



Instituto Superior de Engenharia de Lisboa

Licenciatura em Engenharia Informática e de Computadores

Projeto e Seminário

Plataforma de Integração Contínua

Autores

Pavel Egorov

Iurie Marcinschi

Orientador

Doutor Porfírio Pena Filipe

Setembro 2014

Resumo

No contexto do desenvolvimento de *software* a tarefa de cada membro da equipa traduz-se diariamente na realização de integrações de código, onde cada integração é verificada por meio de testes unitários. Assim sendo, a integração contínua é uma prática comum no dia-a-dia de um programador.

O presente projeto sugere uma interface *Web* para facilitar a prática de integração contínua através da importação de código para ambientes *Linux* que suportam a compilação e execução de testes unitários.

A realização de um conjunto de testes consiste em compilar e executar o código de forma automática, baseada nos testes unitários, produzindo um ficheiro com o resultado da execução.

Agradecimentos

Queremos agradecer a todos que de alguma forma contribuíram para que este projeto fosse possível, e um particular agradecimento ao Professor Doutor Porfírio Pena Filipe por ter aceitado orientar o projeto.

Índice Figura

Figura 1 - Modelo de integração contínua comum.....	2
Figura 2 - Modelo solução.....	3
Figura 3 - Linux Container	6
Figura 4 - Camada Docker	7
Figura 5 – NodeJs com um único fio de execução	9
Figura 6 - Paralelismo NodeJs com N Cluster.....	10
Figura 7 - Fluxo abstrato do protocolo OAuth2.0	11
Figura 8 – Ambiente de desenvolvimento	12
Figura 9 - Vagrant up.....	14
Figura 10 - Vagrant destroy.....	14
Figura 11 - Vagrant ssh.....	15
Figura 12 - Vagrant halt.....	15
Figura 13 - Aplicações desenvolvidas.....	17
Figura 14 – Diagrama do fluxo do trabalho entre componentes.....	18
Figura 15 - Usabilidade Web	19
Figura 16 - Fluxo Autorização e Registo	19
Figura 17 - Pagina Registo	20
Figura 18 - Autenticação GitHub	20
Figura 19 – Arquitetura geral	21
Figura 20 - Modelo de esquemas.....	22
Figura 21 - Exemplo de ficheiro de migração.....	23
Figura 22 - Arquitetura do Servidor Web.....	24
Figura 23 - Exemplo visualização 320x480.....	24
Figura 24 - Arquitetura geral Worker.....	25
Figura 25 - Processos Worker	25
Figura 26 – Exemplo de representação do trabalho na fila Redis	26
Figura 27 - Formato Log stream	26
Figura 28 – Formato do estado de execução.....	27
Figura 29 - Arquitetura geral Hub	27

Sumário

Resumo	i
Agradecimentos	ii
Índice Figura	iii
1 Introdução	1
1.1 Motivação	1
1.2 Objetivo	3
1.3 Organização do documento	4
2 Estado da Arte	5
2.1 Emulador	5
2.2 Gestor da máquina virtual	5
2.3 Gestor de dependências	6
2.3 Ambiente virtual	6
2.4 Armazenamento de dados	8
2.5 Plataforma	8
2.6 Protocolo de autenticação	10
2.8 Framework Web	12
3. Ambiente de desenvolvimento	12
4 Projeto	17
4.1 Solução	17
3.3 Arquitetura	21
3.3.1 Servidor Web	24
3.3.2 Aplicação Worker	25
3.3.3 Aplicação Hub	27
5 Conclusão	28
5.1 Limitações	28
5.2 Trabalho futuro	28
6 Referencias	30

1 Introdução

O presente capítulo contextualiza o projeto apresentando a motivação e os objetivos.

Por motivos de internacionalização do conteúdo deste relatório, a nomenclatura utilizada está em conformidade com a terminologia inglesa.

1.1 Motivação

Normalmente o desenvolvimento de *software* é baseado na implementação de módulos que são integrados num produto. Na fase de integração, as alterações no código, realizadas por vários programadores, são combinadas numa base de código comum o que na maioria das vezes origina conflitos e erros.

A prática mostra que integrar e testar com mais frequência diminui a relevância dos conflitos e erros na base de código comum. Na década de 90 a compilação diária do código tornou-se uma prática frequente. No início dos anos 2000 esta prática foi levada ao extremo empregando a integração contínua e a validação das integrações com testes unitários.

No dia de hoje, o fluxo mais comum de trabalho no processo de integração contínua consiste em equipas de programadores desenvolverem aplicações de um produto, testarem o código localmente e submeterem as alterações no repositório de código do produto, tal como exemplifica a ilustração abaixo.

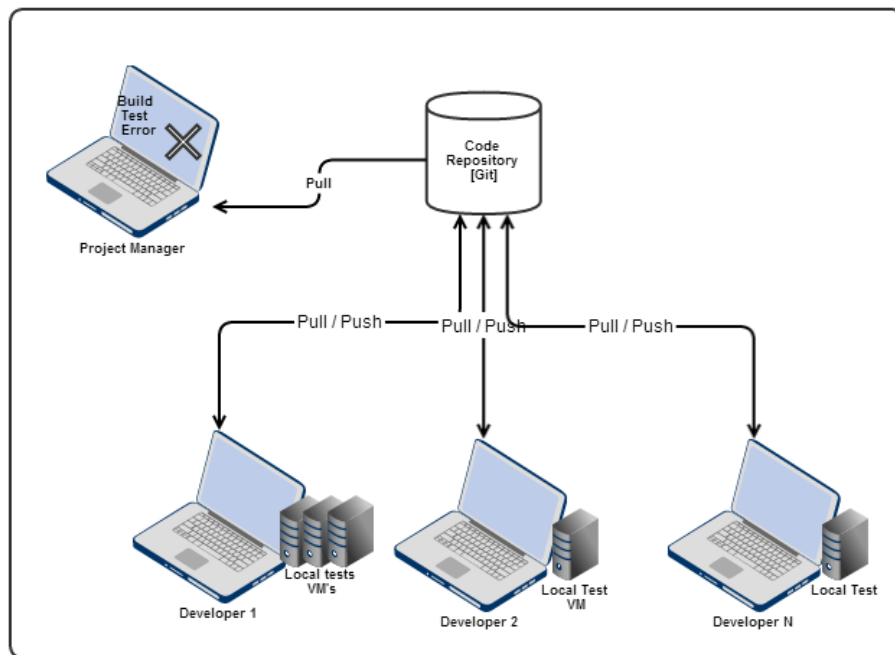


Figura 1 - Modelo de integração contínua comum

Cenário possível de erro:

1. Os programadores A e B descarregaram uma cópia do código.
2. O programador A cria uma classe C2 que faz uso da classe C1.
3. O programador B altera o código da classe C1 adicionando uma dependência.
4. Os dois programadores testam o código localmente nas suas máquinas com as cópias de base de código comum inicialmente descarregadas.
5. Depois de os testes localmente efetuados terem sido bem-sucedidos, os programadores submetem as alterações para o repositório de código comum.

Neste cenário exemplifica-se como as alterações realizadas na classe C1 não são consideradas pelo programador A até serem testadas num determinado momento detetando o erro. Adicionalmente, pode ser necessário testar o mesmo código em ambientes com propriedades diferentes, o que se traduz em gasto de tempo nas configurações dos ambientes ou gastos financeiros adicionais para preparação/aquisição dos mesmos.

1.2 Objetivo

O objetivo consiste em disponibilizar um serviço na internet de alta disponibilidade, que uma vez configurado para um determinado projeto, sabe automaticamente detetar alterações submetidas no repositório do código do produto, efetuar uma cópia do mesmo juntamente com os testes a realizar para um ambiente isolado de execução pré-configurado, executar a construção do código e os testes dentro do ambiente isolado, reportar de seguida os resultados e os *LOGs* da execução.

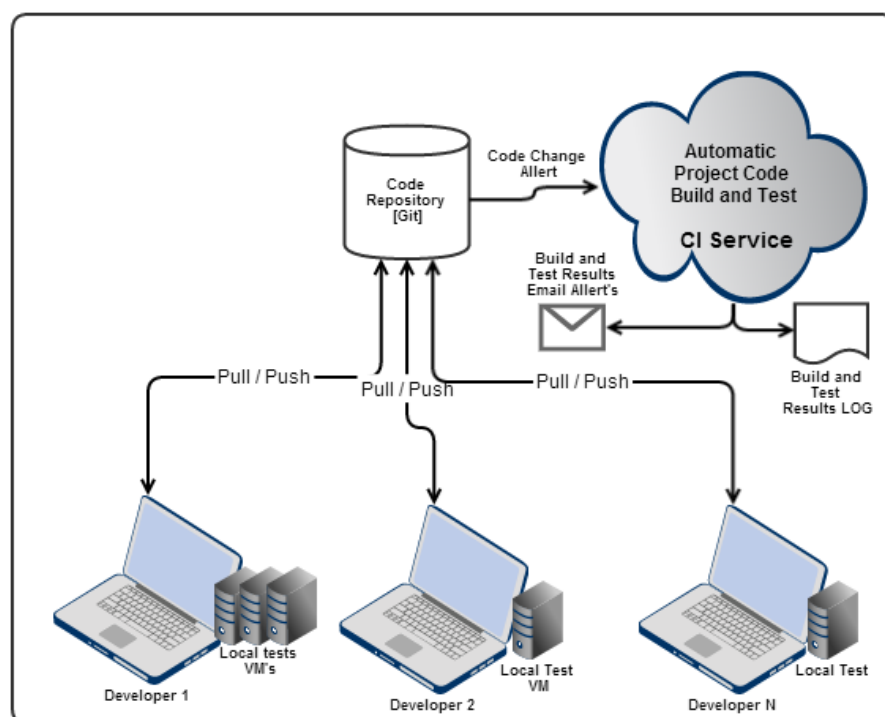


Figura 2 - Modelo solução

Assim sendo, o utilizador deve ter possibilidade de criar um ou mais projetos na plataforma, especificando as definições do ambiente necessário para execução, respeitando o conjunto de testes unitários, na sequência é gerado um ficheiro com os resultados produzidos. O contexto de realização dos testes pode ser, por exemplo, uma máquina Linux com suporte para PHP. [1]

1.3 Organização do documento

Este documento encontra-se organizado nos capítulos:

1. Capítulo 1 – Introdução: capítulo atual. Contextualiza e apresenta a motivação e os objetivos a alcançar.
2. Capítulo 2 - Estado da arte: descreve as tecnologias escolhidas para realização do projeto e o propósito das mesmas.
3. Capítulo 3 – Ambiente de desenvolvimento: descreve o próprio ambiente em que o projeto foi desenvolvido e as técnicas de provisionamento.
4. Capítulo 4 - Projeto: apresenta a solução proposta e arquitetura geral do sistema, descrevendo os seus componentes e a interação entre os mesmos.
5. Capítulo 5 – Conclusão: apresenta a análise crítica sobre o projeto e as limitações do mesmo assim como possíveis desenvolvimentos futuros.

2 Estado da Arte

Este capítulo descreve as tecnologias escolhidas para realização do projeto e o propósito das mesmas.

2.1 Emulador

VirtualBox

É um *software* de virtualização que cria ambientes para instalação e utilização de um ou mais sistemas operativos dentro do sistema operativo da máquina física, compartilhando desta maneira o mesmo hardware.[2]

2.2 Gestor da máquina virtual

Vagrant

Para garantir que os ambientes de desenvolvimento sejam idênticos nos membros da equipa, utilizou-se a ferramenta *Vagrant* [3] que é basicamente um gestor para máquinas virtuais. No ficheiro de configuração *Vagrantfile* descreve-se o tipo de máquina a utilizar (exemplo *Ubuntu-amd64 bits*), as aplicações a instalar e a forma de acesso ao ambiente. Desta forma garante-se que o aprovisionamento das ferramentas e dependências seja automático e equivalente em todas as estações de trabalho onde está a ser desenvolvido o projeto.

O aprovisionamento (*Provisioning*) neste contexto significa instalar e configurar as aplicações necessárias para desenvolvimento dentro da máquina virtual, para que esta esteja pronta para o lançamento e trabalho. De outra forma dizendo, em vez de instalar e configurar manualmente as aplicações como *NodeJs* [9], *Docker* [10][20], *MySQL* [11], *Redis* [12] dentro da máquina virtual, optou-se em utilizar a ferramenta *Chef* [4] que o *Vagrant* [3] suporta no seu processo de aprovisionamento.

Chef

É um dos sistemas mais populares de gestão de configurações em máquinas Linux. É usado para simplificar a tarefa de configuração e manutenção de servidores, e pode se integrar com plataformas baseadas em nuvem, como *Amazon EC2* [5], *Google Cloud* [6], *Microsoft Azure* [7] entre outras, para provisionar automaticamente e configurar novas máquinas.

2.3 Gestor de dependências

Berkshelf

Berkshel é um gestor de dependências para o Chefe, aprovisiona o Chefe com livros de receitas focados para um determinado componente, reutilizável e configurável. *Berkshelf* [8] encara os livros de receitas como bibliotecas de aplicações.

2.3 Ambiente virtual

No âmbito de virtualização o sistema operativo Linux oferece diversas soluções de virtualização, como *Xen* [13] ou *KVM* [14], de ambientes completos com CPU, Disco, Placa de rede, Adaptador de gráficos para uso privado ou privilegiado.

LXC - Linux Containers

O LXC [15] oferecido pelo sistema operativo Linux para "virtualização" leve e rápida de ambientes isolados de execução. Com este recurso é possível executar múltiplas unidades virtuais simultaneamente dentro do sistema operativo hospedeiro.

As unidades virtuais chamadas de “Contentores” são isoladas juntamente com grupos de controlo *Kernel* (*Kernel Cgroups*) [16] [18] e *Kernel Namespaces* [17].

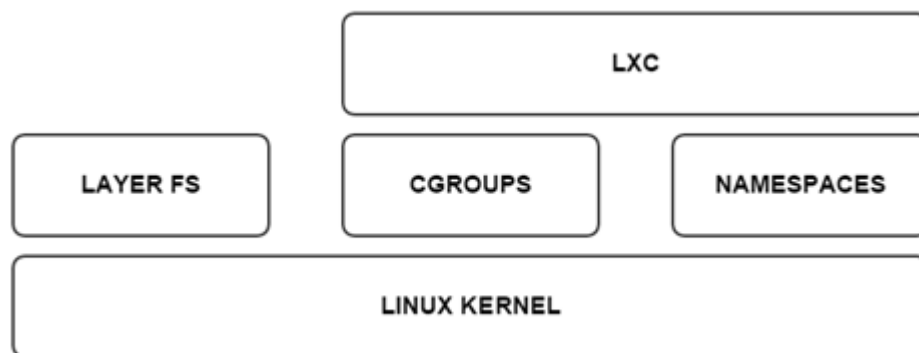


Figura 3 - Linux Container

LXC oferece um nível de virtualização de um sistema operativo *Linux* onde o *Kernel*, do sistema operativo hospedeiro, controla estes recipientes isolados (Contentores).

Concetualmente, LXC pode ser visto como uma técnica aperfeiçoada *chroot*. A diferença é que um ambiente *chroot* separa apenas o sistema de arquivos, enquanto LXC vai mais além e fornece gestão de recursos e controle via *cgroups*.

Benefícios de LXC:

- Isola aplicações e sistemas operativos através de contentores.
- Proporciona um desempenho quase nativo uma vez que faz gestão da alocação de recursos em tempo real.

Controla a interfaces de rede e isola os recursos de *hardware* dentro de contentores através de *cgroups*.

Docker

Para criar e gerir ambientes isolados para execução de aplicações, optou-se pela ferramenta *Docker* [10][20]. Esta ferramenta permite executar um ou mais sistema (s) operativo (s) Linux dentro de um sistema operativo Linux hospedeiro. Para este efeito, o *Docker*, usa um recurso *LXC* - *Linux Containers* que são uma espécie de contentores (ambientes virtuais) que possuem próprio CPU, memória, I/O, rede, espaço etc. fornecidos pelo *Kernel* do *SO Linux* hospedeiro.

Docker faz gestão do espaço físico dos contentores tendo acesso a camada *filesystem* do sistema operativo hospedeiro, usa a estratégia *copy-on-write filesystem* [19] para monitorizar alterações nos dados do utilizador. Contentores Docker [20] também são autosuficientes, possuem o mínimo base do sistema operativo, bibliotecas e *frameworks*.

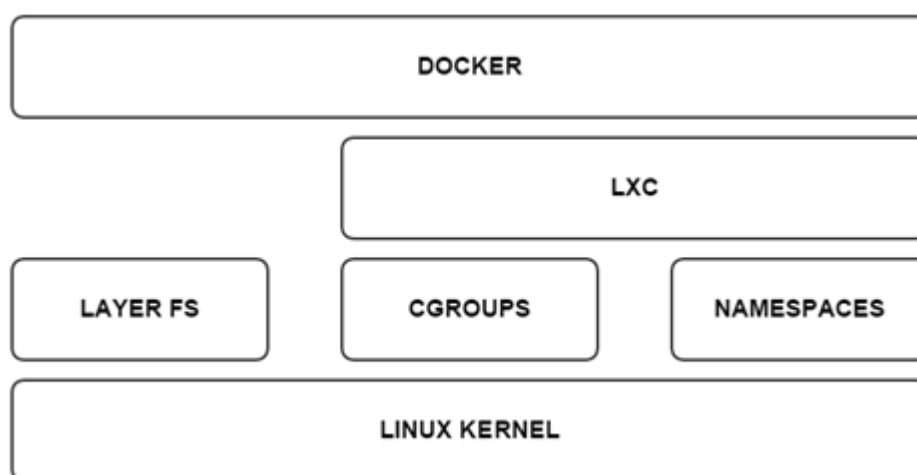


Figura 4 - Camada Docker

2.4 Armazenamento de dados

Redis

É um sistema de armazenamento de dados em pares chave-valor, oferecendo algumas estruturas de dados diferentes como strings, hashes, lists, sets and ordered sets. Cada um tipo de estrutura tem características únicas e suporta comandos únicos. Uma das características relevantes é o Redis possuir funcionalidades de Publicação/Subscrição [21] em canais de troca de dados “messaging” onde todo o interessado pode publicar mensagens e todo o interessado pode ler.

Sequelize

É uma biblioteca que oferece fácil acesso a bases de dados de tipo *MySQL* [11], *MariaDB* [22], *SQLite* [23] ou *PostgreSQL* [24] mapeando as tabelas de bases de dados em representações de classes onde cada registo nestas tabelas é representado como instâncias da classe correspondente.

Dizendo por outras palavras, é um ORM (*Object-Relational Mapper*) [25].

A biblioteca é escrita inteiramente em *JavaScript* e pode ser usado no ambiente de *Node.js*.

2.5 Plataforma

NodeJs

é uma plataforma assente na linguagem *JavaScript* com natureza totalmente assíncrona e que fornecer funcionalidades amigáveis para construção de aplicações web de carater escalável.

A arquitetura orientada a eventos assíncronos permitem ao *NodeJs* ter só um único processo que atende múltiplos pedidos de forma concorrente. [26]

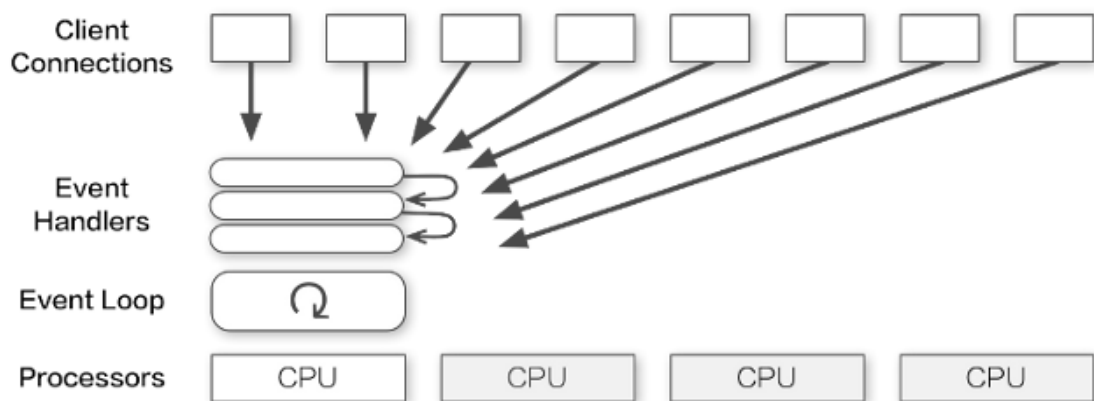


Figura 5 – NodeJs com um único fio de execução

NodeJs é extremamente rápido graças a mecanismos I/O assíncronos não bloqueantes [28] e a tecnologia do motor “*Google Chrome V8*”. [27]

[Fonte - Wikipedia]

Google Chrome V8 - é o nome do interpretador JavaScript, também chamado de máquina virtual Javascript (ou engine), desenvolvido pela Google e utilizado no seu navegador Google Chrome. Google Chrome V8 é uma ferramenta desenvolvida na linguagem C++ e distribuída no regime de código aberto.

A proposta do Google Chrome V8 é acelerar o desempenho de uma aplicação compilando o código Javascript para o formato nativo de máquina antes de executá-lo, permitindo que mesma velocidade de um código binário compilado.

Nas últimas duas décadas os processadores evoluíram de forma surpreendente. Hoje nós deparamos com tecnologias dotadas de grande poder de processamento graças a novos processadores multi-core.

Uma vez que a plataforma NodeJs corre numa única thread, e para ter aproveitamento do paralelismo em sistemas multi-core, a plataforma NodeJs possui o módulo *Cluster* [29] que automaticamente trata do balanceamento de conexões entre múltiplos processos. O próprio Cluster é uma instância de Node que corre com uma única thread.

O módulo Cluster permite facilmente criar Clusters filhos do processo Node pai que partilham entre eles os portos do servidor.

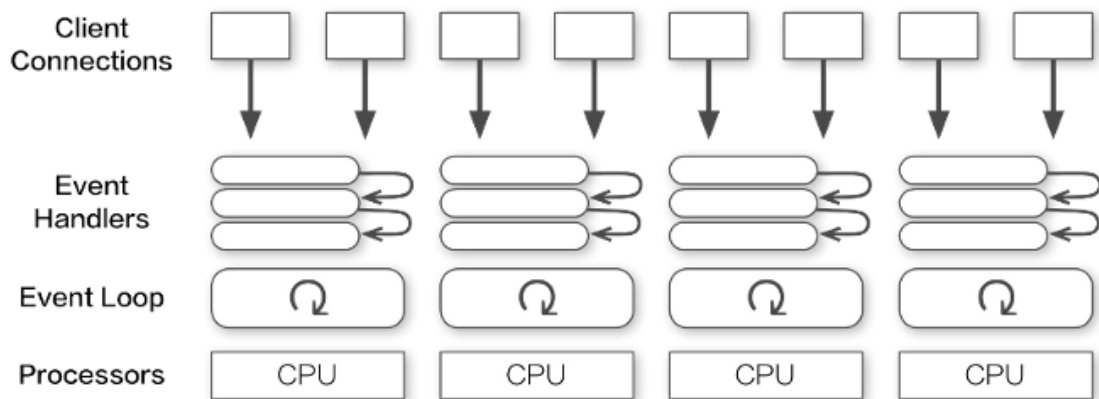


Figura 6 - Paralelismo NodeJs com N Cluster

Cada processo *Clusters* filho é gerado usando o método `child_process.fork`, de modo que eles possam comunicar com o processo *Clusters* pai via *IPC* (*Inter-process communication*). [29][30][31]

2.6 Protocolo de autenticação

OAuth2.0

No modelo tradicional de autenticação cliente-servidor, a solicitação do cliente para aceder a um recurso de acesso restrito (recurso protegido) no servidor é feita através de autenticação com o servidor recorrendo as credenciais do proprietário do recurso.

A fim de acordar o acesso a aplicação de terceiros no uso dos recursos de acesso restrito, o proprietário do recurso tem que partilhar os seus credenciais. Isso cria vários problemas e limitações:

- As aplicações de terceiros são obrigadas a guardar os credenciais do proprietário para uso futuro do recurso, geralmente uma senha no texto claro.
- As aplicações de terceiros ganham excessivamente amplo acesso aos recursos protegidos do dono, deixando-o sem capacidade de restringir o acesso a um só subconjunto de recursos.
- A única maneira de revogar o acesso a uma aplicação de terceiros, sem afetar o acesso a outras aplicações de terceiros, é alterar a senha desta mesma aplicação a revogar.

Com o protocolo *OAuth2.0* [32][33], em vez de usar os credenciais do proprietário no acesso ao recurso protegido, o cliente (aplicação de terceiros) obtém um *Token* [34] de acesso que possui tempo de vida, e outros atributos de acesso. Os *Tokens* de acesso são emitidos para o

cliente por um servidor de autorização com a aprovação do proprietário do recurso. O cliente usa o *Token* de acesso para aceder aos recursos protegidos hospedados pelo servidor de recursos.

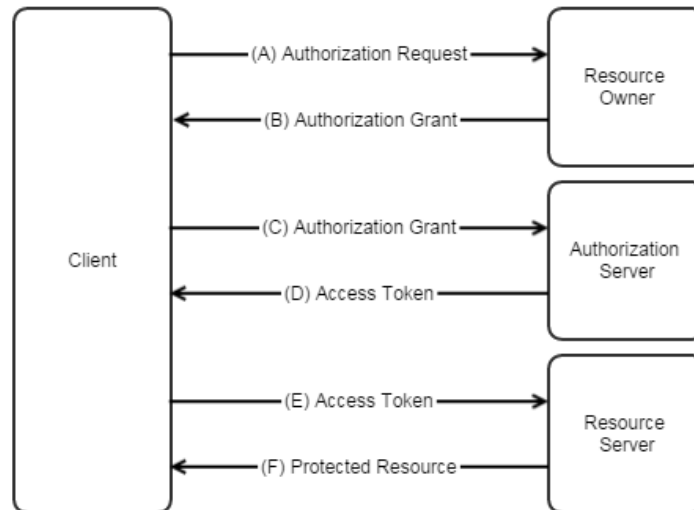


Figura 7 - Fluxo abstrato do protocolo OAuth2.0

A ilustração acima descreve o fluxo de ações de uma aplicação de terceiros (Cliente) na obtenção de um recurso protegido com os seguintes passos:

- a) O cliente pede autorização ao proprietário do recurso. O pedido de autorização pode ser feito diretamente para o proprietário do recurso ou de preferência indiretamente através do servidor de autorizações como intermediário.
- b) O cliente recebe a concessão (*Grant*) de autorização, que são credenciais que representam a autorização do dono do recurso.
- c) O cliente pede o *Token* de acesso através da autenticação no servidor de autorizações apresentando o *Grant* de autorização.
- d) O Servidor de autorização autentica o *Grant* e se for valido concede o *Token* de acesso.
- e) O cliente pede o recurso protegido ao servidor de recursos e autentifica-se apresentando o *Token* de acesso.
- f) O servidor de recursos valida o *Token* de acesso e se valido fornece então o recurso protegido.

2.8 Framework Web

Express

No desenvolvimento da aplicação servidora *Web* utilizou-se a *framework Express* para *NodeJs*. *Express* [35] é uma leve e flexível *framework* para desenvolvimento de aplicação web, que promove um conjunto robusto de recursos para a construção de aplicações simples, complexas ou híbridas.

3. Ambiente de desenvolvimento

A parte significativa no fluxo de desenvolvimento é montagem de um ambiente com características próprias, investigação das ferramentas de automatismo na configuração e aprovisionamento do ambiente com componentes e recursos necessários para o funcionamento do sistema e partilha dos mesmos de forma simples e comoda.

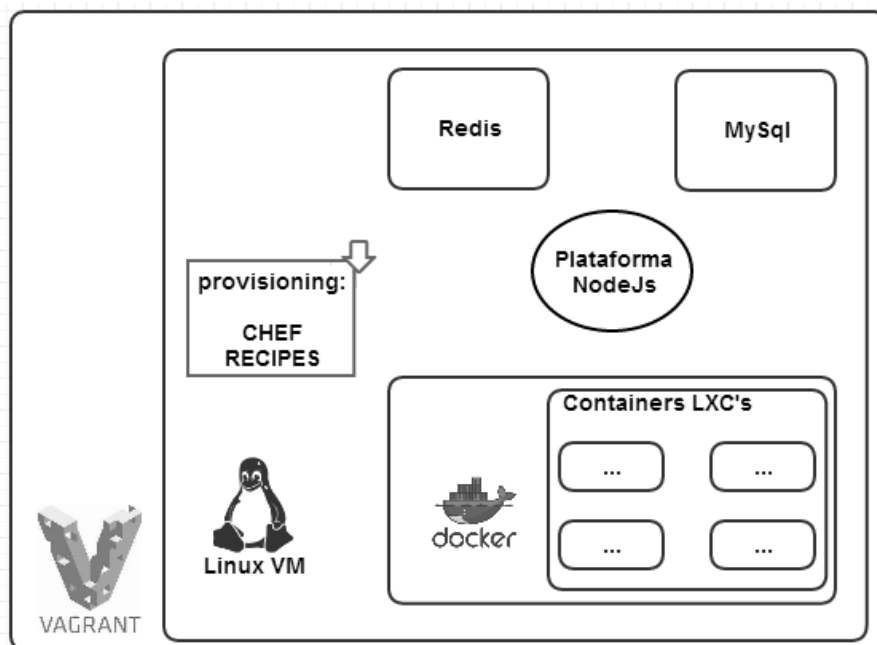


Figura 8 – Ambiente de desenvolvimento

O recurso fundamental do sistema a desenvolver é o sistema operativo *Linux*, assim sendo, a primeira necessidade que surgiu foi montar um emulador para o mesmo.

Por ser já reconhecido e o mais utilizado foi escolhido o emulador “*Oracle VM VirtualBox*” [2].

O próximo passo foi investigar como é que podíamos garantir que todas as alterações efetuadas no ambiente de desenvolvimento sejam facilmente replicadas em todas as estações de trabalho em que o projeto é desenvolvido e como ter um gestor que permite gerir a máquina virtual Linux sem interface gráfica a partir do ambiente de trabalho da máquina hospedeira, oferecendo a possibilidade em desenvolver o código a partir da máquina hospedeira e executa-lo na máquina virtual. A solução encontrada foi o gestor de máquinas virtuais *Vagrant*.

O *Vagrant* permite configurar o mapeamento de portos de encaminhamento da máquina virtual para serem acedidos a partir dos portas específicos da máquina hospedeira, permite também configurar o aprovisionamento automático da máquina virtual com ferramentas necessárias para o funcionamento da aplicação.

Ao executar o comando “*vagrant up*” é executado o “*start*” da máquina virtual.

```

$ vagrant up
Bringing machine 'default' up with 'virtualbox' provider...
==> default: Importing base box 'phusion/ubuntu-14.04-amd64'...
==> default: Matching MAC address for NAT networking...
==> default: Checking if box 'phusion/ubuntu-14.04-amd64' is up to date...
==> default: Setting the name of the VM: ciaas_default_1410548304520_46618
==> default: Clearing any previously set forwarded ports...
Updating Vagrant's berksshelf: 'c:/Users/Iurie/.berksshelf/default/vagrant/berkshe
lf-20140912-6048-1znxfq-default'

Vendoring yum (3.3.2) to c:/Users/Iurie/.berksshelf/default/vagrant/berksshelf-201
40912-6048-1znxfq-default/yum
Vendoring yum-epel (0.5.1) to c:/Users/Iurie/.berksshelf/default/vagrant/berkshe
lf-20140912-6048-1znxfq-default/yum-epel
Vendoring yum-mysql-community (0.1.10) to c:/Users/Iurie/.berksshelf/default/vagr
ant/berksshelf-20140912-6048-1znxfq-default/yum-mysql-community

==> default: Clearing any previously set network interfaces...
==> default: Preparing network interfaces based on configuration...
default: Adapter 1: nat
==> default: Forwarding ports...
default: 3000 => 8086 (adapter 1)
default: 3306 => 3306 (adapter 1)
default: 6379 => 6379 (adapter 1)
default: 22 => 2222 (adapter 1)
==> default: Booting VM...
==> default: Waiting for machine to boot. This may take a few minutes...
default: SSH address: 127.0.0.1:2222
default: SSH username: vagrant
default: SSH auth method: private key
default: Warning: Connection timeout. Retrying...
==> default: Machine booted and ready!
==> default: Checking for guest additions in VM...
default: Guest Additions Version: 4.3.6
default: VirtualBox Version: 4.2
==> default: Mounting shared folders...
default: /vagrant => C:/Users/Iurie/Desktop/ISEL_PROJ/CIAAS/CODE/ciaas
default: /tmp/vagrant-chef-3/chef-solo-2/roles => C:/Users/Iurie/Desktop/ISE
L_PROJ/CIAAS/CODE/ciaas/roles
default: /tmp/vagrant-chef-3/chef-solo-1/cookbooks => C:/Users/Iurie/.berksh
elf/default/vagrant/berksshelf-20140912-6048-1znxfq-default
==> default: Installing Chef 11.16.0 Omnibus package...
==> default: Downloading Chef 11.16.0 for ubuntu...

```

Figura 9 - Vagrant up

Via o comando “vagrant destroy” é lançado o processo destruição da maquina virtual montada pelo “vagrant up”:

```

$ vagrant destroy
default: Are you sure you want to destroy the 'default' VM? [y/N] y
==> default: Destroying VM and associated drives...
Cleaning Vagrant's berksshelf
==> default: Running cleanup tasks for 'chef_solo' provisioner...

```

Figura 10 - Vagrant destroy

Via o comando “vagrant ssh” acede-se para dentro da máquina virtual tendo lá já mapiado os ficheiros e estrutura das pastas do projeto da máquina hospedeira.

```

$ vagrant ssh
Welcome to Ubuntu 14.04 LTS (GNU/Linux 3.13.0-24-generic x86_64)

 * Documentation:  https://help.ubuntu.com/

Last login: Sat Sep 20 12:25:21 2014 from 10.0.2.2
vagrant@ubuntu-14:/$ ls
bin      etc      lib      media   proc     sbin     tmp      var
boot    home     lib64    mnt     root     srv      usr      VBoxLinuxAdditions
dev      initrd.img  lost+found  opt     run      sys      vagrant  vmlinuz
vagrant@ubuntu-14:/vagrant$ ll
total 600037
drwxrwxrwx  1 vagrant vagrant    4096 Sep 20 12:41 ./
drwxr-xr-x 24 root     root    4096 Sep 20 11:27 ../
drwxrwxrwx  1 vagrant vagrant      0 Sep 20 12:04 app/
drwxrwxrwx  1 vagrant vagrant    4096 Sep 20 12:21 containers/
drwxrwxrwx  1 vagrant vagrant      0 Sep 12 18:55 cookbooks/
-rwxrwxrwx  1 vagrant vagrant  48197 Sep 12 18:55 crash.log*
drwxrwxrwx  1 vagrant vagrant    4096 Sep 20 11:17 .git/
-rwxrwxrwx  1 vagrant vagrant    147 Sep  7 16:36 .gitignore*
-rwxrwxrwx  1 vagrant vagrant 614364160 Sep 20 12:42 nodejs.tar*
drwxrwxrwx  1 vagrant vagrant      0 Sep 20 12:22 packer_cache/
-rwxrwxrwx  1 vagrant vagrant    2250 Aug 31 16:19 README.md*
drwxrwxrwx  1 vagrant vagrant      0 Sep  7 16:36 roles/
drwxrwxrwx  1 vagrant vagrant      0 Sep 12 18:56 .vagrant/
-rwxrwxrwx  1 vagrant vagrant    4988 Sep 12 19:02 Vagrantfile*

```

Figura 11 - Vagrant ssh

Via o comando “*vagrant halt*” encera-se a maquina virtual:

```

$ vagrant halt
==> default: Attempting graceful shutdown of VM...

```

Figura 12 - Vagrant halt

O *Vagrant* é um gestor de máquinas virtuais, isto implica que o mesmo utiliza instruções especificadas pelo utilizador para configurar as máquinas virtuais. Para tal efeito, o *Vagrant*, disponibiliza um ficheiro próprio de configuração chamado *vagrantfile*.

Assim sendo sempre que houver alguma alteração nas configurações da máquina virtual ou até substituição por uma outra, basta partilhar o ficheiro *vagrantfile* entre membros da equipa.

O *Vagrant* possui um repositório público com uma coleção de imagens de máquinas virtuais chamadas *boxes*. Assim ao executar o comando “*vagrant up*”, o *Vagrant* descarrega, só pela primeira vez, a *box* especificada no ficheiro *vagrantfile* e aplica as configurações descritas no mesmo, facilitando e evitando desta maneira a partilha física da própria maquina virtual que sempre é uma chatice devido ao tamanho que está pode ter.

O processo de aprovisionamento das máquinas virtuais. O processo de aprovisionamento consiste em configurar um sistema de aprovisionamento como o *Chef* ou *Puppet* no processo de aprovisionamento *Vagrant*.

O *Chef* funciona a base de receitas que são ficheiro escritos em linguagem de programação *Ruby*, em que se descreve de forma programática a gestão das aplicações e como elas devem ser configuradas.

As receitas são agrupadas em coleções chamados *cookbook*. A *cookbook* [37] é uma unidade fundamental de configuração e de políticas de distribuição. Cada *cookbook* define um cenário, como por exemplo o necessário (dependências) para instalação e configuração do MySQL, contendo todos os componentes que são obrigatórios para suportar o tal cenário e garantindo que cada recurso está devidamente configurado.

As unidades *cookbook* são fornecidas pela ferramenta *Berkshelf* [8] que sabe resolver dependências de *cookbooks*.

4 Projeto

Este capítulo apresenta a solução proposta e arquitetura geral do projeto, descrevendo os seus componentes e a interação entre os mesmos.

4.1 Solução

A solução proposta possui três componentes desenvolvidos que em conjunto com outros recursos compõem a plataforma. Os componentes são *Web*, *Worker* e *Hub*. Estes componentes são aplicações autónomas cujo funcionamento não depende uma das outras.

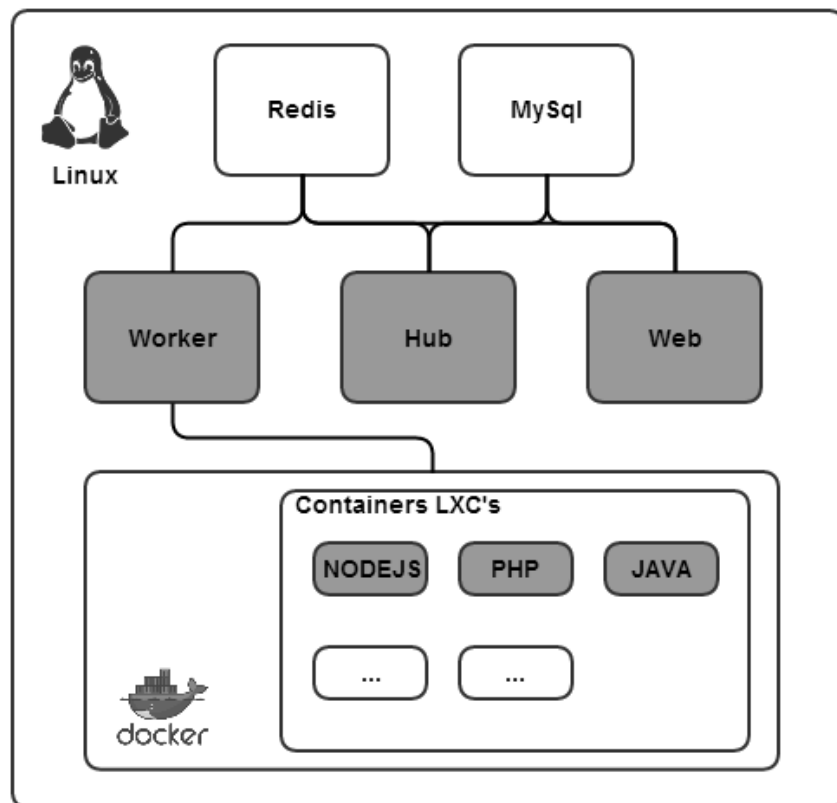


Figura 13 - Aplicações desenvolvidas

O componente *Web* é um servidor que trata da usabilidade da interface Web, autenticação, agendamento de trabalho e apresentação dos resultados.

O *Worker* é um serviço que prepara os ambientes virtuais para execução do trabalho agendado via *Docker*, executa-o dentro dos mesmos e publica os resultados.

O *Hub* é um serviço que trata em notificar por correio eletrónico os utilizadores com os resultados das execuções, guardando-os também na base de dados persistentes.

Sendo aplicações autónomas, tem que haver algo que os une no seu funcionamento, algo que possui funcionalidade de uma conduta de informação que as três partes sabem interpretar e que lhes faz sentido. Para este efeito faz-se uso das funcionalidades do Redis, tais com Publicação/Subscrição em canais de troca de dados “*messaging*” onde todo o interessado pode publicar mensagens e todo o interessado pode ler. Assim, a informação das execuções dos trabalhos e os resultados dos mesmos são propagados nos canais do Redis.

Para dados persistentes como informação acerca das contas de utilizadores, projetos, execuções e resultados, faz-se uso da base de dados relacional *MySQL*.

A imagem seguinte demonstra como os componentes interagem entre si via *Redis* e *MySQL* num fluxo de criação, execução e retorno de resultados de um trabalho despoletado pelo *GitHub* quando neste são submetidos alterações de código.

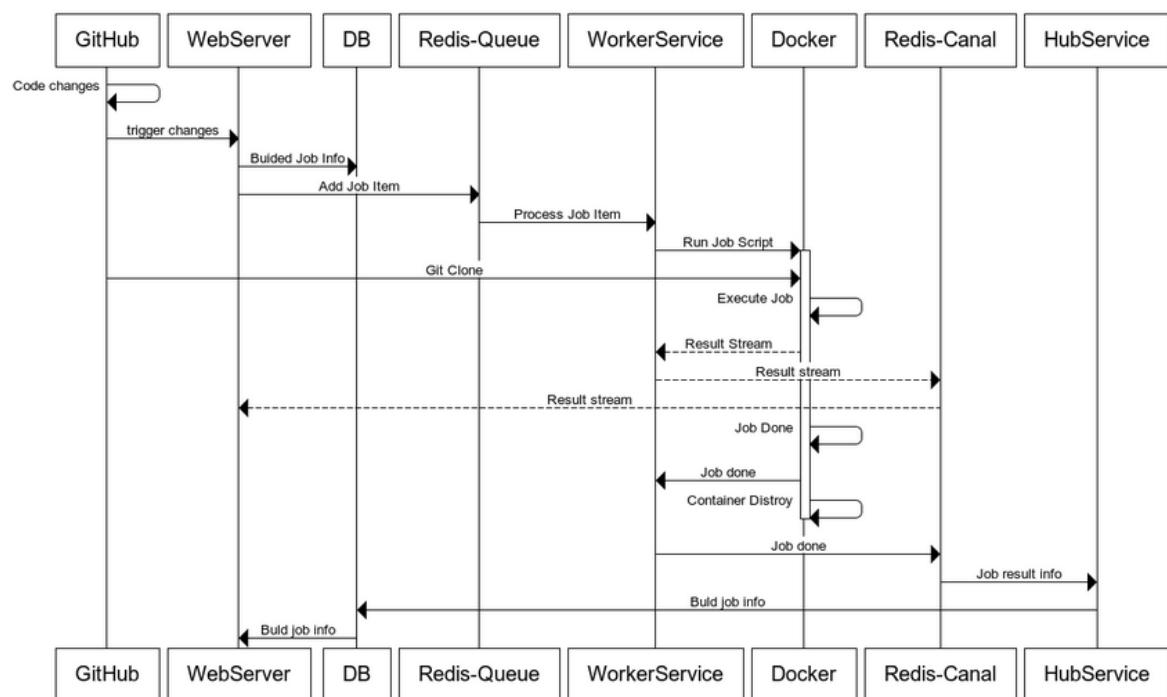


Figura 14 – Diagrama do fluxo do trabalho entre componentes

A usabilidade da interface *Web* consiste em criar, editar, executar projetos tendo possibilidade de seguir a evolução da execução visualizando os logs fornecidos em tempo real, visualizar o histórico de execuções (*Builds*) e eliminar projetos.

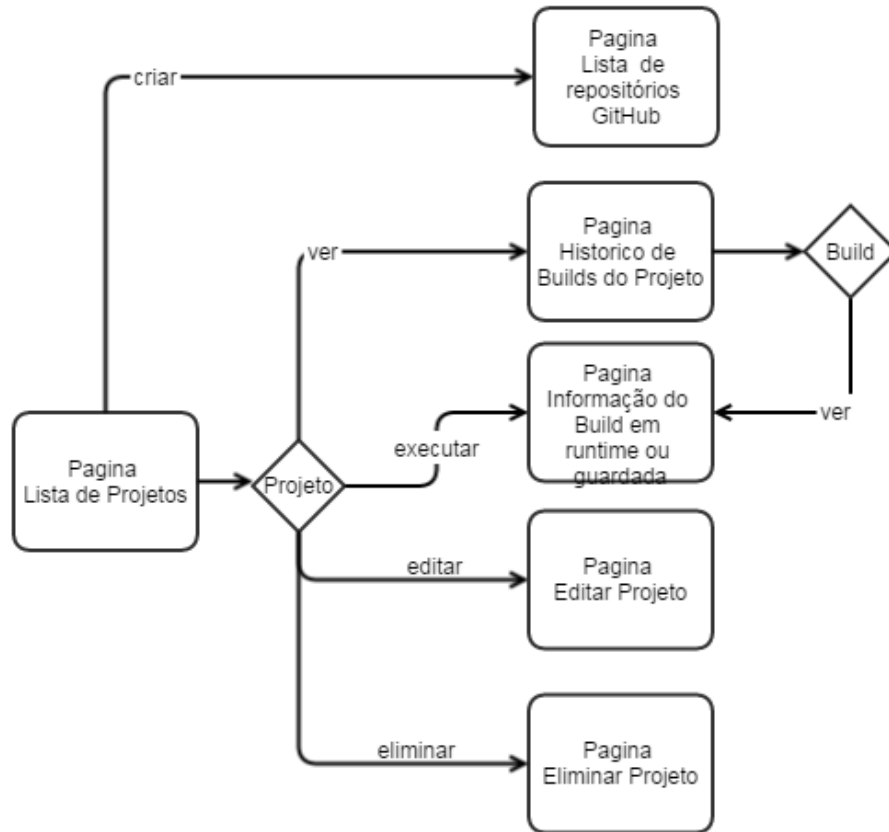


Figura 15 - Usabilidade Web

Ao utilizador disponibilizam-se duas formas de registo e autenticação, local e via aplicação *GitHub* usufruindo do protocolo *OAuth2*.

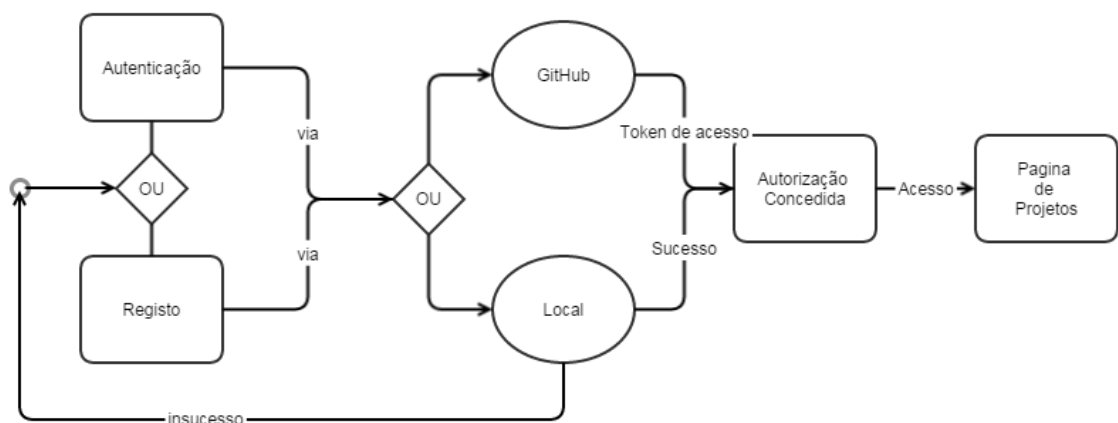
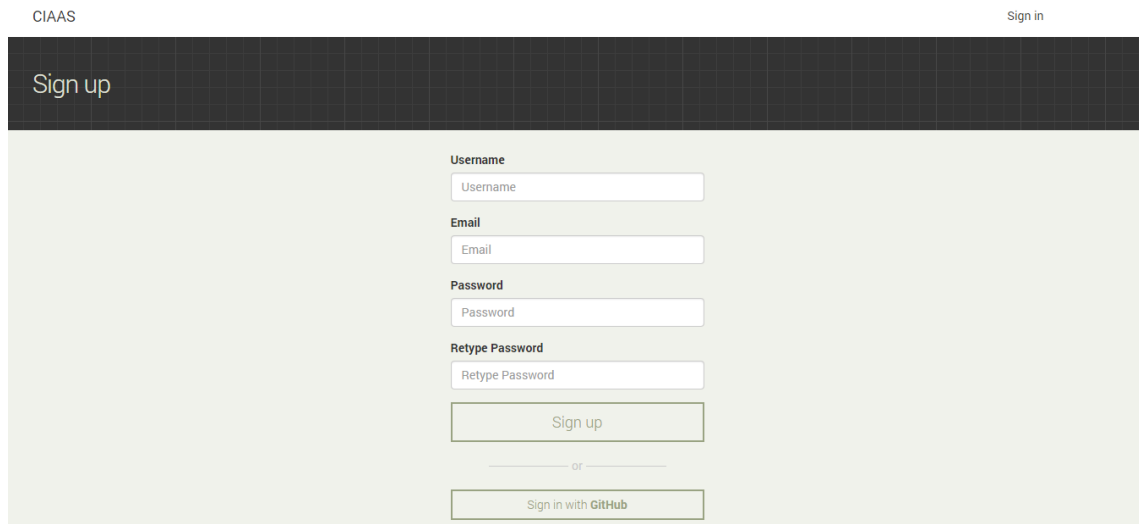


Figura 16 - Fluxo Autorização e Registo

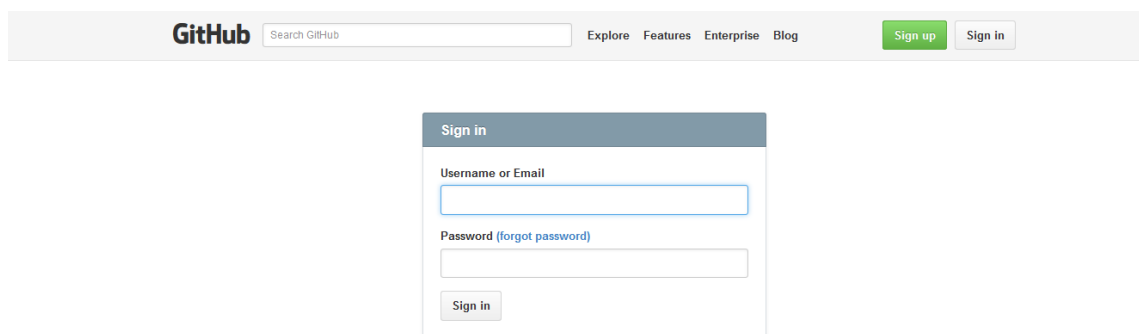
O registo local consiste em fornecer um correio eletrónico válido e credenciais de acesso, nome do utilizador e palavra-chave.



The screenshot shows the 'Sign up' page for CIAAS. At the top left is the 'CIAAS' logo and at the top right is a 'Sign in' link. The main heading is 'Sign up'. Below it, there are four input fields: 'Username', 'Email', 'Password', and 'Retype Password'. Each field has a placeholder text matching its label. Below the fields is a 'Sign up' button. Underneath the button is the word 'or' flanked by horizontal lines. At the bottom is a button labeled 'Sign in with GitHub'.

Figura 17 - Pagina Registo

No registo via *GitHub*, o utilizador será encaminhado para página de autenticação *GitHub*



The screenshot shows the GitHub 'Sign in' page. At the top is the GitHub logo, a search bar, and links for 'Explore', 'Features', 'Enterprise', and 'Blog'. On the right are 'Sign up' and 'Sign in' buttons. The main content is a 'Sign in' modal box with a title bar. Inside, there are two input fields: 'Username or Email' and 'Password (forgot password)'. Below the fields is a 'Sign in' button.

Figura 18 - Autenticação GitHub

onde, após autenticação e autorização da aplicação *CIAAS (Continuous Integration as a Service)* para ceder aos seus conteúdos públicos, será reencaminhado para pagina de projetos no plataforma *CIAAS*.

3.3 Arquitetura

Como arquitetura da solução concebeu-se uma arquitetura possível de configurar para alta disponibilidade e tolerância as falhas dos componentes do sistema, apresentada na figura 19.

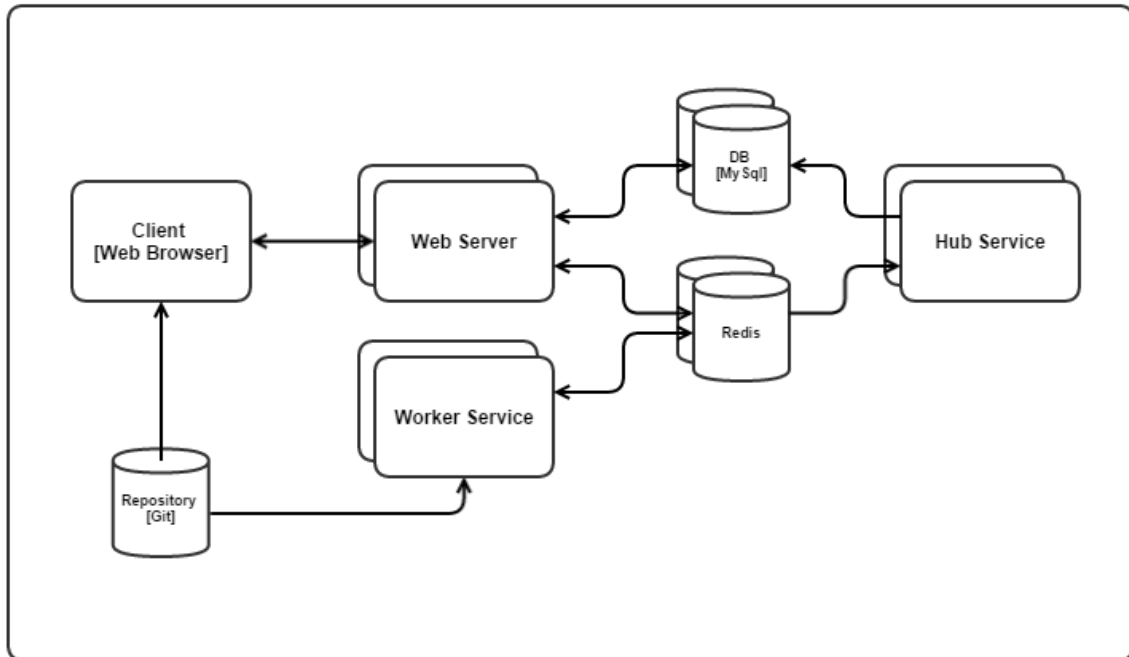


Figura 19 – Arquitetura geral

Por outras palavras, para cada componente podem existir N instâncias a funcionar em paralelas. Mesmo que uma das instâncias por alguma razão falhe, existem outra (as) a atender os pedidos.

A base de dados persistente que suporta a logica do funcionamento é constituída por quatro entidades base, Figura 20.

1. Accounts: Dados acerca dos utilizadores.
2. Projects: Dados acerca dos projetos que os utilizadores possuem.
3. Builds: Dados acerca das execuções dos projetos.
4. Containers: Coleção dos contentores disponíveis no sistema

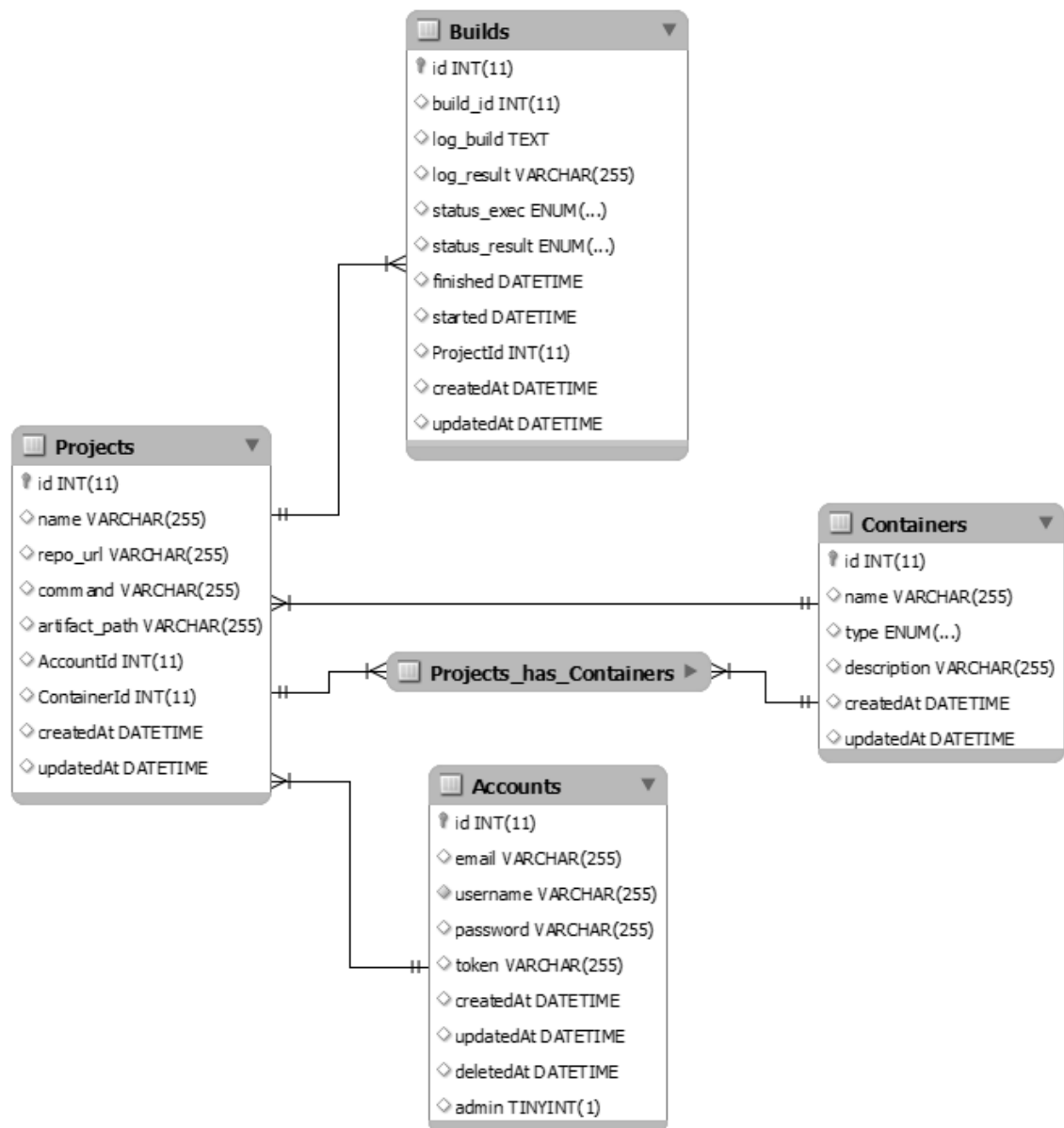


Figura 20 - Modelo de esquemas

A figura acima mostra o modelo de relações, em que:

1. Um projeto "Project" pode ter uma só conta de utilizador "Account"
2. Uma conta de utilizador "Account" pode estar associada a múltiplos projetos "Project"
3. Um projeto "Project" pode ter só um tipo de contentor "Container" primário.
4. Um tipo de contentor "Container" primário pode estar associado a diversos projetos "Projects"
5. Um projeto "Project" pode ter diversos tipos de contentores "Container" secundários.

6. Um tipo de contentor “Container” secundário pode estar associado a múltiplos projetos “Project”
7. Um projeto “Project” pode ter múltiplas execuções “Build”
8. Uma execução “Build” só pode estar associada a um projeto “Project”.

Para manter a consistencia, a transferência do modelo de bases de dados de um estado para outro e vice-versa, utiliza-se o sistema de migrações oferecido pela biblioteca *Sequelize*. As transições de estado são salvas em arquivos de migração, que descrevem de forma programatica, como chegar ao novo estado e como reverter as mudanças, a fim de voltar para o estado anterior. A imagem seguinte mostra um exemplo de um ficheiro de migração “20140919212845-commit_info.js”

```
module.exports = {
  up: function(migration, DataTypes, done) {
    migration.addColumn('Projects', 'default_branch', {type:DataTypes.STRING,defaultValue:"master"})
    .then(function(){
      return migration.addColumn('Builds', 'branch', {type:DataTypes.STRING,defaultValue:"master"});
    }).then(function(){
      return migration.addColumn('Builds', 'commit_id', {type:DataTypes.STRING});
    }).then(function(){
      return migration.addColumn('Builds', 'commit_message', {type:DataTypes.STRING});
    }).then(function(){
      return migration.addColumn('Builds', 'commit_author', {type:DataTypes.STRING});
    }).then(function(){
      done()
    });
  },
  down: function(migration, DataTypes, done) {
    migration.removeColumn('Projects', 'default_branch').then(function(){
      return migration.removeColumn('Builds', 'branch');
    }).then(function(){
      return migration.removeColumn('Builds', 'commit_id');
    }).then(function(){
      return migration.removeColumn('Builds', 'commit_message');
    }).then(function(){
      return migration.removeColumn('Builds', 'commit_author');
    }).then(function(){
      done()
    })
  }
}
```

Figura 21 - Exemplo de ficheiro de migração

As funcionalidades de migração UP e DOWN perimetem a transição de estado dos *Schemas*. Com o comando “*npm run-script migrateUp*” são executados os procedimentos que alteram as definições de schema para um novo estado.

Com commando “*npm run-script migrateDown*” são executados os procedimentos que alteram as definições de schema para um estado anterior.

A criação de um novo ficheiro de migração onde se deve descrever o codigo de migração é executado via comando “*sequelize -c <NOME>*”.

3.3.1 Servidor Web

A interação entre a aplicação cliente (*Browser*) e o serviço *Web* baseia-se no paradigma pedido/resposta, utilizando o protocolo *HTTP*.

Para o utilizador visualizar no *Browser* os resultados (*Build*) de uma execução em tempo real é efetuada uma comunicação bidirecional via *WebSocket*.

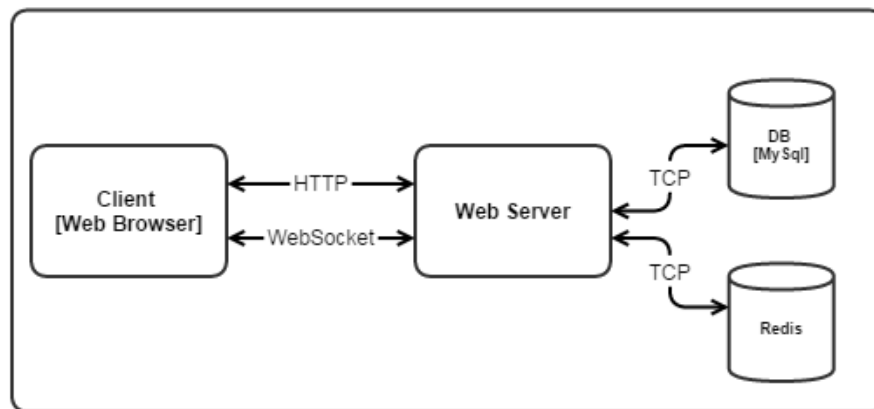


Figura 22 - Arquitetura do Servidor Web

No desenvolvimento do front-end da aplicação web foi utilizado a framework *Bootstrap* para obter experiência de visualização com fácil leitura e navegação em uma ampla gama de dispositivos como em monitores de computador e *smartphones*.

A captura de tela mostra a interface de usuário para o sistema CIAAS. No topo, o nome 'CIAAS' está à esquerda e um ícone de menu (três linhas horizontais) está à direita. Abaixo, há uma seção de cabeçalho com o título 'Sign up' em um fundo escuro. O formulário principal, com fundo claro, contém quatro campos de entrada: 'Username', 'Email', 'Password' e 'Retype Password'. Cada campo é precedido por seu respectivo rótulo em negrito.

Figura 23 - Exemplo visualização 320x480

3.3.2 Aplicação Worker

A aplicação *Worker* foi desenhada de maneira a ter o menor número de dependências de maneira a permitir a execução do mesmo em múltiplas instâncias servidoras de forma simples.

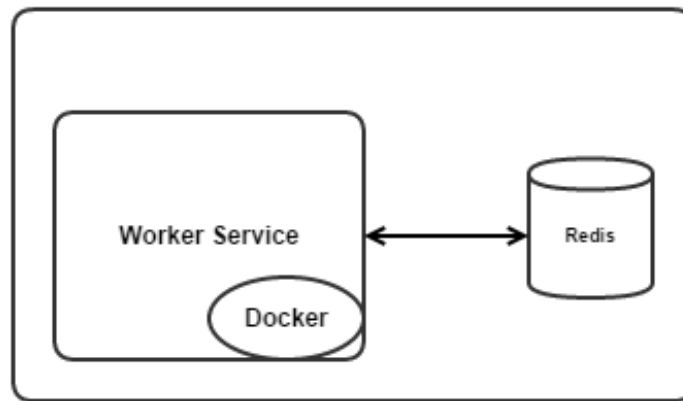


Figura 24 - Arquitetura geral Worker

Worker desenvolvido de maneira a funcionar com um único fio de execução e processando um trabalho de cada vez. Assim foram aproveitadas as características da plataforma *NodeJs* (modulo Cluster) que possibilitam tratar as questões de paralelismo sem termos a preocupação com a *Thread safety*, isto é o trabalho na fila *Redis* pode ser processado por vários processos *Worker*. U nível da concorrência define-se numa variável de ambiente “*CONCURENCY*” que indica o número de processos *Worker* a serem lançados.

```
vagrant@ubuntu-14:/vagrant/app/worker$ sudo node server.js
info: STARTED WORKER CLUSTER WITH CONCURENCY = 10
info: WORKER SLAVE IS STARTED ID= 2144
info: WORKER SLAVE IS STARTED ID= 2148
info: WORKER SLAVE IS STARTED ID= 2145
info: WORKER SLAVE IS STARTED ID= 2147
info: WORKER SLAVE IS STARTED ID= 2153
info: WORKER SLAVE IS STARTED ID= 2157
info: WORKER SLAVE IS STARTED ID= 2156
info: WORKER SLAVE IS STARTED ID= 2150
info: WORKER SLAVE IS STARTED ID= 2152
info: WORKER SLAVE IS STARTED ID= 2155
```

Figura 25 - Processos Worker

O objetivo do *Worker* consiste em estar a escuta do novo trabalho na fila *Redis* para ser executado via *Docker*. Para tal, o *Worker* possui um *Handler* que é despoletado pela fila *Redis* sempre que haja novo trabalho.

A informação do trabalho na fila *Redis* é representada via objetos pares chave/valor, contendo o identificador da execução, configurações, tipo de contentor, informação do repositório associado ao projeto e os comandos, Figura 26.

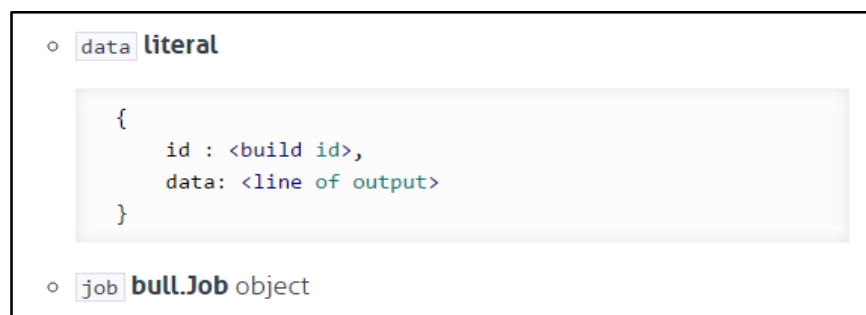
```
var job = {
  _id: _id,
  id: _buildid,
  config: {
    language: "JS",
    timeout: 500000
  },
  container: {
    primary: container.name
  },
  repository: {
    uri: _project.repo_url,
    name: _project.name
  },
  skipSetup: false,
  payload: {
    commands: _project.command.split("\n")
  }
}
```

Figura 26 – Exemplo de representação do trabalho na fila Redis

Os resultados da execução de um trabalho são propagados num canal Redis associado ao trabalho.

O Docker oferece dois canais de *stream standart* de entrada e saída que *Worker* utiliza para ler os resultados provenientes do contentor *Docker* e transmitir para os canais do Redis.

Durante a execução de um trabalho, o fluxo *LOG* é propagado com seguinte formato:



O diagrama mostra a estrutura de um log stream. No topo, há um ícone de círculo com o rótulo 'data' e o texto 'literal'. Abaixo, um bloco cinza contém um objeto JSON: { id : <build id>, data: <line of output> }. Na base, há outro ícone de círculo com o rótulo 'job' e o texto 'bull.Job object'.

Figura 27 - Formato Log stream

A cada linha de *Log* de saída do *Docker stdout* é atribuído o ID da execução a que se refere e de seguida é inserida no canal respetivo de *Redis*.

Para cada execução é atribuído um canal Redis para resultados de execução, em que cada fim de execução é propagado com informação do estado.

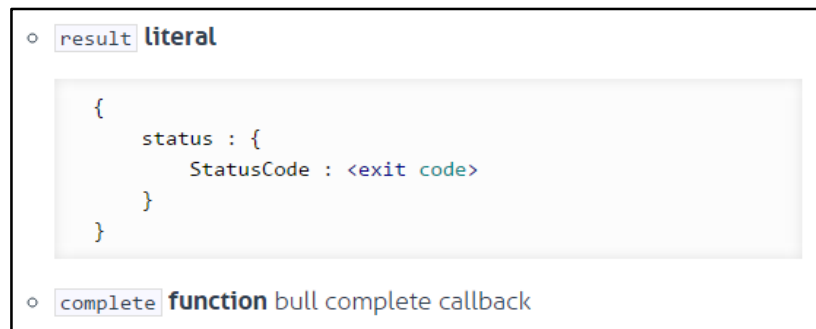


Figura 28 – Formato do estado de execução

3.3.3 Aplicação Hub

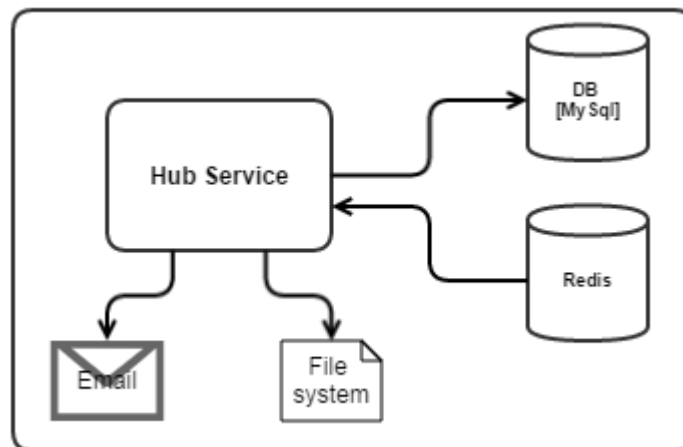


Figura 29 - Arquitetura geral Hub

A funcionalidade *Hub* consiste numa sequência de ações de persistência e notificação de resultados das execuções.

A notificação pode ser realizada via diferentes estratégias: notificação por correio eletrónico ou por exemplo *SMS*, etc.

A persistência dos dados da execução consiste em guarda-los em forma de ficheiro, podendo ser utilizadas vários tipos de estratégias de armazenamento, tais como sistema de ficheiros do sistema operativo, *Amazon S3*, etc.

5 Conclusão

Na área de programação e nomeadamente no contexto de integração contínua, é difícil evitar a ocorrência de erros quando o desenvolvimento é efetuado em equipa ou equipas. Assim sendo a frequência dos testes no processo de desenvolvimento é uma prática importante, pois contribui para deteção de erros provenientes de múltiplas integrações.

Na forma mais comum, para além dos testes que cada programador efetua localmente na sua máquina, é necessário testar o código do produto na íntegra. A forma ideal seria efetuar o teste sempre que ocorre uma integração e dependendo dos critérios do projeto, efetua-los em ambientes com propriedades distintas dependente dos critérios dos mesmos.

O presente projeto descreve uma possível solução para tornar o processo de integração contínua tradicional num processo automatizado. No sistema proposto, os programadores configuram o sistema de maneira a tornar os testes automatizados sempre que houver uma integração e se for necessário em ambientes de execução com propriedades diferentes oferecidos pelo sistema.

A solução proposta apresenta vantagens evidentes em relação ao modelo tradicional de integração contínua, tendo capacidade de alertar sempre na hora de ocorrência de conflitos e erros no código desenvolvido. Também uma vantagem relevante é a diminuição do tempo e recursos que normalmente associados que se traduzem em custos financeiros.

5.1 Limitações

A maior limitação que faz a diferença é o sistema ser destinado a projetos desenvolvidos para serem aplicados em máquina *Unix*.

Outra limitação é não suportar execução de código de projetos que façam uso da ferramenta *Docker*, uma vez que, o mesmo necessita de acesso a recursos *Kernel* que os contentores onde o código se executa não possuem.

5.2 Trabalho futuro

Como trabalho futuro pode se considerar os seguintes aspetos:

- Diversificar as formas de notificação dos utilizados com os resultados.

- Acrescentar ao sistema o conceito de grupo de trabalho para equipas a desenvolver módulos diferentes para mesmo produto, tomando em consideração a hierarquia de cargos (*Roles*)
- “*Code Quality Service*” capacidade de detetar mas praticas de programação e sugerir alternativas.
- Acrescentar sistema de *Billing* para recursos de alta performance e outro serviço extra como por exemplo “*Code Quality Service*” mencionado acima.

6 Referencias

- [1] - PHP - <http://php.net/>
- [2] - Virtualbox - <https://www.virtualbox.org/>
- [3] - Vagrant - <https://docs.vagrantup.com/v2/>
- [4] - Chefe - <https://www.getchef.com/chef/>
- [5] - Amazon EC2 - <http://aws.amazon.com/ec2/purchasing-options/spot-instances/>
- [6] - Google Cloud - <https://cloud.google.com/>
- [7] - Microsoft Azure - <https://azure.microsoft.com/en-us/>
- [8] - Berkshelf - <http://berkshelf.com/>
- [9] - Nodejs - <http://nodejs.org/>
- [10] - Docker - <https://www.docker.com/whatisdocker/>
- [11] - MySQL - <http://pt.wikipedia.org/wiki/MySQL>
- [12] - Redis - <http://en.wikipedia.org/wiki/Redis>
- [13] - Xen - <http://en.wikipedia.org/wiki/Xen>
- [14] - KVM - http://www.linux-kvm.org/page/Main_Page
- [15] - LXC - <https://linuxcontainers.org/>
- [16] - Kernel Cgroups - <http://en.wikipedia.org/wiki/Cgroups>
- [17] - Kernel Namespaces - <http://www.kbartocha.com/tag/linux-kernel-namespaces/>
- [18] - Linux Kernel - [http://pt.wikipedia.org/wiki/Linux_\(n%C3%BAcleo\)](http://pt.wikipedia.org/wiki/Linux_(n%C3%BAcleo))
- [19] - Copy-on-write filesystem - <http://en.wikipedia.org/wiki/Copy-on-write>
- [20] - Docker Containers - <http://www.linuxjournal.com/content/docker-lightweight-linux-containers-consistent-development-and-deployment>
- [21] - Redis Publish/Subscribe messaging paradigm - <http://redis.io/topics/pubsub>
- [22] - MariaDB - <http://pt.wikipedia.org/wiki/MariaDB>
- [23] - SQLite - <http://www.sqlite.org/>
- [24] - PostgreSQL - <http://pt.wikipedia.org/wiki/PostgreSQL>
- [25] - Object Relational Mapper(ORM) - <http://hibernate.org/orm/what-is-an-orm/>
- [26] - Why NodeJs? - <http://www.haneycodes.net/to-node-js-or-not-to-node-js/>
- [27] - Google V8 - [http://pt.wikipedia.org/wiki/V8_\(JavaScript\)](http://pt.wikipedia.org/wiki/V8_(JavaScript))
- [28] - Asynchronous I/O with Node - <https://www.inkling.com/read/javascript-definitive-guide-david-flanagan-6th/chapter-12/asynchronous-io-with-node>
- [29] - Node Cluster - <https://devcenter.heroku.com/articles/node-cluster>
- [30] - Process Fork - [http://en.wikipedia.org/wiki/Fork_\(system_call\)](http://en.wikipedia.org/wiki/Fork_(system_call))

- [31] - IPC (Inter Process Communication) - http://en.wikipedia.org/wiki/Inter-process_communication
- [32] - OAuth2.0 - <http://oauth.net/2/>
- [33] - OAuth2.0 rfc6749 - <http://tools.ietf.org/html/rfc6749>
- [34] - Access Token - <http://en.wikipedia.org/wiki/Token>
- [35] - Express Js - <http://expressjs.com/>
- [36] - Bootstrap - <http://getbootstrap.com/>
- [37] – Chef Cookbook - https://docs.getchef.com/essentials_cookbooks.html