****

Instituto Superior de Engenharia de Lisboa

Licenciatura em Engenharia Informática e de Computadores

Projeto e Seminário

***Plataforma de Integração Contínua***

Pavel Egorov

Iurie Marcinschi

Orientador

Doutor Porfírio Pena Filipe

Julho 2014

# Resumo

No contexto de construção de *Software*, normalmente, o desenvolvimento de cada membro da equipa traduz-se em múltiplas integrações de código por dia, em que cada integração é verificada por meio de testes unitários. Assim sendo, a integração contínua é uma prática que desempenha um papel indispensável no dia-a-dia de um programador.

O presente projeto propõe uma plataforma *Web* que oferece suporte para facilitar a prática de integração contínua, disponibilizando ambientes *Linux* para compilação de código e execução dos testes unitários.

O programador terá possibilidade de importar o código dos seus repositórios para poder testar em ambientes *Linux* à medida, que a plataforma oferece.

Um teste na plataforma consiste em compilar o código e executar testes unitários escritos pelo programador de forma automática, disponibilizando-lhe o log da execução e o resultado.

# 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 – NodeJs com um único fio de execução 4](#_Toc398056508)

[Figura 2 - Paralelismo NodeJs com N Cluster 5](#_Toc398056509)

[Figura 3 - Fluxo abstrato do protocolo OAuth2.0 6](#_Toc398056510)

[Figura 4 - Linux Container 8](#_Toc398056511)

[Figura 5 - Modelo antes da solução. 9](#_Toc398056512)

[Figura 6 - Modelo depois da solução 10](#_Toc398056513)

[Figura 7 – Ferramentas e Tecnologias 11](#_Toc398056514)

[Figura 8 - Ambiente de desenvolvimento 14](#_Toc398056515)

[Figura 9 – Modelo de relações entre componentes via Redis e Bases de Dados 15](#_Toc398056516)

[Figura 10 – Diagrama do fluxo do trabalho entre componentes 16](#_Toc398056517)

[Figura 11 - Camada Docker 17](#_Toc398056518)

**Sumário**

[Resumo i](#_Toc398081027)

[Agradecimentos ii](#_Toc398081028)

[Índice Figura iii](#_Toc398081029)

[1 Introdução 1](#_Toc398081030)

[1.1 Enquadramento e Motivação 1](#_Toc398081031)

[1.2 Objetivos 2](#_Toc398081032)

[1.3 Organização do documento 3](#_Toc398081033)

[2 Estado da Arte 4](#_Toc398081034)

[2.1 Plataforma de programação 4](#_Toc398081035)

[2.2 Sistema multi-core 5](#_Toc398081036)

[2.3 Acesso a recursos via OAuth2.0 5](#_Toc398081037)

[2.4 LXC – Contentores Linux 7](#_Toc398081038)

[3 Solução e Arquitetura 8](#_Toc398081039)

[3.1 Solução proposta 8](#_Toc398081040)

[3.2 Ambiente e Ferramentas 10](#_Toc398081041)

[3.2.1 Ferramentas e Tecnologia 10](#_Toc398081042)

[3.2.2 Montagem do ambiente, Partilha e Aprovisionamento 13](#_Toc398081043)

[3.2.2.1 Ambiente 13](#_Toc398081044)

[3.2.2.2 Partilha 14](#_Toc398081045)

[3.2.2.3 Aprovisionamento 14](#_Toc398081046)

[3.3 Arquitetura Geral 15](#_Toc398081047)

[3.3.1 Componente Web 16](#_Toc398081048)

[3.3.2 Componente Worker 19](#_Toc398081049)

[3.3.3 Componente Hub 20](#_Toc398081050)

[4 Implementação 20](#_Toc398081051)

[4.1 Aplicação Worker 20](#_Toc398081052)

[4.2 Aplicação Hub 20](#_Toc398081053)

[4.3 Aplicação Web 20](#_Toc398081054)

[5 Conclusão 22](#_Toc398081055)

[5.1 Limitações 22](#_Toc398081056)

[5.2 Trabalho futuro 22](#_Toc398081057)

[6 Referencias 22](#_Toc398081058)

# 1 Introdução

O presente capítulo contextualiza e apresenta a motivação e os objetivos a alcançar.

## 1.1 Enquadramento e Motivação

Há poucas décadas atrás, equipas de desenvolvimento de Software desenvolviam código das aplicações individualmente até chegar a hora de integrar num produto. Na fase de integração, as mudanças no código de dezenas ou centenas de programadores seriam fundidas em uma base de código comum o que na maioria das vezes originava conflitos e erros durante a compilação.

Felizmente, aprendeu-se que, integrar e testar com mais frequência evita-se o impacto na base de código comum. Na década de 90 a compilação diária do código tornou-se uma prática normal e no início dos anos 2000 este princípio levou-se ao extremo: a integração contínua e a validação destas integrações com uma construção rápida e casos de teste.

Exemplo cenário:

1. O programador A e B descarregaram uma cópia do código comum na mesma altura.
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 forem bem-sucedidos, submetem as alterações para o repositório de base de código comum.

Neste exemplo demonstra-se como as alterações feitas na classe C1 não são consideradas nos desenvolvimentos do programador A, até que alguém testar todas as alterações feitas até um determinado ponto e detetar o erro.

Às vezes é 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.

Neste contexto surge a necessidade de existir um sistema, com uma simples interface de configuração disponível na Internet, que disponibilizará recursos necessários de forma rapidamente e simples, sempre prontos em realizar testes de forma automática a medida que este é alterado no repositório *Git*, desta forma poupando tempo do programador na execução dos testes sobre o código comum, na obtenção de recursos e configuração dos mesmos.

## 1.2 Objetivos

Com este projeto pretende-se conceber e implementar um sistema aplicacional *Web* que ofereça ambientes *Linux* sem interface gráfica, facilmente configuráveis para construção (*build*) e execução automática de testes.

Com uma simples e fácil configuração no sistema, o utilizador terá sempre recursos necessários prontos para realização dos seus testes de forma automática pois o sistema fará isto por ele cada vez que detetar alterações no código do repositório.

Assim sendo, o utilizador deve ter possibilidade de criar um ou mais projetos, especificando as definições do ambiente necessário para execução do código, como por exemplo necessidade de ter uma máquina Linux com suporte para PHP, e especificar o repositório *Git* que contém o código.

O sistema deve usar *Git Hooks* para ser notificado sempre que haverá alterações no código do repositório *Git* e cada vez que notificado, automaticamente descarrega-o para ser compilado e testado.

O sistema deve disponibilizar o resultado da execução e os *logs* com informaçãotanto no portal do sistema como por correio eletrónico, permitindo desta maneira o mais rapidamente possível notificar acerca dos supostos erros nas alterações recentes do código.

Como praticamente toda aplicação *Web*, o sistema deve possuir procedimentos de registo e autenticação local e como opção via contas existentes *GitHub* dos utilizadores. Também como já referido deve possuir funcionalidade para criação/configuração/eliminação dos projetos, execuções de testes, visualização da execução, recolha dos resultados e visualização do histórico das execuções.

O sistema deve permitir ao utilizador criar vários projetos configurados para deferentes repositórios, dando-lhe desta maneira a possibilidade de testar código isoladamente e se for necessário em ambientes com propriedades diferentes, como por exemplo ambientes com versões Java diferentes.

## 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. Capitulo 3 - Solução e Arquitetura: apresenta a solução proposta e arquitetura geral do sistema, descrevendo os seus componentes e a interação entre os mesmos.
4. Capítulo 3 – Implementação: descreve a implementação dos componentes que fazem parte da solução.

1. 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 Plataforma de programação

Rápido, assente em *JavaScript*, totalmente assíncrono e com API amigável! São três razões que tornam, na nossa opinião, o *NodeJs* vantajoso relativamente a outras plataformas de programação *Web* de código aberto.

*NodeJs* é uma plataformaassente 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.

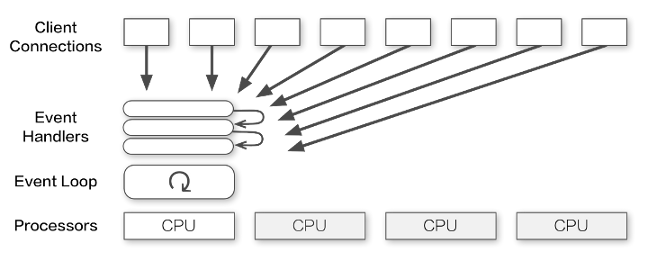


Figura – NodeJs com um único fio de execução

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

*[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.*

## 2.2 Sistema multi-core

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* que automaticamente trata do balanceamento de conceções 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.

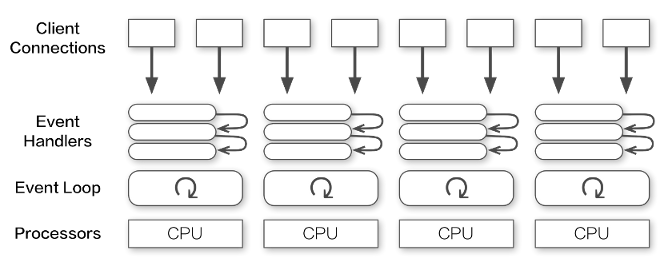


Figura - Paralelismo NodeJs com N Cluster

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*).

## 2.3 Acesso a recursos via 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, em vez de usar os credenciais do proprietário no acesso ao recurso protegido, o cliente (aplicação de terceiros) obtém um *Token* 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.

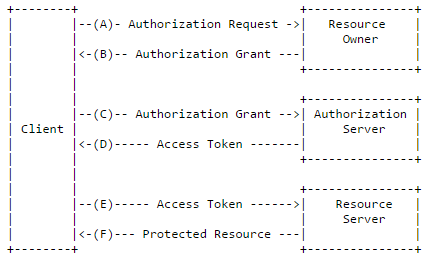


Figura - Fluxo abstrato do protocolo OAuth2.0

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

1. 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.
2. O cliente recebe a concessão (*Grant*) de autorização, que são credenciais que representam a autorização do dono do recurso.
3. O cliente pede o *Token* de acesso através da autenticação no servidor de autorizações apresentando o *Grant* de autorização.
4. O Servidor de autorização autentica o *Grant* e se for valido concede o *Token* de acesso.
5. O cliente pede o recurso protegido ao servidor de recursos e autentifica-se apresentando o *Token* de acesso.
6. O servidor de recursos valida o *Token* de acesso e se valido fornece então o recurso protegido.

## 2.4 LXC – Contentores Linux

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

Uma vez que nosso sistema deve oferecer serviço rápido de criação de ambientes virtuais isolados para execução, seria muito dispendioso em termos de recursos e tempo instanciar máquinas virtuais integra para oferecer ambiente isolado de execução.

A solução mais viável é o recurso LXC 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*) e *Kernel Namespaces*.

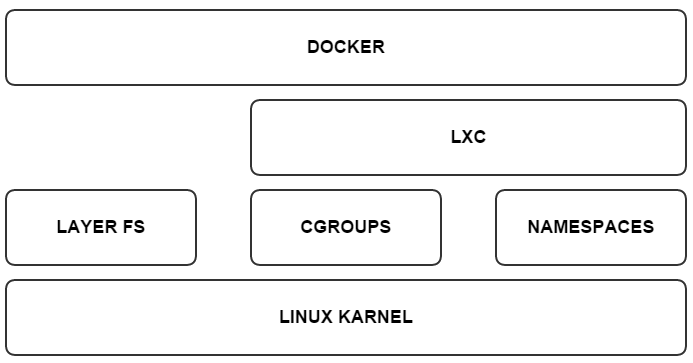


Figura - Linux Container

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

Conceitualmente, 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*.

# 3 Solução e Arquitetura

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

## 3.1 Solução proposta

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.

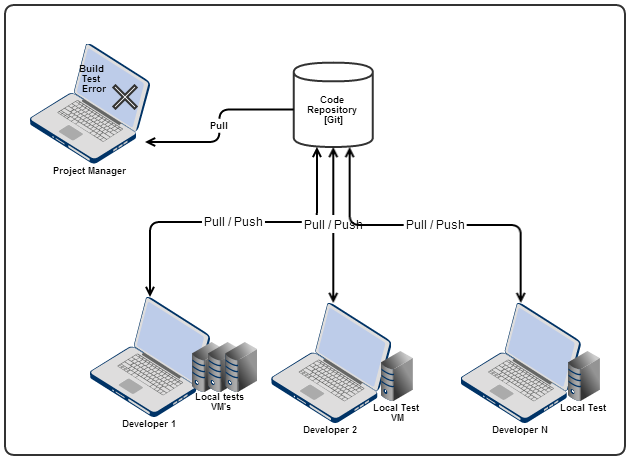


Figura - Modelo antes da solução.

Nestas situações, e dependendo da grandeza do produto desenvolvido, normalmente existem outros intervenientes que executam periodicamente testes do produto na íntegra, reportando erros ou conflitos no código, por exemplo como erros na resolução das dependências entre entidades das aplicações.

Como já foi referido no capítulo “*1.1 Enquadramento e Motivação*”, a solução vem diminuir a intervenção dos programadores das equipas na realização dos testes do código de um produto na sua íntegra ou parcialmente, oferecendo ambientes de execução e automatismo na execução dos testes.

A solução então 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, efetua uma cópia do mesmo juntamente com os testes a realizar para um ambiente isolado de execução pré-configurado, executa a construção do código e os testes, reportando de seguida os resultados e os *LOGs* da execução.

Assim o sistema oferece uma rápida deteção e notificação de supostos erros sempre que ocorra uma submissão de alteração no repositório do código produto.

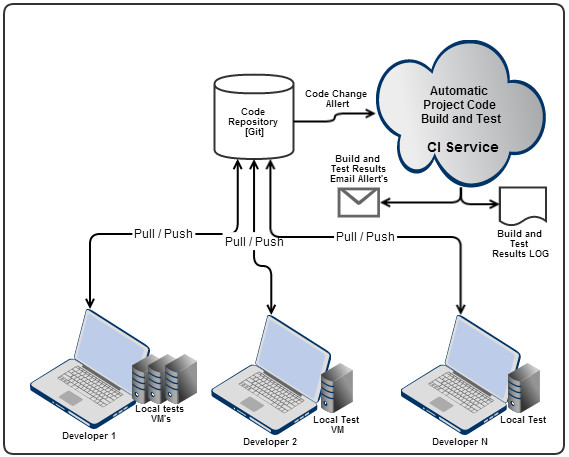


Figura - Modelo depois da solução

Como exeplificado na ilustração acima, o sistema em tempo real é automaticamente notificado quando ocorre o *GIT PUSH* no repositório do produto.

Git tem uma forma de disparo de eventos quando ocorre algo importante no repositório. Para haver possibilidade de subscrição para estes eventos, o *Git* utiliza *Web Hooks*, que são *callback* *HTTP*: *HTTP POST* que ocorre quando algo acontece.

O sistema utiliza o “*post-receive hook*” que notifica após de occore um commit publicado no repositório.

## 3.2 Ambiente e Ferramentas

Este capítulo tem como objetivo apresentar as tecnologias e ferramentas utilizadas e aprovisionamento das mesmas.

### 3.2.1 Ferramentas e Tecnologia

Todas as ferramentas descritas abaixo são de caráter de código aberto (*Open Source*).

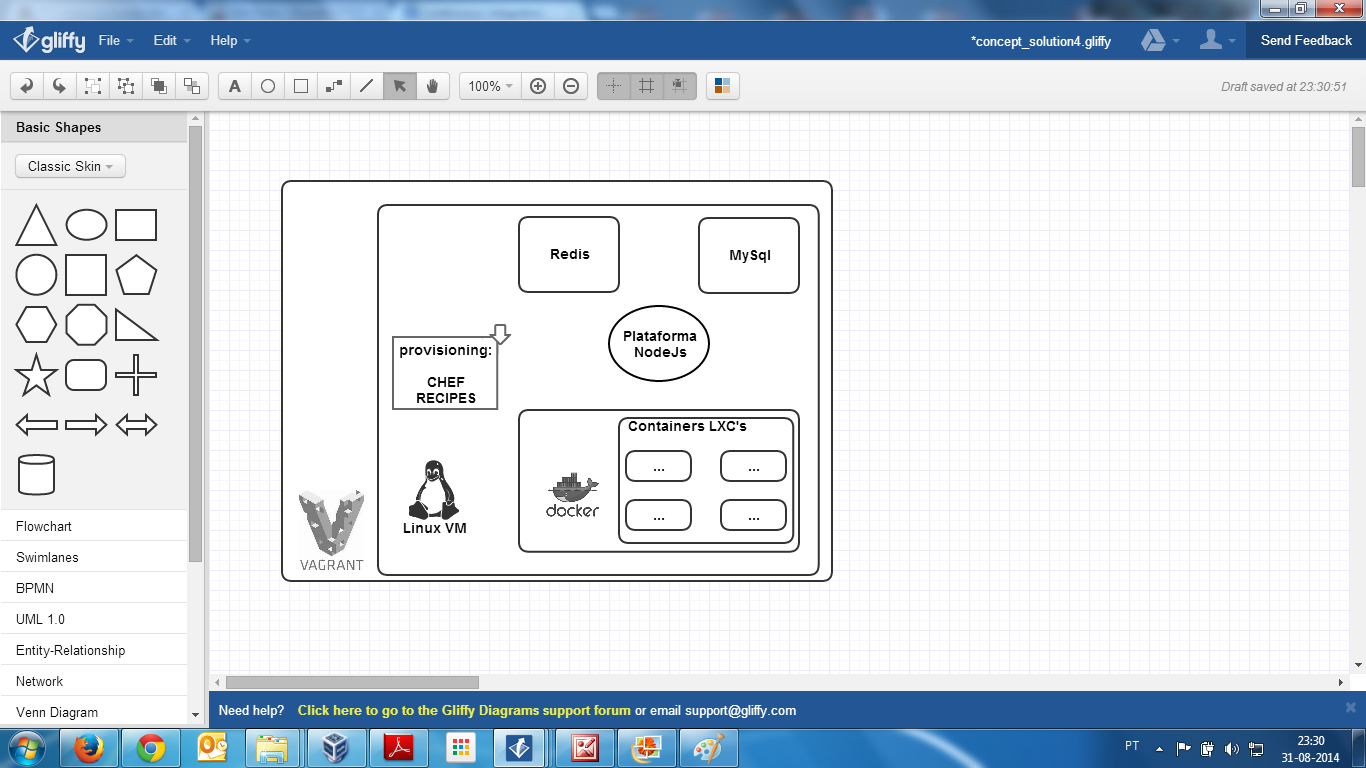


Figura – Ferramentas e Tecnologias

**VirtualBox**

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

**Vagrant**

Para garantir que os ambientes de desenvolvimento sejam idênticos nos membros da equipa, utilizou-se a ferramenta *Vagrant* 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, Docker, MySql, Redis* dentro da máquina virtual, optou-se em utilizar a ferramenta *Chef* que o *Vagrant* 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*, *Google Cloud*, *Microsoft Azure* entre outras, para provisionar automaticamente e configurar novas máquinas.

**Berkshelf**

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

**Docker**

Para criar e gerir ambientes isolados para execução de aplicações, optou-se pela ferramenta *Docker*. 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 do sistema operativo Linux chamado *LXC - Linux Containers* que são uma espécie de contentores (ambientes virtuais) que possuem próprio CPU, memória, I/O, rede, espaço etc. fornecidos pelo *Karnel* do *SO* *Linux* hospedeiro.

**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 para este projeto é Redis possuir funcionalidades de Publicação/Subscrição em canais de troca de dados “messaging*”* onde todo o interessado pode publicar mensagens e todo o interessado pode ler.

**MySql**

É uma das mais populares aplicações de código aberto (*Open Source*) de base de dados relacionais.

**Mocha**

Tal como *JUnit* para *Java* e *NUnit* para *Microsoft .NET*, *Mocha* em plataformas *NodeJS* é uma *framework* de teste para código *JavaScript*.

### 3.2.2 Montagem do ambiente, Partilha e Aprovisionamento

A parte significante 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.

#### 3.2.2.1 Ambiente

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”*.

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.

Via o comando “vagrant provision” é lançado o processo de aprovisionamento.

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.

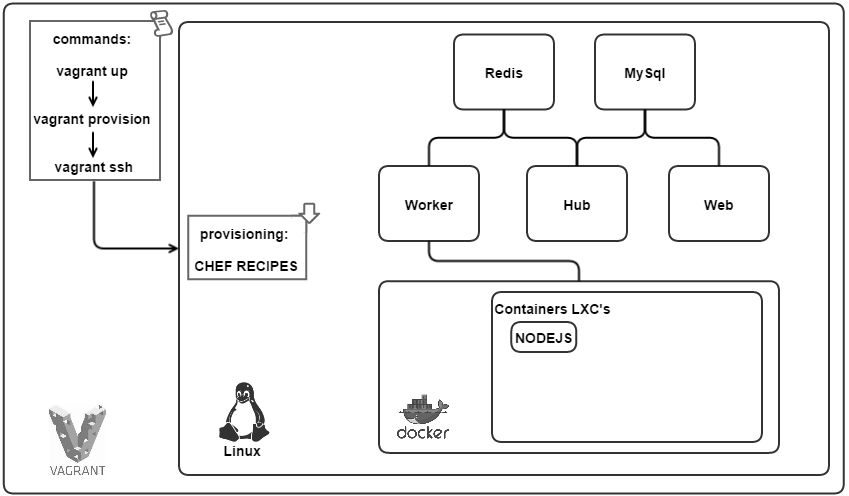


Figura - Ambiente de desenvolvimento

#### 3.2.2.2 Partilha

Como já foi referido 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.

#### 3.2.2.3 Aprovisionamento

Como já foi referido, o *Vagrant* possui 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* foi o sistema de aprovisionamento escolhido. O mesmo 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* 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* que sabe resolver dependências de *cookbooks*.

## 3.3 Arquitetura Geral

O sistema CI desenvolvido possui três componentes desenvolvidos que tornam o conjunto de recursos servirem para o propósito. Os componentes são *Web Server*, *Worker Service* e *Hub Service*. Estes componentes são aplicações autónomas que não dependem uma das outras mas em conjunto servem para o propósito do sistema.

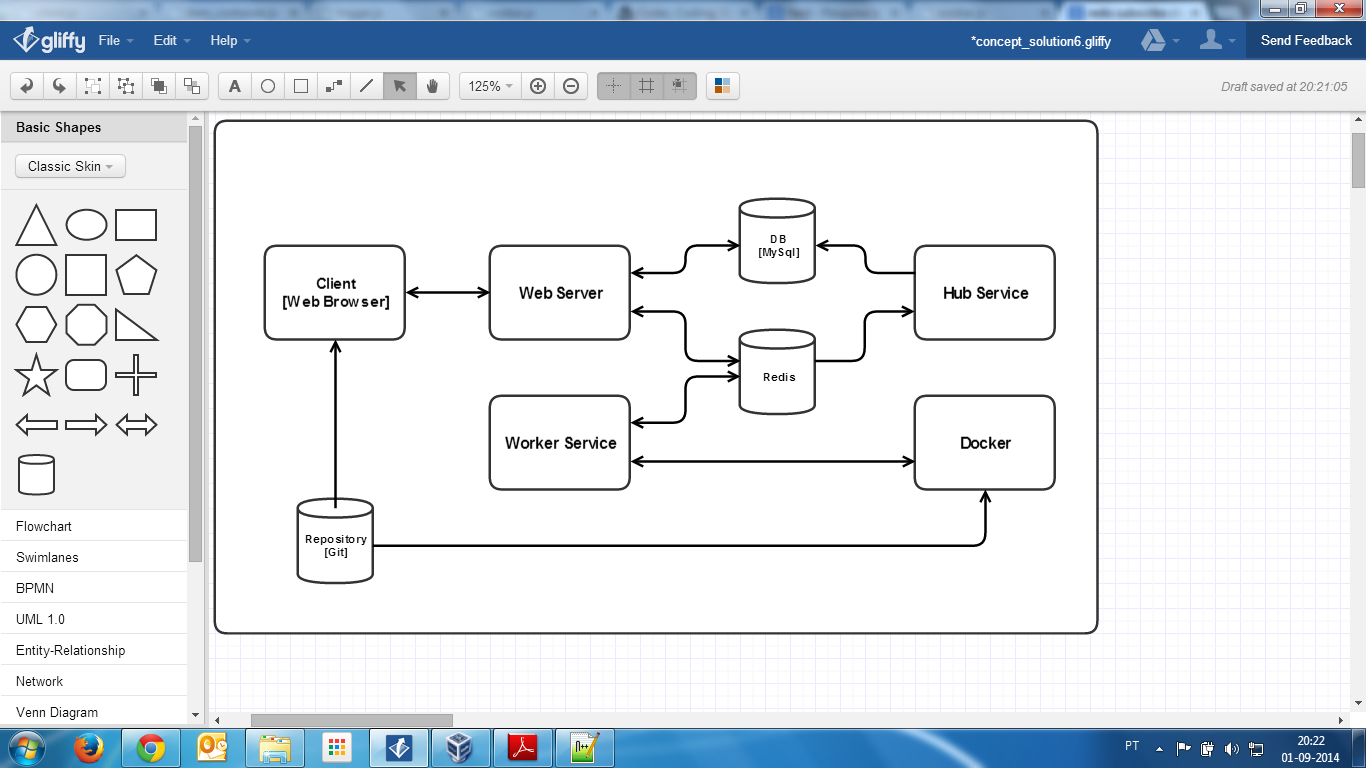


Figura – Modelo de relações entre componentes via Redis e Bases de Dados

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 do *Redis*.

*Redis* como já referido no capitulo *“2.1 Ferramentas e Tecnologia”*, para além de ser um sistema de armazenamento de dados em pares chave-valor, possui também funcionalidades de Publicação/Subscrição em canais de troca de dados “*messaging”* onde todo o interessado pode publicar mensagens e todo o interessado pode ler. Assim sendo, a informação do trabalho agendado pelo utilizador e os resultados dos mesmos são propagados nos canais do Redis e cada um dos componentes extrai o necessário.

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

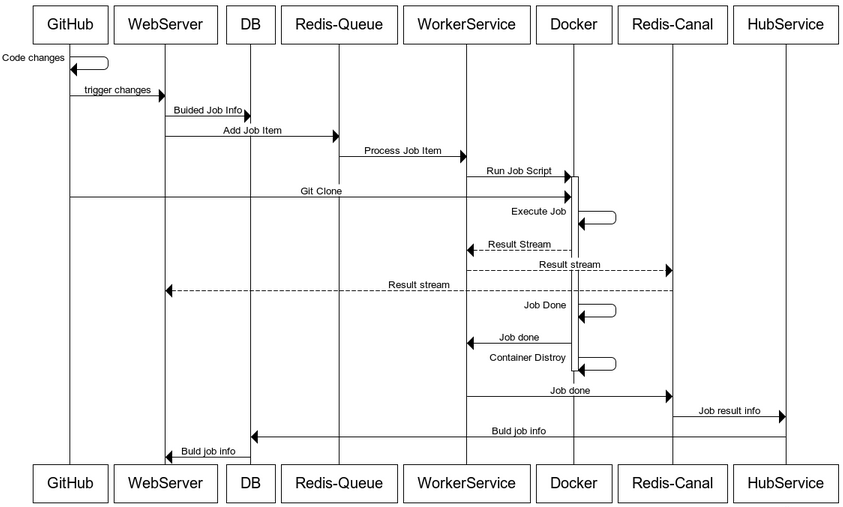


Figura – Diagrama do fluxo do trabalho entre componentes

Os trabalhos são processados por ordem de chegada e são executados de forma assíncrona.

Para fazer frente a grandes volumes de trabalho e fazer aproveitamento de sistemas *multi-core*, faz-se proveito do módulo *Cluster* do *NodeJS* .

### 3.3.1 Componente Web

No desenvolvimento da aplicação servidora *Web* utilizou-se a *framework* *Express* para *NodeJs*.

*Express* 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.

O fluxo de interações do utilizador com a plataforma consiste em criar projetos, associa-los aos projetos no seu repositório *GitHub* e lançar o *Build* automático dos mesmos.

Para tal o utilizador terá que se registar na plataforma que tem o fluxo esquematizado na figura a seguir:

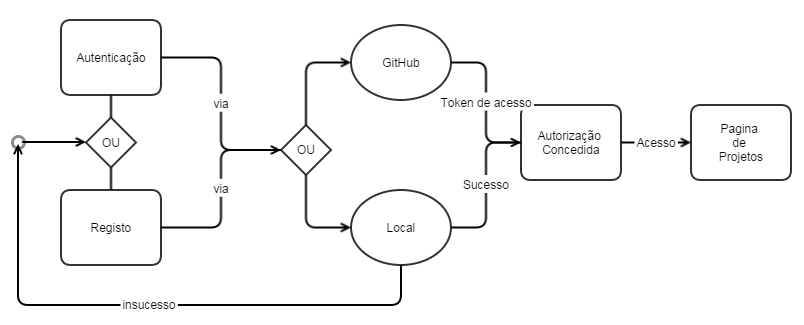


Figura - Fluxo Autorização e Registo

Ao utilizador disponibiliza-se duas formas de registo, registo local e via autenticação na aplicação *GitHub*.

O registo local consiste em fornecer um correio eletrónico valido e credenciais de acesso, nome do utilizador e palavra-chave. A palavra-chave é encriptada utilizando o *Bcrypt*.

*Bcrypt* é um método de criptografia do tipo *hash* para senhas, baseado no *Blowfish* que é uma cifra simétrica de blocos.

Quando o utilizador opta em registar-se na plataforma via associação com uma conta existente na aplicação *GitHub,* a aplicação Servidora Web redireciona-o para página de login *GitHub,* onde o utilizador se autentica e de seguida é-lhe apresentado o pedido de autorização de acesos para o sistema aplicacional *CI*. Neste processo é utilizada a API *GitHub OAuth2.0*, que através da qual são obtidas as informações publicas sobre utilizador, o *Token* de autorização no acesso aos recursos tais como os seus repositórios. Ao autorizar o acesso a aplicação *CI*, o utilizador é redirecionado para página de projetos da plataforma Web do nosso sistema.

A plataforma Web tem uma estrutura de páginas que permitem ao utilizador criar projetos, editar, executar projetos tendo possibilidade de seguir a evolução da execução visualizando os *logs* fornecidos em tempo de execução, visualizar o histórico de execuções (*Builds*) de projetos e eliminar projetos.

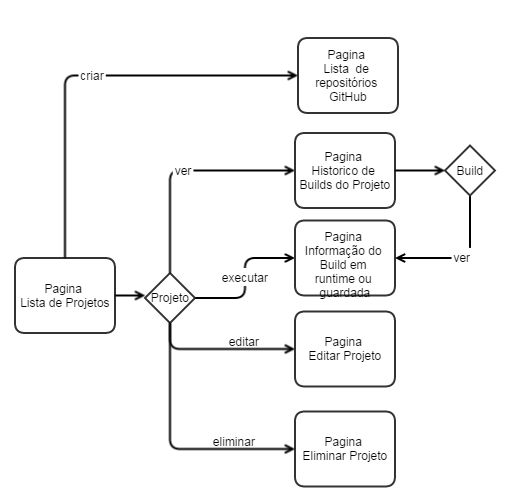


Figura - Usabilidade da plataforma Web

Estrutura das páginas Web da plataforma:

**Pagina “Sign In”:**

* Formulário para autenticação local
* Botão “SignIn with GitHub” para autenticação via *GitHub*

**Pagina “Sign Up”:**

* Formulário para registo local
* Botão “SignUp with GitHub” para registo via *GitHub*

**Pagina “Projects”:**

* Lista de projetos

Cada projeto listado tem informação visível como o Nome, Nome do contentor Docker associado, “*HTTPS clone URL”* através do qual foi clonado, como também opções para ser editado, visualização do histórico de execuções (*Builds*) e opção de eliminação do próprio.

* Butão “*New project*”

Link para a pagina com lista de repositórios do *GitHub*.

* Butão “*Log Out*”

Termina a sessão do utilizador.

**Pagina “History”:**

* Lista de execuções (*Builds*), cada *Build* com informação “*Git Brunch*”, “*Commited By*” e massagem.
* Botão “*View Log*” para cada *Build*.
* Botão “*Build*” como possibilidade de repetir a execução do projeto.

**Pagina “Select Repository”:**

**Pagina “Edit”:**

**Pagina “Delete”:**

### 3.3.2 Componente Worker

Para garantir rapida construção de ambientes de execução necessarios aos utilizadores, recorese aos contentores LXC ofericidos pelo sistema operativo Linux.

Por oferecer uma *API* amigavel e por ser o gestor de contentores LXC mais adequado para este efeito, foi escolhido o *Docker*.

Contentores Docker possuem tudo o que é necessario para um ambiente de execução, o minimu de computação necessario (cpu/io/network), uma camada *filesistem* para guardar espaço, usa a estrategia *copy-on-write filesystem* para monitorizar alterações nos dados do utilizador. Contentores Docker também são autosuficientes, posuem o minimu base do sistema operativo, bibliotecas e *frameworks.*

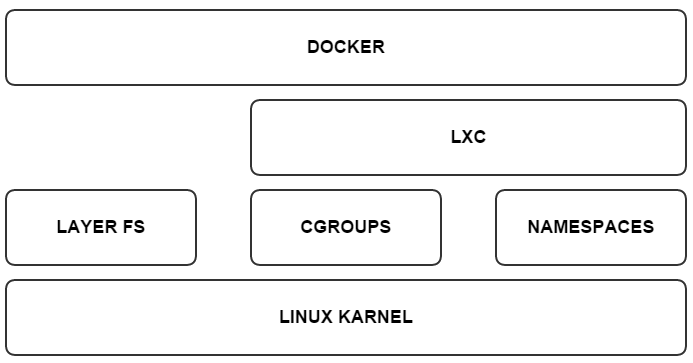


Figura - Camada Docker

### 3.3.3 Componente Hub

# 4 Implementação

Neste capítulo descrevem-se os componentes implementados que fazem parte da solução.

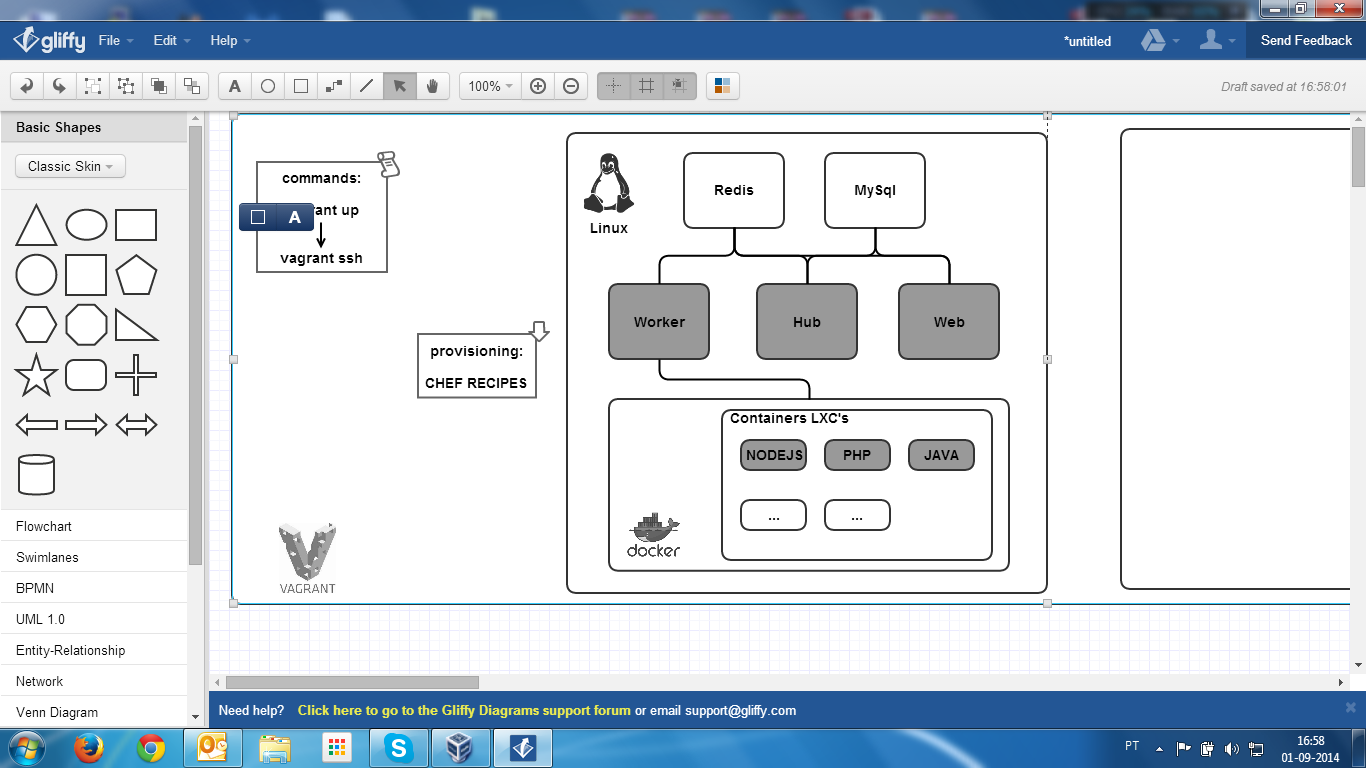


Figura - Aplicações desenvolvidas

## 4.1 Aplicação Worker

## 4.2 Aplicação Hub

## 4.3 Aplicação Web

# 5 Conclusão

Na área de programação e nomeadamente no contexto de integração continua, 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 aceso 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 resultas.
* 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

<http://pt.wikipedia.org/wiki/V8_(JavaScript)>

<http://www.haneycodes.net/to-node-js-or-not-to-node-js/>

<http://webapplog.com/php-vs-node-js/>

<http://en.wikipedia.org/wiki/Inter-process_communication>

<https://devcenter.heroku.com/articles/node-cluster>

<http://oauth.net/2/>

<http://tools.ietf.org/html/rfc6749>

<http://www.linux-kvm.org/page/Main_Page>

<http://en.wikipedia.org/wiki/Xen>

<http://en.wikipedia.org/wiki/Continuous_integration>

<http://en.wikipedia.org/wiki/Vagrant_(software)>

<https://docs.vagrantup.com/v2/>

<https://wiki.opscode.com/display/chef/Home>

<http://berkshelf.com/>

<https://docs.getchef.com/essentials_cookbooks.html>

<https://linuxcontainers.org/>

<https://www.docker.com/whatisdocker/>

<https://www.suse.com/documentation/sles11/singlehtml/lxc_quickstart/lxc_quickstart.html>

<http://nodejs.org/documentation/>

<http://visionmedia.github.io/mocha/>

<http://en.wikipedia.org/wiki/Redis>

<http://redis.io/topics/introduction>

<http://redis.io/documentation>

<http://pt.wikipedia.org/wiki/MySQL>

<http://slides.com/stevenborrelli/docker/fullscreen#/>

<http://lwn.net/Articles/570558/>

<http://expressjs.com/>

<http://en.wikipedia.org/wiki/Blowfish_(cipher)>

<http://en.wikipedia.org/wiki/Bcrypt>