 28 e 29.

MOUAT, A. *Using Docker*. [S.l.]: O Reilly Media, Inc, 2015. Citado 6 vezes nas páginas 14, 15, 16, 17, 18 e 29.

NICKOLOFF, J. *Docker in action*. [S.l.]: Shelter Island, 2016. Citado 9 vezes nas páginas 14, 15, 16, 17, 18, 19, 20, 28 e 29.

PROFISSIONAISTI. *A importância de um ambiente de homologação*. 2016. Disponível em: <<http://www.profissionaisti.com.br/2013/06/a-importancia-de-um-ambiente-de-homologacao/>>. Acesso em: 5 de junho de 2016. Citado na página 24.

SOFTWAREQUALITY. *development environment*. 2016. Disponível em: <<http://searchsoftwarequality.techtarget.com/definition/development-environment>>. Acesso em: 5 de junho de 2016. Citado na página 23.

SOFTWAREQUALITY. *A good QA team needs a proper software staging environment for testing*. 2016. Disponível em: <<http://searchsoftwarequality.techtarget.com/tip/A-good-QA-team-needs-a-proper-software-staging-environment-for-testing>>. Acesso em: 3 de junho de 2016. Citado na página 24.

TECHTARGET. *brief history of Docker Containers overnight success*. 2016. Disponível em: <<http://searchservervirtualization.techtarget.com/feature/A-brief-history-of-Docker-Containers-overnight-success>>. Acesso em: 19 de maio de 2016. Citado na página 15.

TUTUM. *How to Optimize Your Dockerfile*. 2016. Disponível em: <<https://blog.tutum.co/2014/10/22/how-to-optimize-your-dockerfile/>>. Acesso em: 6 de junho de 2016. Citado na página 27.

W3TECHS. *W3Techs*. 2016. Disponível em: <https://w3techs.com/technologies/overview/operating_system/all/>. Acesso em: 19 de maio de 2016. Citado na página 11.

APÊNDICE A CÓDIGO FONTE

Códigos referentes aos arquivos de configuração do Docker, como o Dockerfile e o docker-compose.yaml.

```
# Dockerfile
```

```
FROM python:3.5-alpine
ENV PYTHONUNBUFFERED 1
```

```
RUN apk add --update \
    nginx \
    supervisor \
    openssl \
    postgresql-client \
    postgresql-dev \
    gcc \
    musl-dev \
    linux-headers \
    && rm -rf /var/cache/apk/*
```

```
RUN mkdir -p /etc/supervisor.d/
```

```
EXPOSE 8000
```

```
CMD ["supervisord", "-n"]
```

```
# Dockerfile
```

```
FROM feliperuhland/python-uwsgi-nginx-supervisor
ENV PYTHONUNBUFFERED 1
RUN mkdir /app
WORKDIR /app
ADD requirements.txt /app/
RUN pip install --upgrade pip
RUN pip install -r requirements.txt
RUN pip install uwsgi
ADD . /app/
RUN rm /etc/nginx/nginx.conf
RUN ln -s /app/config/nginx.conf /etc/nginx/
RUN ln -s /app/config/supervisor-app.conf /etc/supervisor.d/supervisor-app.ini
EXPOSE 8000
CMD ["supervisord", "-n"]
```

```
# Dockerfile
```

```
FROM nginx
```

```
RUN rm /etc/nginx/conf.d/default.conf
COPY production.conf /etc/nginx/conf.d/production.conf
COPY development.conf /etc/nginx/conf.d/development.conf
```

```
# Dockerfile
```

```
FROM nginx
ARG port=8081
COPY www/ /usr/share/nginx/html
RUN sed -i -r "s/localhost:8000/piunj.softplan.com.br:$port/g" /usr/share/nginx/html/js/app.js
RUN sed -i -r 's/js\ionic.bundle.js/js\ionic.bundle.min.js/g' /usr/share/nginx/html/index.htm
RUN sed -i -r 's/css\ionic.css/css\ionic.min.css/g' /usr/share/nginx/html/index.html
```

```
# Dockerfile
```

```
FROM ubuntu
```

```
RUN apt-get update && \
    apt-get install -y git wget curl && \
    apt-get clean

RUN curl -sL https://deb.nodesource.com/setup | bash -

RUN apt-get update && \
    apt-get install -y nodejs npm build-essential && \
    ln -s /usr/bin/nodejs /usr/local/bin/node && \
    apt-get clean
```

```
RUN npm install -g cordova ionic
RUN npm install -g grunt-cli
RUN npm install -g gulp
RUN npm install -g bower
```

```
EXPOSE 8100 35729
```

```
RUN mkdir /code
WORKDIR /code
COPY . /code
```

```
CMD ["ionic", "serve", "--all", "--port", "8100", "--livereload-port", "35729"]
```

```
# Dockerfile
```

```
FROM debian:jessie
```

```
# explicitly set user/group IDs
RUN groupadd -r postgres --gid=999 && useradd -r -g postgres --uid=999 postgres
```

```
# grab gosu for easy step-down from root
```

```
ENV GOSU_VERSION 1.7
```

```
RUN set -x \
    && apt-get update && apt-get install -y --no-install-recommends ca-certificates wget && \
    && wget -O /usr/local/bin/gosu "https://github.com/tianon/gosu/releases/download/$GOSU_VERSION/gosu-$GOSU_VERSION" \
    && wget -O /usr/local/bin/gosu.asc "https://github.com/tianon/gosu/releases/download/$GOSU_VERSION/gosu-$GOSU_VERSION.asc" \
    && export GNUPGHOME="$(mktemp -d)" \
    && gpg --keyserver ha.pool.sks-keyservers.net --recv-keys B42F6819007F00F88E364FD4036A90 \
    && gpg --batch --verify /usr/local/bin/gosu.asc /usr/local/bin/gosu \
    && rm -r "$GNUPGHOME" /usr/local/bin/gosu.asc \
    && chmod +x /usr/local/bin/gosu \
    && gosu nobody true \
    && apt-get purge -y --auto-remove ca-certificates wget
```

```

# make the "en_US.UTF-8" locale so postgres will be utf-8 enabled by default
RUN apt-get update && apt-get install -y locales && rm -rf /var/lib/apt/lists/* \
    && localedef -i en_US -c -f UTF-8 -A /usr/share/locale/locale.alias en_US.UTF-8
ENV LANG en_US.utf8

RUN mkdir /docker-entrypoint-initdb.d

RUN apt-key adv --keyserver ha.pool.sks-keyservers.net --recv-keys B97B0AFCAA1A47F044F244A07FCC

ENV PG_MAJOR 9.5
ENV PG_VERSION 9.5.3-1.pgdg80+1

RUN echo 'deb http://apt.postgresql.org/pub/repos/apt/ jessie-pgdg main' $PG_MAJOR > /etc/apt/sources.list.d/pgdg.list

RUN apt-get update \
    && apt-get install -y postgresql-common \
    && sed -ri 's/#(create_main_cluster) .*$/\1 = false/' /etc/postgresql-common/createcluster.conf \
    && apt-get install -y \
        postgresql-$PG_MAJOR=$PG_VERSION \
        postgresql-contrib-$PG_MAJOR=$PG_VERSION \
    && rm -rf /var/lib/apt/lists/*

# make the sample config easier to munge (and "correct by default")
RUN mv -v /usr/share/postgresql/$PG_MAJOR/postgresql.conf.sample /usr/share/postgresql/ \
    && ln -sv ../postgresql.conf.sample /usr/share/postgresql/$PG_MAJOR/ \
    && sed -ri "s!^#?(listen_addresses)\s*=\s*\S+.*!\1 = '*'!" /usr/share/postgresql/postgresql.conf.sample

RUN mkdir -p /var/run/postgresql && chown -R postgres /var/run/postgresql

ENV PATH /usr/lib/postgresql/$PG_MAJOR/bin:$PATH
ENV PGDATA /var/lib/postgresql/data
VOLUME /var/lib/postgresql/data

COPY docker-entrypoint.sh /

ENTRYPOINT ["/docker-entrypoint.sh"]

EXPOSE 5432
CMD ["postgres"]

# Docker-compose

version: '2'
services:
  db:
    image: postgres
    environment:
      - POSTGRES_DB=app_db
    ports:
      - "5432:5432"
  web:
    build: .
    command: python manage.py runserver 0.0.0.0:8000
    volumes:
      - .:/app
    ports:

```

```
    - "8000:8000"
depends_on:
  - db
  - rabbitmq
environment:
  DATABASE_URL: postgres://postgres@db:5432/app
update_issue:
  build: .
  command: celery -A app worker -l info -Q update_issue -c 1
  environment:
    DATABASE_URL: postgres://postgres@db:5432/app
  volumes:
    - ./app
depends_on:
  - db
  - rabbitmq
process_metric:
  build: .
  command: celery -A app worker -l info -Q process_metric -c 4
  environment:
    DATABASE_URL: postgres://postgres@db:5432/app
  volumes:
    - ./app
depends_on:
  - db
  - rabbitmq
rabbitmq:
  image: rabbitmq:3-management
  ports:
    - "15672:15672"
```