UNIVERSIDADE FEDERAL DE SANTA CATARINA

Ferramenta de Gerenciamento de Aplicações em um Cluster DC/OS na Nuvem

Augusto Pacheco Santos de Souza

Felipe Duarte Silveira

Florianópolis - SC

2017/1

UNIVERSIDADE FEDERAL DE SANTA CATARINA

DEPARTAMENTO DE INFORMÁTICA E ESTATÍSTICA CURSO DE SISTEMAS DE INFORMAÇÃO

Ferramenta de Gerenciamento de Aplicações em um Cluster DC/OS na Nuvem

Augusto Pacheco Santos de Souza

Felipe Duarte Silveira

Trabalho de Conclusão de Curso apresentado como parte dos requisitos para obtenção de grau de Bacharel em Sistemas de Informação

Florianópolis - SC

2017/1

Augusto Pacheco Santos de Souza

Felipe Duarte Silveira

Ferramenta de Gerenciamento de Aplicações em um Cluster DC/OS na Nuvem

Trabalho de Conclusão de Curso apresentado como parte dos requisitos para obtenção de grau de Bacharel em Sistemas de Informação

Orientador: Carlos Becker Westphall

Banca examinadora

Carla Merkle Westphall

Rafael Souza Mendes

Sumário

Lista de Figuras

Resumo

Várias empresas necessitam testar suas aplicações antes de colocá-las em produção e disponibilizar uma versão final ao usuário ou cliente.

Em alguns casos os recursos são escassos, a empresa possui somente um ambiente de homologação e uma fila de testadores querendo homologar diferentes versões do mesmo *software*.

Este projeto visa implementar uma solução baseada na tecnologia de *container Datacenter Operating Systems* (DC/OS), que permita aos testadores homologar diferentes versões do mesmo *software* ao mesmo tempo obedecendo regras de níveis de acesso e permissão. Com isso uma empresa conseguiria lançar mais versões de suas aplicações, otimizando a velocidade de correção de *bugs* e o desenvolvimento de novas funcionalidades.

Palavras-Chave: ambiente de homologação, *container*, DC/OS.

1. Introdução

1.1 Motivação

As empresas de tecnologia em geral necessitam de um ambiente para homologar suas aplicações, ambiente este que precisa oferecer condições e ferramentas que possibilitem testes de funcionalidades e desempenho de uma maneira otimizada e que não gere um custo operacional elevado.

Vários ambientes de homologação utilizam o conceito da tecnologia de *containers*. A tecnologia de *container* permite a criação de várias instâncias isoladas de uma aplicação dentro de um único servidor de maneira simples e leve. Um *container* é um pacote que possui um conjunto de configurações, código, tempo de execução, bibliotecas, tudo que é necessário para executar o software de uma determinada aplicação (DOCKER, 2016). Desse modo, a estrutura gerada através do *container* será independente do ambiente em que se encontra, evitando conflitos entre testadores que operam diferentes aplicações utilizando a mesma infraestrutura.

Como várias aplicações são testadas ao mesmo tempo, ocorre a criação de diversos *containers* no mesmo ambiente. Por isso se faz necessário a utilização de um *software* de gerenciamento.

Atualmente no mercado existem alguns *softwares* de gerenciamento de *containers*, dentre eles o *Datacenter Operating Systems* (DC/OS). Ele age como um sistema operacional que gerencia diversas máquinas de forma distribuída, possibilitando executar aplicações abstraindo a alocação de recursos. Assim equipes de testadores conseguem executar suas aplicações e gerenciá-las de um modo simples (MESOSPHERE, 2017a) (MESOSPHERE, 2017b).

O problema é que em algumas ocasiões, os recursos de uma empresa são escassos, com apenas um ambiente de homologação e uma fila de testadores querendo testar diferentes versões de um mesmo *software*, tornando o processo de homologação de novas versões um pouco lento.

Este projeto tem como motivação otimizar o tempo que leva para se lançar correções ou novas funcionalidades de uma aplicação, implementando uma solução para que várias versões de uma mesma aplicação possam ser testadas e homologadas em um mesmo ambiente de testes, sem gerar nenhum conflito de versões, obedecendo regras de acesso.

1.2 Justificativa

A utilização de um ambiente de homologação é muito comum no processo de desenvolvimento de *software*. Várias empresas deste ramo possuem equipes para testar suas aplicações antes de liberá-las para seus clientes.

Em certos casos, podem existir diferentes versões de uma mesma aplicação que precisam ser homologadas antes de serem liberadas para produção. Existindo apenas um ambiente, a solução é homologar uma de cada vez, criando assim uma fila.

Tendo como objetivo acelerar o processo de homologação de versões de uma mesma aplicação, este trabalho se justifica através do desenvolvimento de uma ferramenta que permita executar diferentes versões simultaneamente, em um *cluster* DC/OS, otimizando o tempo para o lançamento em produção de novas versões da aplicação.

1.3 Objetivos

O presente projeto tem como objetivo desenvolver uma ferramenta que permita executar diferentes versões de uma mesma aplicação em um ambiente de homologação, respeitando níveis de acesso e quantidade máxima de aplicações que podem ser executadas no *cluster*. Utiliza a tecnologia de *container* DC/OS, possibilitando a uma empresa otimizar o tempo de lançamento de novas versões. Para alcançar tais objetivos, serão utilizadas as etapas descritas na seção 1.4.

1.4 Métodos de Pesquisa

As seguintes etapas foram utilizadas para a realização deste trabalho:

Etapa 1: Estudo e compreensão dos conceitos de um ambiente de testes baseado na tecnologia de *container* e seus *frameworks*, *softwares* e aplicações atualmente utilizados por equipes de testadores de diversas empresas no mercado de trabalho.

Etapa 2: Desenvolvimento da ferramenta, em Ruby(RUBY, 2017), que permita o lançamento de diferentes versões de uma mesma aplicação em um mesmo *cluster* DC/OS.

Etapa 3: Realização de testes e validação das funcionalidades da aplicação desenvolvida na etapa anterior.

Etapa 4: Elaboração da monografia referente a este projeto.

1.5 Estrutura do Documento

O documento deste projeto está organizado em mais 4 capítulos. O capítulo 2 trata da fundamentação para o desenvolvimento do projeto, apresentando conceitos sobre *frameworks*, sistemas operacionais e aplicações que são utilizadas em conjunto com as tecnologias de *container* que também serão tratadas neste capítulo.

O capítulo 3 apresenta a ferramenta, detalhando o processo de desenvolvimento e as funcionalidades da mesma. Será tratado também um estudo de caso da aplicação, explicando como ela funciona e como ela alcança os resultados esperados.

Por fim, o capítulo 4 apresenta a conclusão do projeto.

2. Fundamentação Teórica

2.1 Método Ágil

"Nossa maior prioridade é satisfazer o cliente através da entrega contínua e adiantada de software com valor agregado (MANIFESTO ÁGIL, 2001)."

Desde sua oficialização através da publicação do Manifesto Ágil (2001), o desenvolvimento Ágil, ou Método Ágil, é amplamente utilizado como metodologia de desenvolvimento de projetos em diversas empresas. Um de seus princípios, citado acima, prioriza a entrega do produto ao cliente de maneira rápida e eficiente utilizando iterações curtas, geralmente com prazo de no máximo semanas. Cada iteração pode ser tratada como um projeto de software reduzido que busca o desenvolvimento de uma nova funcionalidade ou a correção de algum erro da versão anterior, e segue, baseado em Cohen (2004) e Karlstrom (2005), as etapas de um projeto de software brevemente citados abaixo:

i. Planejamento: geralmente envolve análise de custos, levantamento de pessoal, definição do escopo.

ii. Análise de Requisitos: juntamente com o cliente, são definidos todos os requisitos necessários que o projeto deverá atender.

iii. Projeto: especifica as funcionalidades do *software* e seu modelo de interação afim de atender os requisitos levantados pela etapa anterior.

iv. Codificação: etapa na qual as funcionalidades definidas são codificadas e o desenvolvimento do projeto é realizado.

v. Teste: tudo que foi feito na etapa de codificação é submetido a testes em ambientes de homologação por equipes de testadores e até o próprio cliente.

vi. Implantação e operação: depois de homologados os testes o *software* é implantado e a partir daí uma equipe de operação é responsável pelo suporte.

Sem muita burocracia, buscando sempre um contato direto dentro da equipe de desenvolvimento, as etapas citadas podem ser adaptadas para a realidade de cada empresa, mas sempre com o objetivo de otimizar o tempo e a qualidade do *software* para satisfazer o cliente. A qualidade do *software* garantida através da etapa de testes, será discutida na seção 2.2.

2.2 Qualidade de Software

Qualidade de *software* se refere às características desejadas de produtos de *software*, a extensão em que um produto de *software* em particular possui essas características e aos processos, ferramentas e técnicas que são usadas para garantir essas características. (BOURQUE, 2014).

Para se determinar a qualidade de um *software* podem ser utilizadas diversas normas formais como a ISO/IEC 25010 (ISO ,2011), que possui normas para verificação e autenticação da qualidade de um software. Baseada na norma ISO/IEC 25010 são apresentadas abaixo algumas características importantes para a qualidade de um *software*:

i. Funcionalidade: o *software* deve conter funcionalidades que satisfaçam os requisitos levantados com o cliente de maneira que atendam todas as suas necessidades, levando em consideração características fundamentais como segurança das informações e acurácia das funcionalidades, ou seja, executar o que foi acordado de forma precisa.

ii. Confiabilidade: o nível de desempenho do *software* deve manter as condições previamente estabelecidas com o cliente.

iii. Usabilidade: o funcionamento do *software* deve ser de fácil entendimento ao usuário final, bem como o modo de operação e sua interface, se possuir, deve ser amigável.

iv. Eficiência: o tempo de execução das tarefas do *software*, bem como os recursos utilizados devem estar de acordo com as especificações para garantir um bom desempenho.

v. Manutenibilidade: se refere a facilidade com que o *software* pode ser corrigido, caso haja algum erro, e na facilidade de implementar uma nova funcionalidade em uma nova etapa de desenvolvimento.

vi. Portabilidade: o *software* deve ter a capacidade de manter a qualidade do desempenho em diferentes ambientes, com diferentes infraestruturas e sistemas operacionais.

Uma das maneiras de se cumprir as ordens exemplificadas é a realização de testes por meio de uma equipe de testadores, e por vezes do próprio cliente, que executam o *software* em um ambiente de homologação que consiga simular o ambiente de produção. Os testes geralmente seguem alguns princípios e técnicas que são definidos pelas equipes de testadores de modo que se adequem à realidade do projeto de *software*. Segundo Myers (2004), ao se realizar um teste deve-se definir a saída esperada, ou seja, o resultado da execução do teste, com o objetivo de estabelecer os critérios de sucesso. Este princípio é considerado por Myers (2004) como sendo vital, pois direciona os testes, de modo objetivo, a saber se o software cumpre as normas de qualidade de maneira satisfatória.

Para se garantir a qualidade é importante contar com um ambiente de testes que possibilite a equipe de testadores realizar os testes de maneira eficaz e otimizada. Existem algumas tecnologias que focam na otimização do processo de testes a que será tratada neste trabalho é a tecnologia de *container* que será detalhada na seção 2.3.

2.3 Container

Como apresentado na introdução deste projeto, um *container* é um pacote que possui um conjunto de configurações, código, tempo de execução, bibliotecas, tudo que é necessário para executar o *software* de uma determinada aplicação (DOCKER, 2016).

Criada em 2008, a tecnologia de *Linux Containers* (LXC)(LINUX CONTAINER, 2008), permite a virtualização de uma aplicação tendo como hospedeiro um servidor *Linux*. Essa tecnologia é um meio termo entre o antigo comando *chroot* (Unix V7, 1979) e uma máquina virtual convencional. Com o comando *chroot* é possível encapsular um sistema dentro de uma estrutura de diretório possibilitando o sistema hospedeiro acessar somente a parte que é especificada nesta estrutura. Existem algumas diferenças entre o LXC e o comando *chroot* sendo a principal delas a questão da segurança das informações, onde o *chroot* se destaca negativamente por ser conhecido por sua vulnerabilidade (DU, 2009).

Comparando o LXC com uma máquina virtual, o LXC não necessita de uma camada de sistema operacional para cada aplicação tornando-se uma ferramenta que ocupa menos espaço em disco que uma máquina virtual convencional, demanda menos recursos e tem um nível de portabilidade superior. A figura 1 ilustra esta comparação (ALMEIDA, 2015).

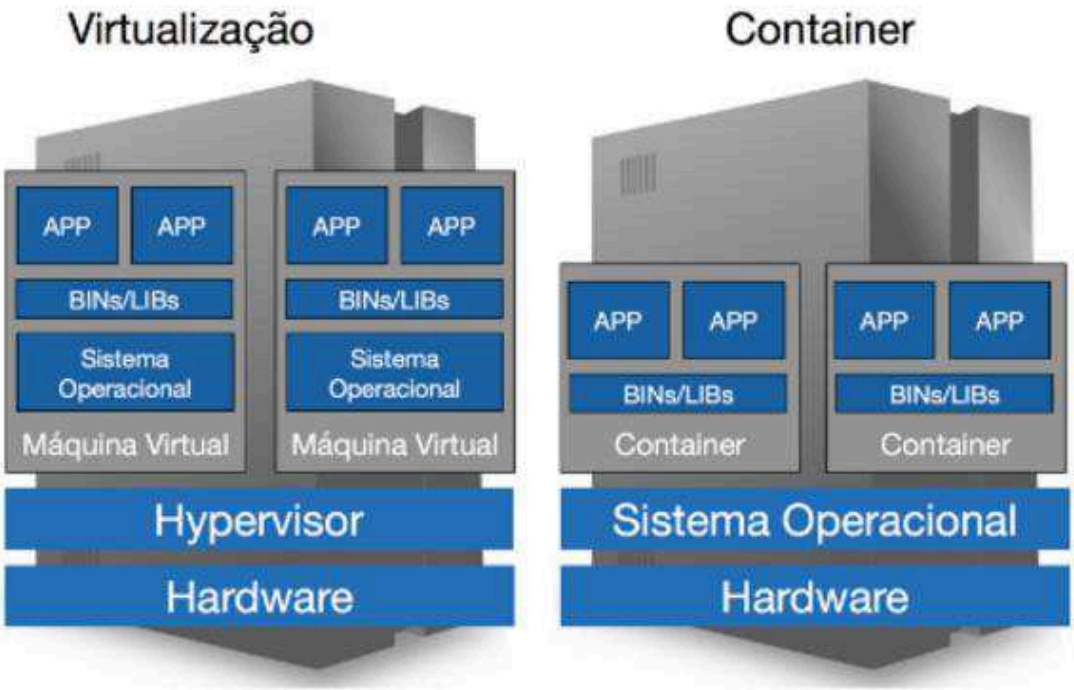

Figura 1: comparação Container x Virtualização

O LXC possui diversas características que fazem dele uma ferramenta poderosa para testar e criar aplicações atraindo desenvolvedores (LINUX CONTAINER, 2008). Entre elas se destacam algumas:

i. *Kernel NameSpaces*: abstrai processos dentro do *kernel* possibilitando identificar *id* de processos e usuários, *hostname* e fila de processos isolados de outros processos ou grupos de processos.

ii. *Apparmor* e *SELinux*: define regras de acesso dentro de determinados diretórios e arquivos de um *host*. Isso previne que uma possível brecha de segurança de um *container* possa visualizar ou executar algo no *host*.

iii. *Seccomp policies*: é realizada uma filtragem nas chamadas de sistema que obedecem a políticas específicas, a nível de interface do *kernel*, pois o *container* e o *host* compartilham do mesmo *kernel*. Sendo assim o *container* não pode escalar privilégios dentro do *host*.

iv. *Chroots* (*pivo_root*): cria a árvore de diretórios na qual o *container* terá acesso.

v. *Kernel Capabilities*: possibilita o *container* executar alguns comandos no *host* de forma privilegiada.

vi. *Cgroups*: permite executar diferente *containers* com diferentes configurações de limite de uso como CPU e memória.

Com a consolidação da virtualização e dos ambientes em nuvem, o LXC foi se difundindo cada vez mais nas empresas e no mercado de trabalho. É possível portar uma aplicação direto do *notebook* do desenvolvedor para o servidor de produção ou um ambiente de testes, assim como acessá-la. A partir do LXC foram criadas outras tecnologias que se adequam às demandas de cada empresa. Uma das tecnologias baseadas em LXC mais difundidas no mercado é o *Docker* (DOCKER, 2013). A figura 2

mostra uma breve comparação entre *Docker* e máquina virtual.

Figura 2: comparação entre Docker(esquerda) e máquina virtual(direita).

2.4 Docker

O *Docker* é uma plataforma de código aberto lançada em 2013, escrita na linguagem de programação Go desenvolvida pela Google e possui um alto desempenho, que proporciona a criação e o gerenciamento de ambientes isolados. Foi concebida pela empresa DotCloud que inicialmente tinha o interesse de somente usá-lo para aplicações internas, mas que acabou sendo muito bem aceito no mercado e tomou um rumo diferente. Sua facilidade em gerenciar *containers* acabou levando a empresa a desenvolver sua própria biblioteca, deixado as nativas do LXC de lado e assumindo o controle dos drivers diretamente com o *kernel* do *host* (ALMEIDA, 2015).

A portabilidade do *Docker* é uma de suas principais características e uma das principais diferenças para o LXC, como mostrado na figura 3, pois além de possibilitar o

empacotamento de uma aplicação dentro de um *container*, é possível portá-lo para qualquer *host* que possua a plataforma *Docker* instalada sem a necessidade de ficar configurando o *host* ou o *container* a todo momento. Essa característica diminui muito o tempo de execução das aplicações ou até mesmo da implantação de infraestruturas, pois uma vez configurado o *container*, basta o *host* possuir o *Docker* instalado que o desenvolvedor só precisa replicar os *containers* e executá-los sem realizar nenhum tipo de configuração (DOCKER, 2017).

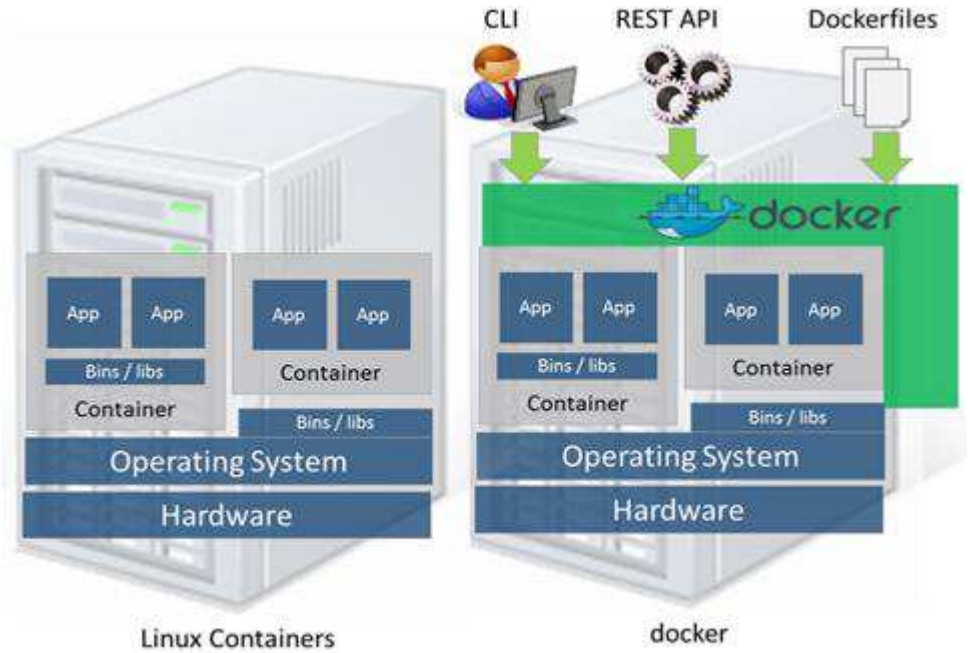

Figura 3: comparação entre LXC e Docker

O funcionamento do *Docker* é baseado em chamadas de cliente e servidor entre o *Docker Daemon* e o *Docker Client* através de uma *API*. Basta ter o *Docker* instalado em algum lugar e direcionar ro seu *Docker Client* para esse servidor (DOCKER, 2017).

Uma das principais bibliotecas do *Docker* e a mais relevante para este projeto é a biblioteca *libconteiner*, que é responsável por fazer a comunicação entre o *Docker Daemon* e o *backend* da plataforma que utiliza o LXC. Esta biblioteca é responsável pela criação dos *containers* e ela permite a configuração dos limites de recursos de cada *container* isoladamente. A figura 4 mostra o funcionamento do *Docker* descrito (DOCKER, 2017).

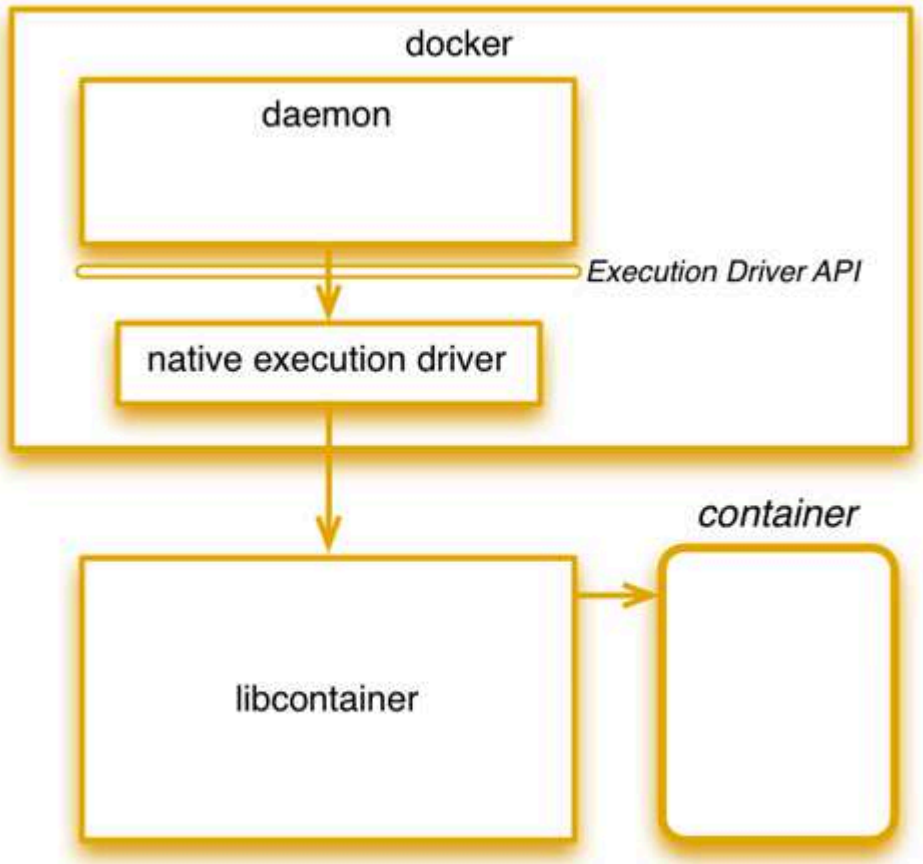

Figura 4: esquema de funcionamento do Docker

Assim como várias empresas do mercado atualmente, este projeto utiliza a plataforma *Docker* para realizar a criação de *containers* e gerenciá-las na simulação de um ambiente de testes através de uma plataforma que age como um sistema operacional de gerenciamento de *data centers* chamado DC/OS.

Como citado na Introdução, o DC/OS age como um sistema operacional que gerencia diversas máquinas de forma distribuída, possibilitando executar aplicações abstraindo a alocação de recursos. Seu desenvolvimento foi feito pela empresa americana *Mesosphere*, que desenvolve *softwares* para *data centers* utilizando o *Apache Mesos*, e lançado em 2016.

2.5 Apache Mesos

O *Mesos* é um *software* de código aberto desenvolvido por estudantes e um professor da Universidade da Califórnia em Berkeley, que tem como principal funcionalidade o gerenciamento de *clusters*. Foi apresentado em 2009 na publicação de um artigo (HINDMAN; KONWINSKI; ZAHARIA; STOICA, 2009) mas somente em 2016 a Apache Software Foundation lançou a primeira versão do software divulgando uma nota em seu blog oficial.

A arquitetura do *Mesos* é baseada em um processo mestre que executa em um nodo mestre. Existe um grupo de nodos mestres que elegem um líder entre eles para gerenciar os recursos dos *clusters* e facilitar a orquestração de tarefas. Além dos nodos

mestres existem também os agentes, que contêm os recursos disponíveis de um nodo agente. E por fim os *frameworks* que são compostos de *schedulers*, que recebem os recursos do nodo líder, e de executores, que executam as tarefas do *framework* no nodo agente. A figura 5 ilustra a arquitetura descrita (APACHE, 2017).

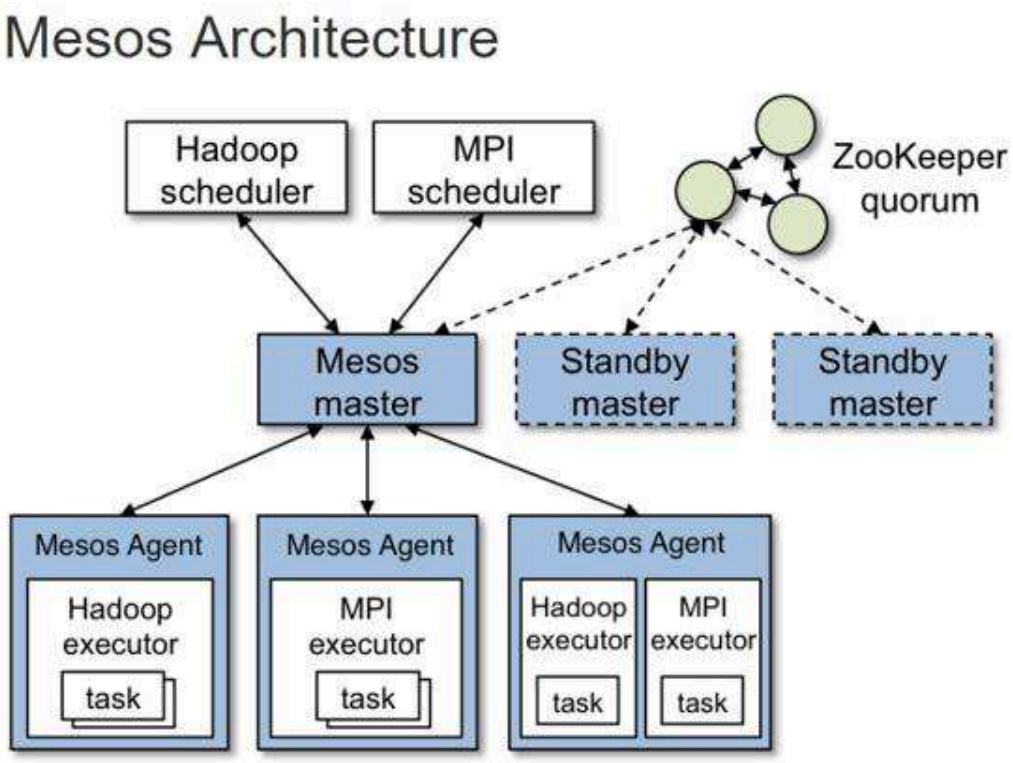

Figura 5: arquitetura básica do Mesos

O fluxo das rotinas que o Mesos executa se dá da seguinte maneira:

i. O agente envia ao nodo líder os recursos disponíveis do nodo agente.

ii. O nodo líder recebe os recursos e envia para o *framework* definido por um módulo de política de alocação.

iii. O *scheduler* do *framework* responde o nodo líder enviando as tarefas a serem executadas e a quantidade de recursos que elas irão consumir.

iv. O nodo líder envia ao agente a quantidade de recursos necessárias para a execução das tarefas.

v. O agente aloca os recursos necessários no nodo agente para que o executor do *framework* consiga realizar as tarefas necessárias.

A figura 6 ilustra as etapas descritas.

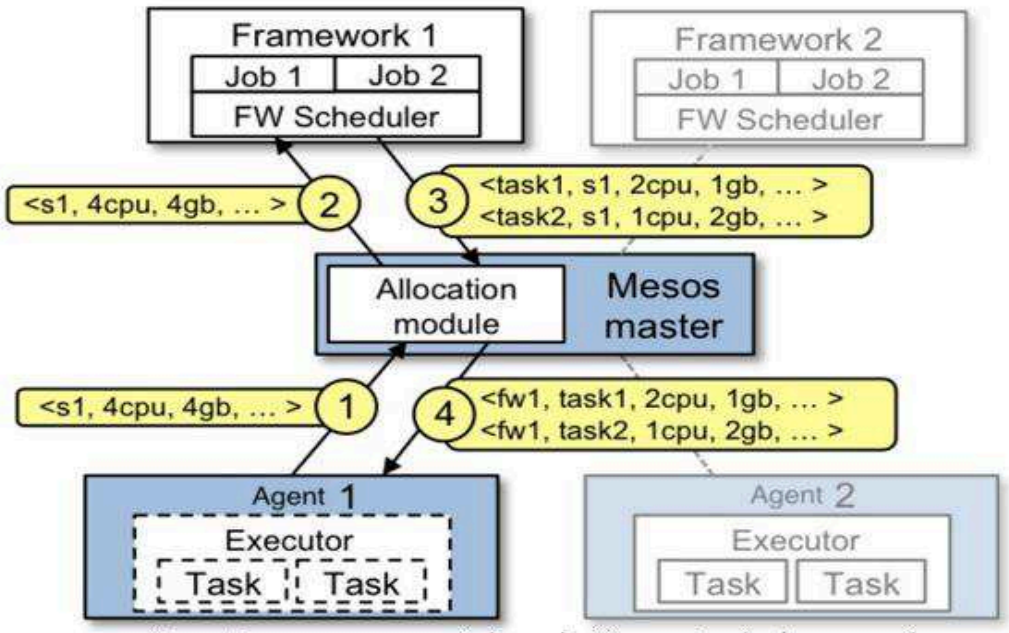

Figura 6: processos executados pelo Mesos através de seus nodos

Uma vantagem desse tipo de arquitetura é que se o nodo líder ficar inoperante, os outros nodos mestres candidatos fazem uma nova eleição para tomar o lugar de líder e continuar as operações fazendo com que os *clusters* nunca fiquem sem gerenciamento. Outra característica importante desta arquitetura é que os *frameworks* são independentes e têm a capacidade de rejeitar as ofertas de recursos do nodo líder que não seguirem as restrições impostas pelo próprio *framework*. Um dos *frameworks* mais utilizados pelo *Mesos* e pelo DC/OS é o *Marathon*, que realiza a orquestração de *containers*. O *Marathon* será descrito na próxima seção juntamente com o DC/OS.

Devido às características e vantagens apresentadas, o *Mesos* tem grande aceitação no mercado. É utilizado por diversas grandes empresas do ramo da tecnologia entre elas a rede social Twitter, o aplicativo Airbnb de aluguel de imóveis, o serviço de reconhecimento de voz Siri da Apple, e a empresa Yelp que utiliza um aplicativo para avaliar estabelecimentos comerciais.

2.6 DC/OS

O *Datacenter Operating Systems* (DC/OS), como dito em capítulos anteriores, é um sistema operacional para *data centers*. É composto de vários componentes e *softwares* de código aberto que o fazem um software completo em várias áreas diversificadas (MESOSPHERE, 2017b).

Funciona como um sistema distribuído que é executado em um *cluster* de nodos e não somente em uma máquina. Como em diversos sistemas distribuídos, possui nodos mestres que comandam nodos agentes, e há uma eleição entre os nodos mestres para definir um nodo líder, como no *Mesos*. Também possui a funcionalidade de um gerenciador de *clusters*, que gerencia tanto os recursos quanto as tarefas executadas dos nodos agentes, utilizando o *Apache Mesos* descrito na seção anterior. Outra funcionalidade é a capacidade do DC/OS de orquestrar *containers*, utilizando o *Marathon* como *scheduler*, e o *Docker* e o *Mesos* como tecnologias de *container*. O DC/OS também permite a customização dos *schedulers* o que amplia as possibilidades dos desenvolvedores, e os permite adequar seu *schedulers* de acordo com suas necessidades específicas. Como um sistema operacional, abstrai os recursos de *software* e *hardware* e provê alguns serviços comuns às aplicações como serviços de rede, gerenciamento de pacotes, armazenamento e segurança. A interface é amigável ao usuário e permite o gerenciamento dos *clusters* do DC/OS de maneira simples como mostra a figura 7 (MESOSPHERE, 2017b):

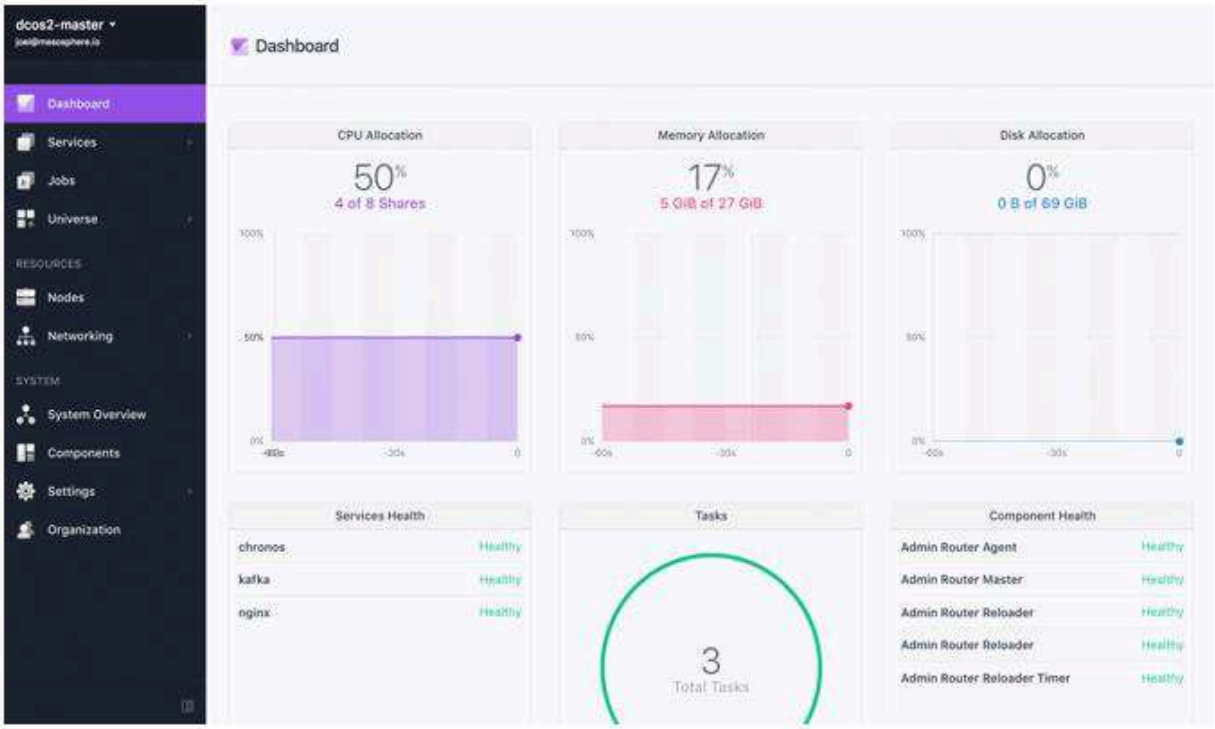

Figura 7: interface gráfica do DC/OS em um navegador web

A arquitetura do DC/OS pode ter um *hardware* tanto físico quanto virtual, desde que forneça os serviços comuns como rede e armazenamento. Independentemente de ser físico ou virtual, a arquitetura do DS/OS apresenta as seguintes camadas:

i. Camada de *Software*: provê gerenciamento e repositório de pacotes para instalar e gerenciar diversos tipos de serviços tais como banco de dados, *logs*, ferramentas de integração contínuas. Além desses serviços, o usuário pode instalar serviços e aplicações personalizados.

ii. Camada da Plataforma: essa camada possui diversos componentes que estão separados nas categorias de gerenciamento de *cluster*,

orquestração de *containers*, ambiente de *containers*, rede, gerenciamento de pacotes, segurança e armazenamento. Esses componentes também são distribuídos entre os tipos de nodos que são definidos como mestres, agentes públicos ou privados. Para instalar o DC/OS um dos nodos deve conter um dos sistemas operacionais suportados, *CentOS*, *RHEL* e *CoreOS*. A figura 8 mostra os requisitos de *hardware* de cada tipo de nodo.

Nodo Mestre

	Minimum	Recommended
RHEL/CentOS	7.2, 7.3	7.2, 7.3
CoreOS	1235.9.0	1235.9.0
Nodes	1	3 or 5
Processor	4 cores	4 cores
Memory	32 GB RAM	32 GB RAM
Hard disk	120 GB	120 GB

Nodo Agente

	Minimum	Recommended
RHEL/CentOS	7.2, 7.3	7.2, 7.3
CoreOS	1235.9.0	1235.9.0
Nodes	1	6 or more
Processor	2 cores	2 cores
Memory	16 GB RAM	16 GB RAM
Hard disk	60 GB	60 GB

Figura 8: tabela de requisitos de hardware dos nodos

iii. Camada de Infraestrutura: o DC/OS pode ser instalada em *hardware* local e em nuvens públicas ou privadas desde que esses ambientes possuam rede IPv4 compartilhada e máquinas x86.

A figura 9 exemplifica a arquitetura apresentada.

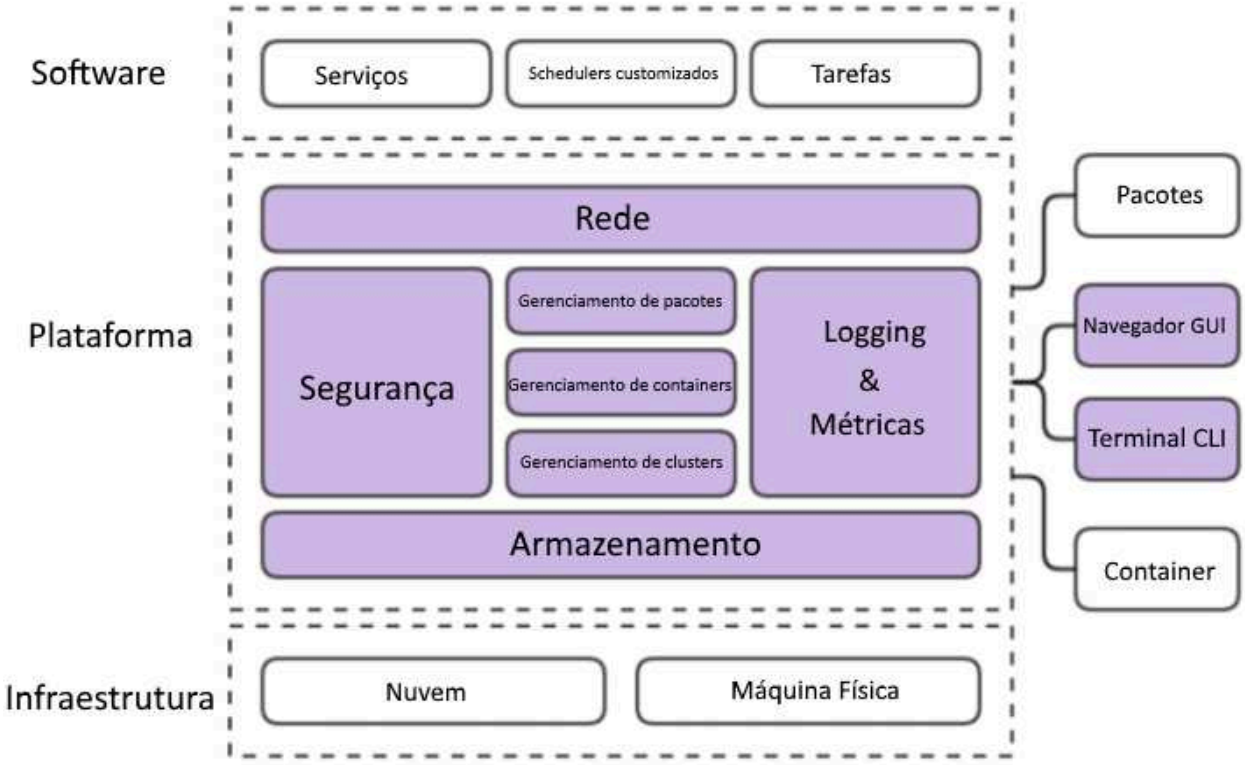

Figura 9. Camadas de arquitetura do DC/OS

Com a descrição dos conceitos apresentados neste capítulo de fundamentação teórica é possível compreender as tecnologias utilizadas na ferramenta desenvolvida neste projeto. No próximo capítulo será apresentada a ferramenta de orquestração que foi desenvolvida para apresentar uma ideia de solução para o problema de gerenciamento de fila de testadores em um ambiente de homologação.

3 Ferramenta

3.1 Problema e Solução

Como opção para que vários testadores consigam homologar versões diferentes do *software* ao mesmo tempo, é possível utilizar tecnologias de virtualização e *containers* para executar as versões de forma independente e automatizada, sem a necessidade de configuração, controle de servidores e ambientes para executar novas aplicações.

Uma possibilidade seria utilizar o DC/OS com o *Marathon* para gerenciamento destes *containers*. Entretanto, mesmo com toda a abstração que esta plataforma traz, ainda se torna custoso, por parte da equipe de teste, o aprendizado e uso de suas *APIs* para realizar tarefas como executar e derrubar as aplicações no *cluster* DC/OS. Além disto, como os recursos no *cluster* podem ser escassos, deve-se limitar a quantidade de execuções de aplicações no *cluster* por equipe. Portanto, é importante o controle sobre o uso, para que todas as equipes consigam testar em paralelo evitando a criação de filas, prática comum quando se tem apenas um ambiente de teste.

Com isto, surge a necessidade de uma ferramenta para realizar este gerenciamento e controle, através de perfis de acesso, das aplicações que são executadas no *cluster* DC/OS. Esta ferramenta seria responsável por receber, através de uma *API RESTful*, pedidos para executar aplicações no *cluster* através de uma identificação de conta. Desta forma, cada conta poderia ter um limite máximo de aplicações sendo executadas simultaneamente, com esta regra sendo controlada pela ferramenta.

3.2 Especificação

Como forma de solucionar o problema apontado anteriormente, foi desenvolvido um *software* que permite o *deploy* de aplicações em um cluster DC/OS, através de uma imagem *Docker*, com o uso restrito à uma limitação pré-estabelecida. Um *deploy*, neste caso, é a execução de uma aplicação em um cluster DC/OS. Esta aplicação recebe estas requisições através de uma *API*, e se comunica com o *Marathon* do *cluster* DC/OS para executar ou interromper a execução das aplicações.

Para mapear corretamente as aplicações de cada usuário, o sistema deve permitir que um *host* seja especificado na chamada para a realização do *deploy.* Este *host* seria utilizado para mapear, dentro do *cluster*, para a aplicação correta do usuário.

Também deve ser possível requisitar o cancelamento de uma aplicação, ou seja, interromper a execução de uma aplicação, por parte do usuário, para liberar os recursos conforme o necessário.

A ferramenta deve se comunicar com um banco de dados PostgreSQL que possui duas tabelas, uma que se refere aos usuários chamada *User* e uma referente às aplicações que serão executadas no *cluster* chamada *App*.

3.3 Implementação

Para implementar a solução, foi desenvolvido uma aplicação *web* com uma *API* que permite este controle. A linguagem de programação escolhida foi o *Ruby*, utilizando o *framework web Sinatra* e o servidor *Unicorn*. Foram definidas duas entidades principais para o modelo de dados:

i. User: o usuário cadastrado, contendo o limite de aplicações simultâneas permitido. O usuário contém também uma chave de acesso única chamada token de acesso, que permite a utilização da api e das funcionalidades da ferramenta. Com isso é possível controlar o acesso, e o limite das aplicações simultâneas garante um gerenciamento eficaz das aplicações que estão sendo executadas no cluster DC/OS.

ii. App: *host* e *status* da aplicação, relacionado com o usuário que a requisitou. O *host*, indica de onde o contêiner que está subindo deve receber requisições. O status pode ser:

    a. *not running:* A aplicação foi criada, mas ainda não está no ar.

    b. *running:* A aplicação se encontra no ar (através do Marathon no DC/OS)

c. *done:* A aplicação já esteve no ar, mas neste momento se encontra desligada.
d. *failed:* Ocorreu uma falha ao executar esta aplicação

Para que os usuários possam ser gerenciados, foram criados alguns *endpoints* exclusivos para estas tarefas de administração. O controle acontece através do *token* de acesso compartilhado que identifica os administradores. Para realizar a requisição, deve-se identificar com o *token* de acesso através de um cabeçalho de autenticação, utilizando o método de autenticação básica *HTTP*. Os *endpoints* serão descritos abaixo:

1. GET /users:
    a. Retorna um JSON contendo todos os usuários, com os campos *id*, representando o identificador do usuário na tabela Users, *name* representando o nome do usuário, e *max_deploys*, que representa o número máximo de aplicações que o usuário pode executar simultaneamente no cluster DC/OS.
    b. Se for enviado um parâmetro no *querystring* de “access_token” com o valor “true”, o JSON de resposta retornará também o access_token de cada usuário, que é uma chave única representada por um hash que serve para autenticar o usuário.

Na figura X podemos perceber na *querystring* que o parâmetro *acess_token* está definido com o valor *true* e por isso retornará o *token* de acesso dos usuários no *JSON*. No caso temos somente um usuário cadastrado.

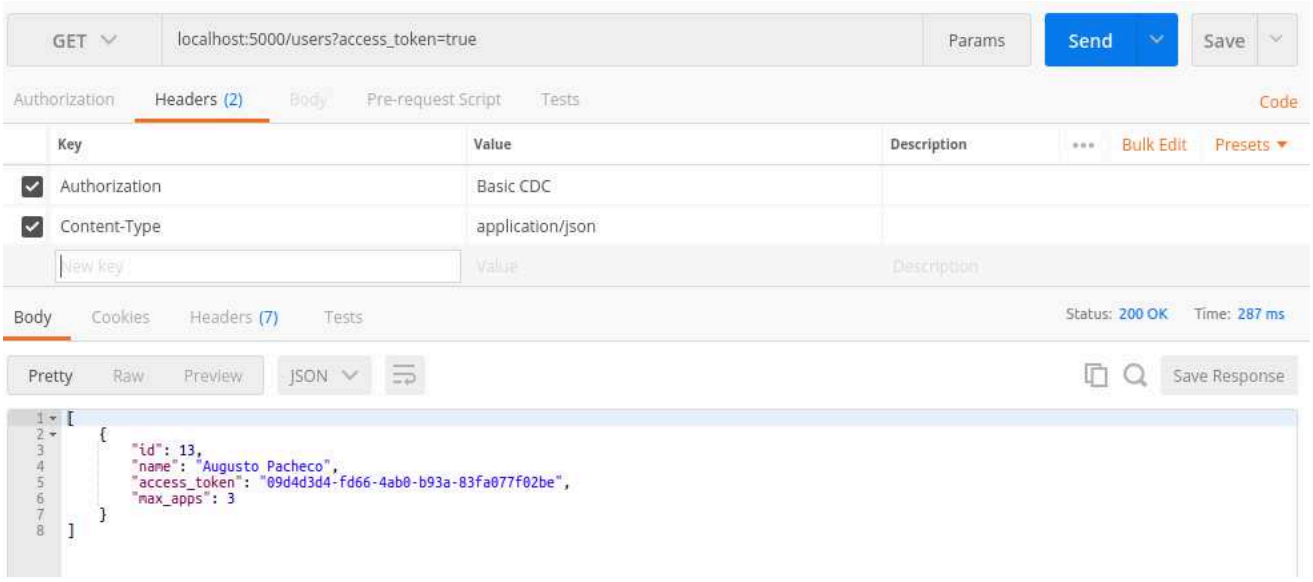

Figura 10 - Requisição exibindo os *access token* dos usuários

2. POST /users:
    a. Parâmetros (JSON):
        i. name: nome do usuário.
        ii. max_apps: quantidade máxima de aplicações que este usuário pode manter simultaneamente.
        iii. Ambos são obrigatórios. Retornará o código HTTP 422, caso ambos não estejam presentes.
    b. Retorna o usuário criado, contendo os campos *id*, *name*, *max_apps* e *access_token*. Ex.: {"id": 14,"name": "Felipe Duarte", "max_apps": 1, "access_token": "cd265ce6-5b85-4c1f-cd3bd6b120b7”}.

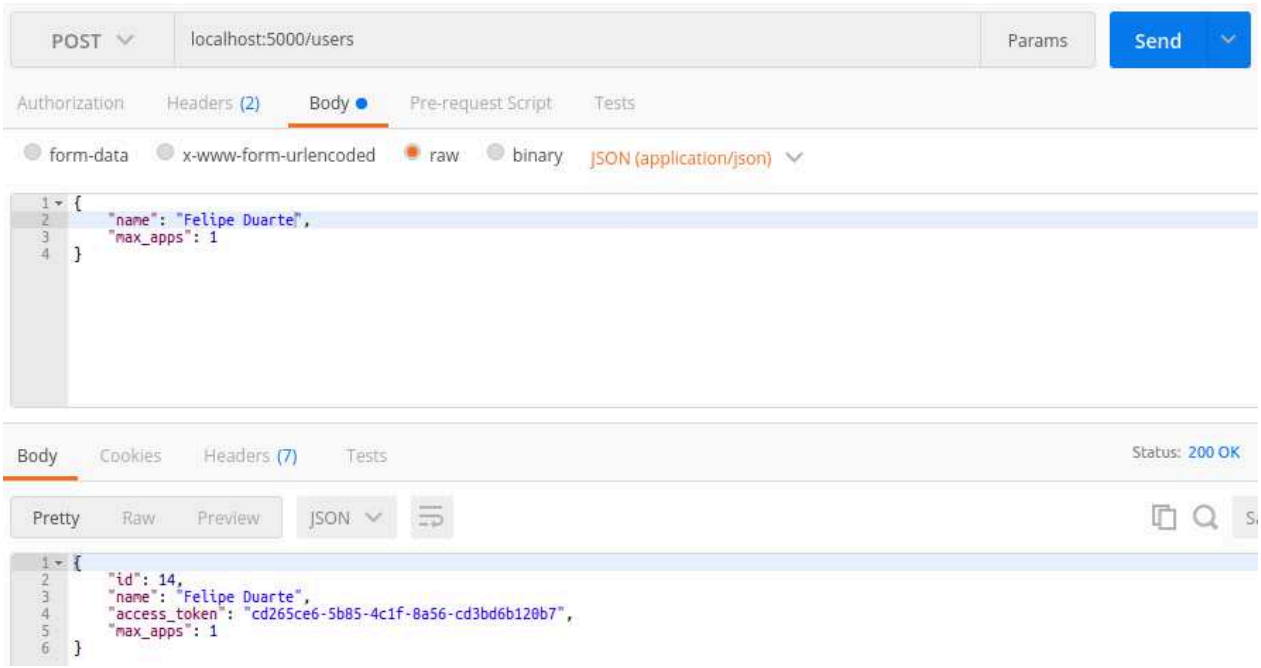

Figura 11 - Criação de um usuário na ferramenta

3. PUT /users/:id
    a. Parâmetros (JSON):
        i. id (via o *path*): ID do usuário
        ii. name: nome do usuário
        iii. max_apps: quantidade máxima de aplicações que este usuário pode manter simultaneamente
    b. Retorna o usuário atualizado, contendo os campos *id*, *name*, *max_apps* e *access_token*. Ex.: {"id": 1,"name": "Bob", "max_apps": 10, "access_token": "abc-123" }

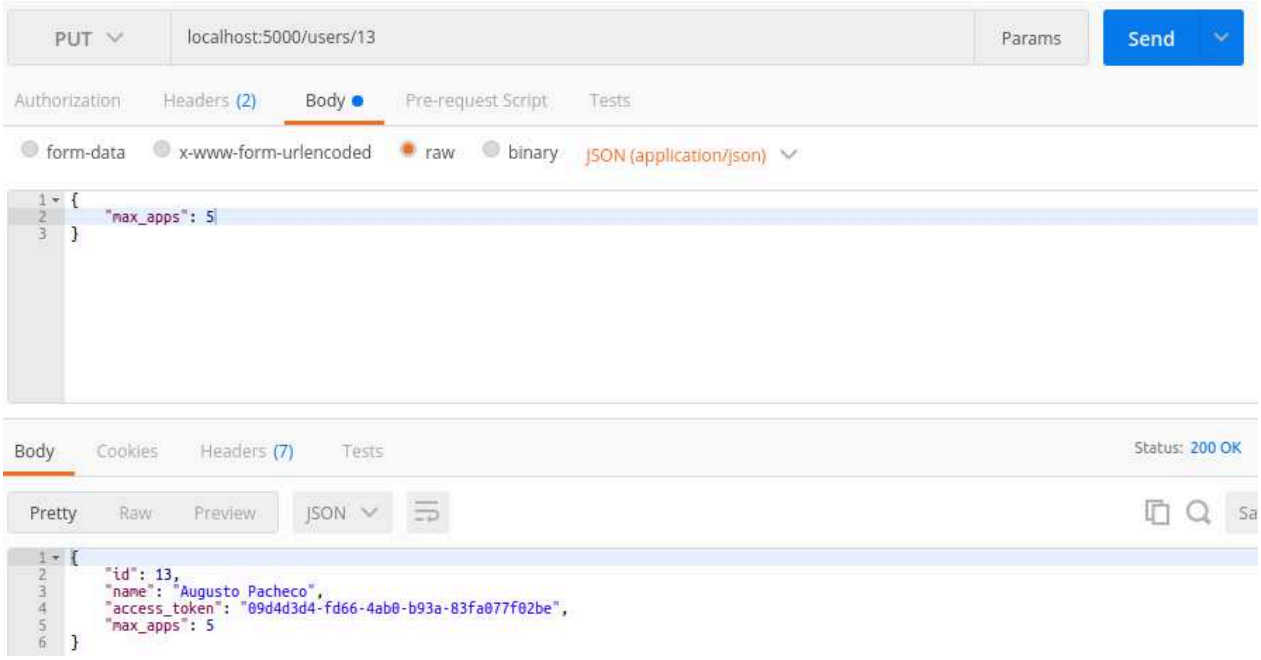

Figura 12 - Atualização de um usuário

Assim que um usuário é cadastrado, ele recebe um *access_token*. Este *token* deve ser informado através de um parâmetro *access_token* no *payload* da requisição, em todas as requisições realizadas pelo usuário, para autenticar e identificá-lo. Os *endpoints* para utilização do usuário são:

1. GET /apps/:id
    a. Parâmetros:
        i. id (via o *path*): ID da aplicação.
    b. Retorna um JSON contendo o campo *id* e *status* da aplicação. Ex.: { “id”: 29, “status”: “running” }.
    c. Se não existir uma aplicação registrada com este ID retorna código HTTP 404.

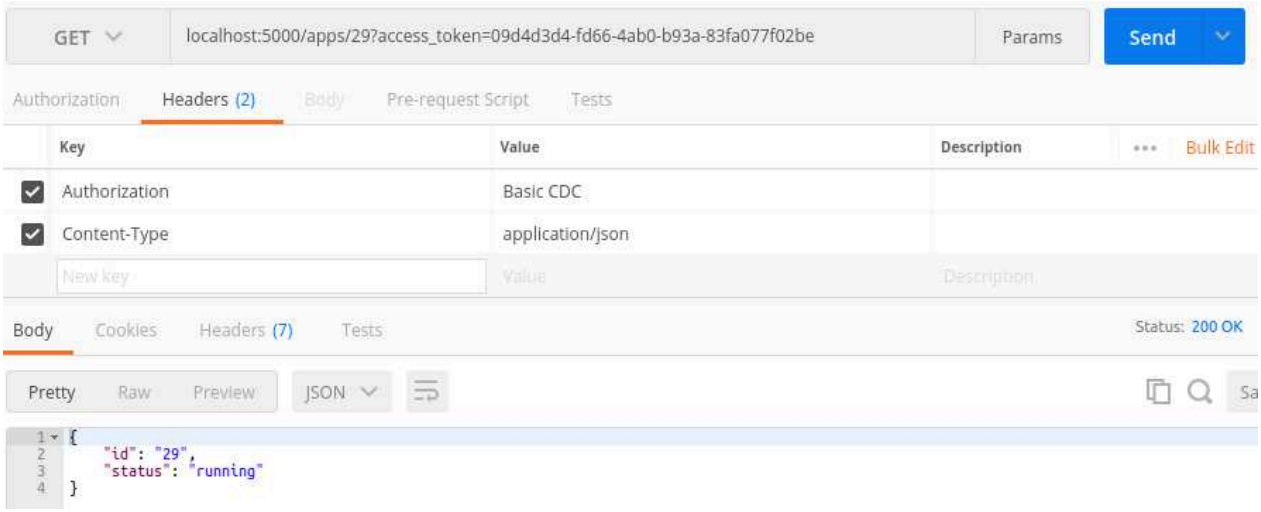

Figura 13 - Informações da aplicação de id 29

2. GET /apps

    a. Retorna todas as aplicações para este usuário, com os campos *id*, *status* e *host*. Ex.: { “apps”: [{ “id”: 1, “status”: “running”, “host”: “app-1.staging” }, { “id”: 2, “status”: “done”, “host”: “app-2.staging” }] }.

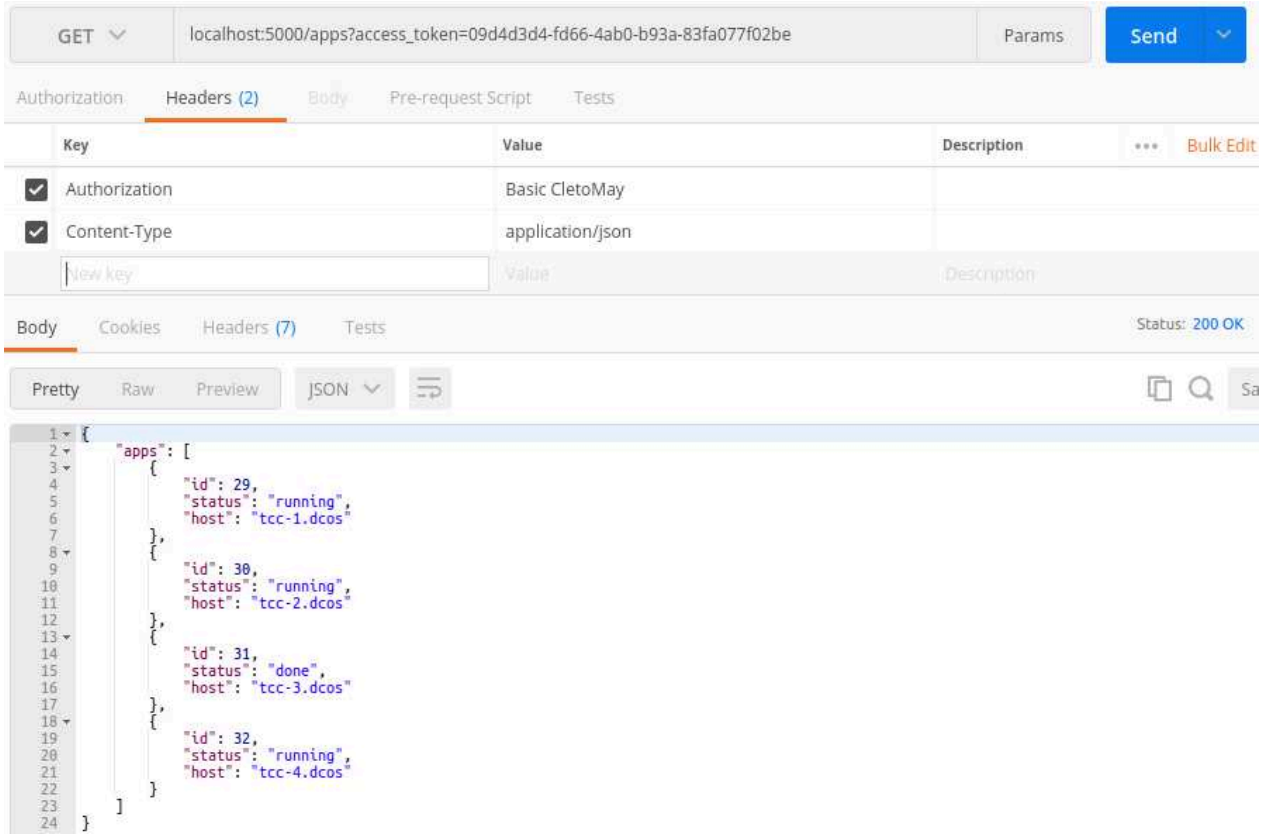

Figura 14: Todas as aplicações do usuário

3. POST /apps
    a. Parâmetros:
        i. acess_token
        ii. image: nome da imagem Docker que o DC/OS deverá executar no formato “repository/image_name:tag”.
        iii. *host*: o host que o Marathon usará para mapear a aplicação.
        iv. Ambos são obrigatórios. Retornará o código HTTP 422 caso ambos não estejam presentes.

b. Caso o usuário já esteja no limite de aplicações (número máximo de aplicações = número de aplicações no ar), retornará código HTTP 400 e o *deploy* será rejeitado.

c. Realiza o *deploy* da aplicação no *Marathon*, utilizando uma configuração mínima de CPU e memória. Retorna um JSON com a aplicação criada, e os campos *id*, *user_id*, *status* e *host*. Ex.: { “id”: 1, “user_id”: 2, “status”: “running”, “host”: “app-1.staging” }.

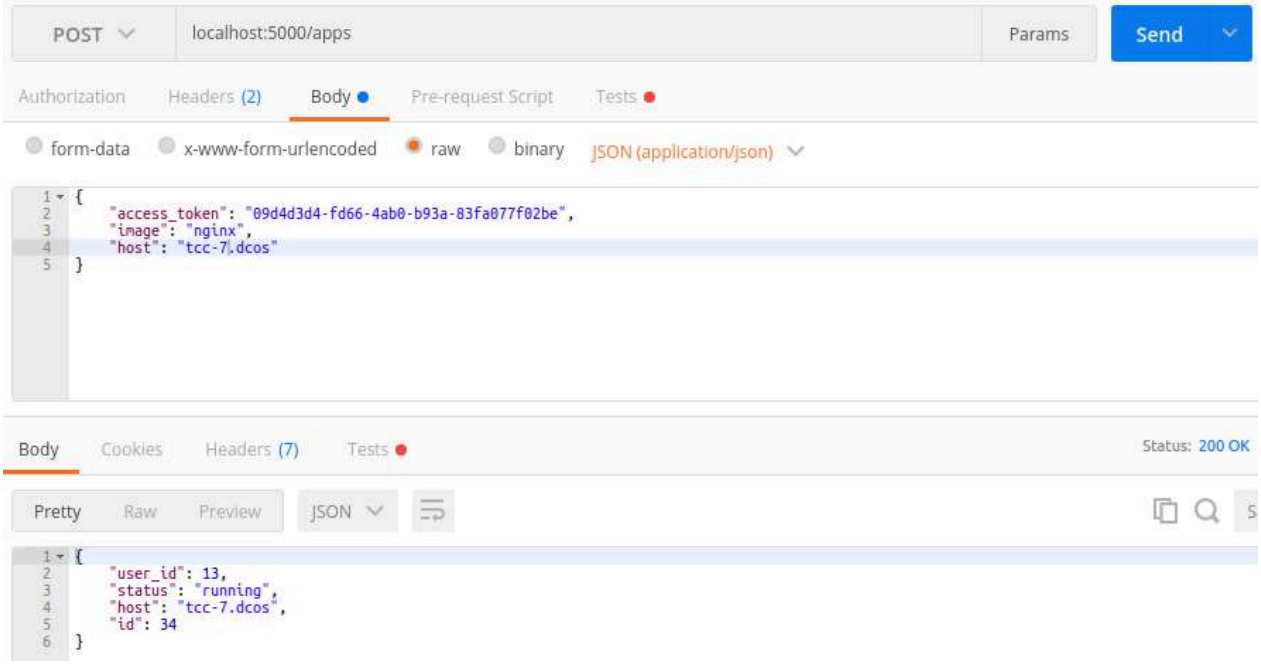

Figura 15 - Aplicação executando com sucesso para o host informado

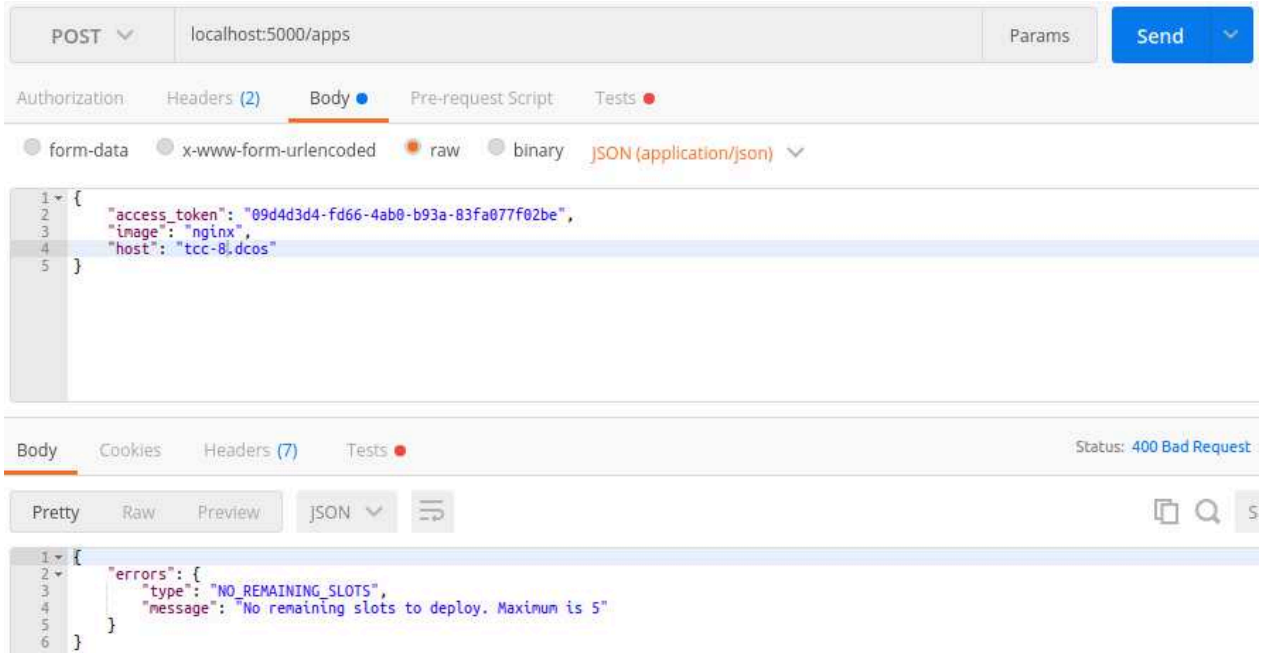

Figura 16 - Execução da aplicação sendo recusada devido ao limite de aplicações do usuário

4. DELETE /apps/:id
    a. Parâmetros:
        i. id (via o *path*): ID da aplicação.
        ii. acess_token
    b. Derruba a aplicação com o id informado.
    c. Retorna erro HTTP 404 se não existir uma aplicação com esse id.
    d. Se não houver uma aplicação rodando com esse id retorna erro HTTP 400.

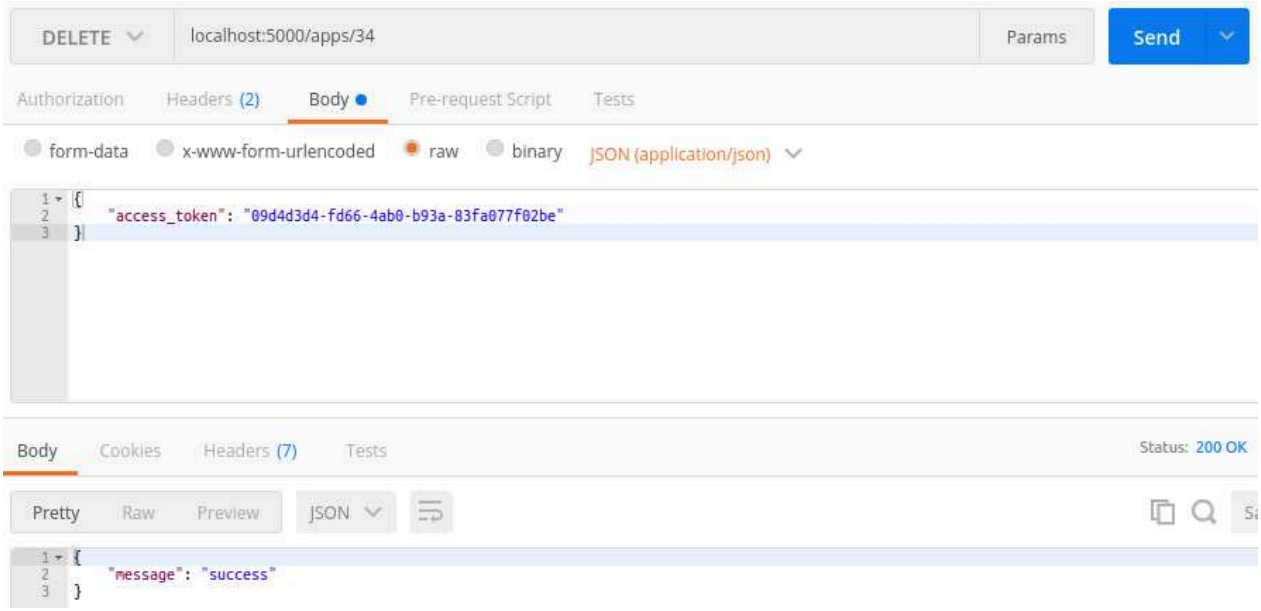

Figura 17 - Remoção de uma aplicação

Para utilização da ferramenta, deve-se especificar algumas variáveis de ambiente para configurar o servidor, o banco de dados e o endereço do *Marathon*:

1. WEB_CONCURRENCY: número de *workers* (concorrência) do servidor Unicorn. Ex.: 5.
2. WEB_TIMEOUT: Valor, em milissegundos, de *timeout* para as requisições. Ex.: 30000.
3. DATABASE_URL: Connection *string* do banco de dados, a ser utilizado pela biblioteca *Sequel*. Ex.: postgres://user:pass@localhost:5432.
4. API_TOKEN: *Token* necessário para autenticação HTTP, utilizado nos *endpoints* administrativos (gerenciamento dos usuários). Ex.: a99c8482-ca44-4928-91da-9e778d4c5dbf.

5. MARATHON_URL: URL do *Marathon*, para ser utilizado pela ferramenta para subir e remover as aplicações. Ex.: http://m1.dcos/marathon.

3.4 Caso de Uso

A ferramenta foi desenvolvida dentro da arquitetura demonstrada na figura 18. O usuário se comunica com a ferramenta que, através de um banco de dados PostgreSQL, gerencia e controla as execuções de aplicações. A ferramenta, por sua vez, se comunica com o DC/OS através da API do Marathon. É nesta etapa que ocorre a execução da aplicação com Docker, conforme detalhado na figura 19.

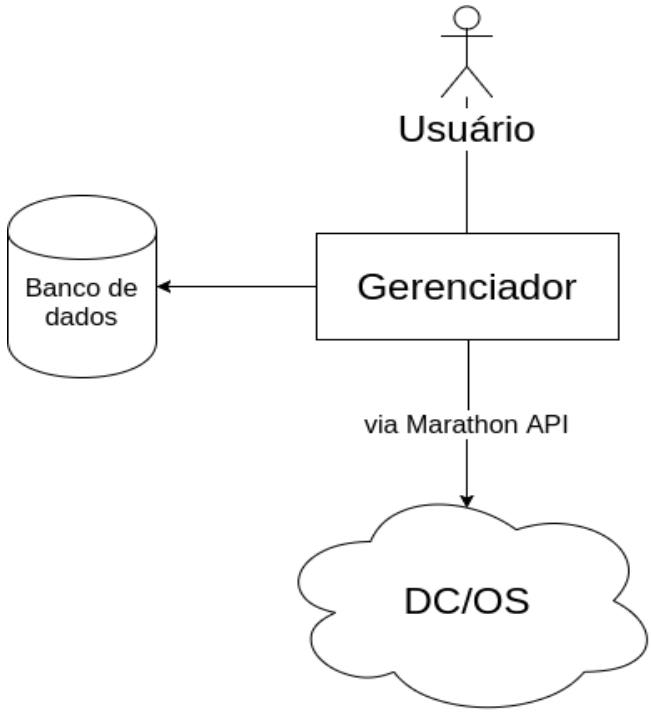

Figura 18 - Diagrama da ferramenta

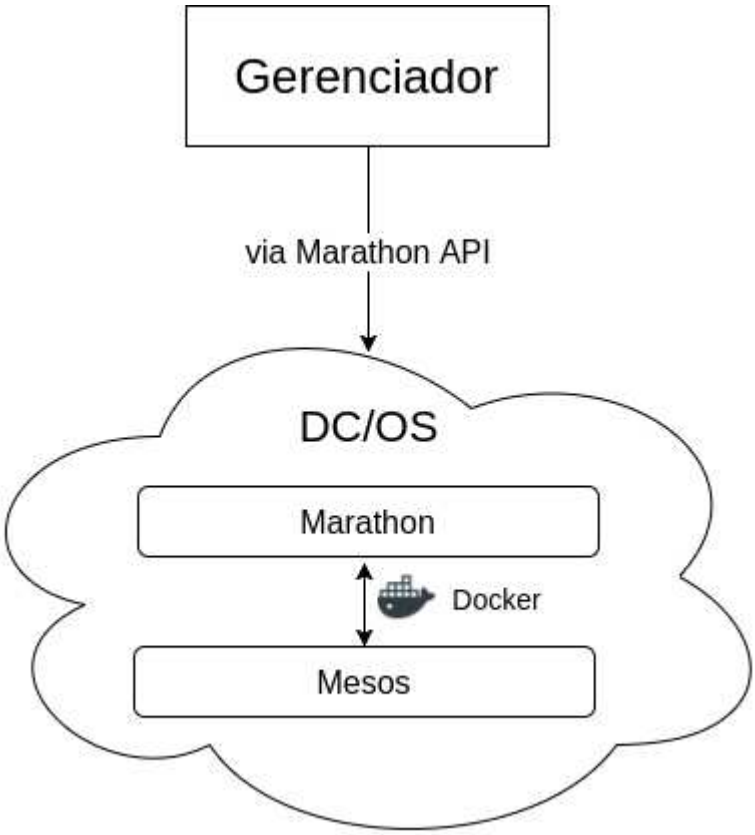

Figura 19 - Detalhe da iteração entre a ferramenta e o DC/OS, através do Marathon

Na etapa de testes do processo de desenvolvimento de *software* de uma empresa, por exemplo, as equipes de testadores precisam homologar suas aplicações em um ambiente de teste. Para isso eles executam versões da aplicação na qual estão desenvolvendo novas funcionalidades.

O problema é que somente uma versão pode ser executada por vez pois não há um meio de gerenciar as versões de maneira automatizada, de modo a criar uma fila de testadores querendo homologar suas versões, atrasando o lançamento de novas funcionalidades da aplicação.

A ferramenta desenvolvida neste projeto possibilita que mais de uma versão de uma mesma aplicação possa ser lançada no ambiente de homologação. Ela gerencia todas as versões respeitando restrições de acesso e número máximo de aplicações que um usuário pode executar no cluster, de maneira a acabar com a fila de testadores e otimizando o processo de homologação acarretando no lançamento de mais funcionalidades da aplicação com um tempo de espera menor.

A figura 20 exemplifica uma requisição, utilizando o programa *Postman*, para a ferramenta:

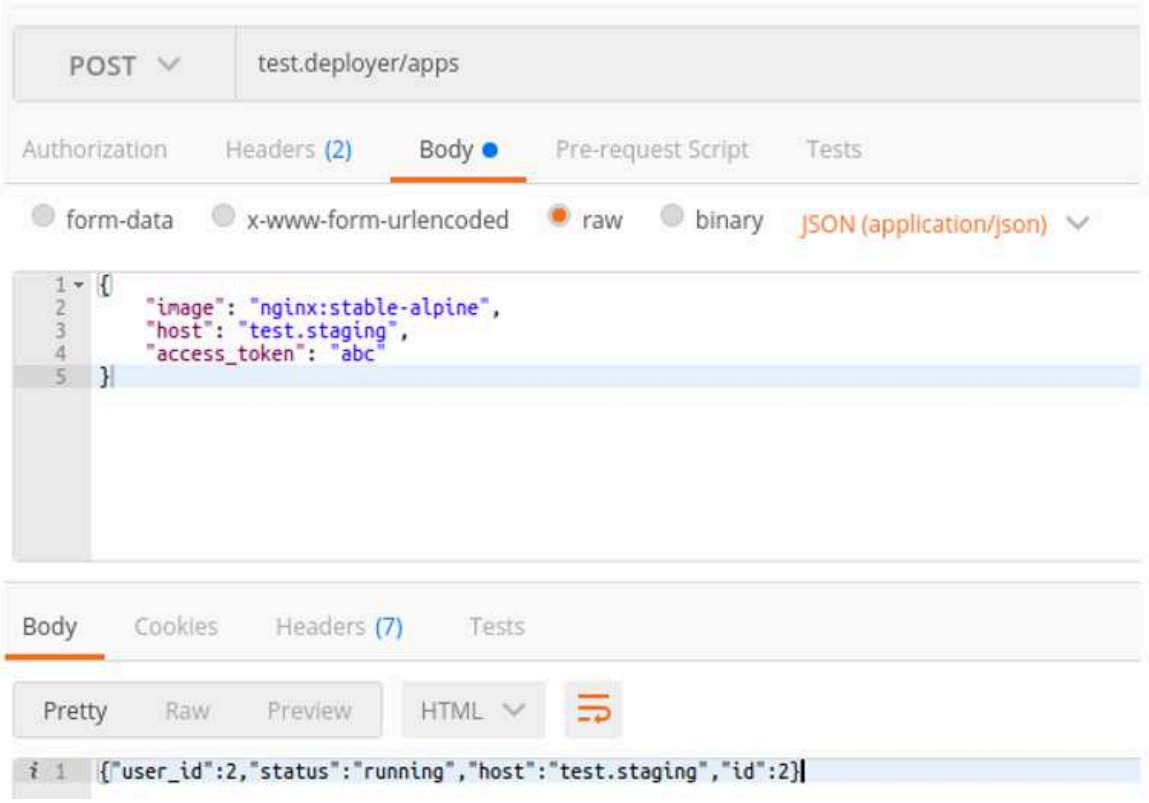

Figura 20 – Postman executando o endpoint POST/apps

Isto deve executar a aplicação no *cluster* DC/OS configurado na ferramenta, associando com o usuário do respectivo *access_token* informado. O mapeamento para o *host* também será efetuado no *Marathon*, garantindo que a aplicação solicitada esteja acessível pelo testador.

Supondo que este usuário tenha um limite máximo de 1 aplicação, ao tentar novamente ele recebe como resposta um erro, com a mensagem de erro segundo a figura 21 abaixo:

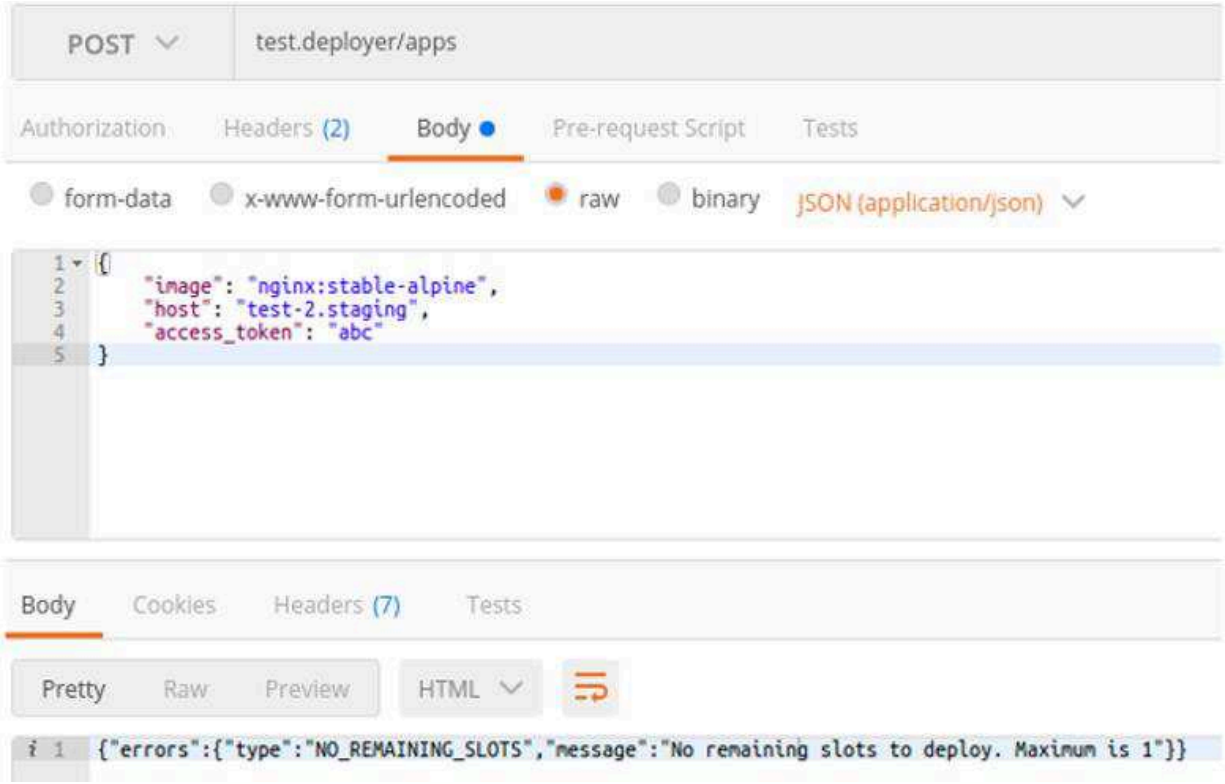

Figura 21 - Erro de um usuário tentando executar mais aplicações que seu limite permitido

Mesmo assim, caso um outro usuário, ou seja, com outro *access_token* e com limite suficiente, tente executar a aplicação da mesma forma, ele conseguirá devido ao limite individual de aplicações. Conforme demonstrado na figura 22:

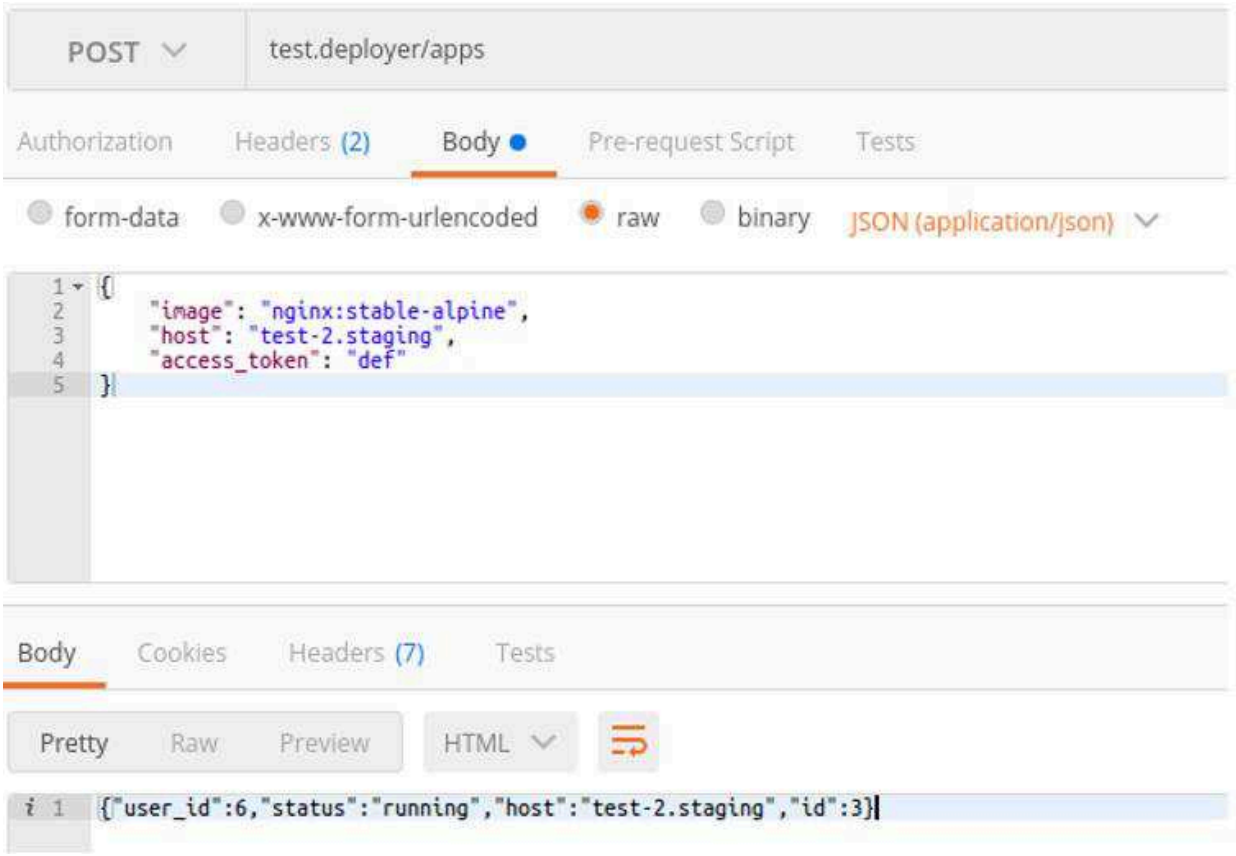

Figura 22 - Usuário diferente consegue executar aplicações no cluster pois cada um possui limites independentes

Desta forma, a ferramenta consegue gerenciar com sucesso a fila de testadores, impondo regras de acesso e limites de aplicações - definidas na hora do cadastro do usuário. Com isto, além de acelerar o processo de desenvolvimento, também permite a

economia de recursos, e custos operacionais, ao limitar as execuções por usuário, fazendo uma distribuição justa conforme o critério desejado.

Com estes métodos, e os demais citados no capítulo 3.3, a ferramenta possui uma gama de funcionalidades básicas para gerenciar uma fila de testadores que querem executar suas aplicações, automatizando o processo e permitindo que o fluxo de homologação seja mais fluido.

4 Conclusão

Este trabalho teve como objetivo a implementação de uma ferramenta de orquestração em ambiente de testes na nuvem. Esta ferramenta tem o objetivo de gerenciar várias versões de uma mesma aplicação em um ambiente de teste, respeitando níveis de acesso e o número máximo de aplicações que cada usuário pode executar em um cluster, para que a fila de testadores aguardando a vez para testarem suas versões acabe, e o tempo de lançamento de novas funcionalidades da aplicação seja menor.

A implementação dessa ferramenta requereu um estudo aprofundado de diversas tecnologias muito atuais, como a tecnologia de *container* representada pelo *Docker*, plataformas de orquestração de *containers* como o *Marathon*, gerenciadores de *clusters* como o *Mesos* e o DC/OS que é um sistema operacional muito útil e consegue unificar as tecnologias citadas acima em um único ambiente.

A finalização deste projeto representa uma alternativa para equipes de desenvolvedores que possuem uma fila de espera para homologar suas versões. Um embrião que utiliza de tecnologias atuais e muito utilizadas no mercado, que permite gerenciar diferentes versões de uma mesma aplicação em um ambiente de testes de maneira simples. Com isso, a empresa pode acelerar os lançamentos de novas funcionalidades e correções, acelerando o processo de desenvolvimento.

Para trabalhos futuros, poderia ser realizada a criação de uma interface gráfica e amigável para o usuário, suporte a manipulação de banco de dados, escalar *clusters* DC/OS inteiros ao invés de aplicações, escalar *clusters* com banco de dados individuais para cada *cluster*, possibilitar a inclusão de uma funcionalidade que permita o gerenciador escolher o serviço de nuvem que ele quer lançar as aplicações de maneira que ele possa gerenciá-los de forma automatizada alocando recursos de um serviço para outro simplesmente usando a API da ferramenta.

Referências

ALMEIDA, J.M.B. **Container, um novo passo para a virtualização**. IBM Academy of Technology Affiliated. 2015. Disponível em: < https://www.ibm.com/developerworks/community/blogs/tlcbr/entry/mp234?lang=en>.

APACHE SOFTWARE FOUNDATION (Org). **Mesos.** Disponível em:< http://mesos.apache.org/>. Acessado em 15 mai. 2017.

BOURQUE, PIERRE; FAIRLEY, DICK. **SWEBOK 3.0 Guide to the Software Engineering Body of Knowledge**. [S.l.]: IEEE Computer. 2014

CENTOS (Org). Disponível em: < https://www.centos.org >. Acessado em 16 mai.2017

COHEN, D., LINDVAL, M., COSTA, P. **An introduction to agile methods. In Advances in Computers**. New York: Elsevier Science. p. 1-66. 2004

COREOS (Inc). Disponível em <https://coreos.com>. Acessado em 16 mai.2017.

DOCKER (Inc). **eBook: Docker for the Virtualization Admin**. 2016

DOCKER (Inc). **Container.** Disponível em: <https://www.docker.com/what-container>. Acessado em: 12 mai. 2017.

DOCKER (Inc). **Docker.** Disponível em: < https://www.docker.com/>. Acessado em 13 mai. 2017.

DU , W. **Chroot Sandbox Vulnerability Lab**. Syracuse University. 2009 Disponível em: <http://www.cis.syr.edu/~wedu/seed/Labs/Vulnerability/Chroot/Chroot.pdf>

HINDMAN, B., KONWINSKI, A., ZAHARIA, M., STOICA, I. **A Common Substrate for Cluster Computing**. University of Californa, Berkeley. 2009

ISO (Org): **ISO/IEC 25010** Disponível em: <https://www.iso.org/standard/35733.html>. 2011. Acessado em 15 mai. 2017.

KARLSTROM, D., RUNESON P. **Combining agile methods with stage-gate project management**. IEEE Software. v.22. 2005

LINUX (Org). Disponível em < https://www.linuxfoundation.org/>. Acessado em 13 mai. 2017.

LINUX CONTAINER (Org). Disponível em: < https://linuxcontainers.org/>. 2008. Acessado em 12 mai. 2017.

MANIFESTO ÁGIL (Org). Disponível em: < http://agilemanifesto.org/>. 2001. Acessado em 13 mai. 2017.

MESOSPHERE (Inc). **Marathon.** Disponível em: < https://mesosphere.github.io/marathon/>. Acessado em 15 mai. 2017

MESOSPHERE (Inc). **DC/OS.** Disponível em: <https://dcos.io/>. Acessado em: 12 mai. 2017a.

MESOSPHERE (Inc). **DC/OS.** Disponível em: <https://dcos.io/docs/1.9/overview/concepts/#dcos>. Acessado em: 12 mai. 2017b.

MESOSPHERE (Inc). **DC/OS.** Disponível em: <https://dcos.io/docs/1.9/overview/what-is-dcos/>. Acessado em: 12 mai. 2017c.

MESOSPHERE (Inc). Disponível em: <https://mesosphere.com/>. Acessado em: 12 mai. 2017.

MESOSPHERE (Inc). **DC/OS.** Disponível em: < https://dcos.io/docs/1.9/overview/architecture >. Acessado em: 12 mai. 2017d

MYERS, GLENFORD J. **The Art of Software Testing**. Ed. John Wiley & Sons, Nova Jérsei. 2004

NAIK, KSHIRASAGAR; TRIPATHY, PRIVADARSHI. **Software Testing and Quality Assurance**. Hoboken, New Jersey: Wiley. 2008

PRESSMAN, ROGER. **Engenharia de Software**. Ed. Makron Books. 1995

RED HAT (Inc). Disponível em: <https://www.redhat.com>. Acessado em 20 mai.2017.

RUBY (Org). Disponível em: < https://www.ruby-lang.org/pt/>. Acessado em: 20 mai. 2017.

THE APACHE SOFTWARE FOUNDATION BLOG (Org). Disponível em < https://blogs.apache.org/foundation/entry/the_apache_software_foundation_announces 97 >. 2016

Ferramenta para Gerenciamento de Aplicações em um Cluster DC/OS na Nuvem

Augusto Pacheco Santos de Souza[1], Felipe Duarte Silveira[1]

[1]Departamento de Informática e Estatística – Universidade Federal de Santa Catarina (UFSC) – Florianópolis – SC – Brasil

augusto.pss@grad.ufsc.br, felipe.duarte@grad.ufsc.br

Abstract. *Many companies need to test their applications before putting them into production and delivering a final version to the user or customer. In some cases the resources are scarce, the company has only one homologation environment and a queue of testers wanting to approve different versions of the same software. This project aims to implement a solution based on the Datacenter Operating Systems (DC/OS) technology, which allows to approve different versions of the same software at the same time obeying rules of access levels and permission. With this, a company could launch more versions of its applications, optimizing the speed of correction of bugs and the development of new features.*

Resumo. *Várias empresas necessitam testar suas aplicações antes de colocá-las em produção e disponibilizar uma versão final ao usuário ou cliente. Em alguns casos os recursos são escassos, a empresa possui somente um ambiente de homologação e uma fila de testadores querendo homologar diferentes versões do mesmo software. Este projeto visa implementar uma solução baseada na tecnologia Datacenter Operating Systems (DC/OS), que permita homologar diferentes versões do mesmo software ao mesmo tempo obedecendo regras de níveis de acesso e permissão. Com isso uma empresa conseguiria lançar mais versões de suas aplicações, otimizando a velocidade de correção de bugs e o desenvolvimento de novas funcionalidades.*

1. Introdução

As empresas de tecnologia em geral necessitam de um ambiente para homologar suas aplicações, ambiente este que precisa oferecer condições e ferramentas que possibilitem testes de funcionalidades e desempenho de uma maneira otimizada e que não gere um custo operacional elevado.

Vários ambientes de homologação utilizam o conceito da tecnologia de containers. A tecnologia de container permite a criação de várias instâncias isoladas de uma aplicação dentro de um único servidor de maneira simples e leve. Um container é um pacote que possui um conjunto de configurações, código, tempo de execução, bibliotecas, tudo que é necessário para executar o software de uma determinada aplicação (DOCKER, 2016). Desse modo, a estrutura gerada

através do container será independente do ambiente em que se encontra, evitando conflitos entre testadores que operam diferentes aplicações utilizando a mesma infraestrutura.

Como várias aplicações são testadas ao mesmo tempo, ocorre a criação de diversos containers no mesmo ambiente. Por isso se faz necessário a utilização de um software de gerenciamento.

Atualmente no mercado existem alguns softwares de gerenciamento de containers, dentre eles o Datacenter Operating Systems (DC/OS). Ele age como um sistema operacional que gerencia diversas máquinas de forma distribuída, possibilitando executar aplicações abstraindo a alocação de recursos. Assim equipes de testadores conseguem executar suas aplicações e gerenciá-las de um modo simples (MESOSPHERE, 2017a) (MESOSPHERE, 2017b).

O problema é que em algumas ocasiões, os recursos de uma empresa são escassos, com apenas um ambiente de homologação e uma fila de testadores querendo testar diferentes versões de um mesmo software, tornando o processo de homologação de novas versões um pouco lento.

Este projeto tem como objetivo otimizar o tempo que leva para se lançar correções ou novas funcionalidades de uma aplicação, implementando uma solução para que várias versões de uma mesma aplicação possam ser testadas e homologadas em um mesmo ambiente de testes, sem gerar nenhum conflito de versões, obedecendo regras de acesso.

2. Container

Criada em 2008, a tecnologia de Linux Containers (LXC) (LINUX CONTAINER, 2008), permite a virtualização de uma aplicação tendo como hospedeiro um servidor Linux. Essa tecnologia é um meio termo entre o antigo comando chroot (Unix V7, 1979) e uma máquina virtual convencional. Com o comando chroot é possível encapsular um sistema dentro de uma estrutura de diretório possibilitando o sistema hospedeiro acessar somente a parte que é especificada nesta estrutura. Existem algumas diferenças entre o LXC e o comando chroot sendo a principal delas a questão da segurança das informações, onde o chroot se destaca negativamente por ser conhecido por sua vulnerabilidade (DU, 2009).

Comparando o LXC com uma máquina virtual, o LXC não necessita de uma camada de sistema operacional para cada aplicação tornando-se uma ferramenta que ocupa menos espaço em disco que uma máquina virtual convencional, demanda menos recursos e tem um nível de portabilidade superior. A figura 1 ilustra esta comparação (ALMEIDA, 2015).

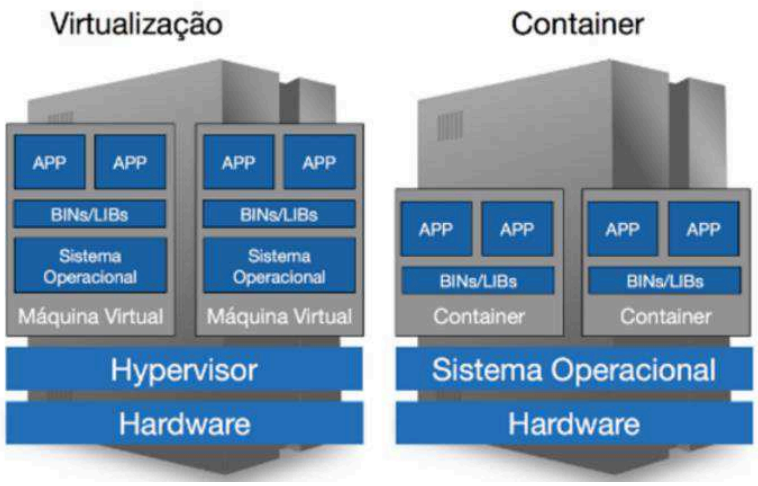

Figura 1. Comparação Container x Virtualização

Com a consolidação da virtualização e dos ambientes em nuvem, o LXC foi se difundindo cada vez mais nas empresas e no mercado de trabalho. É possível portar uma aplicação direto do notebook do desenvolvedor para o servidor de produção ou um ambiente de testes, assim como acessá-la. A partir do LXC foram criadas outras tecnologias que se adequam às demandas de cada empresa. Uma das tecnologias baseadas em LXC mais difundidas no mercado é o Docker (DOCKER, 2013). A figura 2 mostra uma breve comparação entre Docker e máquina virtual.

Figura 2. Comparação entre Docker (esquerda) e máquina virtual (direita)

3. Docker

O Docker é uma plataforma de código aberto lançada em 2013, escrita na linguagem de programação Go desenvolvida pela Google e possui um alto desempenho, que proporciona a criação e o gerenciamento de ambientes isolados. Foi concebida pela empresa DotCloud que inicialmente tinha o interesse de somente usá-lo para aplicações internas, mas que acabou sendo muito bem aceito no mercado e tomou um rumo diferente. Sua facilidade em gerenciar containers

acabou levando a empresa a desenvolver sua própria biblioteca, deixado as nativas do LXC de lado e assumindo o controle dos drivers diretamente com o kernel do host (ALMEIDA, 2015).

A portabilidade do Docker é uma de suas principais características e uma das principais diferenças para o LXC, como mostrado na figura 3, pois além de possibilitar o empacotamento de uma aplicação dentro de um container, é possível portá-lo para qualquer host que possua a plataforma Docker instalada sem a necessidade de ficar configurando o host ou o container a todo momento. Essa característica diminui muito o tempo de execução das aplicações ou até mesmo da implantação de infraestruturas, pois uma vez configurado o container, basta o host possuir o Docker instalado que o desenvolvedor só precisa replicar os containers e executá-los sem realizar nenhum tipo de configuração (DOCKER, 2017).

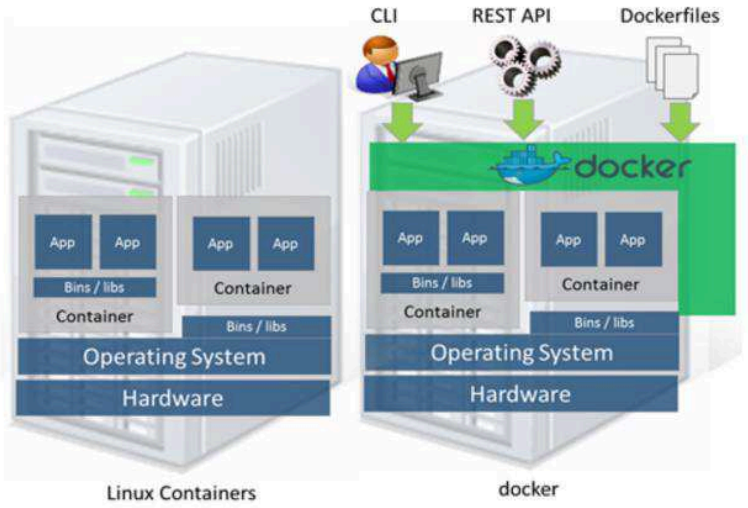

Figura 3. Comparação entre LXC e Docker

O funcionamento do Docker é baseado em chamadas de cliente e servidor entre o Docker Daemon e o Docker Client através de uma API. Basta ter o Docker instalado em algum lugar e direcionar ro seu Docker Client para esse servidor (DOCKER, 2017).

Uma das principais bibliotecas do Docker e a mais relevante para este projeto é a biblioteca libconteiner, que é responsável por fazer a comunicação entre o Docker Daemon e o backend da plataforma que utiliza o LXC. Esta biblioteca é responsável pela criação dos containers e ela permite a configuração dos limites de recursos de cada container isoladamente (DOCKER, 2017).

Assim como várias empresas do mercado atualmente, este projeto utiliza a plataforma Docker para realizar a criação de containers e gerenciá-las na simulação de um ambiente de testes através de uma plataforma que age como um sistema operacional de gerenciamento de data centers chamado DC/OS.

Este age como um sistema operacional que gerencia diversas máquinas de forma distribuída, possibilitando executar aplicações abstraindo a alocação de recursos. Seu desenvolvimento foi feito pela empresa americana Mesosphere, que desenvolve softwares para data centers utilizando o Apache Mesos, e lançado em 2016.

4. Apache Mesos

O Mesos é um software de código aberto desenvolvido por estudantes e um professor da Universidade da Califórnia em Berkeley, que tem como principal funcionalidade o gerenciamento de clusters. Foi apresentado em 2009 na publicação de um artigo (HINDMAN; KONWINSKI; ZAHARIA; STOICA, 2009) mas somente em 2016 a Apache Software Foundation lançou a primeira versão do software divulgando uma nota em seu blog oficial.

A arquitetura do Mesos é baseada em um processo mestre que executa em um nodo mestre. Existe um grupo de nodos mestres que elegem um líder entre eles para gerenciar os recursos dos clusters e facilitar a orquestração de tarefas. Além dos nodos mestres existem também os agentes, que contêm os recursos disponíveis de um nodo agente. E por fim os frameworks que são compostos de schedulers, que recebem os recursos do nodo líder, e de executores, que executam as tarefas do framework no nodo agente (APACHE, 2017).

O fluxo das rotinas que o Mesos executa se dá da seguinte maneira:

i. O agente envia ao nodo líder os recursos disponíveis do nodo agente.

ii. O nodo líder recebe os recursos e envia para o framework definido por um módulo de política de alocação.

iii. O scheduler do framework responde o nodo líder enviando as tarefas a serem executadas e a quantidade de recursos que elas irão consumir.

iv. O nodo líder envia ao agente a quantidade de recursos necessárias para a execução das tarefas.

v. O agente aloca os recursos necessários no nodo agente para que o executor do framework consiga realizar as tarefas necessárias.

Uma vantagem desse tipo de arquitetura é que se o nodo líder ficar inoperante, os outros nodos mestres candidatos fazem uma nova eleição para tomar o lugar de líder e continuar as operações fazendo com que os clusters nunca fiquem sem gerenciamento. Outra característica importante desta arquitetura é que os frameworks são independentes e têm a capacidade de rejeitar as ofertas de recursos do nodo líder que não seguirem as restrições impostas pelo próprio framework. Um dos frameworks mais utilizados pelo Mesos e pelo DC/OS é o Marathon, que realiza a orquestração de containers.

5. DC/OS

O Datacenter Operating Systems (DC/OS) é um sistema operacional para data centers. É composto de vários componentes e softwares de código aberto que o fazem um software completo em várias áreas diversificadas (MESOSPHERE, 2017b).

Funciona como um sistema distribuído que é executado em um cluster de nodos e não somente em uma máquina. Como em diversos sistemas distribuídos, possui nodos mestres que comandam nodos agentes, e há uma eleição entre os nodos mestres para definir um nodo líder, como no Mesos. Também possui a funcionalidade de um gerenciador de clusters, que gerencia tanto os recursos quanto as tarefas executadas dos nodos agentes, utilizando o Apache Mesos. Outra funcionalidade é a capacidade do DC/OS de gerenciar containers, utilizando o Marathon como scheduler, e o Docker e o Mesos como tecnologias de container. O DC/OS também permite a customização dos schedulers o que amplia as possibilidades dos desenvolvedores, e os permite adequar os mesmos de acordo com suas necessidades específicas. Como um sistema operacional, abstrai os recursos de software e hardware e provê alguns serviços comuns às aplicações como serviços de rede, gerenciamento de pacotes, armazenamento e segurança. A interface é amigável ao usuário e permite o gerenciamento dos clusters do DC/OS de maneira simples (MESOSPHERE, 2017b).

A arquitetura do DC/OS pode ter um hardware tanto físico quanto virtual, desde que forneça os serviços comuns como rede e armazenamento. Independentemente de ser físico ou virtual, a arquitetura do DS/OS apresenta as seguintes camadas:

i. Camada de Software: provê gerenciamento e repositório de pacotes para instalar e gerenciar diversos tipos de serviços tais como banco de dados, logs, ferramentas de integração contínuas. Além desses serviços, o usuário pode instalar serviços e aplicações personalizados.

ii. Camada da Plataforma: essa camada possui diversos componentes que estão separados nas categorias de gerenciamento de cluster, orquestração de containers, ambiente de containers, rede, gerenciamento de pacotes, segurança e armazenamento. Esses componentes também são distribuídos entre os tipos de nodos que são definidos como mestres, agentes públicos ou privados. Para instalar o DC/OS um dos nodos deve conter um dos sistemas operacionais suportados, CentOS, RHEL e CoreOS.

iii. Camada de Infraestrutura: o DC/OS pode ser instalada em hardware local e em nuvens públicas ou privadas desde que esses ambientes possuam rede IPv4 compartilhada e máquinas x86.

A figura 4 exemplifica a arquitetura apresentada.

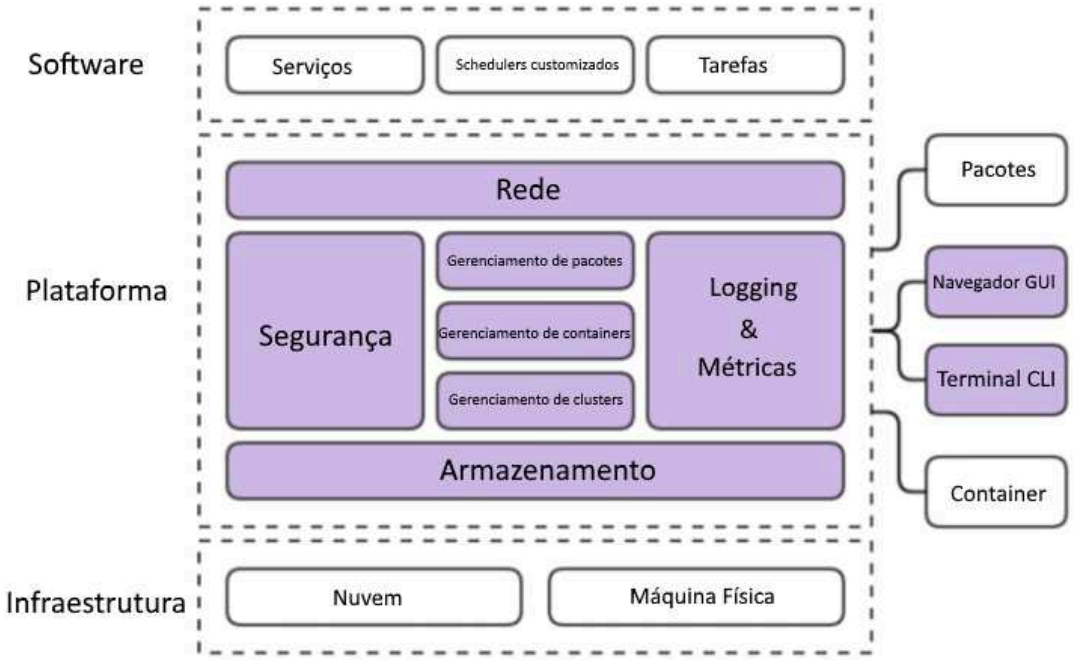

Figura 4. Camadas de arquitetura do DC/OS

6. Ferramenta Gerenciadora de Aplicações em Clusters DC/OS na Nuvem

Foi desenvolvido um software que permite o deploy de aplicações em um cluster DC/OS, através de uma imagem Docker, com o uso restrito à uma limitação pré-estabelecida. Um deploy, neste caso, é a execução de uma aplicação em um cluster DC/OS. Esta aplicação recebe estas requisições através de uma API, e se comunica com o Marathon do cluster DC/OS para executar ou interromper a execução das aplicações.

Para mapear corretamente as aplicações de cada usuário, o sistema deve permitir que um host seja especificado na chamada para a realização do deploy. Este host seria utilizado para mapear, dentro do cluster, para a aplicação correta do usuário.

Também deve ser possível requisitar o cancelamento de uma aplicação, ou seja, interromper a execução de uma aplicação, por parte do usuário, para liberar os recursos conforme o necessário.

A ferramenta deve se comunicar com um banco de dados Postgres que possui duas tabelas, uma que se refere aos usuários chamada User e uma referente às aplicações que serão executadas no cluster chamada App.

Para implementar a solução, foi desenvolvido uma aplicação web com uma API que permite este controle. A linguagem de programação escolhida foi o Ruby, utilizando o framework web Sinatra e o servidor Unicorn.

Foram definidas duas entidades principais para o modelo de dados:

i. User: o usuário cadastrado, contendo o limite de aplicações simultâneas permitido. O usuário contém também uma chave de acesso única chamada token de acesso, que permite a utilização da api e das funcionalidades da ferramenta. Com isso é possível controlar o acesso, e o limite das aplicações simultâneas garante um gerenciamento eficaz das aplicações que estão sendo executadas no cluster DC/OS.

ii. App: host e status da aplicação, relacionado com o usuário que a requisitou. O host, indica de onde o contêiner que está subindo deve receber requisições. O status pode ser:

a. Not running: A aplicação foi criada, mas ainda não está no ar.
b. Running: A aplicação se encontra no ar (através do Marathon no DC/OS).
c. Done: A aplicação já esteve no ar, mas neste momento se encontra desligada.
d. Failed: Ocorreu uma falha ao executar esta aplicação.

Para que os usuários possam ser gerenciados, foram criados alguns endpoints exclusivos para estas tarefas de administração. O controle acontece através do token de acesso compartilhado que identifica os administradores. Para realizar a requisição, deve-se identificar com o token de acesso através de um cabeçalho de autenticação, utilizando o método de autenticação básica HTTP. Os endpoints serão descritos abaixo:

1. GET /users:
    a. Retorna um JSON contendo todos os usuários, com os campos id, representando o identificador do usuário na tabela Users, name representando o nome do usuário, e max_deploys, que representa o número máximo de aplicações que o usuário pode executar simultaneamente no cluster DC/OS.
    b. Se for enviado um parâmetro no querystring de “access_token” com o valor “true”, o JSON de resposta retornará também o access_token de cada usuário, que é uma chave única representada por um hash que serve para autenticar o usuário.

Na figura 5 podemos perceber na querystring que o parâmetro acess_token está definido com o valor true, e por isso retornará o token de acesso dos usuários no JSON. No caso temos somente um usuário cadastrado.

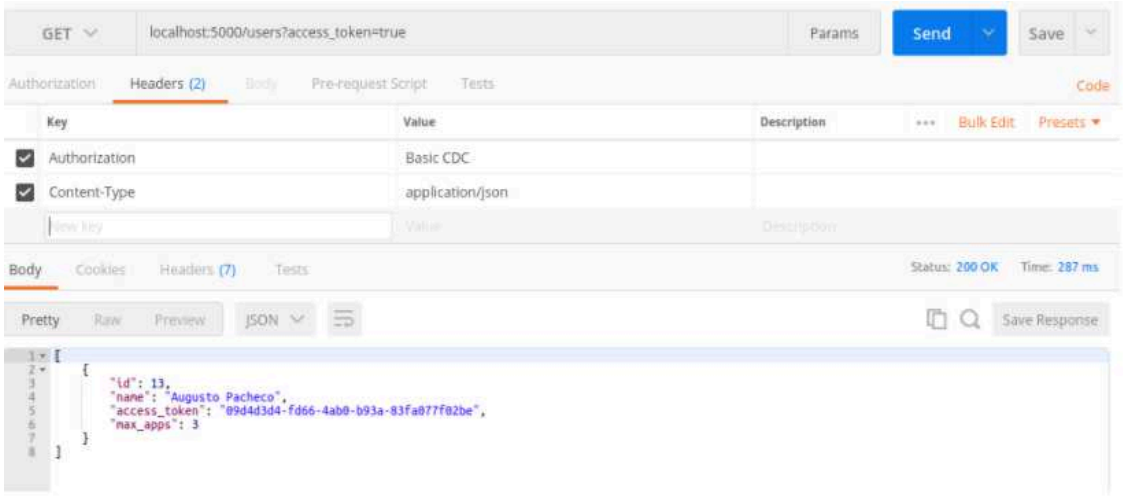

Figura 5. EndPoint GET/users

2. POST/users:
    a. Parâmetros (JSON):
        i. name: nome do usuário.
        ii. max_apps: quantidade máxima de aplicações que este usuário pode manter simultaneamente.
        iii. Ambos são obrigatórios. Retornará o código HTTP 422, caso ambos não estejam presentes.
    b. Retorna o usuário criado, contendo os campos id, name, max_apps e access_token. Ex.: {"id": 14,"name": "Felipe Duarte", "max_apps": 1, "access_token": "cd265ce6-5b85-4c1f-cd3bd6b120b7”}.

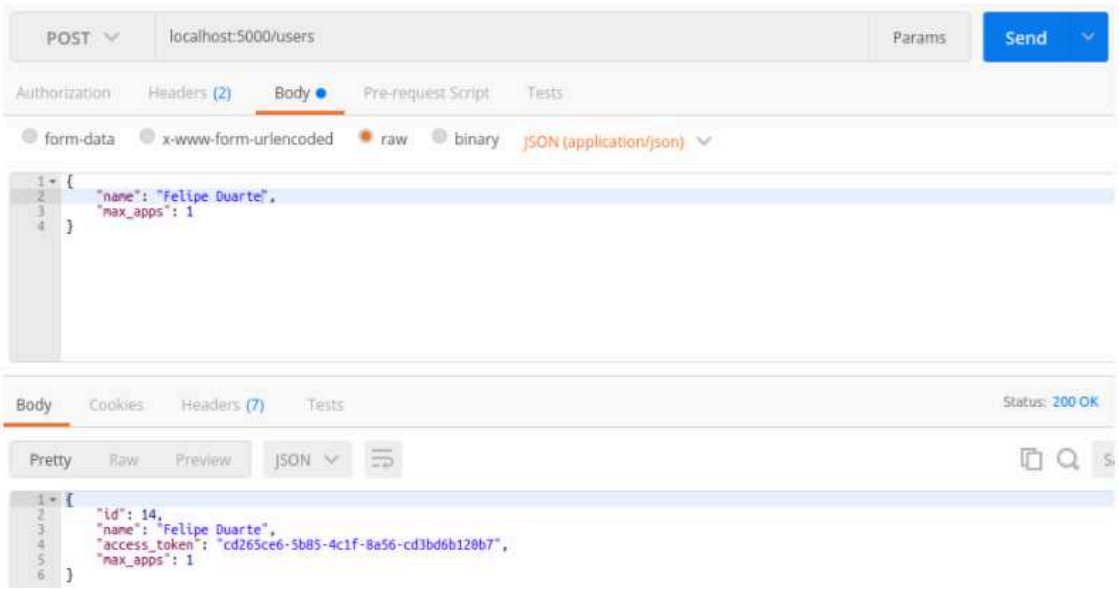

Figura 6. EndPoint POST/users

3. PUT /users/:id
    a. Parâmetros (JSON):
        i. Id (via o path): ID do usuário
        ii. name: nome do usuário
        iii. max_apps: quantidade máxima de aplicações que este usuário pode manter simultaneamente
    b. Retorna o usuário atualizado, contendo os campos id, name, max_apps e access_token. Ex.: {"id": 1,"name": "Augusto Pacheco", "max_apps": 10, "access_token": "cd265ce6-5b85-4c1f-8a56-cd3bd128b7"}.

Assim que um usuário é cadastrado, ele recebe um access_token. Este token deve ser informado através de um parâmetro access_token no payload da requisição, em todas as requisições realizadas pelo usuário, para autenticar e identificá-lo. Os endpoints para utilização do usuário são:

1. GET /apps/:id
    a. Parâmetros:
        i. Id (via o path): ID da aplicação.
    b. Retorna um JSON contendo o campo id e status da aplicação. Ex.: {“id”: 29, “status”: “running”}.
    c. Se não existir uma aplicação registrada com este ID retorna código HTTP 404.
2. GET /apps
    a. Retorna todas as aplicações para este usuário, com os campos id, status e host conforme mostra a figura 7:

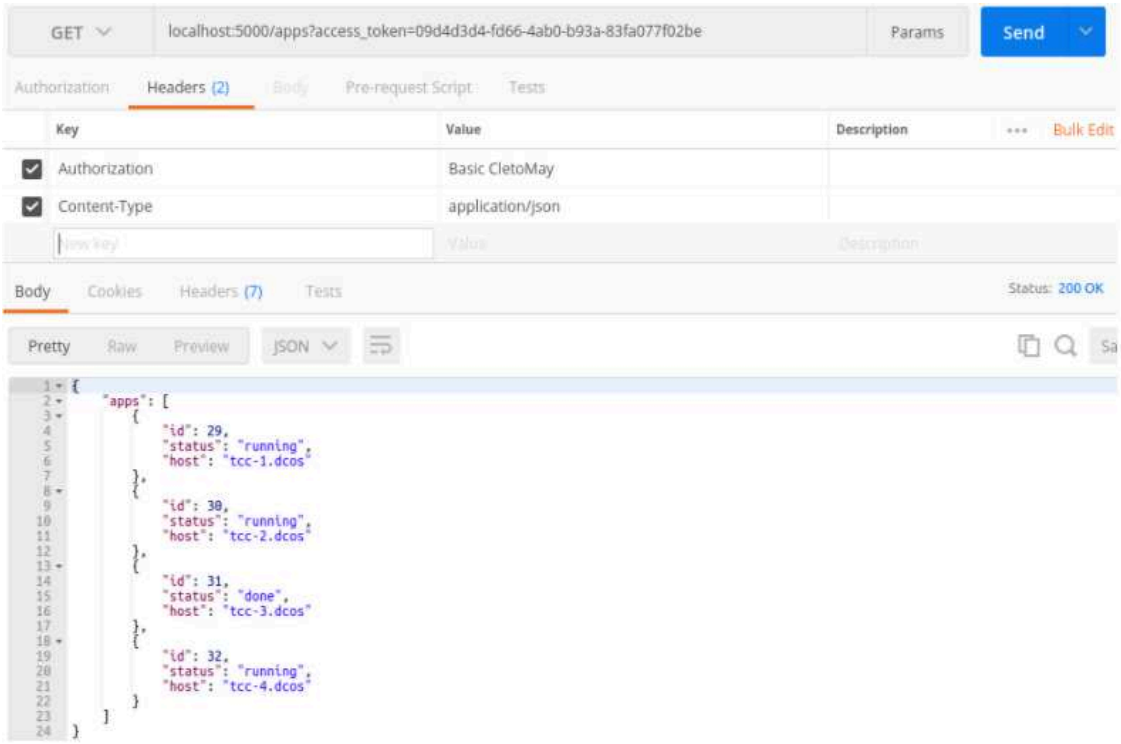

Figura 7. EndPoint GET/apps

3. POST /apps
    a. Parâmetros:
        i. acess_token
        ii. image: nome da imagem Docker que o DC/OS deverá executar no formato “repository/image_name:tag”.
        iii. host: o host que o Marathon usará para mapear a aplicação.
        iv. Ambos são obrigatórios. Retornará o código HTTP 422 caso ambos não estejam presentes.
    b. Caso o usuário já esteja no limite de aplicações, número máximo de aplicações executando no cluster, retornará código HTTP 400 e o deploy será rejeitado mostrando uma mensagem de erro conforme mostra a figura 8:

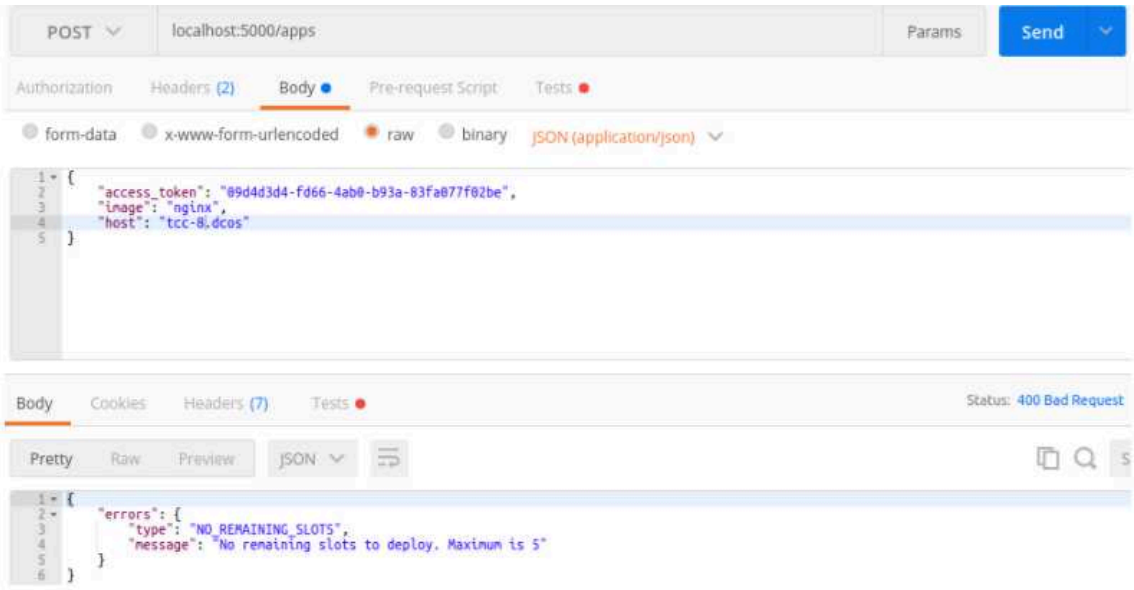

Figura 8. EndPoint POST/apps rejeitando deploy por exceder o limite

c. Realiza a execução da aplicação no Marathon, utilizando uma configuração mínima de CPU e memória. Retorna um JSON com a aplicação criada, e os campos id, user_id, status e host conforme mostra a figura 9:

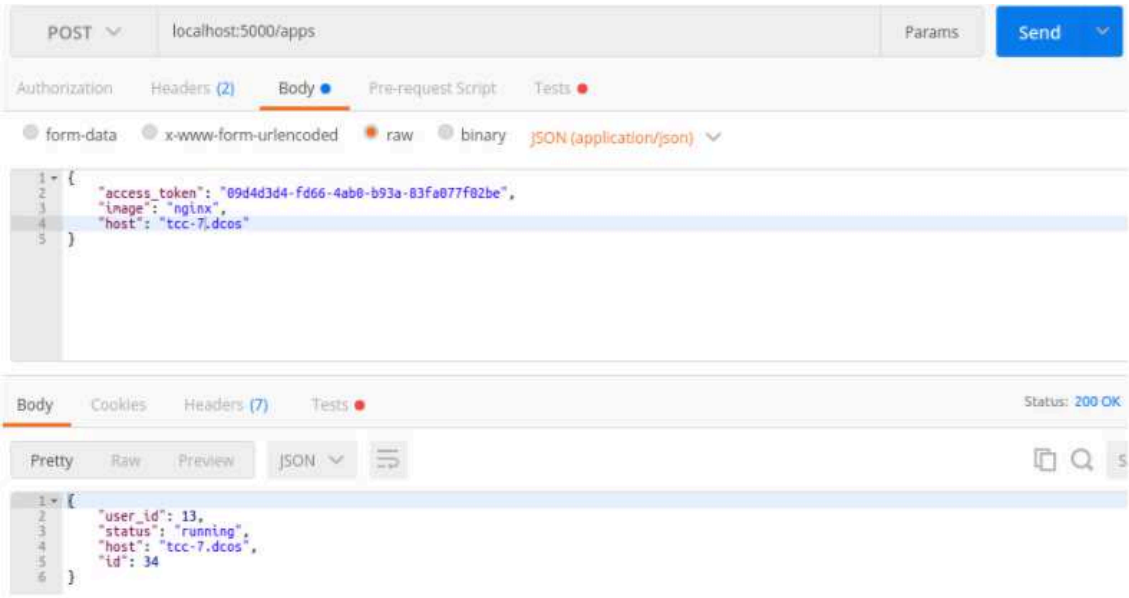

Figura 9. EndPoint POST/apps realizando um deploy com sucesso

4. DELETE /apps/:id
    a. Parâmetros:
        i. Id (via o path): ID da aplicação.

ii. acess_token

b. Derruba a aplicação com o id informado.
c. Retorna erro HTTP 404 se não existir uma aplicação com esse id.
d. Se não houver uma aplicação rodando com esse id retorna erro HTTP 400.

Para utilização da ferramenta, deve-se especificar algumas variáveis de ambiente para configurar o servidor, o banco de dados e o endereço do Marathon:

1. Web_concurrency: número de workers (concorrência) do servidor Unicorn. Ex.: 5.
2. Web_timeout: Valor, em milissegundos, de timeout para as requisições. Ex.: 30000.
3. Database_url: Connection string do banco de dados, a ser utilizado pela biblioteca Sequel. Ex.: postgres://user:pass@localhost:5432.
4. Api_token: Token necessário para autenticação HTTP, utilizado nos endpoints administrativos (gerenciamento dos usuários). Ex.: a99c8482-ca44-4928-91da-9e778d4c5dbf.
5. Marathon_url: URL do Marathon, para ser utilizado pela ferramenta para subir e remover as aplicações. Ex.: http://m1.dcos/marathon.

Em um cenário prático de utilização da ferramenta, foi desenvolvida a ferramenta dentro da arquitetura demonstrada na figura 10. O usuário se comunica com a ferramenta que, através de um banco de dados PostgreSQL, gerencia e controla as execuções de aplicações. A ferramenta, por sua vez, se comunica com o DC/OS através da API do Marathon. É nesta etapa que ocorre a execução da aplicação com Docker, conforme detalhado na figura 11:

Figura 10. Diagrama da ferramenta

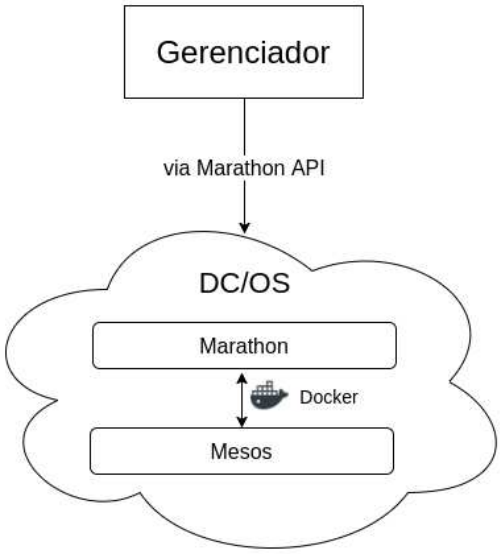

Figura 11. Detalhe da iteração entre a ferramenta e o DC/OS, através do Marathon

Na etapa de testes do processo de desenvolvimento de software de uma empresa, por exemplo, as equipes de testadores precisam homologar suas aplicações em um ambiente de teste. Para isso eles executam versões da aplicação na qual estão desenvolvendo novas funcionalidades.

O problema é que somente uma versão pode ser executada por vez pois não há um meio de gerenciar as versões de maneira automatizada, de modo a criar uma fila de testadores querendo homologar suas versões, atrasando o lançamento de novas funcionalidades da aplicação.

A ferramenta desenvolvida neste projeto possibilita que mais de uma versão de uma mesma aplicação possa ser lançada no ambiente de homologação. Ela gerencia todas as versões respeitando restrições de acesso e número máximo de aplicações que um usuário pode executar no cluster, de maneira a acabar com a fila de testadores e otimizando o processo de homologação acarretando no lançamento de mais funcionalidades da aplicação com um tempo de espera menor. A figura 12 exemplifica uma requisição, utilizando o programa Postman, que possibilita executar as requisições efetuadas para a ferramenta e retorna os resultados definidos nos endpoints:

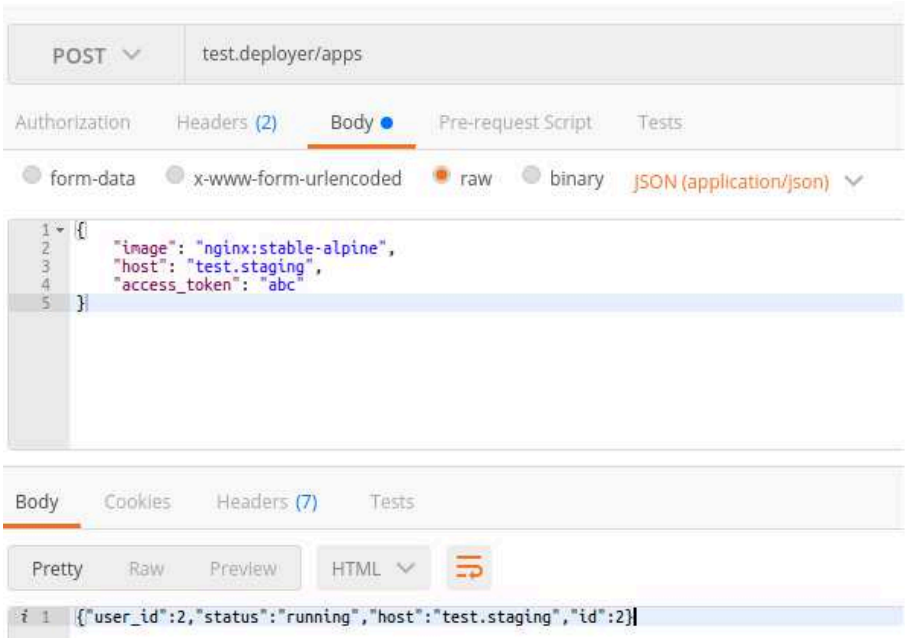

Figura 12. Postman executando o endpoint POST/apps

Isto deve executar a aplicação no cluster DC/OS configurado na ferramenta, associando com o usuário do respectivo access_token informado. O mapeamento para o host também será efetuado no Marathon, garantindo que a aplicação solicitada esteja acessível pelo testador.

Supondo que este usuário tenha um limite máximo de 1 aplicação, ao tentar novamente ele recebe como resposta um erro, com a mensagem de erro segundo a figura 13:

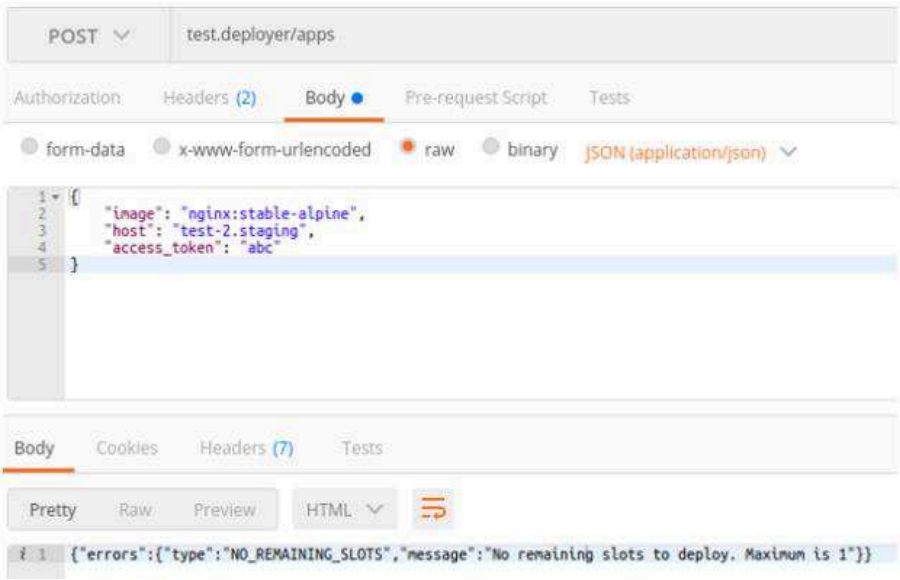

Figura 13. Erro de um usuário tentando executar mais aplicações que seu limite permitido

Mesmo assim, caso um outro usuário, ou seja, com outro access_token e com limite suficiente, tente executar a aplicação da mesma forma, ele conseguirá devido ao limite individual de aplicações. Conforme demonstrado na figura 14:

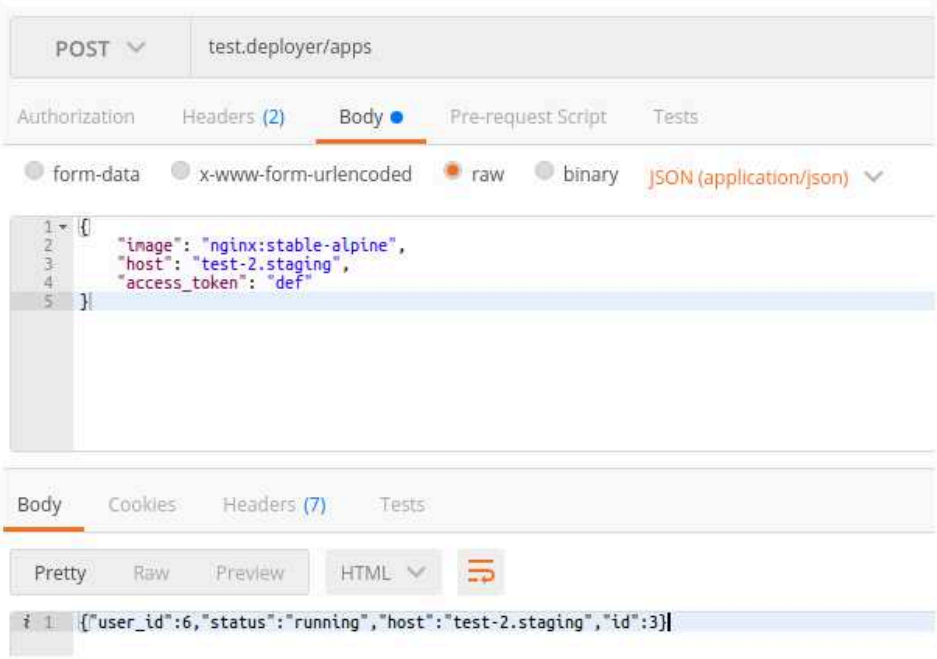

Figura 14. Usuário diferente consegue executar aplicações no cluster pois cada um possui limites independentes

Desta forma, a ferramenta consegue gerenciar com sucesso a fila de testadores, impondo regras de acesso e limites de aplicações definidas na hora do cadastro do usuário. Com isto, além de acelerar o processo de desenvolvimento, também permite a economia de recursos, e custos operacionais, ao limitar as execuções por usuário, fazendo uma distribuição justa conforme o critério desejado.

7. Conclusão e Trabalhos Futuros

Esta ferramenta tem o objetivo de gerenciar várias versões de uma mesma aplicação em um ambiente de teste, respeitando níveis de acesso e o número máximo de aplicações que cada usuário pode executar em um cluster, para que a fila de testadores aguardando a vez para testarem suas versões acabe, e o tempo de lançamento de novas funcionalidades da aplicação seja menor.

A implementação dessa ferramenta requereu um estudo aprofundado de diversas tecnologias muito atuais, como a tecnologia de container representada pelo Docker, plataformas de orquestração de containers como o Marathon, gerenciadores de clusters como o Mesos e o

DC/OS que é um sistema operacional muito útil e consegue unificar as tecnologias citadas acima em um único ambiente.

A finalização deste projeto representa uma alternativa para equipes de desenvolvedores que possuem uma fila de espera para homologar suas versões. Um embrião que utiliza de tecnologias atuais e muito utilizadas no mercado, que permite gerenciar diferentes versões de uma mesma aplicação em um ambiente de testes de maneira simples. Com isso, a empresa pode acelerar os lançamentos de novas funcionalidades e correções, acelerando o processo de desenvolvimento.

Para trabalhos futuros, poderia ser realizada a criação de uma interface gráfica e amigável para o usuário, suporte a manipulação de banco de dados, escalar clusters DC/OS inteiros ao invés de aplicações, escalar clusters com banco de dados individuais para cada cluster, possibilitar a inclusão de uma funcionalidade que permita o gerenciador escolher o serviço de nuvem que ele quer lançar as aplicações de maneira que ele possa gerenciá-los de forma automatizada alocando recursos de um serviço para outro simplesmente usando a API da ferramenta.

8. Referências

ALMEIDA, J.M.B. Container, um novo passo para a virtualização. (2015) IBM Academy of Technology Affiliated. Disponível em: < https://www.ibm.com/developerworks/community/blogs/tlcbr/entry/mp234?lang=en>.

APACHE SOFTWARE FOUNDATION (Org). Mesos. Disponível em:< http://mesos.apache.org/>. Acessado em 15 mai. 2017.

BOURQUE, PIERRE; FAIRLEY, DICK. (2014) SWEBOK 3.0 Guide to the Software Engineering Body of Knowledge. [S.l.]: IEEE Computer.

CENTOS (Org). Disponível em: < https://www.centos.org >. Acessado em 16 mai.2017

COHEN, D., LINDVAL, M., COSTA, P. (2004) An introduction to agile methods. In Advances in Computers. New York: Elsevier Science. p. 1-66.

COREOS (Inc). Disponível em <https://coreos.com>. Acessado em 16 mai.2017.

DOCKER (Inc). eBook: Docker for the Virtualization Admin. (2016)

DOCKER (Inc). Container. Disponível em: <https://www.docker.com/what-container>. Acessado em: 12 mai. 2017.

DOCKER (Inc). Docker. Disponível em: < https://www.docker.com/>. Acessado em 13 mai. 2017.

DU, W. Chroot Sandbox Vulnerability Lab. Syracuse University. (2009). Disponível em: <http://www.cis.syr.edu/~wedu/seed/Labs/Vulnerability/Chroot/Chroot.pdf>

HINDMAN, B., KONWINSKI, A., ZAHARIA, M., STOICA, I. A Common Substrate for Cluster Computing. (2009) University of Californa, Berkeley.

ISO (Org): ISO/IEC 25010 Disponível em: <https://www.iso.org/standard/35733.html>. 2011. Acessado em 15 mai. 2017.

KARLSTROM, D., RUNESON P. (2005) Combining agile methods with stage-gate project management. IEEE Software. v.22.

LINUX (Org). Disponível em < https://www.linuxfoundation.org/>. Acessado em 13 mai. 2017.

LINUX CONTAINER (Org). (2008). Disponível em: < https://linuxcontainers.org/>. Acessado em 12 mai. 2017.

MANIFESTO ÁGIL (Org). (2001). Disponível em: < http://agilemanifesto.org/>. Acessado em 13 mai. 2017.

MESOSPHERE (Inc). Marathon. Disponível em: < https://mesosphere.github.io/marathon/>. Acessado em 15 mai. 2017

MESOSPHERE (Inc). DC/OS. Disponível em: <https://dcos.io/>. Acessado em: 12 mai. 2017a.

MESOSPHERE (Inc). DC/OS. Disponível em: <https://dcos.io/docs/1.9/overview/concepts/#dcos>. Acessado em: 12 mai. 2017b.

MESOSPHERE (Inc). DC/OS. Disponível em: <https://dcos.io/docs/1.9/overview/what-is-dcos/>. Acessado em: 12 mai. 2017c.

MESOSPHERE (Inc). Disponível em: <https://mesosphere.com/>. Acessado em: 12 mai. 2017.

MESOSPHERE (Inc). DC/OS. Disponível em: < https://dcos.io/docs/1.9/overview/architecture >. Acessado em: 12 mai. 2017d

MYERS, GLENFORD J. The Art of Software Testing. Ed. John Wiley & Sons, Nova Jérsei. 2004

NAIK, KSHIRASAGAR; TRIPATHY, PRIVADARSHI. (2008). Software Testing and Quality Assurance. Hoboken, New Jersey: Wiley.

PRESSMAN, ROGER. (1995) Engenharia de Software. Ed. Makron Books.

RED HAT (Inc). Disponível em: <https://www.redhat.com>. Acessado em 20 mai.2017.

RUBY (Org). Disponível em: < https://www.ruby-lang.org/pt/>. Acessado em: 20 mai. 2017.

THE APACHE SOFTWARE FOUNDATION BLOG (Org). Disponível em < https://blogs.apache.org/foundation/entry/the_apache_software_foundation_announces97 >. 2016

```
$:.unshift(File.expand_path File.join(File.dirname(__FILE__),'helpers'))

$:.unshift(File.expand_path File.join(File.dirname(__FILE__),'models'))

$:.unshift(File.expand_path File.join(File.dirname(__FILE__),'initializers'))

$:.unshift(File.expand_path File.join(File.dirname(__FILE__),'lib'))



require 'sinatra'

require 'database'

require 'rack'

require 'authentication'

require 'deployer'

require 'access_control'

require 'marathon_client'

require 'app'

require 'user'



class TestManager < Sinatra::Application

  include Sinatra::Authentication
```

```
helpers do
  def access_token?
    access_token == 'true'
  end

  def access_token
    params['access_token'] || payload['access_token']
  end

  def missing_params(param)
    return 422, generate_error("MISSING_PARAM", "Missing param: #{param}")
  end

  def payload
    @payload ||= JSON.parse(request.body.read)
  end
```

```
  def validate_param(param, name)
    return param unless param.nil?

    missing_params(name)
  end

  def resource_not_found(resource)
    halt 404, generate_error("RESOURCE_NOT_FOUND", "#{resource} not found")
  end

  def no_remaining_slots
    return 400, generate_error("NO_REMAINING_SLOTS", "No remaining slots to deploy. Maximum is #{current_user.max_apps}")
  end

  def host_in_use(host)
```

```
    return 400, generate_error("HOST_IN_USE", "There is already an app running with provided
host: #{host}")
  end

  def app_not_running(app_id)
    return 400, generate_error("APP_NOT_RUNNING", "App for deploy #{app_id} is not
running")
  end

  def generate_error(type, msg)
    { errors: { type: type, message: msg } }.to_json
  end
end

get '/ping' do
  return 200, { ping: 'pong' }.to_json
end
```

```
get '/users' do
  authenticate!

  users = User.all_to_hash(access_token?)

  "#{users.to_json}"
end

post '/users' do
  authenticate!

  user_name = validate_param(payload['name'], 'name')
  max_apps = validate_param(payload['max_apps'], 'max_apps')

  user = AccessControl.create_user(user_name, max_apps)
```

```
  "#{user.values.to_json}"
end

put '/users/:id' do
  authenticate!

  user_id = validate_param(params[:id].to_i, 'id')
  user_name = payload['name']
  max_apps = payload['max_apps']

  user = AccessControl.update_user(user_id, user_name, max_apps)

  "#{user.values.to_json}"
end

get '/apps/:id' do
  authenticate_access_token!
```

```
  status = Deployer.status(params[:id].to_i, current_user.id)

  return resource_not_found('app') if status.nil?

  { id: params[:id], status: status }.to_json
end

get '/apps' do
  authenticate_access_token!

  app_status = Deployer.all_status(current_user.id)

  { apps: app_status }.to_json
end

post '/apps' do
```

```
authenticate_access_token!


image = payload["image"]

host = payload["host"]


return missing_params('image') if image.nil?

return missing_params('host') if host.nil?


begin

  app = Deployer.deploy!(image, host, current_user.id)


  app.values.tap { |v| v[:status] = app.real_status }.to_json

rescue Deployer::NoSlotError

  no_remaining_slots

rescue Deployer::HostInUseError

  host_in_use(host)

end
```

```
end

delete '/apps/:id' do
  authenticate_access_token!

  app_id = params[:id]

  begin
    Deployer.stop!(app_id, current_user.id)
  rescue Deployer::NotRunningError
    return app_not_running(app_id)
  rescue Deployer::AppNotFoundError
    return resource_not_found("app")
  end

  return 200, { message: "success" }.to_json
end
```

```
end

require 'dotenv/load'

require 'sequel'


worker_processes Integer(ENV['WEB_CONCURRENCY'].to_i || 5)

timeout Integer(ENV['WEB_TIMEOUT'].to_i || 30000)

preload_app true


before_fork do |server, worker|

  Signal.trap 'TERM' do

    Process.kill 'QUIT', Process.pid

  end


  if defined?(Sequel::Model)

    DB.disconnect

  end

end
```

```
module Sinatra
  module Authentication
    attr_reader :current_user

    def authenticate!
      deny unless api_token_authorized?
    end

    def authenticate_access_token!
      user = User.where(access_token: access_token).first

      return deny if user.nil?

      @current_user = user
    end

    private
```

```
def deny
  headers['WWW-Authenticate'] = 'Basic realm="Restricted Area"'
  halt 401, "Not authorized\n"
end

def api_token_authorized?
  @auth ||=  Rack::Auth::Basic::Request.new(request.env)
  @auth.provided? && @auth.basic? && @auth.credentials && @auth.params == api_token
end

def api_token
  ENV['API_TOKEN']
end

def access_token
  params['access_token'] || payload['access_token']
```

```
    end
  end
end
class AccessControl
  def self.create_user(user_name, max_apps)
    user = User.new

    user.name = user_name
    user.max_apps = max_apps
    user.access_token = SecureRandom.uuid

    user.save
  end

  def self.update_user(user_id, user_name, max_apps)
    user = User.where(id: user_id).first
```

```
    user.name = user_name unless user_name.nil?
    user.max_apps = max_apps unless max_apps.nil?

    user.save
  end
end
class Deployer
  def self.deploy!(image, host, user_id)
    raise NoSlotError unless has_slot?(user_id)
    raise HostInUseError unless host_available?(host)

    app = create_app(host, user_id)
    ship(image, host)

    app
  end
```

```
def self.stop!(app_id, user_id)
  app = App.where(user_id: user_id, id: app_id).first

  raise AppNotFoundError if app.nil?
  raise NotRunningError unless app.status == App::RUNNING

  app.status = App::DONE
  app.save

  MarathonClient.stop_app_by_host(app.host)
end

def self.status(app_id, user_id)
  app = App.where(id: app_id, user_id: user_id).select(:status)

  return nil if app.empty?
```

```
  app.first.real_status
end

def self.all_status(user_id)
  App.where(user_id: user_id).select(:id, :status, :host).map do |app|
    { id: app.id, status: app.real_status, host: app.host }
  end
end

private

def self.has_slot?(user_id)
  current_apps = App.where(user_id: user_id, status: App::RUNNING).count
  max_apps = User.where(id: user_id).select(:max_apps).first.values[:max_apps]

  current_apps < max_apps
end
```

```
def self.host_available?(host)
  App.where(status: App::RUNNING, host: host).empty?
end

def self.create_app(host, user_id)
  app = App.new
  app.user_id = user_id
  app.host = host
  app.status = App::RUNNING
  app.save

  app
end

def self.ship(image, host)
  MarathonClient.new(image, host).deploy
```

```
end


class NoSlotError < StandardError
  def initialize(msg="No remaining slots")
    super(msg)
  end
end


class HostInUseError < StandardError
  def initialize(msg="App with this host already running")
    super(msg)
  end
end


class NotRunningError < StandardError
  def initialize(msg="App is not running")
    super(msg)
```

```
    end
  end


  class AppNotFoundError < StandardError
    def initialize(msg="App does not exist")
      super(msg)
    end
  end
end
require 'marathon'


class MarathonClient
  attr_reader :image, :host


  def initialize(image, host)
    @image = image
    @host = host
```

```
  # Marathon.url = MarathonClient.marathon_url
end

def deploy
  # Marathon::App.new(app_config).start!(true)
end

def self.stop_app_by_host(host)
  # Marathon.url = MarathonClient.marathon_url

  # Marathon::App.delete("web-#{host.downcase}")
end

private

def self.marathon_url
```

```
  ENV["MARATHON_URL"]
end


def app_config
  {
    "id" => "/web-#{host.downcase}",
    "cmd" => nil,
    "cpus" => 0.1,
    "mem" => 64,
    "disk" => 0,
    "instances" => 1,
    "acceptedResourceRoles" => [
      "slave_public",
      "*"
    ],
    "container" => {
      "type" => "DOCKER",
```

```
"volumes" => [],
"docker" => {
  "image" => image,
  "network" => "BRIDGE",
  "portMappings" => [
    {
      "containerPort" => 80,
      "hostPort" => 0,
      "servicePort" => 10101,
      "protocol" => "tcp",
      "labels" => {}
    }
  ],
  "privileged" => false,
  "parameters" => [],
  "forcePullImage" => true
}
```

```
},
"env" => {
},
"healthChecks" => [
  {
    "path" => "/",
    "protocol" => "HTTP",
    "portIndex" => 0,
    "gracePeriodSeconds" => 300,
    "intervalSeconds" => 60,
    "timeoutSeconds" => 10,
    "maxConsecutiveFailures" => 3,
    "ignoreHttp1xx" => false
  }
],
"labels" => {
  "HAPROXY_GROUP" => "external,internal",
```

```
        "HAPROXY_0_VHOST" => host
      }
    }
  end
end
Sequel.migration do
  change do
    create_table(:users) do
      primary_key :id
      String :name, null: false
      Integer :max_deploys, null: false
    end
  end
end
Sequel.migration do
  change do
    create_table(:apps) do
```

```
      primary_key :id
      foreign_key :user_id, :users, null: false
      Integer :status, null: false
      String :host, null: false
      index [:user_id, :status]
    end
  end
end
Sequel.migration do
  change do
    add_column :users, :access_token, String, null: false
  end
end
require 'sequel'
require 'pg'

DB = Sequel.connect(ENV['DATABASE_URL'])
```

```
class App < Sequel::Model

  NOT_RUNNING = 0

  RUNNING = 1

  DONE = 2

  FAILED = 3



  def real_status

    App.status_described[self.status.to_s]

  end



  private



  def self.status_described

    {

      "#{NOT_RUNNING}" => 'not running',

      "#{RUNNING}" => 'running',

      "#{DONE}" => 'done',
```

```
      "#{FAILED}" => 'failed',
    }.freeze
  end
end
class User < Sequel::Model
  def self.all_to_hash(with_access_token)
    users = User.all.map(&:values)

    return users if with_access_token

    remove_access_token(users)
  end

  private

  def self.remove_access_token(users)
    users.map do |user|
```

```
      user.tap { |u| u.delete(:access_token) }
    end
  end
end
source 'https://rubygems.org'


ruby '2.2.6'


gem 'pry'
gem 'sinatra'
gem 'sequel'
gem 'rack'
gem 'unicorn'
gem 'pg'
gem 'marathon-api', :require => 'marathon'


gem 'dotenv', groups: [:development, :test]
```

```
require './test_manager'


run TestManager
```