Universidade de Aveiro
Departamento de Eletrónica, Telecomunicações e Informática
2015

João André Santos César

# PRESTO: sistema para provisão automática de ambientes de testes de aceitação

# PRESTO: a system for automatic provisioning of acceptance testing environments

**Universidade de Aveiro**
**2015**

Departamento de Eletrónica,
Telecomunicações e Informática

**João André Santos César**

# PRESTO: sistema para provisão automática de ambientes de testes de aceitação

# PRESTO: a system for automatic provisioning of acceptance testing environments

Dissertação apresentada à Universidade de Aveiro para cumprimento dos requisitos necessários à obtenção do grau de Mestre em Engenharia de Computadores e Telemática, realizada sob a orientação científica do Doutor Ilídio Castro Oliveira, Professor auxiliar do Departamento de Eletrónica, Telecomunicações e Informática da Universidade de Aveiro.

Dedico este trabalho aos meus pais e irmãs.

**o júri / the jury**

presidente / president

Professor Doutor Luís Filipe de Seabra Lopes
Professor Associado, Universidade de Aveiro

vogais / examiners committee

Mestre Ricardo Azevedo Guerra Raposo Pereira
Especialista, Critical Software

Professor Doutor Ilídio Fernando de Castro Oliveira
Professor Auxiliar, Universidade de Aveiro

**agradecimentos** /
**acknowledgements**

Primeiramente gostaria de agradecer à minha família, em especial ao meu pai e à minha mãe que sempre acreditaram em mim, mesmo nos momentos mais difíceis do meu percurso académico. Sem eles este momento não seria possível.

Um agradecimento especial ao meu orientador, Professor Doutor Ilídio Oliveira, o seu auxílio foi crucial para que fosse possível concluir este trabalho, muito obrigado por toda a sua disponibilidade e simpatia.

Um agradecimento igualmente especial ao Mestre Rui Gonçalves, pelos seus conselhos, apoio e ajuda prestadas. A sua experiência e conhecimento foram fundamentais no desenvolvimento do Presto.

Um obrigado à Beubi, empresa que ao longo destes últimos meses me acolheu e forneceu o servidor alojado na infraestrutura da Amazon, o que permitiu a colocação do sistema Presto num ambiente de produção. A todos os colegas da Beubi, que desde o início se disponibilizaram para me auxiliar, prestaram ajuda e com que tive o prazer de conviver, o meu muito obrigado.

A todos os meus amigos, com quem convivi diariamente, que me ajudaram e apoiaram durante o meu percurso académico, o meu sincero agradecimento, sem vocês teria sido muito mais difícil encarar e ultrapassar as dificuldades com que nos deparamos todos os dias.

**Palavras Chave**

Entrega Contínua, Sistemas de virtualização, Sistemas de provisão automática, Sistemas de orquestração, Sistemas de versionamento de código, Ambiente de testes, Testes de aceitação

**Resumo**

As abordagens ágeis na engenharia de *software* valorizam o envolvimento do cliente, através da entrega frequente de valor e a sua participação na aceitação dos incrementos.

A validação por parte do cliente inclui a realização de testes manuais sobre novas funcionalidades do produto. Operacionalmente, requer a criação de um ambiente de testes dedicado para o efeito, atualizado sempre que há um incremento a apresentar ao cliente. A criação e a configuração de um ambiente assim implica geralmente a dedicação de uma pessoa para realizar o processo manualmente, o que não é eficiente, nem escalável e é permeável a erros.

No âmbito desta dissertação, propomos uma plataforma *web* para automatizar e agilizar o processo de criação do ambiente a usar nos testes de aceitação pelo cliente. A solução é transparente para a equipa de desenvolvimento e não depende de ações adicionais do programador.

O sistema desenvolvido observa o repositório partilhado de gestão de código e é notificado da aceitação de pedidos de integração de incrementos na solução (*pull requests*). Em função disso, e utilizando especificações de instalação que são incluídas juntamente com o projeto de código (e por isso elas mesmas sujeitas a controlo de versões), o sistema desenvolvido configura *containers* virtuais com o ambiente necessário e faz a instalação de dependências e da solução. O sistema reconhece a existência de incrementos baseando-se na abordagem GitFlow. O gestor de projeto pode, a qualquer altura, pedir a instanciação do ambiente de teste e indicar ao cliente um endereço para acesso de modo a realizar os testes de aceitação.

A utilização de tecnologia de virtualização baseada em *containers*, e especialmente o Docker, permitiu criar um sistema de provisão de recursos muito eficiente. A solução, implementada e utilizada em contexto de empresa, mostrou ser capaz de substituir a configuração manual, repetitiva e demorada, por um processo automático, sem disrupção das práticas existentes.

**Keywords**

Continuous Delivery, Virtualization systems, Provisioning systems, Orchestration systems, Version control system, Testing Environments, User Acceptance Tests

**Abstract**

Agile approaches to software engineering value customers engagement with frequent delivery of value and their participation in the acceptance of increments.

Customer validation includes performing manual testing on new product features. Operationally, this requires setting up a test environment dedicated for this purpose, updated whenever there is an increment to present to the client. The creation and setting up of such environment usually involves the dedication of a person to perform the process manually, which is not efficient, nor scalable, and it is error-prone.

In this work, we propose a web platform to automate and streamline the preparation of the environment that will be used by the client for acceptance testing. The solution is seamless to the development team and does not depend on additional actions from the developers.

The developed system observes the shared code repository and is notified of the acceptance of application increments integration in the solution (pull requests). Upon this trigger, it uses the deployment specifications included in the code base (and also under version control) to configure virtual containers with the required enviroment, and to install dependencies and the solution. The system recognizes the existence of increments based on the GitFlow approach. The Project Manager may at any time instantiate the test environment and give the customer the web address to perform the acceptance testing.

The use of virtualization technology based on "containers" and especially the Docker, enabled a very efficient resource provisioning system. The solution, implemented and used in the context of a company, proved to be able to replace the manual and repetitive configuration, by an automatic process without disruption of the existing practices.

# Conteúdos

# Lista de Figuras

# Lista de Tabelas

# Glossário

| | |
|---|---|
| **IPC** | *Interprocess comunication* |
| **PID** | *Process identifier* |
| **UTS** | *UNIX Timesharing System* |
| **LXC** | *Linux Containers* |
| **Union FS** | *Union File System* |
| **YAML** | *YAML Ain't Markup Language* |
| **SVN** | Subversion |
| **UAT** | *User Acceptence Tests* |
| **MVC** | *Model View Controller* |
| **SSH** | *Secure Shell* |
| **IEETA** | Instituto de Engenharia Electrónica e Telemática de Aveiro |
| **API** | *Application Programming Interface* |
| **SWE** | *Software With Emotion* |
| **TCP** | *Transmission Control Protocol* |
| **JSON** | *JavaScript Object Notation* |
| **ERB** | *Embedded RuBy* |
| **ORM** | *Object Relation Mapping* |

CAPÍTULO 1

# INTRODUÇÃO

## 1.1 MOTIVAÇÃO

Hoje em dia com a proliferação dos sistemas de informação e com o crescente número de empresas que se dedicam ao desenvolvimento dos mesmos, surge a necessidade destas se manterem competitivas e se diferenciarem em relação às restantes.

Um fator de competitividade é o desenvolvimento eficiente de *software*, cumprindo prazos e desenvolvendo o produto correto. Para que isto aconteça é essencial a colaboração entre toda a equipa e a automatização de processos repetitivos. Ao garantirmos a qualidade do *software* desenvolvido, estamos também a promover a eficiência da equipa de desenvolvimento da nossa empresa, pois a probabilidade de no futuro sermos forçados a corrigir erros é menor.

A diferenciação também é obtida através da satisfação do cliente, oferecendo um produto final que satisfaça todos os requisitos definidos e que seja pouco permeável a erros.

Uma das formas de garantir a satisfação do cliente é através da validação das funcionalidades desenvolvidas por meio de testes de aceitação, de forma a garantir que as mesmas vão de encontro às necessidades deste e acrescentam valor ao produto.

No atual estado da técnica, ainda é relativamente moroso o processo de preparação do ambiente de testes de aceitação para ser utilizado pelo cliente final, envolvendo processos manuais. Por essa razão não é muitas vezes prático envolver o cliente no teste de funcionalidades isoladas, apesar disso ser vantajoso para uma abordagem ágil ao desenvolvimento de software.

Esta dissertação foi proposta pela empresa Beubi que pretende automatizar o processo de provisão dos ambientes de testes de aceitação e criar uma plataforma na qual os ambientes possam ser controlados pelo gestor do projeto. Desta forma é pretendido que o tempo de espera de preparação do ambiente seja reduzido e abstrair os detalhes subjacentes à sua criação.

## 1.2 OBJETIVOS

O objetivo desta dissertação é criar um sistema para automatizar o processo de criação do ambiente de testes, no qual o cliente ou gestor do projeto possam realizar os testes de aceitação, e que possa ser integrado no atual modelo de desenvolvimento da empresa Beubi. Com o sistema Presto proposto, deixa de ser necessário que uma pessoa seja responsável por esta tarefa, ficando liberta para realizar outras tarefas que tragam valor à empresa.

O nosso estudo será focado nos sistemas de virtualização, provisão e orquestração, que permitirão a criação destes ambientes e a automatização do processo. A solução terá que ser capaz de reconhecer as diferentes versões do projeto através do sistema de versionamento de código e despoletar as ações de criação e provisão (de uma forma manual ou automatizada) do ambiente de testes, para que quando for efetivamente pretendido realizar os testes manuais a execução do ambiente seja expedita.

Apesar de se partir do contexto real e necessidades da Beubi, empresa que acolheu esta dissertação, pretende-se chegar a uma solução genérica, que possa ser usada noutras equipas de desenvolvimento.

## 1.3 CONTRIBUIÇÃO

O principal resultado desta dissertação é um sistema computacional designado Presto que, em italiano, tem o significado ”em breve”, ”rápido”. Na música clássica, o termo Presto é usado para marcar andamentos que devem ser interpretados de forma rápida e viva, que é exatamente o objetivo da ferramenta desenvolvida, agilizar o processo de criação dos ambientes de teste. Uma orquestra é composta por diversos intérpretes e é a sua articulação que faz um bom concerto. No contexto deste trabalho, a orquestra será a equipa de desenvolvimento, e o sistema Presto, uma contribuição para o ”andamento” fluído na criação do ambiente de testes.

O sistema Presto deverá ser capaz de identificar as diferentes versões do projeto e preparar ambientes de teste ajustados, o que é possível pois o repositório de código segue algumas regras que permitem a identificação dos incrementos. A partir de uma lista (de versões) será possível gerir os diferentes ambientes de *staging* e desencadear o processo de provisão manualmente ou automaticamente através da criação de um *pull request* no sistema de versionamento remoto.

A criação do ambiente de testes é automatizado sempre que surge um evento de *pull request* no sistema de versionamento de código remoto por parte da equipa de desenvolvimento e o mesmo represente um incremento importante no desenvolvimento do produto (uma nova funcionalidade a ser testada), é criado automaticamente o ambiente que permite a validação por parte do cliente, através de testes aceitação, das funcionalidades desenvolvidas.

O Presto elimina o problema de termos que criar o ambiente manualmente, o que é um processo demorado, necessita de recursos humanos especializadas e é bastante sujeito a erros.

Através do sistema Presto conseguimos abstrair todos os detalhes subjacentes à configuração do ambiente de testes. Qualquer utilizador pode iniciar um ambiente para executar testes de aceitação sobre novas funcionalidades, sem que para isso precise de dominar as tecnologias de virtualização e orquestração subjacentes.

Para além disso foram utilizados múltiplas ferramentas de provisão, orquestração e virtualização, o que faz com que este sistema seja compatível com projetos com diferentes tipos de requisitos.

## 1.4 ESTRUTURA

Esta dissertação está dividida em sete capítulos:

- Capítulo 1: Descrição do problema que nos propomos a resolver e objetivos que pretendem ser alcançados com esta dissertação.
- Capítulo 2: Serão estudadas práticas relacionadas com a garantia de qualidade de *software* que permitirão perceber a importância das mesmas e qual a etapa em que o sistema a ser desenvolvido se insere. Para além disso serão abordadas diversas ferramentas de virtualização, provisão, orquestração e versionamento de código.
- Capítulo 3: Apresentação dos requisitos e os casos de uso do sistema a ser desenvolvido.
- Capítulo 4: Apresentação da organização da solução e as ferramentas que foram utilizadas. São ainda descritos os diversos componentes, qual a sua função e como é que eles interagem entre si.
- Capítulo 5: Explicação dos detalhes da implementação, mostra todos os constituintes do sistema e como estes interagem entre si nas principais operações do sistema. No final desta secção é apresentado o *workflow* da plataforma que foi desenvolvida e as suas funcionalidades.
- Capítulo 6: Apresentação dos projetos que foram utilizados para validar a utilização da plataforma, comparação entre a criação manual e automatizada pelo nosso sistema e realização de alguns testes comparativos entre as diferentes ferramentas de virtualização, provisão e orquestração.
- Capítulo 7: Apresentação dos benefícios da utilização da plataforma, algumas conclusões sobre os diferentes sistemas utilizados e possíveis melhoramentos.

capítulo 2

# Revisão do estado da arte

## 2.1 práticas de garantia de qualidade do software

A garantia de qualidade do *software* (Software Quality Assurance) é conseguida através da utilização de diversas ferramentas e práticas e tenta assegurar que o produto desenvolvido está conforme os requisitos definidos.

Durante o processo de desenvolvimento e de forma a promover a qualidade do produto, é essencial que sejam efetuados testes sobre o código desenvolvido para detetar erros [1], de forma automática ou manual. Os primeiros permitem tipicamente verificar o código e são realizados geralmente por um servidor dedicado para este efeito. Já os segundos são tipicamente utilizados para aceitação e garantem que o sistema desenvolvido vai de encontro às expectativas do cliente.

A revisão do código é outra prática importante e existem várias formas de ser realizada. Uma das abordagens é através da utilização de uma ferramenta de análise estática de código, que inspeciona o código através de um conjunto de diretivas definidas pela equipa e/ou específicas da linguagem. Outra é através da análise do código por outros programadores da equipa, realizada depois dos testes automáticos e permite detetar potenciais problemas de implementação.

O desenvolvimento eficiente de um produto requer a utilização de ferramentas que automatizem o processo de instalação, que inclui a configuração do ambiente onde a aplicação será executada. A validação e verificação do código desenvolvido poderá passar por várias fases e cada uma destas fases pode necessitar, ou não, de um ambiente específico onde a aplicação será executada, que deve ser preparado de forma automática, garantindo que tudo é configurado de uma forma correta e que demora o menor tempo possível.

As práticas acima introduzidas não esgotam o leque de práticas relacionadas com a Garantia de Qualidade do *Software*, mas serão aquelas mais relevantes para o nosso trabalho, que desenvolveremos a seguir.

### 2.1.1 GESTÃO DE VERSÕES DO CÓDIGO COM GIT

O Git é um sistema de versionamento de código que permite guardar um histórico das alterações que foram feitas sobre os ficheiros da nossa solução ao longo do desenvolvimento [2], ao contrário de outros sistemas de versionamento (como, por exemplo, o SVN), permite guardar as alterações local e remotamente, e é por isso designada como uma solução distribuída [3].

O sistema de versionamento Git funciona por ramos (*branchs*); existe um ramo principal (normalmente designado de *master*) e existem ramos secundários que são criados a partir do ramo principal ou de outro ramo secundário [2].

Os principais comandos do Git são os seguintes:

- *Add* - adiciona um ou mais ficheiros que atualmente não estejam versionados no repositório local.
- *Delete* - remove um ficheiro, atualmente versionado no repositório local.
- *Commit* - submete as alterações no repositório local.
- *Push* - envia as alterações locais para o repositório de código remoto.
- *Pull* - atualiza o ramo atual a partir do repositório de código remoto. O *pull* é a composição de duas operações, *fetch* e o *merge* [3].
- *Fetch* - atualiza os ramos que estão sincronizados entre o repositório local e o repositório remoto.
- *Reset* - reinicia o estado de um ramo para a versão de um outro ramo.

O GitFlow é um modelo convencionado que melhora a organização do nosso repositório Git através da atribuição de nomes específicos aos ramos. Existem vários tipos de ramos no GitFlow, cada um com uma função específica como pode ser observado pelo diagrama representado na figura 2.1.

- *master* - é o ramo principal do nosso projeto e contém a versão da nossa aplicação que está atualmente em produção [4].
- *develop* - contém a versão da aplicação que será disponibilizada na próxima *release* do produto [4].
- *feature/** - é utilizado pelos programadores para desenvolverem novas funcionalidades do produto. Este ramo é criado a partir do ramo *develop* e quando o seu desenvolvimento é terminado, o código do ramo *feature* é fundido com o ramo *develop* [4].
- *release/** - é utilizado para preparar a próxima *release* do produto, é tal como os ramos *feature* criado a partir do ramo *develop* e quando essa preparação é terminada o código é fundido novamente no ramo *develop* [4].
- *hotfix/** - é utilizado para corrigir possíveis erros que apareçam em produção, este ramo é criado a partir do ramo principal(*master*) e o código desenvolvido é depois fundido nos ramos *develop* e *master* [4].

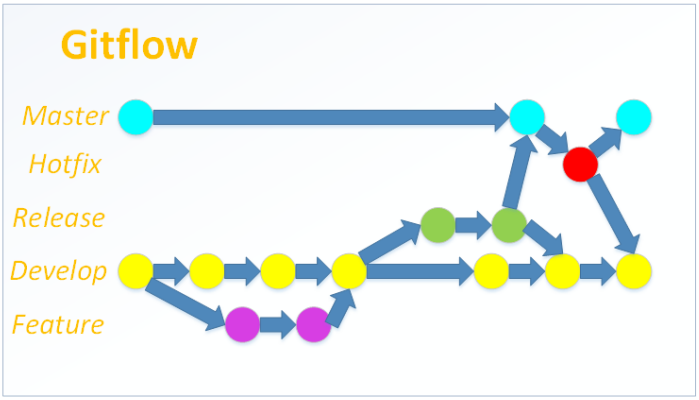



Figura 2.1: Modelo de utilização do Git Flow, imagem adaptada de [5]

Alguns repositórios de código também são capazes de interagir com sistemas externos (como, por exemplo, um servidor Jenkins) o que permite que alguns eventos no repositório de código despoletem ações automáticas; esta facilidade é também conhecida como *hooks*.

Um dos *hooks* que nos interessa especialmente nesta dissertação é o *pull request* POST. O *pull request* é uma ação que nos permite notificar a existência de alterações que foram submetidas ao repositório de código remoto e que necessitam de serem revistas. A eventual aceitação corresponde à integração de um incremento na solução.

### 2.1.2 INTEGRAÇÃO CONTÍNUA

A prática de Integração Contínua surge da tradução do termo *Continuous Integration*, que iremos utilizar preferencialmente na versão inglesa. O ciclo de *Continuous Integration*, representado no diagrama da figura 2.2, permite garantir que todo o código desenvolvido, presente no ramo principal do sistema de versionamento está funcional. Quando é iniciado o desenvolvimento de uma nova versão do produto em que esta representa uma nova funcionalidade ou melhoramento, genericamente são efetuadas as seguintes atividades [6]:

- É criado um novo ramo a partir do ramo principal no sistema de versionamento, que contém a última versão funcional do código. No modelo GitFlow este é identificado pelo prefixo *feature/*, seguido do nome da funcionalidade ou melhoramento.
- O programador ou equipa encarregues dessa tarefa começam o desenvolvimento.
- O código é submetido no sistema de versionamento e desencadeia uma série de mecanismos que permitem verificar se código desenvolvido está ou não funcional.
- O código é submetido a uma série de testes criados pela equipa de qualidade de *software* da empresa, estes testes são automáticos e testam a aplicação cumulativamente, garantindo que o novo código é funcional com o anteriormente desenvolvido.
- Se o código não conseguir passar em algum dos testes o programador ou equipa de desenvolvimento são notificados e corrigem o erro. Este processo é repetido até que o código consiga passar em todos os testes e esteja por isso, num estado funcional.
- Depois de verificado, o código é submetido no ramo principal do sistema de versionamento.

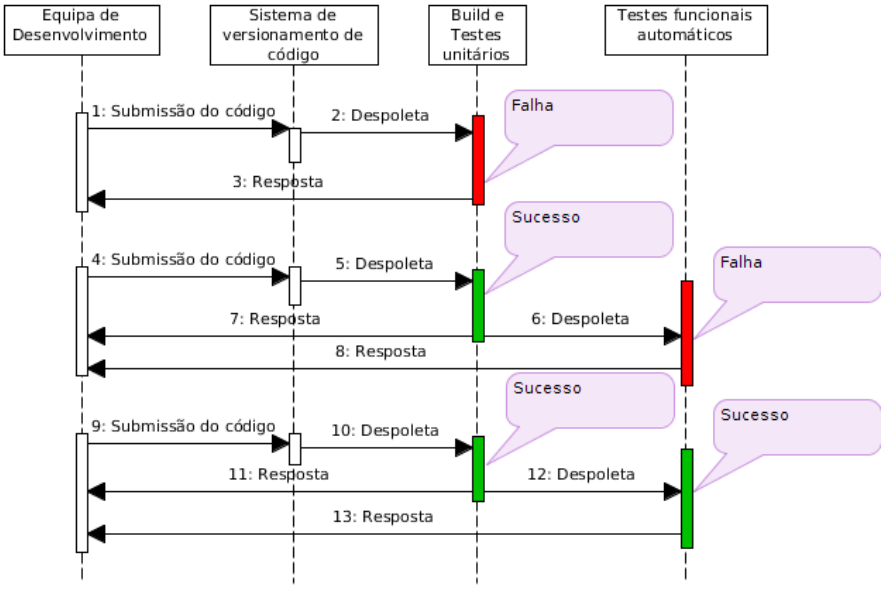



Figura 2.2: Ciclo de vida do processo de *Continuous Integration*, imagem traduzida do livro *Continuous Delivery* [6]

Esta prática permite que os programadores apenas avancem para uma nova tarefa quando o código desenvolvido estiver correto e faz com que a resolução dos problemas seja feita mais eficientemente, pois todas as particularidades da solução desenvolvida estão ainda bem presentes [7]. Como o código do ramo principal é sempre testado e garantimos que está num estado funcional, evitamos que outros programadores ou equipas estejam a desenvolver novas versões sobre código não verificado.

Este processo faz com que o *software* seja menos suscetível a erros uma vez que é testado frequentemente através de testes automatizados e permite aumentar a produtividade, evitando que os erros sejam apenas descobertos posteriormente, quando a sua resolução é menos eficiente e impede a utilização por outras equipas de versões não funcionais que atrasariam o processo de desenvolvimento.

### 2.1.3 ENTREGA CONTÍNUA

A prática de Entrega Contínua surge da tradução do termo *Continuous Delivery*, que iremos utilizar preferencialmente na versão inglesa. O ciclo de *Continuous Delivery*, representado no diagrama da figura 2.3, é uma extensão de uma outra prática apresentada na secção anterior, o *Continuous Integration*. Os objetivos do *Continuous Delivery* são os seguintes [7]:

- Desenvolver *software* que seja menos suscetível a erros.
- Oferecer aos clientes do produto novas versões, mais frequentemente.
- Aumentar a produtividade da equipa de desenvolvimento.
- Garantir que o *software* pode ser posto em produção em qualquer momento.
- Garantir a satisfação do cliente.

O *Continous Delivery* é uma extensão da prática de *Continuous Integration* e para além dos testes automáticos são realizadas as seguintes atividades:

- A nova versão do *software* é instalada e executada num ambiente semelhante ao de produção. Este ambiente é também designado de *staging*.
- A pessoa responsável realiza os testes manuais que permitem validar se a funcionalidade desenvolvida cumpre todos os requisitos.
- A nova versão é colocada em produção e fica disponível para a utilização do cliente.

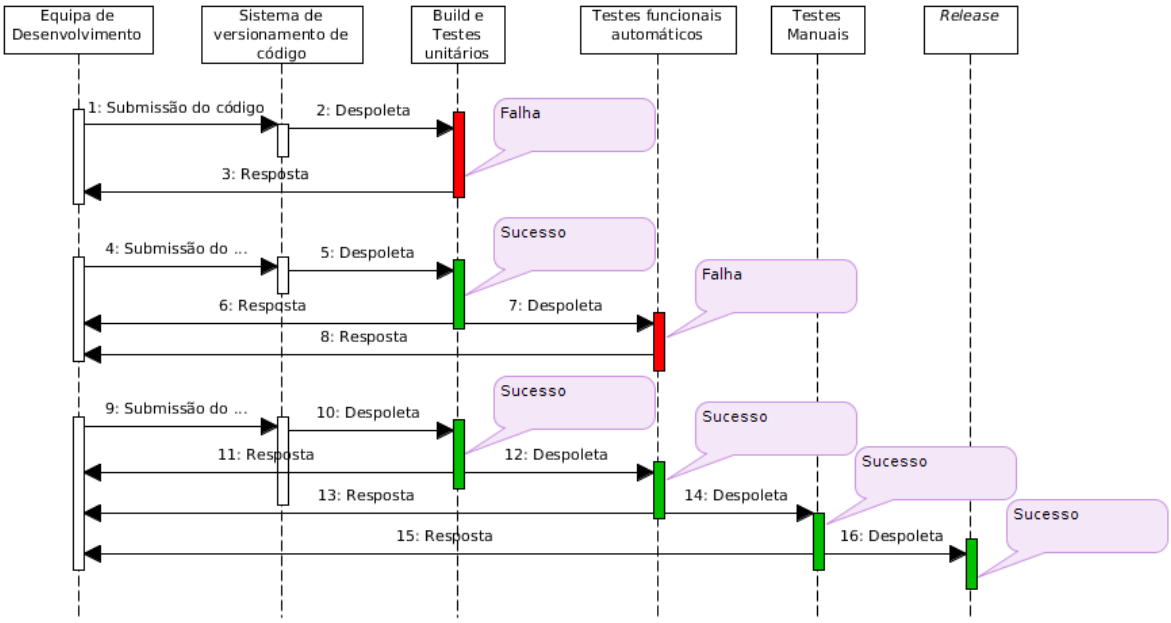



Figura 2.3: Ciclo de vida do processo de *Continuous Delivery*, imagem traduzida do livro *Continuous Delivery* [6]

Tanto o *Continuous Delivery* como o *Continuous Integration* são constituídos por diferentes ambientes onde a aplicação desenvolvida é executada, nomeadamente em ambientes de desenvolvimento, testes, *staging* (apenas utilizado no *Continuous Delivery*) e produção. O ambiente de desenvolvimento é o ambiente no qual os programadores testam o código que estão a desenvolver, este pode ser a sua própria máquina, um sistema de virtualização ou uma máquina externa. O ambiente de testes geralmente é um servidor dedicado para este efeito e que corre uma ferramenta que permita a execução dos testes automáticos. O ambiente de *staging* tenta replicar o ambiente de produção e serve para a execução de testes manuais sobre novas versões do produto. Finalmente o ambiente de produção é onde é executada a última versão funcional e onde o produto pode ser acedido pelos clientes.

Existem dois pontos importantes para que as práticas de *Continuous Delivery* e *Continuous Integration* sejam eficientes. Primeiro é preciso automatizar o processo de configuração dos ambientes nos quais a aplicação será executada. Este processo é suscetível a erros e se não for automatizado requer documentação adicional, a utilização de recursos (humanos) na realização destas tarefas e é dispendioso em termos do tempo que é necessário para preparar estes ambientes [7]. Esta tarefa é também denominada de provisão. O segundo ponto é que os ambientes deverão ser o mais homogéneos possível, mitigando problemas que possam surgir da utilização de diferentes versões das ferramentas[6].

A prática de *Continuous Delivery* promove a satisfação do cliente, pois garantimos que as funcionalidades em desenvolvimento acrescentam qualidade ao produto e vão de encontro às especificações transmitidas pelo cliente e permite aumentar a produtividade das equipas de desenvolvimento, pois evita que se continue a trabalhar sobre funcionalidades que não acrescentam valor ao produto final.

A ferramenta a ser desenvolvida nesta dissertação está focada no problema da criação do ambiente de *staging* e posterior validação por parte do cliente de uma dada funcionalidade. Pretende-se que este processo seja o mais eficiente possível através da automatização da criação e provisão deste ambiente.

### 2.1.4 TESTES MANUAIS DE ACEITAÇÃO

Os testes de aceitação são também denominados de *User Acceptence Tests* (UAT) e são realizados pelo cliente ou uma pessoa delegada por este e que se responsabiliza por testar o produto e garantir que este segue todos os pressupostos para os quais foi construído [1].

Um produto de *software* só pode ser terminado depois de executados testes aceitação que permitem verificar se os critérios de aceitação foram cumpridos [1].

A validação de novas funcionalidades do produto por parte do cliente é importante de forma a perceber o seu real valor para o produto final e se a mesma vai de encontro aos requisitos estipulados. Se os testes de aceitação não forem efetuados de uma forma incremental isto pode levar a que no final do desenvolvimento o número de alterações seja muito superior, pois o sistema foi todo construído segundo pressupostos que não satisfazem o cliente e a sua reconstrução pode implicar alterações mais profundas.

Os testes manuais de aceitação são realizados depois da verificação do código desenvolvido através de testes automáticos, estes apenas garantem a correção e se o mesmo é funcional mas não garantem que o mesmo funciona como esperado para o cliente. Para além disso, a utilização de testes manuais de aceitação pode ser útil pois permite encontrar erros que não foram cobertos através dos testes automáticos [8].

## 2.2 VIRTUALIZAÇÃO COM CONTAINERS

Os *containers* são uma forma de virtualização que permite o isolamento entre o sistema operativo base e o *container* tal como acontece com as típicas máquinas virtuais, mas ao contrário destas últimas não requerem um sistema operativo completo para funcionar [9]. Os sistemas de virtualização baseados nas capacidades do sistema operativo Linux representam uma tecnologia em expansão devido às suas características e capacidades [10], uma vez que estes sistemas são mais leves, conseguimos criar na mesma máquina mais instâncias, sendo transparente para o utilizador final a utilização deste sistema de virtualização ou outro [10].

A figura 2.4a mostra as diferenças entre as duas soluções. À esquerda (figura 2.4a) está representada a utilização de máquinas virtuais onde é necessário, para além da existência de um sistema operativo base (*Host OS*), a existência de um sistema operativo hóspede (*Guest OS*) que suportará a execução do sistema virtualizado o que requer a utilização de um *hypervisor* que traduza as instruções entre o sistema operativo hóspede e base [11]. À direita, a arquitetura utilizando *containers* (figura 2.4b) em que não é necessária a utilização de um sistema operativo hóspede completo, nem um *hypervisor* que


(a) Máquinas Virtuais (b) *Containers*

Figura 2.4: Comparação entre a execução de *containers* e máquinas virtuais [13]

faça a tradução das instruções entre as duas, os processos correm sobre o sistema operativo base de uma forma isolada utilizando funcionalidades do *kernel* Linux [9].

A virtualização utilizando *containers* é também conhecida como *Operating System-Level Virtualization* [12].

Existem várias implementações usando os mecanismos do *kernel* Linux [12]; no âmbito desta dissertação iremos estudar duas delas: LXC e Docker.

A utilização de *containers* traz algumas vantagens relativamente às máquinas virtuais. A primeira vantagem decorre dos *containers* partilharem o mesmo sistema operativo que a máquina onde estão a ser executados (ao contrário de uma máquina virtual). Isto faz com que sejam utilizados menos recursos em termos de espaço de armazenamento e permite, na mesma máquina base, executar um maior número de sistemas de virtualização [9]. A segunda vantagem é que utilizando *containers*, não é necessário traduzir as instruções entre a máquina virtual e o sistema operativo base, o que faz com que as aplicações executadas nos *containers* tenham um desempenho muito idêntico à execução no sistema operativo base, logo superior ao que é possível obter com as máquinas virtuais [14].

### 2.2.1 SISTEMA DE VIRTUALIZAÇÃO LXC

Os *containers* LXC recorrem a mecanismos como o ***Chroot***, ***Cgroups*** e ***Namespaces*** para garantirem o isolamento entre o sistema operativo base e o *container* [11].

- O *chroot* permite definir o diretório *root* para um processo e os seus subprocessos, garantindo o isolamento entre *containers* e o sistema operativo base [9]. Em termos práticos o processo de um *container* não é capaz de aceder aos ficheiros do sistema operativo base nem aos ficheiros de outros *containers*.
- Os *cgroups* permitem gerir a utilização de recursos partilhados como CPU e memória [9].
- Os *namespaces* permitem o isolamento de recursos globais do sistema para que possam ser utilizados pelos *containers*. Na prática os *namespaces* criam uma abstração sobre os recursos de forma a que cada *container* seja capaz de utilizar uma instância daquele recurso e as alterações

feitas sobre essa instância sejam apenas visíveis pelos processos dentro daquele *container*. Existem 6 tipos de *namespaces*:

- IPC - Isola recursos responsáveis pela comunicação entre processos [15].
- *Network* - Isola recursos relativos à rede como *iptables*, *routing tables* e *interface loopback*. [9].
- *Mount* - Isola os *mount points* do sistema de ficheiros [15] de forma a estes não serem visíveis por processos que não pertençam àquele *namespace*.
- PID - Permite o isolamento dos id's dos processos entre diferentes *namespaces*, desta forma é possível haver processos no sistema com os mesmos id's mas em *namespaces* diferentes [15].
- *Users* - Proporciona que um processo tenha um id de utilizador e grupo diferentes no sistema operativo base e no *container* [15].
- UTS - Isola dois identificadores do sistema, *nodename* e *domainname*, desta forma é possível que cada *container* tenha um *nodename* e *domainname* próprios [15].

Os *containers* LXC são criados a partir de *templates*, através do comando `lxc-create` [11].

Os *templates* são *scripts* usados para criar os *containers* LXC que, para além de outras operações, criam um ficheiro de configuração (*config*) e geram o sistema de ficheiros (*rootfs*). Existem vários tipos de *templates* que permitem gerar *containers* com base em diferentes distribuições.

Os ficheiros dos *containers* (*config* e *rootfs*) estão localizados em ~/.local/share/lxc ou em /var/-lib/lxc dependendo do seu tipo (não privilegiado ou privilegiado respetivamente) [11].

A diferença entre os *containers* privilegiados e não privilegiados é que estes segundos mapeiam os id's dos utilizadores e dos grupos entre o sistema base e o *container*, isto significa que o utilizador root dentro do *container* não terá o mesmo id dentro e fora do container (que representará o id de um utilizador não privilegiado). A utilização dos *containers* não privilegiados é recomendada de forma a garantir que mesmo que um atacante consiga sair do *container* as operações que pode realizar são limitadas [11].

O ficheiro *config* permite definir um conjunto de recursos como os *mount points*, a raiz do sistema de ficheiros (que por omissão é a localização da pasta *rootfs* no sistema operativo base), *utsname* (nome do *container*), configurações de rede e grupos de controlo (*control groups* ou *cgroups*) [16], que são depois virtualizados quando o *container* é iniciado.

Depois de criados, os *containers* podem ser inicializados ou parados através dos comandos (`lxc-start` e `lxc-stop`). Quando é iniciado, ao *container* é atribuído um endereço IP que pode ser utilizado para depois aceder às aplicações dentro do *container*.

Através do comando `lxc-ls` podemos ver uma lista de *containers* criados (ver figura 2.5).

```
NAME                STATE    IPV4        IPV6  AUTOSTART
----------------------------------------------------------
78swe-396-elastica  RUNNING  10.0.3.165  -     NO
beubi_swe           STOPPED  -           -     NO
ubuntu-base         STOPPED  -           -     NO
ubuntu-base-chef    STOPPED  -           -     NO
```

Figura 2.5: Exemplo do comando `lxc-ls -fancy`.

Outro comando que também é importante no âmbito da utilização dos *containers* LXC é o *clone*, sendo especialmente útil quando temos um conjunto de ferramentas ou configurações que tem de ser aplicadas à maioria dos nossos *containers*.

Partindo do diagrama 2.6 para explicar a utilização dos *containers* LXC, um *container* é criado a partir do comando `lxc-create` em que podemos escolher o tipo de distribuição que pretendemos utilizar. Se pretendermos criar um *container* que tenha um conjunto de configurações que necessitem de ser reutilizadas em *containers* futuros, o primeiro passo é inicializar o *container* através do comando `lxc-start` e aplicar as alterações necessárias, em seguida paramos o *container* por meio do comando `lxc-stop` e este pode então ser replicado através do comando `lxc-clone`.

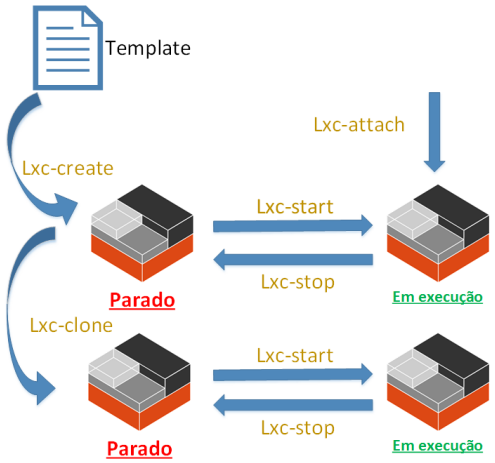



Figura 2.6: Modelo de utilização dos *containers* LXC.

### 2.2.2 SISTEMA DE VIRTUALIZAÇÃO DOCKER

O Docker, tal como os *containers* LXC, utiliza os **Namespaces** e **Cgroups** para garantirem o isolamento e os recursos que cada *container* pode utilizar [17].

Para além disso, o Docker utiliza um Union FS[18]; este tipo de sistema de ficheiros funciona sobrepondo vários sistemas de ficheiros, também conhecidos como *layers*, em que apenas o sistema de ficheiros do topo tem permissões de leitura e escrita tal como representado na figura 2.7 [17].

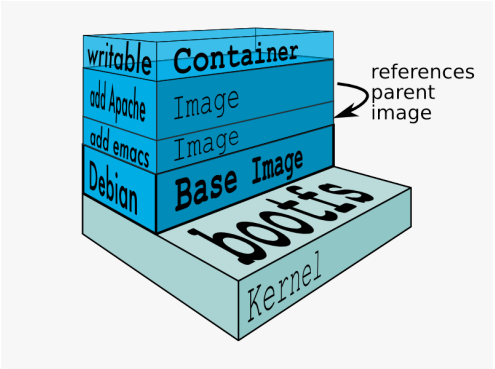



Figura 2.7: Sistema de ficheiros por camadas de uma imagem Docker. [17].

Os *containers* Docker permitem a utilização de diferentes tipos de *execution drivers*, este é o componente que é utilizado para criar o ambiente de isolamento dos *containers* [19].

O Docker é do tipo cliente-servidor [17]. Tal como se pode verificar na figura 2.8, o Docker daemon é o responsável pela gestão dos *containers* e o Docker client é a interface que permite operar sobre os *containers* presentes no sistema onde o Docker daemon está a ser executado. A comunicação entre o Docker daemon e o Docker client é feito através de uma API RESTful ou utilizando *sockets* [17], isto significa que o Docker daemon e Docker client não precisam de estar a ser executados na mesma máquina.

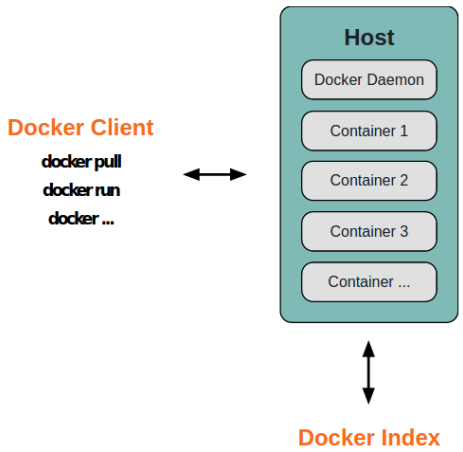



Figura 2.8: Arquitetura cliente-servidor do Docker. [17].

O Docker na sua base é composto por três componentes fundamentais, os Dockerfiles, imagens e *containers*. Os Dockerfiles dão origem a imagens e estas dão origem a *containers* tal como mostra a figura 2.9.

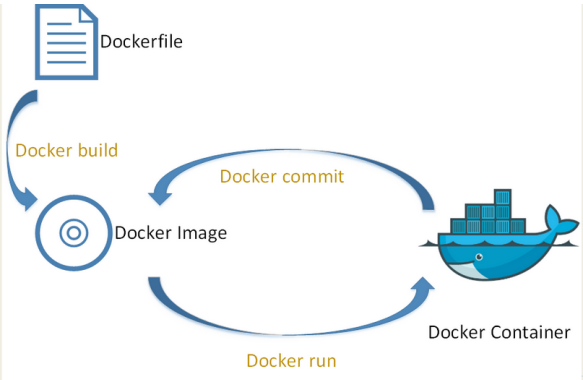



Figura 2.9: Componentes Docker e a relação entre eles.

Os Dockerfiles são ficheiros que contêm instruções de ações que devem ser aplicadas a uma imagem Docker. Uma imagem é um sistema de ficheiros formado por várias *layers* onde cada uma representa uma alteração (uma instrução do ficheiro Dockerfile) que foi feita sobre o sistema de ficheiros. Os *containers* são depois criados a partir destas imagens e múltiplos *containers* podem ser criados a partir da mesma imagem. Fazendo a comparação entre o Docker e programação, podemos dizer que as imagens são as nossas classes onde podemos definir os nossos atributos e métodos e os *containers* são os nossos objetos.

As imagens são criadas a partir de Dockerfiles, através do comando `docker build` e são construídas por camadas (tirando partido do Union FS). O ficheiro Dockerfile é formado por duas partes (tal como pode ser observado no trecho de código 1), a primeira indica o tipo de instrução e a segunda representa a ação propriamente dita.

```
FROM        ubuntu:14.04

RUN         apt-get update && apt-get install -y apache2

RUN         rm -rf /var/www/*

COPY        . /var/www
COPY        ./apache_conf/000-default.conf /etc/apache2/sites-available/

ENTRYPOINT  /usr/sbin/apache2ctl -D FOREGROUND
```

Código 1: Ficheiro Dockerfile

As instruções mais úteis do ficheiro Dockerfile são:

- FROM: indica a imagem base.
- RUN: permite executar comandos do terminal Linux.
- COPY: permite copiar ficheiros dentro do contexto do Dockerfile, do sistema base para o *container*.
- WORKDIR: define o caminho a partir do qual os próximos comandos serão executados.
- ENTRYPOINT: define o comando que deve de ser executado quando o *container* é inicializado.
- EXPOSE: permite definir os portos do *container* que pretendemos expor.
- VOLUME: permite definir uma pasta que será montada entre o *container* e o sistema base.

Cada Dockerfile tem de indicar qual a imagem base a utilizar (por exemplo, Ubuntu), por cada instrução descrita no Dockerfile é lançado um novo *container*, executada a instrução e guardada a imagem intermédia (através do comando `docker commit`) como pode ser visto através da figura 2.9. Para a instrução seguinte é utilizada a imagem gerada pela instrução anterior, sendo este processo repetido até todas a instruções do Dockerfile terem sido executadas.

A criação de imagens por camadas traz algumas vantagens [17]:

- Permite a reutilização das camadas, desta forma são utilizados menos recursos em termos de armazenamento pois camadas idênticas não são replicadas (cada *layer* é identificado por uma *tag* e o *layer* pode estar associado a diferentes imagens) e permite a reconstrução das imagens mais rapidamente.
- Cada imagem tem um histórico de operações. Através do comando `docker history` podemos ver que configurações foram aplicadas em cada camada da nossa imagem.

Uma vez construída a imagem e aplicadas as configurações necessárias podemos inicializar o *container* através do comando `docker run`. O comando `docker run` permite-nos lançar um *container* e com ele podemos definir algumas opções, como por exemplo: os portos que pretendemos mapear, as pastas/ficheiros que queremos partilhar entre o *container* e o sistema base e o seu nome.

A descrição dos portos e dos volumes no Dockerfile não nos permite especificar o porto do sistema base a utilizar nem a pasta que pretendemos partilhar, isto porque os Dockerfiles foram idealizados de forma construírem imagens que possam ser utilizadas independentemente da máquina.

Geralmente cada *container* Docker corre apenas um processo por exemplo, um servidor Apache ou servidor MySQL, porém podem existir situações em que tal não faz sentido e necessitamos de ter múltiplos processos a correr num único *container* Docker. Uma das soluções para contornar esta "limitação" dos *containers* Docker é utilizar um gestor de processos como, por exemplo, o Supervisor para que seja possível executar múltiplos processos no mesmo *container* [17].

Quando utilizamos uma solução com múltiplos *containers* é fundamental que estes comuniquem entre si. Por exemplo, uma aplicação PHP executada num servidor Apache, através de um *container*, é natural que esta necessite de algum tipo de persistência como, por exemplo, uma base de dados MongoDB. Isto é conseguido através da utilização de *links*. Os *links* podem ser especificados através de comandos nativos do Docker utilizando a opção `-link`.

Os *containers* Docker podem ser lançados através dos comandos nativos ou através de uma ferramenta de orquestração como o Docker Compose. O Docker Compose permite descrever múltiplos *containers* e todos os seus parâmetros, como ligações entre eles, volumes e portos, por meio de um único ficheiro YAML, tal como é exemplificado no código 2. Ferramentas deste tipo são fundamentais para a execução de *containers* Docker devido à forma como estes foram idealizados. Cada *container* executa geralmente apenas um processo e por isso, é natural que o ambiente de execução de uma aplicação utilize múltiplos *containers*, fazendo o lançamento de todos, através dos comandos Docker nativos, um processo demorado e complexo (dependendo do tamanho da aplicação).

```
web:
  build: .
  links:
    - db
  ports:
    - "10000:80"
db:
  build: ./containerMysql
```

Código 2: Ficheiro Docker Compose

### 2.2.3 COMPARAÇÃO DOS SISTEMAS DE VIRTUALIZAÇÃO

Os *containers* Docker e LXC são dois tipos de virtualização idênticos na perspetiva em que utilizam as funcionalidades do *kernel* Linux para garantir o isolamento dos processos.

Os *containers* LXC surgiram antes dos *containers* Docker. Nas primeiras versões do Docker, o *execution driver* utilizado era o LXC e, só mais tarde, é que surgiu um *execution driver* criado pelo Docker, o libcontainer.

Uma das vantagens da utilização dos *containers* Docker relativamente ao LXC, e que levou à forte adesão ao Docker, foi a facilidade de interação com os mesmos através de comandos e opções que o LXC não tem. Apesar disso, a abordagem do Docker de executar apenas um processo por *container* pode dificultar um pouco a adoção destes pois é um paradigma um pouco diferente do qual estamos habituados.

Na tabela 2.1 são apresentadas as principais diferenças entre o Docker e o LXC.

| | Docker | LXC |
|---|---|---|
| Multi-processos | O *container* Docker foi idealizado de forma a correr um único processo, porém é possível contornar isto utilizando um gestor de processos como por exemplo o Supervisor. | Os *containers* LXC comportam-se como uma máquina virtual mais leve e é possível executar diversos processos, tal como uma máquina virtual normal. |
| Reutilização | O Docker faz uso das imagens para permitir que vários *containers* possam ser lançados utilizando a mesma configuração. | O LXC contém o comando lxc-clone que permite a reutilização de *containers* porém isto implica a cópia total de todos os ficheiros do *container* original |
| Provisão | O Docker faz uso de um mecanismo (Dockerfile) de provisão que permite configurar os *containers*, para além disso a utilização de um Union FS permite que a recriação das imagens seja feita mais rapidamente. | No LXC não existe esta política de preparação do *container* pelo que é necessário utilizar uma ferramenta externa como por exemplo o Ansible ou o Chef |
| Interação | O Docker tem um conjunto de comandos bastante abrangente que facilitam a interação e permitem ver informação detalhada sobre os *containers* e imagens quando comparado com o LXC. | Os comandos e as opções que o LXC oferece são algo limitados e dificultam a interação com os mesmos. As informações disponibilizadas também são muito poucas quando comparadas com as que o Docker oferece. |
| Orquestração | O Docker, através do Docker Compose, suporta a orquestração de múltiplos *containers*. | O LXC não oferece nenhuma ferramenta que permita a orquestração dos *containers* mas como o LXC foi idealizado para correr múltiplos processos, esta lacuna não é especialmente importante. |
| Portabilidade | A portabilidade dos *containers* Docker é facilitada através dos Dockerfiles e do Docker Hub, o primeiro permite a construção das imagens a partir destes ficheiros o segundo permite enviar as imagens para o repositório Docker Hub e descarregar essas imagens a partir de outro sistema que tenha o Docker instalado. | Em termos de portabilidade o LXC é mais uma vez limitado, a única forma de utilizar o *container* criado numa máquina noutra, é transferindo todos os ficheiros do *container* de uma máquina para outra. Este é um processo nada eficiente e a correta execução do *container* não é garantida pois este é dependente da configuração da máquina onde está a ser executado. |

Tabela 2.1: Comparação dos sistemas de virtualização Docker e LXC.

### 2.2.4 IMPORTÂNCIA DOS SISTEMAS DE VIRTUALIZAÇÃO NO PROCESSO DE ENTREGA CONTÍNUA

A utilização de sistemas de virtualização no processo de *Continuous Delivery* é importante pois permite a imitação do ambiente de produção (ambiente onde a versão final do *software* será executada) num ambiente isolado. Geralmente a não utilização de um sistema de virtualização na ótica do programador implica que todas as configurações relacionadas com um projeto sejam feitas no sistema operativo base, o que pode gerar conflitos entre dependências e diferentes versões do *software* (por exemplo, PHP, MySQL ou Apache) de outros projetos.

O *software* desenvolvido só tem valor quando é posto em produção, geralmente a transição entre sistemas no ciclo *Continuous Delivery* gera problemas pois as configurações entre os sistemas são diferentes e essas diferenças trazem problemas de compatibilidade [20]. O código que funcionava bem na máquina do programador não funciona no ambiente de produção por exemplo.

Os sistemas de virtualização ao nível do sistema operativo (*Operating System-Level Virtualization*) permitem lançar rapidamente um novo meio de execução sem grandes custos em termos de desempenho e onde todas as configurações relativas a um projeto podem ser aplicadas isoladamente.

Outro ponto importante é que se utilizarmos sempre o mesmo ambiente de desenvolvimento com o passar do tempo começamos a ficar com um ambiente de execução viciado e é por isso mais difícil de auditar os requisitos reais do projeto e encontrar possíveis conflitos.

## 2.3 FERRAMENTAS DE ORQUESTRAÇÃO E PROVISÃO

As ferramentas de orquestração e provisão foram criadas para mitigar o problema que advém da gestão e configuração de um grande número de máquinas. Quando feito manualmente, requer equipas dedicadas inteiramente a essa função, perdendo-se mão-de-obra em atividades que não geram valor para a empresa.

Esta secção pretende estudar ferramentas que possam ser utilizadas no processo de configuração e orquestração de *containers* LXC ou Docker.

O Ansible e o Chef são agnósticas em relação ao meio de execução que estão a provisionar (pode ser um *container*, máquina virtual, servidor físico ou máquina na *cloud*) enquanto que o Dockerfile e DockerCompose são ferramentas que são utilizadas especificamente em *containers* geridos através da plataforma Docker.

### 2.3.1 CHEF

O Chef é uma ferramenta que permite a automatização do processo de provisão/configuração dos diferentes meios de execução (servidores, máquinas virtuais, *containers*) que compõem uma aplicação. As configurações são aplicadas através de ficheiros que descrevem as diferentes ações que necessitam de ser executadas para o meio ficar num dado estado, como por exemplo a instalação de pacotes, permissões sobre ficheiros ou diretórios, alterações sobre ficheiros de configuração e a inicialização serviços.

Os meios de execução que o Chef configura são também referidos na documentação oficial [21] como *nodes*.

O Chef aplica as configurações necessárias a um *node* através de *cookbooks*. Os *cookbooks* são formados por vários componentes, entre eles as receitas, atributos e *templates* [21], como pode ser visto pela figura 2.10.

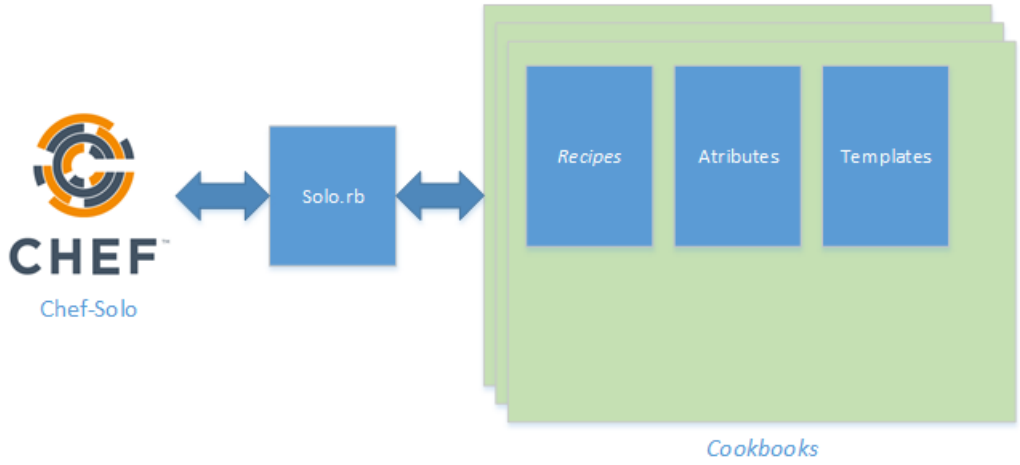



Figura 2.10: Utilização do Chef através do Chef Solo.

As receitas são ficheiros escritos em Ruby e que descrevem configurações (por exemplo, a instalação de um pacote, a alteração de um ficheiro de configuração ou a inicialização de um serviço) a serem executadas [21] para que o *node* fique num dado estado.

Os atributos são variáveis que permitem alterar o comportamento da receita consoante o *node* que está a ser configurado, por exemplo estes atributos permitem definir o utilizador e a *password* do MySQL, o nome da base de dados, os portos que este serviço utilizará ou o nome do utilizador do *node* [22].

Os *templates* são outra funcionalidade interessante, através de ficheiros *template* ERB é possível gerar ficheiros com parâmetros dinâmicos (através dos atributos) para cada *node* da nossa infraestrutura.

Todas as configurações a aplicar num *node* são descritas através de um ficheiro JSON num *array* designado *run list*, este indica a ordem pela qual as receitas devem de ser aplicadas.

Existem dois modos de funcionamento, localmente através do Chef Solo ou remotamente utilizando o Chef Server [22]. No primeiro caso, todos os ficheiros necessários à provisão (*cookbooks*) têm que estar disponíveis localmente. No segundo caso, o Chef Server contém todas as informações e ficheiros necessários à configuração de um *node*, estes são requisitados através do Chef Client e utilizados para o configurar [22]. A maior parte do trabalho, como referido anteriormente, é feito pelo *node* a ser provisionado.

A utilização do Chef Solo é exemplificada através do diagrama representado pela figura 2.10. O Chef Solo é executado através do comando `chef-solo -c`, em que a opção `-c` serve para especificar a localização do ficheiro de configuração, que descreve entre outros, a localização dos *cookbooks* e o ficheiro do *node* a ser provisionado.

Em seguida será exemplificado a configuração de um servidor Apache utilizando as funções do Chef descritas anteriormente. Esta solução é constituída fundamentalmente por quatro ficheiros.

O ficheiro *template* (ver Código 3) permite gerar ficheiros em que são definidas variáveis, as quais é possível substituir por valores, gerando o ficheiro final. Neste exemplo é pretendido definir o diretório

raiz do Apache no ficheiro 000-default.conf. As variáveis são definidas através das diretivas <%= %> em que o Directory indica o nome da variável.

```
...
DocumentRoot <%= @Directory %>

# Available loglevels: trace8, ..., trace1, debug, info, notice, warn,
# error, crit, alert, emerg.
# It is also possible to configure the loglevel for particular
# modules, e.g.
#LogLevel info ssl:warn

<Directory <%= @Directory %>>
  # enable the .htaccess rewrites
  AllowOverride All
  Require all granted
</Directory>
...
```

Código 3: Ficheiro do *Template*

No ficheiro da receita (ver Código 4) são definidas as instruções de provisão. Neste exemplo simples são efetuadas três operações: é executado o comando `apt-get update`; em seguida é instalado o Apache 2 através do recurso `apt_package`; e é gerado o ficheiro 000-default.conf a partir do *template* descrito anteriormente. Adicionalmente é importante referir a utilização do atributo como valor da variável Directory no *template*.

```
execute 'apt_get_update' do
  command 'apt-get update'
end

apt_package 'apache2'

template '/etc/apache2/sites-available/000-default.conf' do
  source '000-default.conf.erb'
  owner 'root'
  group 'root'
  mode '0644'
  variables(:Directory => node['apache']['root_dir'])
end
```

Código 4: Ficheiro da Receita

No ficheiro Node (ver Código 5) definimos os atributos e as receitas que necessitam de ser executadas para configurar um dado sistema.

```
{
  "apache": {
    "root_dir": "/var/www/html"
  },
  "run_list": [
    "recipe[webserver]"
  ]
}
```

Código 5: Ficheiro do *Node*

O ficheiro solo (ver Código 6) é utilizado pelo Chef Solo para obter as configurações com que deve de ser executado.

```
cookbook_path  ["/srv/cheftest/chef/cookbooks"]
json_attribs "/srv/cheftest/chef/nodes/webdocker.json"
```

Código 6: Ficheiro de configuração Chef Solo

Para aplicarmos estas configurações necessitamos de executar o seguinte comando:

```
chef-solo -c solo.rb
```

Código 7: Comando de execução do Chef

### 2.3.2 ANSIBLE

O Ansible é uma ferramenta de configuração, que tal como o Chef, permite descrever configurações que são aplicadas para que o meio de execução cumpra os requisitos necessários para realizar uma dada tarefa.

Os *playbooks* descrevem as configurações e os meios de execução onde estas devem de ser aplicadas. Estes são escritos no formato YAML [23] e são compostos por uma ou mais *plays*.

Cada *play* define a máquina ou conjunto de máquinas e ações (*tasks*) que têm de ser aplicadas. Cada *task* é aplicada através de módulos Ansible que são componentes que permitem executar uma ação específica como mover um ficheiro, instalar uma ferramenta ou iniciar um serviço [23].

As máquinas destino de cada *play* são identificadas através de um ficheiro chamado *inventory*, este define máquinas ou grupos de máquinas que depois são utilizados para indicar quais os meios a que as configurações têm de ser aplicadas [23]. Esta organização do Ansible é representada no diagrama da figura 2.11.

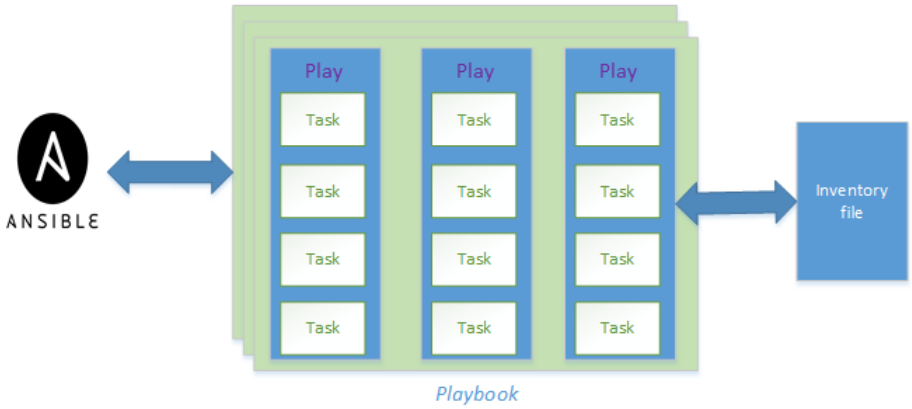



Figura 2.11: Utilização do Ansible.

Uma das características do Ansible, tal como acontecia com o Chef, é a sua idempotência, o que significa que o resultado das configurações é sempre o mesmo e só são executadas as configurações necessárias para que a máquina alvo fique no estado pretendido.

O Ansible funciona de uma forma *agentless* [23], isto significa que não existe uma hierarquia cliente-servidor. A máquina responsável por promover a provisão dos meios de execução comunica através de SSH, enviando os módulos Ansible necessários para executar as configurações e, após serem aplicadas, os módulos são removidos desse sistema [24].

O seguinte exemplo (ver Código 8 e 9) mostra os ficheiros necessários para configurar um servidor Apache utilizando o Ansible.

A partir do ficheiro *template* (ver Código 8) conseguimos definir variáveis que depois podem ser substituídas por valores específicos, definidos no ficheiro *playbook* que utiliza o *template*.

```
ServerAdmin webmaster@localhost
DocumentRoot {{document_root}}

# Available loglevels: trace8, ..., trace1, debug, info, notice, warn,
# error, crit, alert, emerg.
# It is also possible to configure the loglevel for particular
# modules, e.g.
#LogLevel info ssl:warn

<Directory {{document_root}}>
  # enable the .htaccess rewrites
  AllowOverride All
  Require all granted
</Directory>
```

Código 8: Ficheiro do *Template*

No nosso ficheiro *playbook* (ver Código 9) definimos uma *play* que é composta por uma variável que indica o diretório raiz do Apache 2 e um conjunto de *tasks* que tem que ser executadas para configurar o ambiente. Inicialmente é instalado o Apache 2, a opção `update_cache` permite executar o comando `apt-get update` antes de proceder à instalação. O segundo comando indica que deve de ser utilizado aquele *template* de forma a gerar o ficheiro 000-default.conf. Os valores que as variáveis do *template*

assumem são declarados através dos atributos.

```
- hosts: localhost
  vars:
    document_root: /var/www/html
  tasks:
    - apt: name=apache2 update_cache=yes
    - template: src=./templates/000-default.j2
        dest=/etc/apache2/sites-available/000-default.conf owner=root
        group=root mode=0644
```

Código 9: Ficheiro *Playbook*.

Para aplicarmos estas configurações necessitamos de executar o seguinte comando:

```
ansible-playbook yaml_file.yml
```

Código 10: Comando de execução do Ansible.

### 2.3.3 COMPARAÇÃO DAS FERRAMENTAS DE ORQUESTRAÇÃO E PROVISÃO

Tal como referido anteriormente o Chef e o Ansible são duas ferramentas que permitem a orquestração da infraestrutura e provisão/configuração de meios de execução distintos (como por exemplo máquinas virtuais, *containers* ou servidores físicos) por isso podemos dizer que são agnósticas ao meio onde estão a ser executadas.

O Dockerfile & DockerCompose, analogamente ao Chef e ao Ansible, são ferramentas específicas dos *containers* Docker o que é um fator limitativo em termos de portabilidade das configurações para outros sistemas.

A ferramenta Chef inicialmente era apenas destinada à configuração de meios de execução e não à descrição da infraestrutura, sendo só mais tarde anunciado pela equipa do Chef que passaria a ser suportada a descrição da infraestrutura através das receitas.

O Ansible por ser uma ferramenta mais recente, já foi criada com um *mindset* virado não só para a configuração/provisão, mas também para a descrição da infraestrutura.

As principais diferenças entre o Ansible e Chef são as seguintes:

| | Ansible | Chef |
|---|---|---|
| Arquitetura | Arquitetura *agentless*, as configurações podem ser aplicadas a partir de qualquer máquina que tenha o Ansible instalado para outra desde que esta seja capaz de comunicar por SSH. O Ansible também pode ser utilizado para configurar a própria máquina onde está instalado. | O Chef pode ser executado de várias formas. Utilizando o Chef Server e o Chef Client este impõe uma arquitetura cliente-servidor bem definida, porém também podemos executar as receitas localmente a partir do Chef Solo. |
| Provisão | A provisão é despoletada a partir da máquina onde o Ansible está instalado. | Utilizando o Chef Server e o Chef Client o cliente é responsável por verificar periodicamente a existência de alterações. No caso do Chef Solo a configuração é executada explicitamente através do comando chef-solo. |
| Meio de configuração | As configurações são aplicadas através de *playbooks*. | As configurações são aplicadas através de *cookbooks* |
| Ficheiros de configuração | Os *playbooks* são escritos no formato YAML e utilizam o Jinja2 como plataforma dos *templates*. | As receitas são escritas em Ruby e utilizam o Erubis como plataforma dos *templates*. |

Tabela 2.2: Comparação das ferramentas Ansible e Chef.

Existem algumas razões que nos podem levar a utilizar ferramentas como o Chef ou o Ansible em detrimento de formas específicas de provisão como o Dockefile [23]:

- Os *playbooks* do Ansible e as receitas do Chef são portáveis e podem ser executadas em diferentes meios de execução. Os ficheiros Dockerfile são específicos dos *containers* Docker e não podem ser executadas em outros meios de execução.
- Os *playbooks* e as receitas podem ser escritas em vários ficheiros e posteriormente reutilizadas. As configurações dos Dockerfile são escritas num único ficheiro e não existe a possibilidade de as reutilizar.
- O Chef e o Ansible permitem a configuração de diferentes meios de execução. Cada ficheiro Dockerfile apenas é capaz de configurar uma imagem e não múltiplas imagens.

### 2.3.4 IMPORTÂNCIA DAS FERRAMENTAS DE PROVISÃO E ORQUESTRAÇÃO NO PROCESSO DE ENTREGA CONTÍNUA

As ferramentas de provisão e orquestração foram criadas de forma a evitar a configuração manual de meios de execução uma vez que é um processo bastante trabalhoso quando aplicado a várias máquinas e muito suscetível a erros humanos. Tal como referido na secção 2.1, para a existência de um ciclo eficiente de *Continuous Delivery* e uma vez que diferentes etapas podem necessitar de diferentes configurações, é necessário que estas sejam automatizadas, garantindo que todos os meios de execução estão corretamente provisionadas e de acordo com o descrito pelos ficheiros de configuração, tornando todo o ciclo mais eficiente.

Na figura 2.12, são identificadas as etapas em que os sistemas de provisão poderão ser úteis, para garantir máquinas preparadas e equivalentes:

- Sistema de desenvolvimento onde o novo código desenvolvido é testado pelo programador.
- Sistema responsável pelo ambiente de *staging* e ao qual o cliente acederá para realizar testes manuais.
- Sistema de produção no qual a última versão verificada e validada é disponibilizada ao cliente final.

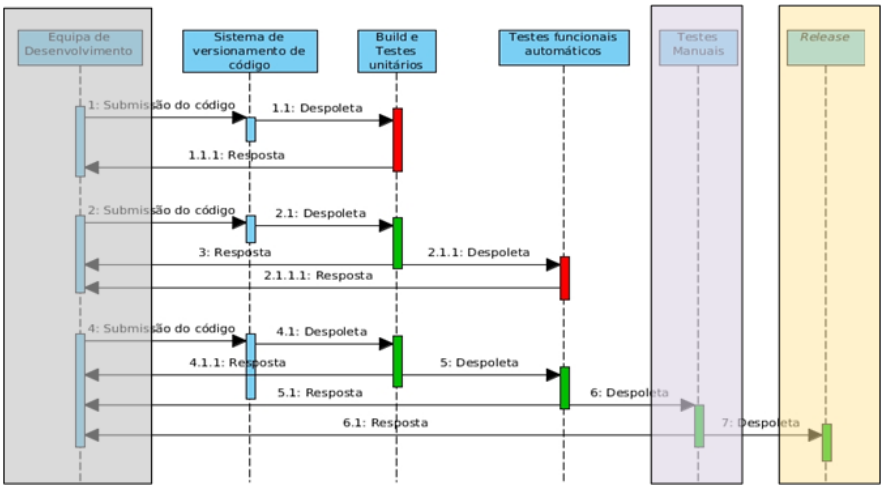



Figura 2.12: Ciclo de vida do processo de *Continuous Delivery* e identificação das etapas em que os sistemas de provisão são importantes (figura adaptada de [6]).

capítulo 3

# PRESTO: Requisitos do produto

O sistema a desenvolver no âmbito desta dissertação pretende facilitar a execução de testes de aceitação pelo utilizador de forma a validar novas funcionalidades desenvolvidas. Esta deve de permitir de uma forma automática a preparação dos ambientes *staging* (através de *pull requests*) e a inicialização destes mesmos ambientes por parte do utilizador, para que este possa interagir com a funcionalidade que pretende testar.

## 3.1 visão geral da solução pretendida

O atual ciclo de desenvolvimento utilizado pela Beubi é demonstrado através do diagrama de atividades UML [25] da figura 3.1:

1. O gestor do projeto cria uma nova tarefa na ferramenta de gestão do projeto.
2. A tarefa é atribuída ao programador ou equipa encarregues de a desenvolver.
3. Paralelamente ao início do desenvolvimento de uma funcionalidade é criado um novo ramo no sistema de gestão de código, seguindo o modelo do Gitflow.
4. A funcionalidade começa a ser desenvolvida bem como os testes unitários e funcionais que permitirão validá-la.
5. Quando o programador ou equipa encarregues de desenvolver essa funcionalidade se sentem confiantes que terminaram o seu trabalho, criam um *pull request* através da ferramenta de gestão de código. Este por sua vez desencadeará três atividades que têm como objetivo a validação e verificação do trabalho realizado.
   a) É criado um ambiente de *staging*, que permite a execução de UATs.
   b) São executados testes funcionais e unitários sobre o código desenvolvido.
   c) É executada uma ferramenta de análise de código que permite verificar se o código desenvolvido segue todos os *coding standards* definidos.

6. Se alguma destas três validações falhar, o programador ou equipa corrigem os erros detetados e submetem o código novamente para validação. Isto é repetido até ser conseguida a aprovação em todos os testes.
7. Depois de validado e verificado, o *pull request* é aprovado pelo gestor do projeto e o código desenvolvido é junto com o ramo principal do sistema de versionamento, a aplicação é provisionada no ambiente de produção e a tarefa é dada como terminada pelo gestor do projeto.

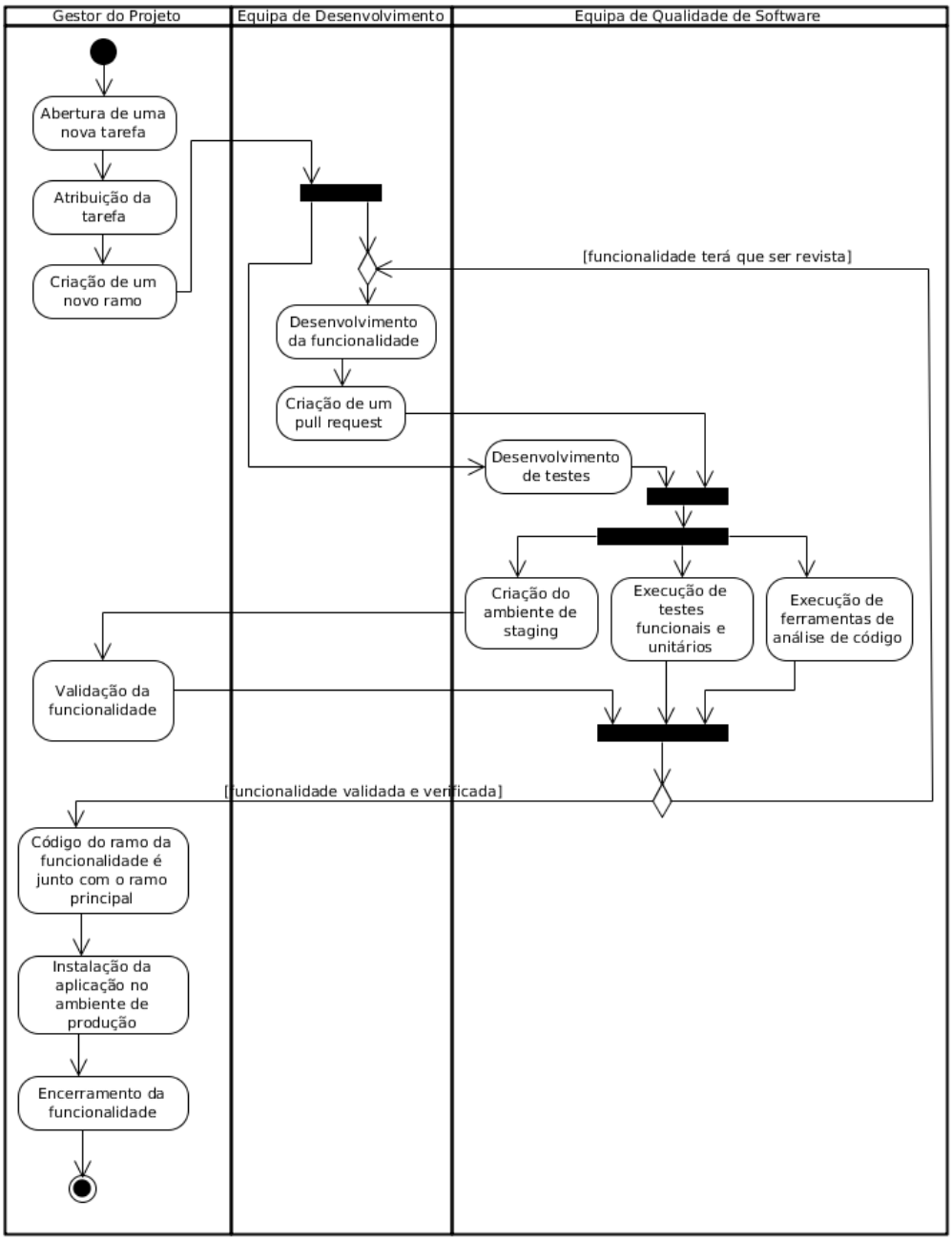



Figura 3.1: Diagrama de atividades do ciclo Continuous Delivery na empresa Beubi.

A ferramenta a desenvolver no âmbito desta dissertação foca-se nas atividades desde a criação do *pull request* por parte da equipa de desenvolvimento, passando pela criação do ambiente de *staging*, até à validação por parte do gestor do projeto ou cliente.

Pretende-se que depois de despoletada a ação de *pull request* no sistema de versionamento de código, o sistema Presto dê início à provisão do ambiente de *staging* para que quando o gestor do projeto ou cliente estiver disponível para testar essa funcionalidade apenas tenha que inicializar esse ambiente.

Os objetivos desta ferramenta são por isso:

- Automatizar o processo de criação do ambiente de *staging*, que deve de evoluir de acordo com a versão do projeto a testar.
- Reduzir o tempo que o gestor do projeto ou cliente tem de esperar até que o ambiente de *staging* esteja pronto.
- Reduzir o tempo de aceitação da funcionalidade permitindo que programador ou equipa responsáveis tenham *feedback* mais rapidamente.
- Utilização de diferentes meios de virtualização e ferramentas de provisão de forma a satisfazer projetos com diferentes requisitos.

O *workflow* geral da aplicação representado na figura 3.2 é o seguinte: o gestor de projeto começa a utilização da nossa aplicação adicionando o projeto à plataforma, sempre que for criado um *pull request* por parte do sistema de gestão de código a plataforma recebe essa informação e dá início à provisão do ambiente de *staging*. Esta provisão está separada em três fases (atualização dos ficheiros do projeto, criação do sistema de virtualização e provisão do sistema), quando o gestor do projeto quiser dar início ao processo de validação da funcionalidade apenas tem que iniciar o ambiente de *staging* e aceder à funcionalidade que pretende testar.

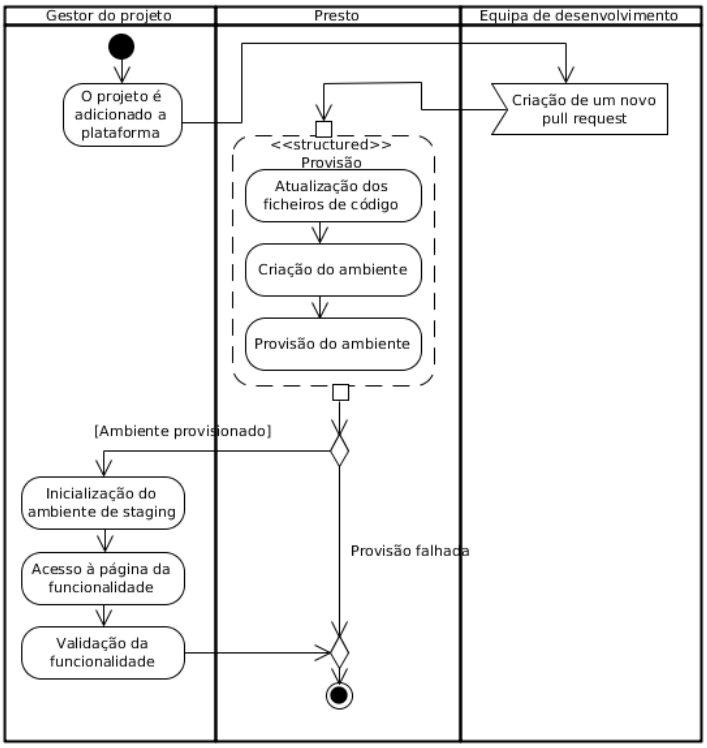



Figura 3.2: Diagrama de atividades do sistema Presto.

## 3.2 CASOS DE UTILIZAÇÃO

O diagrama de casos de utilização apresentado na figura 3.3, representa as interações possíveis que o gestor de projetos tem com o nosso sistema.

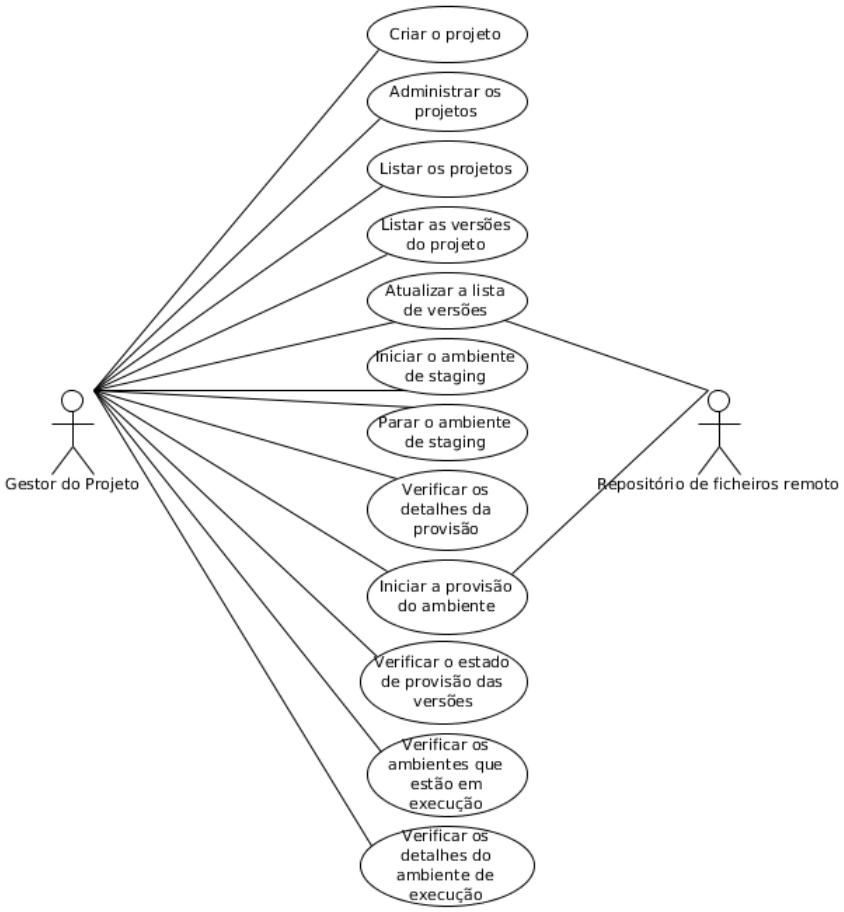



Figura 3.3: Diagrama de casos de uso do sistema Presto.

| Atores | |
|---|---|
| Gestor do projeto | Utilizador do sistema Presto, a sua função na plataforma é gerir os projetos que tem à sua responsabilidade. |
| Repositório remoto | Sistema externo que despoleta a ação de provisão de um ambiente de execução. |

Tabela 3.1: Atores do sistema Presto.

| Casos de Uso | |
|---|---|
| Criar o projeto | O gestor do projeto será capaz de registar um novo projeto. Um projeto no Presto corresponde a um produto atualmente em desenvolvimento e inclui: o endereço do repositório onde os ficheiros estão localizados, o nome do projeto, uma descrição sumária e detalhes específicos da ferramenta de orquestração e provisão utilizada. |
| Administrar os projetos | Será possível ao gestor do projeto gerir os projetos adicionados, englobando ações como editar e remover os projetos. |
| Listar os projetos | Na página principal será possível ver uma listagem de todos os projetos que foram registados na plataforma. |
| Listar as versões do projeto | Na página de detalhes do projeto será possível ver uma lista de versões que poderão ser validadas pelo cliente ou gestor do projeto. Estas versões serão identificadas pelo nome da funcionalidade que pode ser testada. |
| Atualizar a lista de versões | O gestor do projeto será capaz de atualizar a lista de versões do projeto, mantendo a sincronização entre o repositório local e o repositório remoto. |
| Iniciar o ambiente de *staging* | A partir da listagem das funcionalidades o gestor do projeto poderá iniciar o ambiente de *staging*. |
| Parar o ambiente de *staging* | A partir da listagem das funcionalidades o gestor do projeto poderá parar o ambiente de *staging*. |
| Verificar os detalhes da provisão | Depois e durante a provisão do ambiente de *staging* será possível ver um registo que mostre as operações realizadas nesse ambiente. |
| Iniciar a provisão do ambiente | Existirão duas formas de iniciar a provisão do ambiente de *staging*, esta poderá ser despoletada pelo gestor do projeto ou então iniciada através da ação de *pull request* do repositório de código remoto. |
| Verificar o estado da provisão das versões | Para cada versão, de cada projeto, será possível verificar o seu estado de provisão. |
| Verificar os ambientes que estão em execução | O gestor do projeto será capaz de identificar as versões que atualmente estão em execução na plataforma. |
| Verificar os detalhes do ambiente de *staging* | Dependendo do tipo de sistema de virtualização escolhido será possível, através da plataforma verificar detalhes relativos aos ambientes de *staging*. |

Tabela 3.2: Casos de uso do sistema Presto.

## 3.3 REQUISITOS TRANSVERSAIS

Apresentamos de seguida alguns requisitos da solução, transversais aos vários casos de utilização:

- O processo de provisão deverá ser totalmente automatizado e suportado através das configurações do projeto e ficheiros presentes no sistema de versionamento de código.
- A plataforma deverá ser capaz de detetar quando a provisão do ambiente de *staging* não foi corretamente executado e dar informações suficientes que permitam a identificação do problema.

- O processo de provisão deverá garantir que todas as configurações necessárias à execução da funcionalidade são aplicadas nesta fase garantido que quando o gestor do projeto inicializar o ambiente de *staging* este processo requer o menor tempo possível.
- As tecnologias utilizadas no desenvolvimento da plataforma devem ser suportadas em sistemas operativos Linux.
- O sistema operativo Linux onde estará instalada a plataforma deverá suportar a execução de sistemas de virtualização ao nível do sistema operativo.
- Deverá ser utilizada uma linguagem de programação que permita a execução de comandos do terminal Linux.
- A plataforma deverá de ser compatível com diferentes tipos de sistemas de virtualização, provisão e orquestração.
- As operações que necessitem mais de 30 segundos para serem executadas devem ser feitas assincronamente.

capítulo 4

# Arquitetura do sistema Presto

A arquitetura da solução do Presto está condiciona pelas ferramentas selecionadas para suportar a criação dos ambientes de *staging*, sendo por isso necessário primeiramente abordar as ferramentas escolhidas no âmbito do desenvolvimento desta dissertação para posteriormente discutir a sua organização.

## 4.1 seleção de ferramentas

A solução a ser desenvolvida é composta por diversas ferramentas, que permitem que o sistema Presto seja capaz de automatizar o processo de criação do ambiente de *staging* depois utilizado pelo gestor do projeto ou cliente para validar a funcionalidade desenvolvida.

Relativamente aos sistemas de virtualização, foram escolhidos o Docker e o LXC, ambos funcionam com virtualização ao nível do sistema operativo o que permite a criação de ambientes de *staging* mais leves e que necessitam de menos recursos relativamente às típicas máquinas virtuais. Para além disso, os *containers* LXC são o meio de virtualização atualmente utilizado na Beubi e o Docker é uma ferramenta que está atualmente a ter uma forte adesão, pois contém um conjunto de funcionalidades que permitem a interação com os *containers* Docker mais alargado que o sistema LXC. No âmbito desta dissertação optou-se por construir a solução com possibilidade de usar ambos os sistemas.

Sobre a forma como os *containers* são provisionados no caso específico do Docker, é obrigatória a utilização de ficheiros Dockerfile, este Dockerfile pode depois ser utilizado para lançar outras ferramentas de provisão como por exemplo o Chef. O Chef é a ferramenta atualmente utilizada pela Beubi e que permite a provisão dos seus ambientes de execução e daí também termos explorado a sua utilização. O Ansible, nativamente, para além da provisão também permite a orquestração do ambiente que suporta a execução da aplicação, a sua utilização foi requisitada pela empresa que acolheu esta dissertação de forma a podermos explorar as suas versatilidades. O Docker Compose é uma ferramenta oficial que permite a orquestração de *containers* Docker, sendo por isso também interessante a sua exploração e

utilização no contexto desta ferramenta de modo a evitar a execução de comandos demasiado extensos e que não representam a forma correta de lançar estes ambientes mais complexos.

Quanto ao sistema de versionamento de código a escolha recaiu sobre o Git que permite o versionamento dos ficheiros tanto localmente como remotamente e é usado no contexto da empresa. Para além disso, o sistema de versionamento remoto, tem que ser capaz de comunicar com um sistema externo (ao repositório) sempre que um *pull request* é criado (de forma a despoletar o processo provisão), para esse efeito foi escolhido o Bitbucket já que permite a criação de *hooks* e é também o repositório utilizado pela Beubi.

A persistência dos dados, que é mínima para a gestão dos projetos registados, é garantida através de uma base de dados MySQL [26].

## 4.2 COMPONENTES DA SOLUÇÃO

A solução a ser desenvolvida será então composta por diversas ferramentas: Docker, LXC, Chef, Ansible, Docker Compose e Git. Estes são os sistemas fundamentais que permitirão a configuração e criação dos ambientes de *staging* com uma versão específica do projeto em que foi desenvolvida a funcionalidade que queremos testar.

A camada de apresentação Presto deve funcionar como uma aplicação *web*. Para isso, pretende-se usar as ferramentas principais da Beubi que desenvolve principalmente em PHP [27].

O Symfony 2 [28] é uma *framework* de PHP que segue uma arquitetura MVC, contendo componentes importantes que facilitam a organização da nossa solução e a interação com todos os sistemas externos à nossa plataforma (Git, LXC, Docker, Docker Compose e Ansible). O Symfony usa as abstrações Serviço, Comandos, Processos e Repositório, como se define a seguir:

- Os Serviços em Symfony 2 são classes em que são definidos métodos de um propósito geral no sistema. Por exemplo, no caso da provisão do ambiente de *staging*, o método responsável pela provisão recebe os id's do projeto e da versão do projeto, estes identificadores permitem saber qual o tipo de projeto e o comando correto que deve ser utilizado para criar e provisionar o ambiente de *staging*.
- Os Comandos em Symfony 2 são classes que permitem criar comandos que podem ser executados através do terminal. Este tipo de classe é especialmente útil porque permite executar funcionalidades sem recorrer à interface *web* e para além disso, quando utilizados em conjunto com os Processos permitem executar tarefas do nosso sistema de uma forma assíncrona.
- O Processo é um componente do Symfony 2 que permite a execução de comandos do terminal, de forma síncrona ou assíncrona e permite obter o resultado desse comando. É através deste componente que a nossa plataforma comunica com os sistemas externos, formando os *wrappers*.
- O Repositório em Symfony 2 é uma classe onde são declarados vários métodos que permitem a execução de operações sobre a base de dados.

Em termos de persistência é utilizado o Doctrine [29] que é uma biblioteca ORM que permite o mapeamento entre objetos e o esquema relacional da base de dados.

A interação entre os componentes do sistema, pode ser observado no diagrama da figura 4.1. Existem duas formas de comunicar com o sistema Presto: através da interface *web* ou pela linha de

comandos. Estes são os dois pontos de entrada do nosso sistema, através dos quais o utilizador pode interagir.

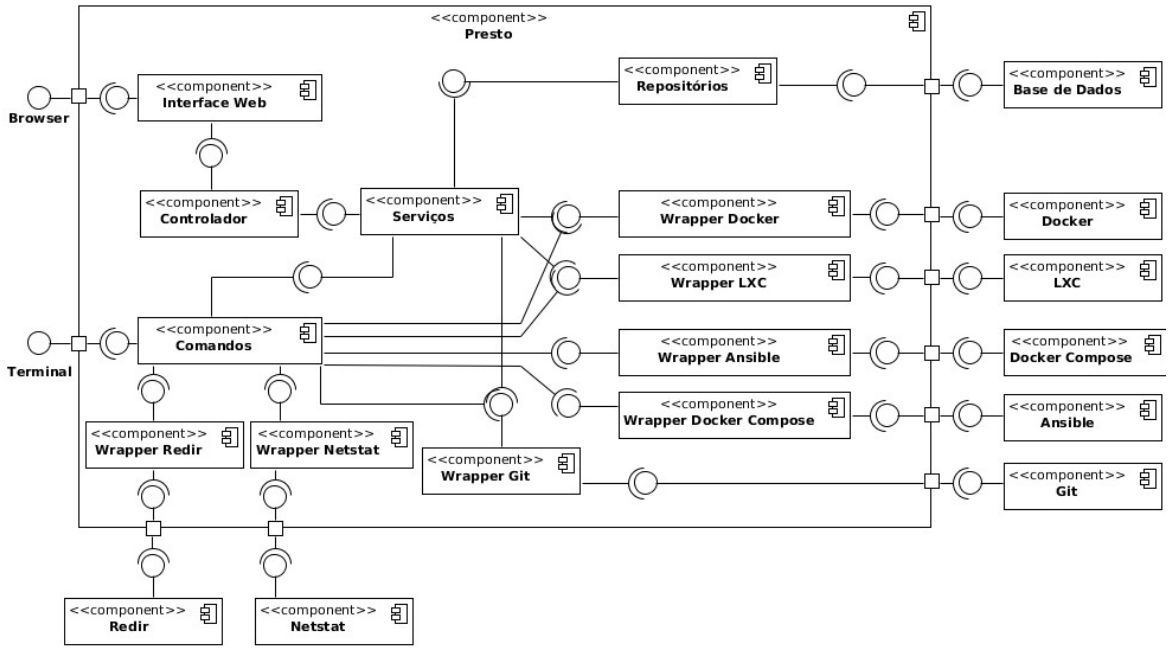



Figura 4.1: Diagrama de componentes do sistema Presto.

O controlador recebe os pedidos despoletados pela interface *web*, esta informação é passada ao serviço correto e este é o responsável por obter as informações pedidas ou executar a operação desejada.

Dependendo do pedido, o serviço pode utilizar diferentes componentes; se este necessitar de informações da base de dados comunica com o repositório, se necessitar de informações relativas aos sistemas de virtualização comunica diretamente com o *wrapper* do LXC ou do Docker. Para operações mais complexas como criar e provisionar o ambiente de *staging* é necessário utilizar comandos de forma a que este processo seja assíncrono e para que a sua execução seja possível através do terminal.

Os comandos abstraem todo o processo de provisão, inicialização e paragem dos sistemas de virtualização, interagindo com os diferentes componentes que permitem estas operações.

Os *wappers* Docker e LXC controlam as operações relativas aos sistemas de virtualização como a sua inicialização, paragem e permitem obter informações específicas dos *containers*. No caso do LXC, estas informações permitem identificar os *containers* existentes no sistema, se os mesmos estão em execução e o IPv4 e IPv6, que permitem o posterior acesso ao *container*. No caso dos *containers* Docker existem mais informações que podemos obter como o tamanho das imagens, há quanto tempo foram construídas e o seu nome, há quanto tempo o *container* foi criado, está inicializado e que portos estão mapeados entre o sistema operativo base e o *container*. Todas estas informações descritas anteriormente são disponibilizadas através da plataforma e são obtidas através dos *wrappers* respetivos.

Os *wrappers* do Ansible (ao nível da orquestração) e do Docker Compose são dois componentes que permitem a construção de múltiplas imagens e a inicialização de múltiplos *containers* a partir de ficheiros de configuração. Desta forma é eliminada a necessidade de executar múltiplos comandos Docker para cada uma das imagens ou *containers* que formem o ambiente de *staging* da nossa aplicação. O *wrapper* Docker e LXC apenas permite controlar a execução de um único *container*.

No caso dos *containers* Docker, o Chef e o Ansible (ao nível da provisão) não são controlados explicitamente através da plataforma, a execução destas ferramentas é feita através dos ficheiros

Dockerfile quando a imagem está a ser criada. No caso dos *containers* LXC, o Chef e o Ansible são executados explicitamente através da execução do comando do sistema de provisão específico sobre o *container*.

O *wrapper* Git permite selecionar diferentes versões da aplicação e atualizar os ficheiros locais a partir do repositório de código. Partimos do pressuposto que cada versão da aplicação que queremos testar é identificada no sistema de versionamento de código através do prefixo *feature/*. Seguindo o modelo do GitFlow, cada ramo do nosso repositório de código que tiver o prefixo *feature/* representará uma versão da aplicação em que existe uma nova funcionalidade para ser testada e são estas versões da aplicação que interessam serem testadas num ambiente de *staging*.

O *wrapper* Netstat é utilizado para verificar se um dado porto TCP está livre no sistema, sempre que um novo ambiente de *staging* é executado, é encontrado um novo porto livre através deste comando. O Netstat[30] é utilizado em conjunto com o Redir[31] no contexto de utilização dos *containers* LXC de forma a mapear um porto entre o sistema base e o *container*.

Todos os componentes aqui descritos foram desenvolvidos em PHP e utilizando as versatilidades da framework Symofony 2.

De forma a tornar mais explícita a interação entre os diferentes *containers* iremos utilizar o exemplo do acesso à página de detalhes do projeto. A informação que esta página apresenta pode ser observada na figura 5.27. No diagrama da figura 4.2 são representadas as operações necessárias para a obter.

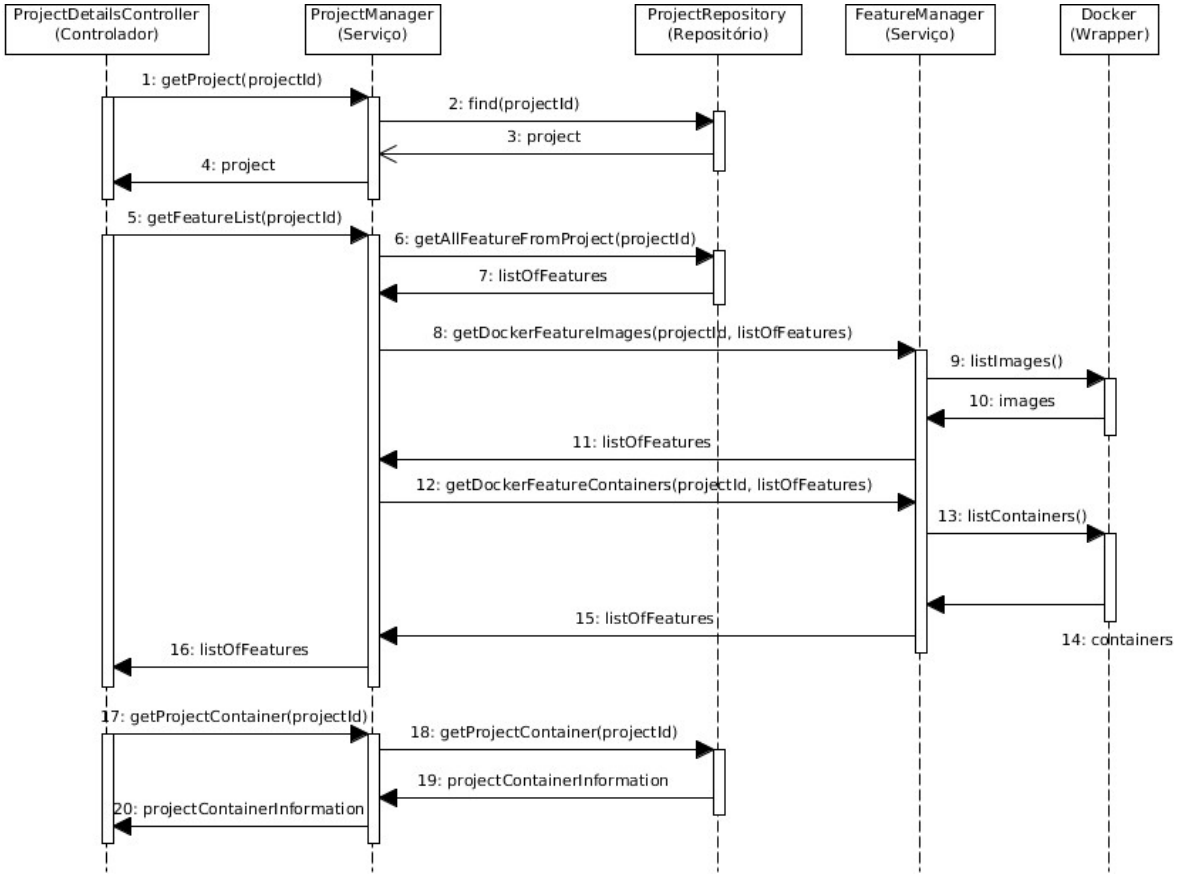



Figura 4.2: Diagrama de sequência das operações necessárias para obter a informação para listar os detalhes de um projeto.

Na primeira metade da página é possível ver diversas informações relativas ao projeto, estas são

persistidas na base de dados sendo que por isso o serviço comunica com o repositório específico para obter essas informações.

Na segunda metade existe uma lista de funcionalidades, estas representam diferentes versões do projeto, o nome de cada versão do projeto permite identificar a funcionalidade a testar. Temos primeiro que obter a lista de funcionalidades do projeto (persistida na base de dados), em seguida e dependendo do sistema de virtualização utilizado, para cada funcionalidade temos que obter as imagens (que permitem determinar se a versão está provisionada ou não, apenas utilizado no caso do Docker) e os *containers* (que permitem determinar se o ambiente de *staging* está em execução ou não) através do *wrapper* do Docker ou LXC.

capítulo 5

# Implementação

A plataforma Presto tem como peça central o servidor *web*, este é o responsável por interligar todos os restantes componentes que são responsáveis pela criação do ambiente de *staging*.

Todas as informações relativas aos projetos são guardadas num servidor MySQL, a aplicação efetua operações sobre este para obter dados sobre os projetos e as suas versões, estas informações permitem saber por exemplo, qual o sistema de virtualização utilizado, que porto devemos utilizar para aceder à aplicação ou no caso do ambiente de *staging* ter mais que um *container*, a qual deles devemos aceder, ou qual o endereço do repositório Git.

A plataforma comunica com o Git para obter todas as versões da nossa aplicação passíveis de serem testadas e garante que quando um novo ambiente de *staging* é provisionado, é utilizada a versão correta dos ficheiros e os mesmos estão atualizados com a versão mais recente, existente no repositório de código remoto.

Com as informações obtidas através da base de dados e com a versão correta dos ficheiros da aplicação conseguimos criar o ambiente de *staging* através dos sistemas de virtualização, provisão e orquestração. Dependendo do sistema de virtualização, existem várias formas do meio de execução ser provisionado.

No caso dos *container* Docker, são construídas imagens através dos ficheiros Dockerfile. A imagem pode ser provisionada unicamente através do Dockerfile ou utilizar, em conjunto com o Dockerfile, outras ferramentas de provisão como o Chef ou o Ansible, utilizando imagens que já estão preparadas para executar estas ferramentas.

Utilizando o Docker, é ainda possível utilizar duas ferramentas de orquestração, o Docker Compose e o Ansible. Estas ferramentas permitem criar um ambiente *multi-container* através de ficheiros YAML e lançá-lo utilizando apenas um comando. A não utilização destas ferramentas em ambientes *multi-container*, faz com que seja necessária a utilização de comandos Docker complexos.

No caso dos *containers* LXC, este não tem a noção de imagens como o Docker, porém o comando `lxc-clone` permite a reutilização dos primeiros. O LXC também não tem nenhum meio de provisão nativo como o Dockerfile no Docker, este processo de provisão é conseguido através da execução de comandos específicos das ferramentas Chef e Ansible dentro do *container*.

No caso específico do Presto decidimos criar um *container* LXC base em que estas duas ferramentas já estão instaladas. Quando queremos criar um novo ambiente de *staging* é criada uma cópia do

*container* base, o que depois permite executar qualquer uma das ferramentas para provisionar o *container*.

Para que seja possível aceder à aplicação provisionada no ambiente de *staging* é necessário o mapeamento de pelo menos um porto entre o sistema de virtualização e o sistema base onde é executado a nossa plataforma, isto permite o acesso a este ambiente a partir de um ponto remoto.

O Netstat é uma aplicação de linha de comandos que permite listar as ligações TCP ativas, desta forma é possível verificarmos se um dado porto está ou não a ser utilizado.

Os *containers* LXC ao contrário do Docker não tem nenhum comando que permita o mapeamento dos portos entre o sistema de virtualização e o sistema base, por isso foi necessário recorrer a outra aplicação que fizesse esse mapeamento por nós, o Redir.

## 5.1 INTERAÇÃO ENTRE COMPONENTES NA PROVISÃO DE AMBIENTES DE STAGING

Iremos demonstrar em seguida através de diagramas de sequência as principais operações do nosso sistema que são a provisão, inicialização e paragem dos ambientes de execução através das diferentes ferramentas (Docker, Docker Compose, Ansible e LXC).

Para melhor expor como o Presto funciona, iremos utilizar como projeto exemplo o Opencart. Esta plataforma *ecommerce* será executada utilizando os diferentes comandos criados no âmbito desta dissertação. Os principais requisitos desta plataforma são um servidor *web* como o Apache 2 e o MySQL Server.

### 5.1.1 ELEMENTOS COMUNS

O processo inicial das ações de provisão, inicialização e paragem dos ambientes de *staging*, despoletados a partir da interface *web*, é demonstrado através do diagrama da figura 5.1.

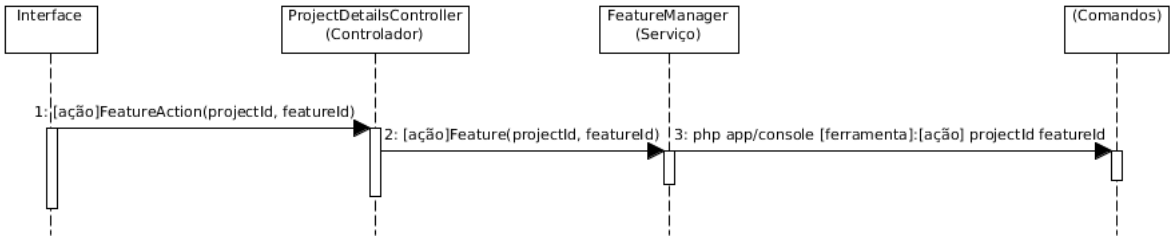



Figura 5.1: Diagrama sequência das principais operações do sistema, despoletadas a partir do navegador.

O controlador recebe o id do projeto e o id da funcionalidade (versão do projeto) e chama o método do serviço (FeatureManager) respetivo, este verifica o tipo de projeto e chama o comando específico.

Existem algumas verificações que são comuns em qualquer comando que seja executado como pode ser observado pelo diagrama da figura 5.2:

- É verificado se os id's passados como argumentos representam projetos e funcionalidades (versões do projeto) existentes no sistema.
- É verificado se o projeto é de um tipo válido, isto é, se por exemplo o projeto utilizar o Docker Compose o comando executado terá que utilizar também esse tipo de ferramenta. Esta verificação é importante quando os comandos são executados diretamente do terminal.

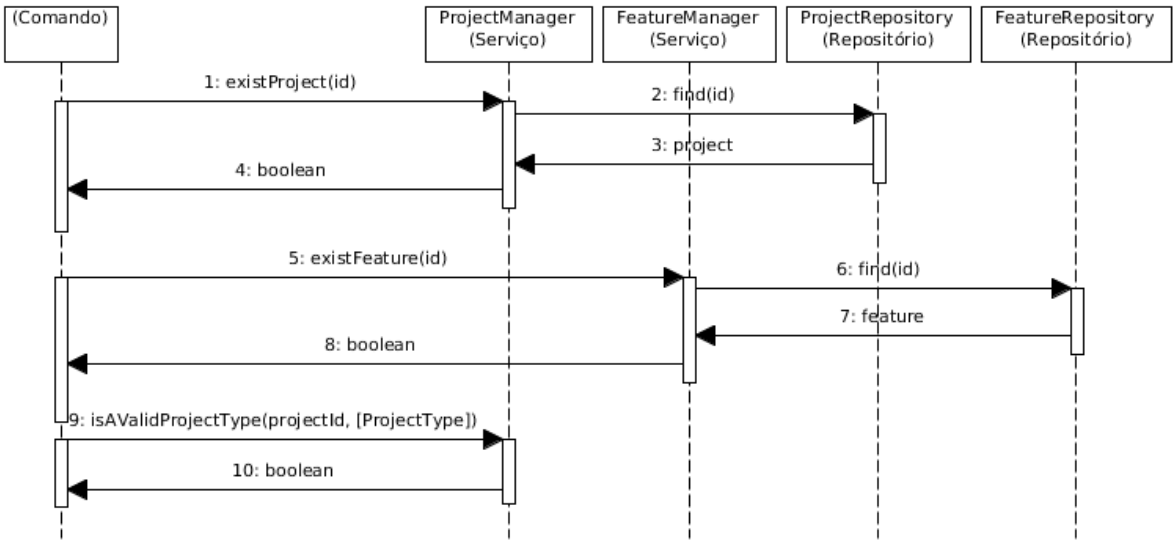



Figura 5.2: Diagrama sequência das operações comuns em todos os comandos.

No caso específico dos comandos de provisão existem mais algumas operações que são comuns às 4 ferramentas como pode ser visto pelo diagrama da figura 5.3:

- É reiniciado o registo da provisão.
- É feito o *checkout* da versão correta dos ficheiros.
- A versão dos ficheiros local é atualizada com o repositório remoto através de um *fetch*
- É feito um *reset* entre o ramo local e o ramo remoto origin/feature/{nome da funcionalidade}, o qual foi atualizado no passo anterior.

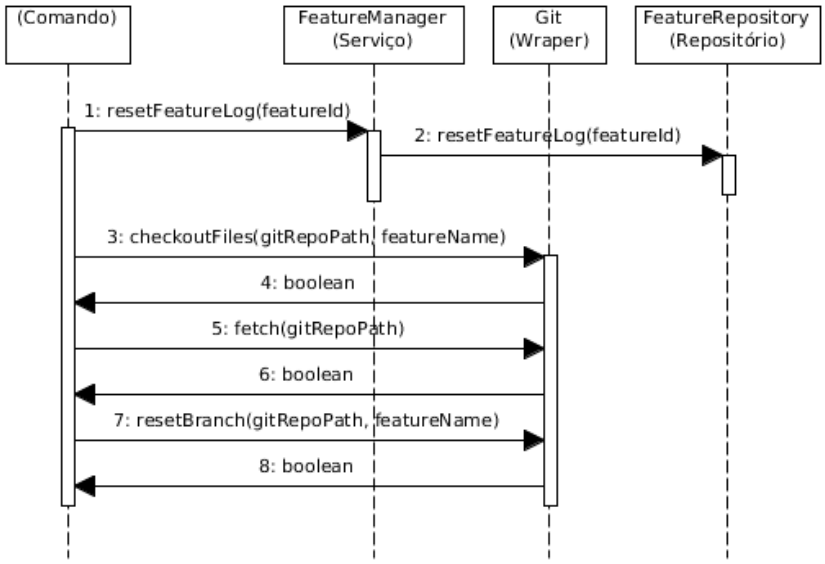



Figura 5.3: Diagrama sequência das operações comuns a todos os comandos de provisão.

### 5.1.2 DOCKER

O comando Docker foi desenvolvido para projetos pequenos que necessitem apenas de um *container* e como tal a sua utilização tem algumas limitações.

Cada imagem e *container* Docker são identificados pela nossa plataforma através da concatenação do id do projeto com o nome da funcionalidade, tal como pode ser observado pela figura 5.4 e 5.6. Esta nomenclatura permite identificar as imagens e os *containers* (assim como os atributos associados a cada um deles) de cada projeto e versão específicas.

```
ubuntu@ip-10-2-0-89:~$ docker images
REPOSITORY                    TAG                 IMAGE ID            CREATED             VIRTUAL SIZE
20carmanagement               latest              9a3c25d59b7f        About a minute ago  464.5 MB
14testbasewebsite_web         latest              9dc26b52d222        5 minutes ago       326 MB
14testbasewebsite_db          latest              c7adab314cdd        5 minutes ago       344.3 MB
ubuntu                        14.04               07f8e8c5e660        3 weeks ago         188.3 MB
ubuntu                        latest              07f8e8c5e660        3 weeks ago         188.3 MB
```

Figura 5.4: Identificação das imagens Docker.

#### PROVISÃO

As imagens são configuradas através dos ficheiros Dockerfile que são versionados juntamente com o restante código desenvolvido.

Pegando no exemplo de configuração do Opencart, todas as configurações que têm que ser aplicadas no *container* são descritas por este ficheiro, isto inclui a cópia dos ficheiros do projeto, instalação de pacotes, alteração dos ficheiros de configuração (como por exemplo do Apache ou MySQL) e definição dos serviços que têm de ser executados quando o *container* for iniciado.

Depois das operações ilustradas na figura 5.3 o passo seguinte da provisão, apresentado pelo diagrama da figura 5.5, é criar a imagem Docker através da invocação do comando `docker build` no terminal, utilizando o *wrapper* previamente criado, onde é necessário especificar a localização do ficheiro Dockerfile e o nome a dar à imagem.

Um exemplo de invocação de um comando do terminal Linux utilizando o componente *process* da *framework* Symfony 2 é mostrado a seguir:

```
$process = new Process('docker␣build␣-t␣' . $dockerImageName . '␣' .
    $gitSourcePath);
$process->setTimeout(3600);
$process->run();

if ($process->getExitCode() != 0) {
  return new DockerResponse(DockerResponse::EXIT_CODE_ERROR,
      $process->getOutput());
}

return new DockerResponse(DockerResponse::OK, $process->getOutput());
```

Código 11: Exemplo da invocação do comando `docker build` através do componente *process* do Symfony 2

Quando a provisão do *container* termina todo o *output* gerado por este é persistido na base de dados para consulta futura.

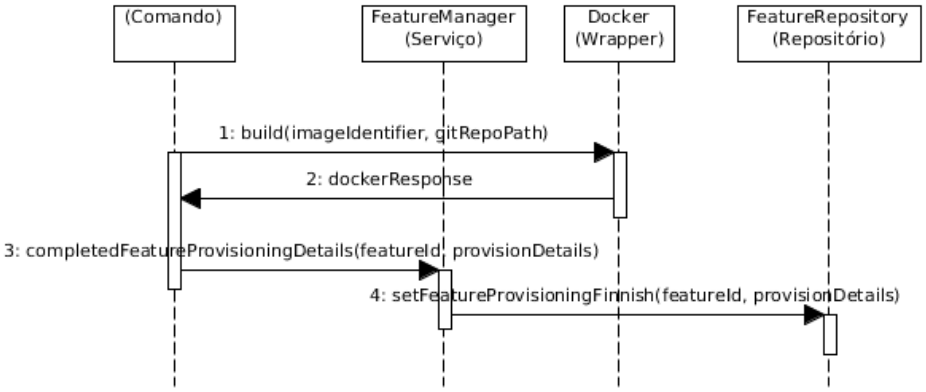



Figura 5.5: Diagrama sequência das operações utilizadas para a provisão do *container* Docker.

Este processo poderá ser lançado através do terminal utilizando o seguinte comando: `php app/console docker:build id_do_projeto id_da_versao`.

### INICIALIZAÇÃO

O processo de inicialização de um *container* é descrito através do diagrama 5.8, a primeira operação a ser realizada é verificar se a versão do projeto já foi provisionada, depois disto verificamos se já não existe um *container* Docker com aquela versão do projeto em execução, isto é feito obtendo uma lista dos *containers* que estão em execução e verificando senão existe nenhum *container* com o mesmo identificador da versão do projeto que queremos executar. O *container* é identificado através do seu nome, tal como é exemplificado na figura 5.6.

```
ubuntu@ip-10-2-0-89:~$ docker ps
CONTAINER ID        IMAGE                    STATUS              PORTS                     NAMES
bee52b6101f8        20carmanagement:latest   Up 12 minutes       0.0.0.0:11000->80/tcp     20carmanagement
```

Figura 5.6: Identificação dos *containers* Docker.

A operação seguinte é encontrar um porto TCP que esteja livre no sistema através do comando Netstat, que é utilizado para depois podermos aceder à aplicação a ser testada. O Presto através do seu ficheiro de configuração tem definido um conjunto de portos, que são utilizados para mapear o acesso entre o sistema base e o serviço disponibilizado pelo *container*. O comando Netstat é utilizado em conjunto com o grep para filtrar e encontrar os portos ocupados no sistema (figura 5.7).

```
joao@joao-K54HR ~ $ sudo netstat -atnp | grep 11000
tcp6       0      0 :::11000                :::*                    LISTEN      13533/docker-proxy
joao@joao-K54HR ~ $ sudo netstat -atnp | grep 11001
joao@joao-K54HR ~ $ docker ps
CONTAINER ID        IMAGE                    COMMAND                  PORTS                     NAMES
68331b725b0c        77carmanagement:latest   "/bin/sh -c /docker-"    0.0.0.0:11000->80/tcp     77carmanagement
```

Figura 5.7: Exemplo da utilização do comando Netstat.

Depois destes dois passos, o *container* é executado através do comando `docker run`. Este receberá como argumentos o porto livre gerado anteriormente, o porto do serviço no *container* e o identificador da versão do projeto que serve para dar o nome ao *container* e identificar a imagem provisionada.

No exemplo do Opencart o porto do serviço seria o 80, referente ao Apache.

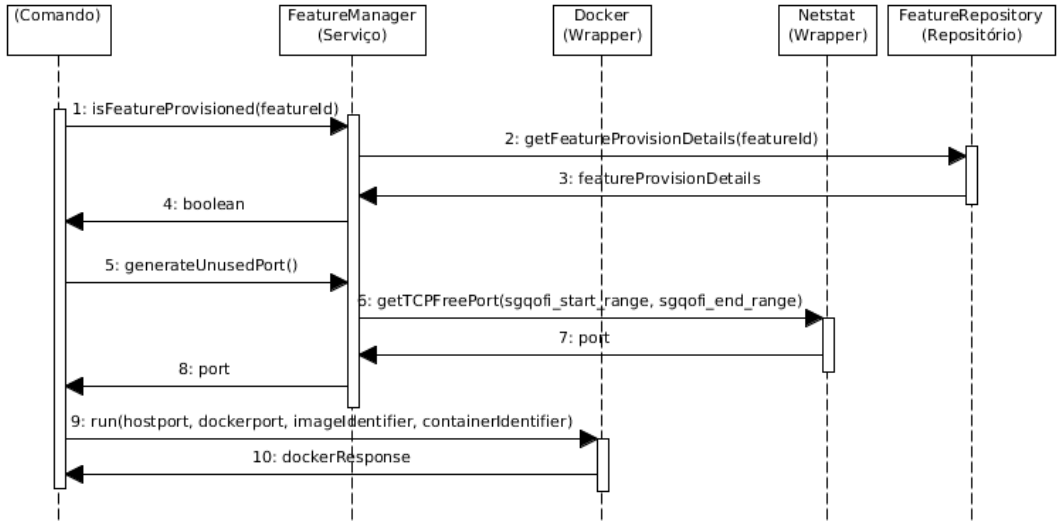



Figura 5.8: Diagrama sequência das operações utilizadas para a inicialização do *container* Docker.

Este processo poderá ser lançado através do terminal utilizando o seguinte comando: `php app/console docker:start id_do_projeto id_da_versao`.

PARAGEM

Para parar um *container* Docker, processo que é exemplificado no diagrama da figura 5.9, apenas é necessário obter o id (Docker) do *container*. Esse id é obtido através do identificador da versão do

projeto, utilizado como nome do *container*, tal como pode ser observado em 5.6, por fim executamos o comando `docker rm` para parar e remover o *container*.

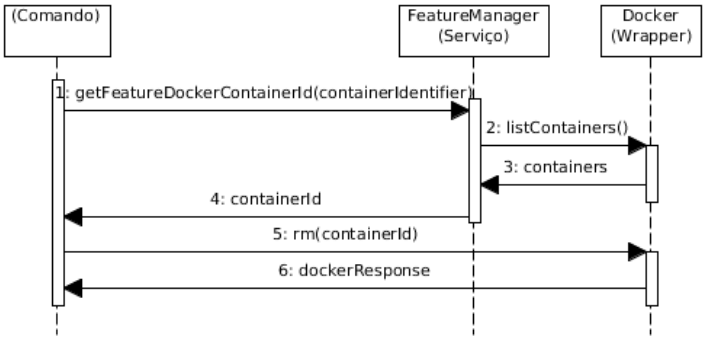



Figura 5.9: Diagrama sequência das operações utilizadas para a paragem do *container* Docker.

Este processo poderá ser lançado através do terminal utilizando o seguinte comando: `php app/console docker:stop id_do_projeto id_da_versao`.

### 5.1.3 DOCKER COMPOSE

Os comandos Docker Compose foram idealizados para serem utilizados em projetos em que o ambiente de *staging* é formado por mais que um *container* e por essa razão necessitam de configurações mais complexas, utilizando por exemplo múltiplos portos, volumes e ligações entre os *containers*.

Estas configurações são descritas através do ficheiro YAML, versionado com os restantes ficheiros do projeto. A partir deste ficheiro definimos os múltiplos *containers* e os seus atributos.

Utilizando o Docker Compose podemos identificar as imagens e os *containers* que pertencem a uma determinada versão do projeto devido ao facto do nome destas ter um prefixo (que por omissão é o nome da pasta do projeto) concatenado com o nome do *container*. Tanto o comando `docker-compose build`, `run` e `stop` permitem definir este prefixo. Nos comandos Docker definimos que o nome do *container* e da respetiva imagem de cada versão do projeto, seria a concatenação entre o id do projeto e o nome da funcionalidade, esta nomenclatura é utilizada também na solução do Docker Compose como prefixo, o que permite identificar os múltiplos *containers* e imagens que formam o ambiente de *staging* de uma dada versão do projeto tal como pode ser observado na figura 5.10.

```
ubuntu@ip-10-2-0-89:~$ docker images
REPOSITORY                    TAG                 IMAGE ID            CREATED             VIRTUAL SIZE
20carmanagement               latest              9a3c25d59b7f        About a minute ago  464.5 MB
14testbasewebsite_web         latest              9dc26b52d222        5 minutes ago       326 MB
14testbasewebsite_db          latest              c7adab314cdd        5 minutes ago       344.3 MB
ubuntu                        14.04               07f8e8c5e660        3 weeks ago         188.3 MB
ubuntu                        latest              07f8e8c5e660        3 weeks ago         188.3 MB
```

Figura 5.10: Identificação das imagens Docker criadas com o Docker Compose.

### PROVISÃO

Partindo do diagrama da figura 5.3 e tal como apresentado no diagrama da figura 5.11, é necessário primeiramente executar o comando `docker-compose build`, identificando a localização do ficheiro YAML e o identificador da versão do projeto.

No caso do Opencart o ficheiro YAML iria descrever dois *containers*: um para a base de dados e outro para o servidor *web*. Para cada um deles é necessário declarar a localização dos Dockerfiles responsáveis por provisionar cada um dos *containers*.

Depois de terminado o processo de provisão o *output* é persistido na base de dados para consulta futura.

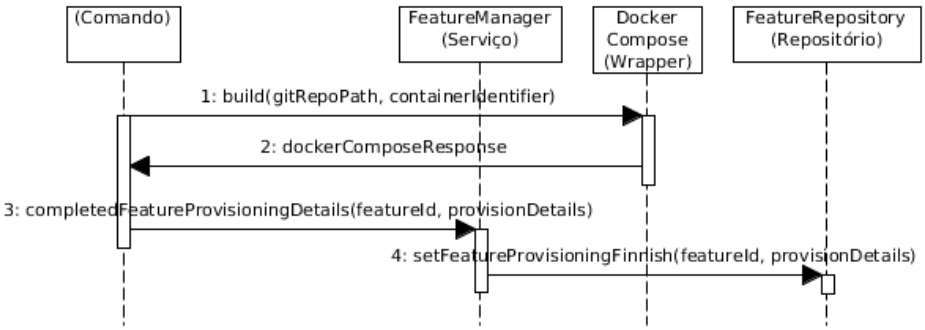



Figura 5.11: Diagrama sequência das operações utilizadas no comando Docker Compose que provisiona os *containers*.

Este processo poderá ser lançado através do terminal utilizando o seguinte comando: `php app/console docker_compose:build id_do_projeto id_da_versao`.

### INICIALIZAÇÃO

Tal como acontecia com o Docker e tal como exemplificado no diagrama da figura 5.12, antes de iniciarmos os *containers* verificamos primeiro se a versão do projeto que queremos executar já foi provisionada e se os mesmos já não estão em execução, depois disto é feito o *checkout* da versão correta dos ficheiros da nossa aplicação, isto porque diferentes versões podem ter ficheiros de configuração YAML diferentes e o comando `docker-compose run` faz uso deste para saber, por exemplo, que portos deve de mapear entre o sistema base e o *container*.

O próximo passo é então encontrar um porto livre, usando novamente o Netstat, que possa ser mapeado entre o sistema base e o *container* tal como descrito anteriormente.

Depois de termos a versão correta do ficheiro YAML temos que substituir todos os portos mapeados e identificar o *container* principal da aplicação e o porto de acesso ao serviço, este porto é então mapeado com o porto livre encontrado no passo anterior. Estas informações, relativas ao *container* principal e do porto do serviço são obtidos através da base de dados.

No contexto do Opencart, o *container* principal, ao qual pretenderíamos aceder seria o que estaria a executar o Apache 2 e o porto a mapear seria o 80.

A modificação do ficheiro YAML é fundamental porque geralmente este é criado e reutilizado em diferentes versões do projeto e se não houvesse este cuidado de substituir os portos acabaríamos por não conseguir executar mais que uma versão ao mesmo tempo.

O último passo para iniciar o ambiente de *staging* é executar o comando `docker-compose up` com o identificador da versão do projeto e indicando a localização do ficheiro YAML.

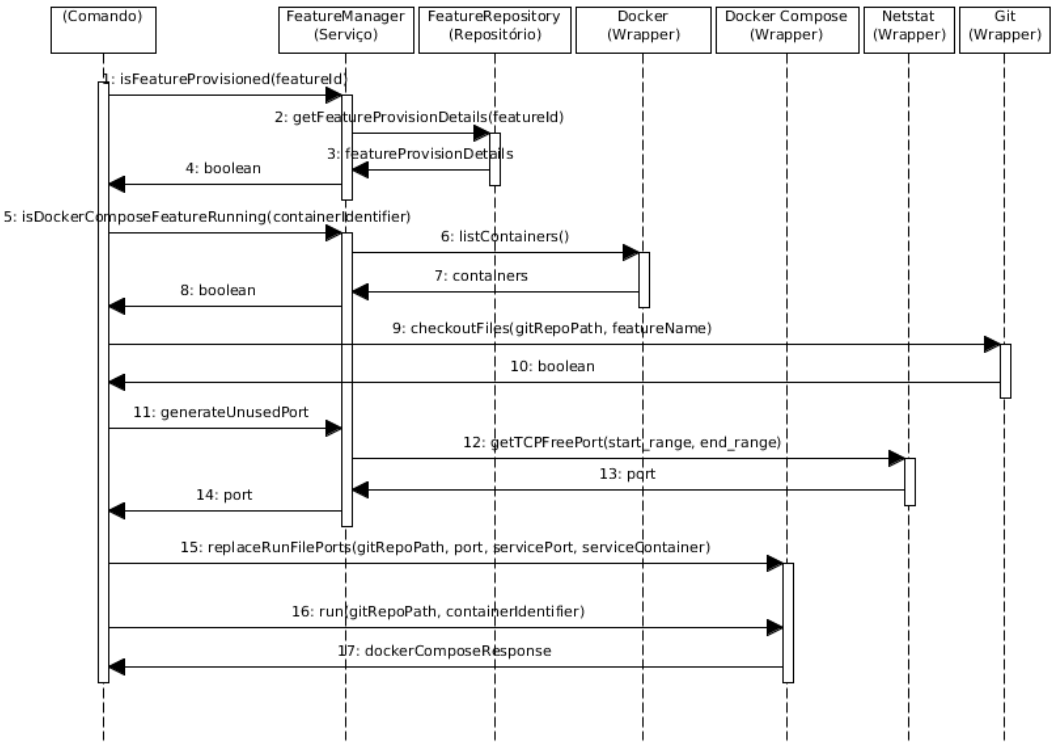



Figura 5.12: Diagrama sequência das operações utilizadas no comando Docker Compose que inicializa os *containers*.

Este processo poderá ser lançado através do terminal utilizando o seguinte comando: `php app/console docker_compose:start id_do_projeto id_da_versao`.

PARAGEM

No diagrama da figura 5.13, é descrito o processo para parar os *containers* do ambiente de *staging* utilizando o Docker Compose, primeiramente verificamos mais uma vez se os *containers* estão em execução e se o ambiente já foi provisionado.

O próximo passo é fazer o *checkout* dos ficheiros do projeto de forma a garantir que utilizamos o ficheiro de configuração YAML correto.

Por fim apenas precisamos chamar o comando `docker-compose stop`, indicando mais uma vez o prefixo e a localização do ficheiro de configuração.

Este processo poderá ser lançado através do terminal utilizando o seguinte comando: `php app/console docker_compose:stop id_do_projeto id_da_versao`.

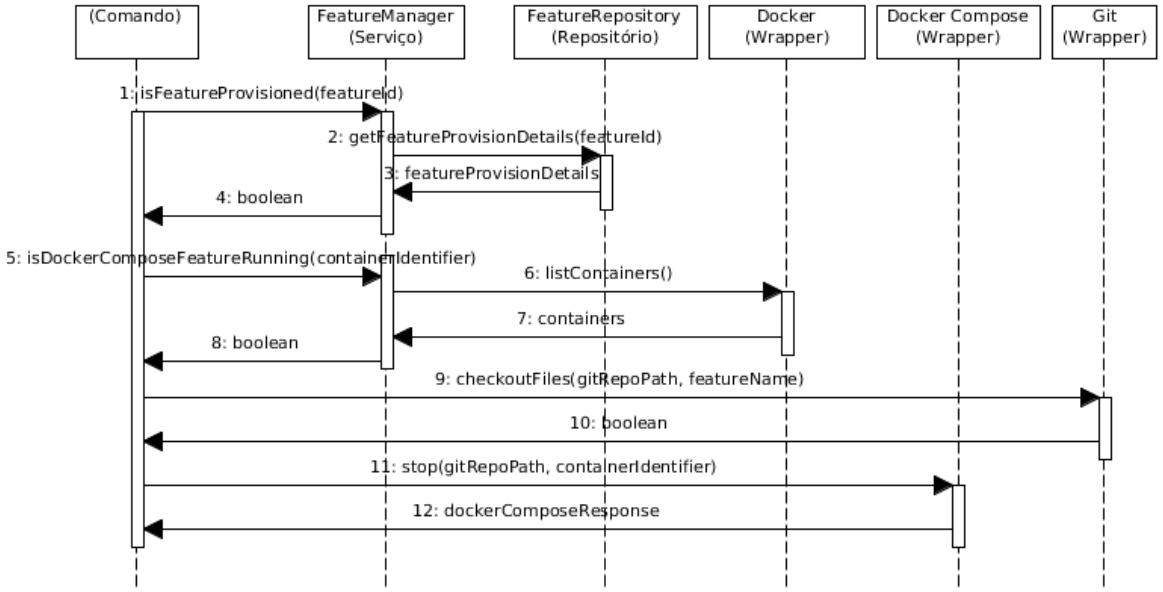



Figura 5.13: Diagrama sequência das operações utilizadas no comando Docker Compose que para os *containers*.

### 5.1.4 ANSIBLE

Apesar de o Ansible facilitar o orquestramento do ambiente de execução quando executamos múltiplos *containers* (tal como o Docker Compose), a utilização desta ferramenta requer a manipulação dos ficheiros YAML para atribuir os nomes às imagens e aos *containers* de forma a que o nome destes sigam o mesmo padrão utilizado nos dois outros casos anteriores. O segundo ponto em que o Ansible é diferente do Docker Compose e que torna a sua utilização mais complexa é que não existem comandos específicos para construir imagens, iniciar *containers* ou pará-los. Estas ações são descritas através dos ficheiros YAML, por esta razão é necessário definir múltiplos ficheiros de configuração YAML, um deles define e provisiona as imagens do nosso ambiente de execução, os outros dois definem os estados em que os *containers* podem estar, ligados ou desligados.

Uma vantagem do Ansible em relação ao Docker Compose é que não precisamos de verificar se os *containers* estão iniciados. A ferramenta faz essa verificação por nós e mesmo que executemos duas vezes o mesmo comando para inicializar os *containers* apenas uma instância será criada.

Com o Docker Compose podíamos definir o prefixo que pretendíamos utilizar através da opção `-p`, com o Ansible tal só é possível se manipularmos o ficheiro diretamente.

#### PROVISÃO

Partindo do diagrama 5.3, o processo de provisão das imagens utilizando o Ansible requer mais um passo adicional como se pode observar pela figura 5.14. Numa fase inicial alteramos o ficheiro de configuração YAML para que o nome das imagens siga o padrão definido, em seguida damos início ao processo de provisão através do comando `ansible-playbook` indicando o ficheiro correto.

No caso do Opencart foram definidas duas *tasks* para cada uma das imagens que tem de ser construídas. Nestas *tasks* estão definidas as localizações dos ficheiros Dockerfile que são usados para configurar o *container*.

Por fim, tal como acontecia nos outros dois casos anteriores o *output* é persistido na base de dados.

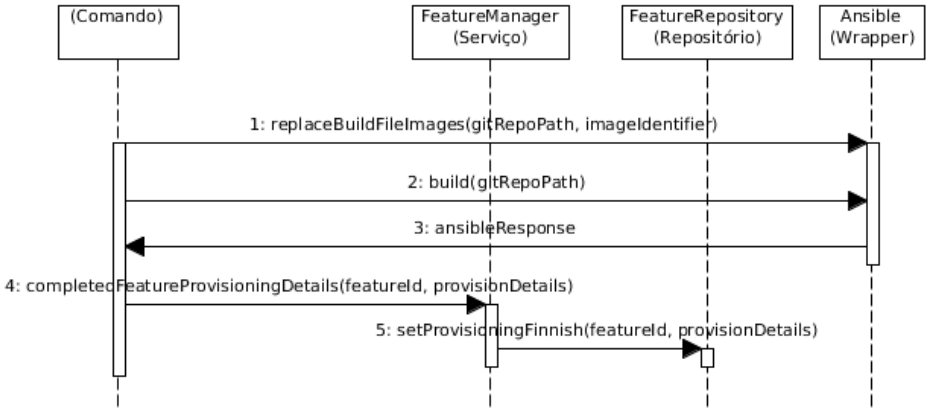



Figura 5.14: Diagrama sequência das operações utilizadas no comando Ansible que provisiona os *containers*.

Este processo poderá ser lançado através do terminal utilizando o seguinte comando: `php app/console ansible:build id_do_projeto id_da_versao`.

### INICIALIZAÇÃO

No diagrama da figura 5.15 são apresentadas as operações realizadas para inicializar os *containers* Docker através do Ansible e tal como acontecia com o Docker Compose, primeiro temos de nos certificar que as imagens respetivas do ambiente já foram criadas e estamos a utilizar a versão correta dos ficheiros do projeto fazendo o *checkout* dos mesmos.

O próximo passo é gerar mais uma vez um porto TCP livre através do Netstat, que depois nos vai permitir aceder à aplicação que queremos testar, uma vez obtido esse porto temos que substituir todos os portos mapeados e identificar o *container* principal e o porto de acesso ao serviço, o qual mapeamos com o anteriormente encontrado. No Opencart este é mais uma vez o *container* onde está instalado o Apache2 e o porto de acesso ao serviço é o 80.

Depois deste processo é necessário alterar mais uma vez o ficheiro YAML de forma a que nome das imagens e o nome dos *containers* siga o padrão definido.

Após este processo apenas temos que indicar qual o ficheiro de configuração correto e executar o comando `ansible-playbook`.

Este processo poderá ser lançado através do terminal utilizando o seguinte comando: `php app/console ansible:start id_do_projeto id_da_versao`.

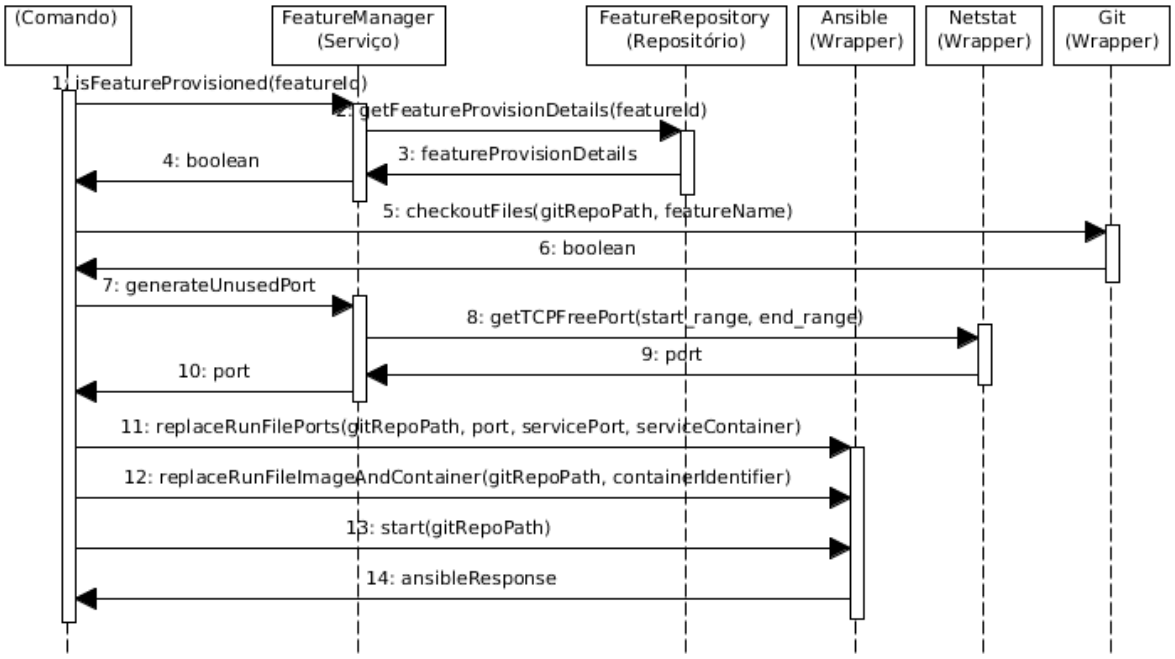



Figura 5.15: Diagrama sequência das operações utilizadas no comando Ansible que inicializa os *containers*.

PARAGEM

O processo de paragem dos *containers* Docker através do Ansible é apresentado no diagrama da figura 5.16. Numa primeira fase temos mais uma vez que fazer *checkout* da versão correta e modificar o ficheiro YAML respetivo para que os nomes dos *containers* sejam coerentes com o padrão definido.

Após este processo apenas temos que indicar qual o ficheiro de configuração correto e executar o comando `ansible-playbook`.

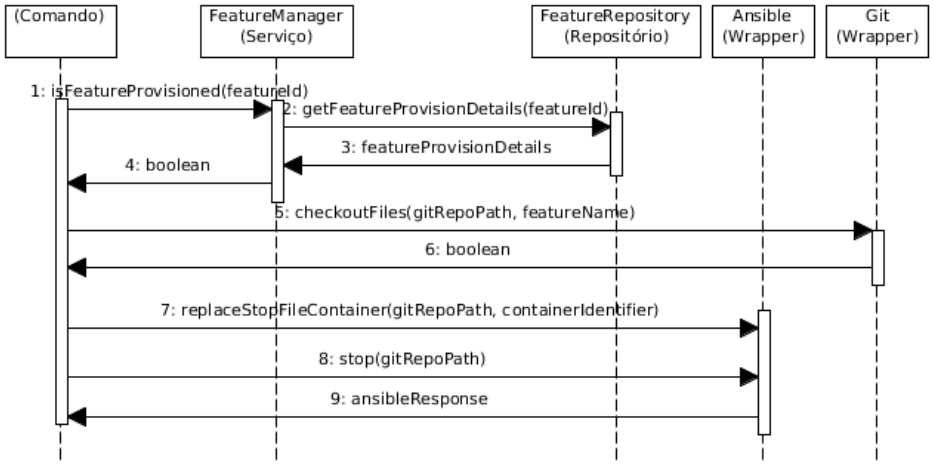



Figura 5.16: Diagrama sequência das operações utilizadas no comando Ansible que para os *containers*.

Este processo poderá ser lançado através do terminal utilizando o seguinte comando: `php app/console ansible:stop id_do_projeto id_da_versao`.

### 5.1.5 LXC

Nos *containers* LXC, para além de não termos a noção de imagem, não existe nenhuma ferramenta específica que permita a configuração dos *containers* como acontece com o Docker através do ficheiro Dockerfile. Ao contrário do Docker, a utilização de ferramentas de orquestração como o Docker Compose ou Ansible não é tão crítica nos *containers* LXC, pois estes foram criados para se comportarem como máquinas virtuais mais leves permitindo a execução de vários processos. Como o Docker foi desenhado para que cada *container* corra apenas um processo (apesar de ser possível correr múltiplos processos através de um gestor de processos como o Supervisor), se necessitarmos de diferentes serviços, é natural que sejam utilizados múltiplos *containers* e por essa razão, a utilização de ferramentas de orquestração torna-se fundamental.

#### PROVISÃO

As operações de provisão de *containers* LXC são mostradas no diagrama da figura 5.17, a primeira operação que temos que fazer é verificar a existência do *container*, obtemos a lista de *containers* LXC e através do id do projeto e da versão, identificamos se já existe um *container* provisionado com aquela versão. Se tal for verdade primeiro temos que destruir o *container*.

A provisão dos *containers* LXC é feita partindo de um *container* base onde estão instaladas as ferramentas de provisão por isso utilizamos o comando `lxc-clone` para criar uma cópia desse *container* e identificamos o novo *container* com o identificador da versão do projeto.

Utilizando os *containers* LXC não temos a facilidade de partilha de ficheiros entre o sistema base e o *container* como acontecia com o Docker, é por isso necessário alterar o ficheiro de configuração do *container* para que seja criado um *mount point* que permitirá a partilha de ficheiros entre o sistema base e uma pasta criada previamente no *container*.

A provisão do *container* LXC requer que o mesmo esteja em execução através do comando `lxc-start`, em que é passado como argumento o nome do *container* que pretendemos iniciar.

Os próximos passos envolvem executar comandos dentro do *container* utilizando o comando `lxc-attach`:

1. É criada a pasta da localização final dos ficheiros do projeto.
2. Os ficheiros que foram mapeados entre a pasta predefinida do *container* e a pasta do projeto no sistema base são copiados para o diretório final criado no passo anterior.
3. É executada a ferramenta de provisão definida na criação do projeto.

Por último, o *container* é parado e os detalhes da provisão são guardados na base de dados para referência futura.

No caso de utilização do Opencart em conjunto com o LXC teríamos que escolher entre as duas ferramentas disponíveis e criar os ficheiros de configuração necessários à provisão do *container*. Para além disso teríamos que definir quando registássemos o projeto no Presto a localização final dos ficheiros (que, neste caso é no diretório /var/www/html) e dentro dessa pasta a localização o ficheiro encarregue pela provisão, no caso do Chef seria o ficheiro solo.rb.

Este processo poderá ser lançado através do terminal utilizando o seguinte comando: `php app/console lxc:build id_do_projeto id_da_versao`.

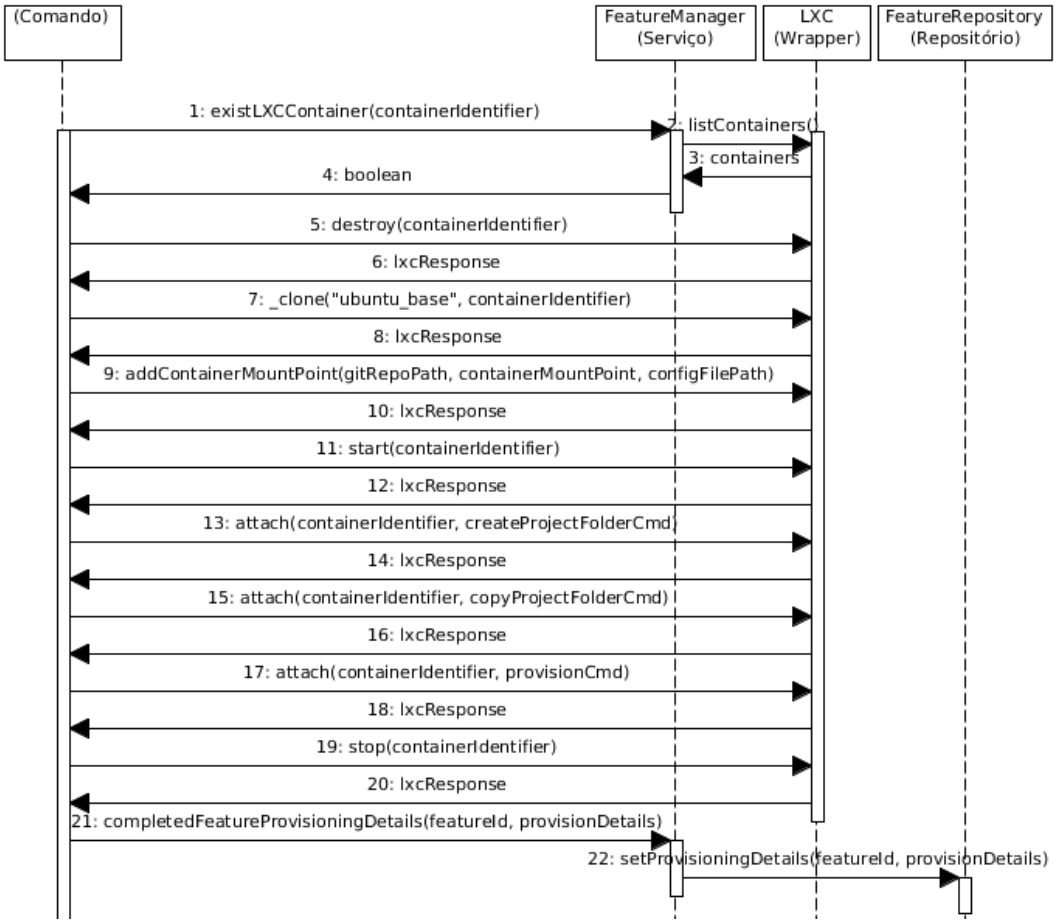



Figura 5.17: Diagrama sequência das operações utilizadas no comando LXC que provisiona o *container*.

### INICIALIZAÇÃO

A inicialização dos *containers* LXC é idêntica à dos *containers* Docker e é demonstrada no diagrama da figura 5.19. Primeiro é verificado se o *container* já não está em execução a partir da lista dos *containers* e em seguida a partir do nome do *container* é executado o comando `lxc-start` que inicia o *container* pretendido.

De seguida, e como o LXC não suporta o mapeamento dos portos entre o sistema base e o *container*, temos que utilizar a aplicação Redir que permite este mapeamento utilizando o porto livre anteriormente encontrado, mais uma vez através comando Netstat.

No exemplo de utilização do Opencart o porto a mapear seria o 80.

Um exemplo da utilização deste comando pode ser visto na figura 5.18.

```
joao@joao-K54HR ~ $ redir --laddr=127.0.0.1 --lport=11001  --caddr=10.0.3.165 --cport=80 &
[1] 6178
joao@joao-K54HR ~ $ sudo netstat -atnp | grep 11001
[sudo] password for joao:
tcp        0      0 127.0.0.1:11001         0.0.0.0:*               LISTEN      6178/redir
joao@joao-K54HR ~ $ lxc-ls --fancy
NAME                STATE    IPV4        IPV6  AUTOSTART
----------------------------------------------------------
78swe-396-elastica  RUNNING  10.0.3.165  -     NO
80opencarttest      STOPPED  -           -     NO
beubi_swe           STOPPED  -           -     NO
ubuntu-base         STOPPED  -           -     NO
ubuntu-base-chef    STOPPED  -           -     NO
```

Figura 5.18: Exemplo de utilização do comando Redir.

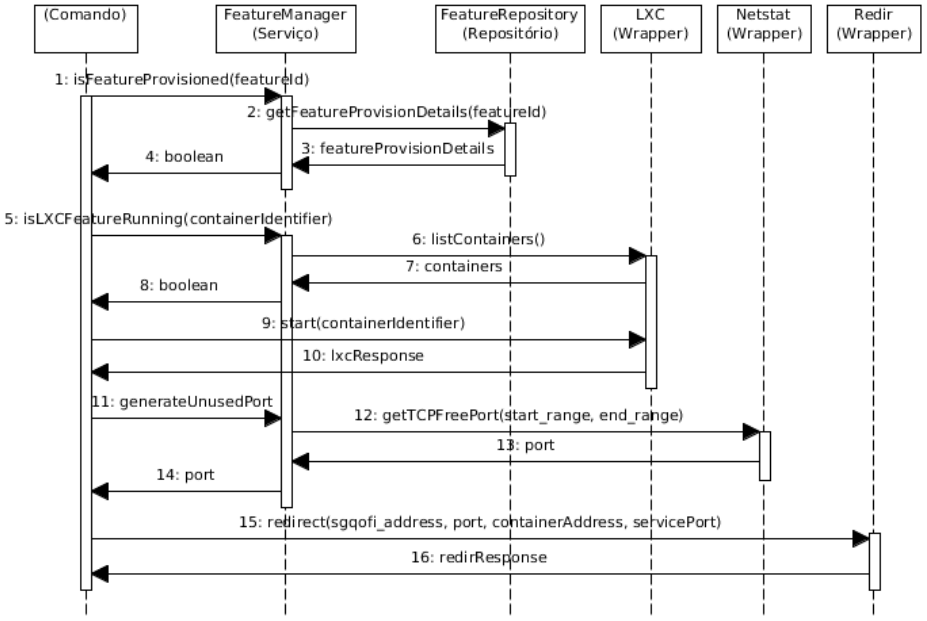



Figura 5.19: Diagrama sequência das operações utilizadas no comando LXC que inicializa o *container*.

Este processo poderá ser lançado através do terminal utilizando o seguinte comando: `php app/console lxc:start id_do_projeto id_da_versao`.

### PARAGEM

A paragem dos *containers* LXC, exposto no diagrama da figura 5.20, é análogo à inicialização. Inicialmente é verificado apenas se o *container* está em execução e se estiver, é chamado o comando `lxc-stop` que permite parar os *containers* através do seu nome.

Após o *container* ser parado é necessário parar o processo Redir que fazia o mapeamento do porto do serviço para o sistema base. Para isso é utilizado mais uma vez o comando Netstat para descobrir o id do processo e em seguida é removido esse processo do sistema fazendo com que o porto fique novamente livre.

Este processo poderá ser lançado através do terminal utilizando o seguinte comando: `php app/console lxc:stop id_do_projeto id_da_versao`.

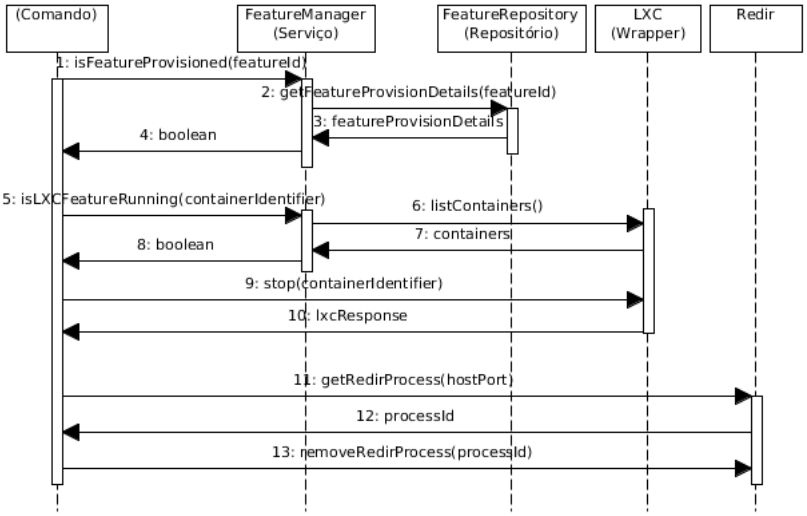



Figura 5.20: Diagrama sequência das operações utilizadas no comando LXC que pára o *container*.

### 5.1.6 PULL REQUEST

O processo de *pull request*, exibido no diagrama da figura 5.21 é uma operação especial no nosso sistema uma vez que não é despoletada a partir da nossa plataforma mas sim a partir do repositório de código remoto. A integração necessária é conseguida através da criação de *hooks* no repositório remoto.

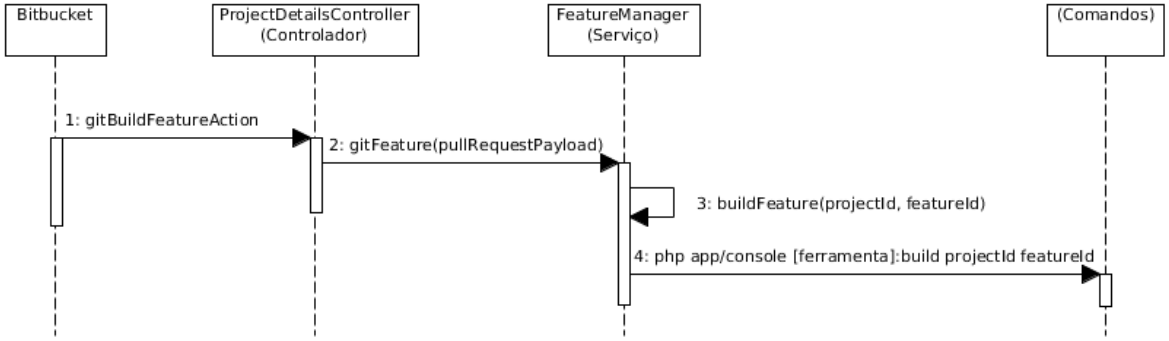



Figura 5.21: Diagrama sequência das operações durante um *pull request*

A figura 5.22 mostra o exemplo da criação de um *hook* no Bitbucket. Os *hooks* são ações executadas sempre que algum evento é despoletado no repositório remoto, neste caso o nosso *hook* deve chamar um serviço do nosso sistema sempre que um *pull request* é criado.

Depois da comunicação entre o Bitbucket e o serviço do nosso sistema, as informações recebidas através do *pull request* permitem identificar o projeto e a versão a provisionar através do nome do repositório e o nome do ramo de onde partiu o pedido de *pull request*. A partir destes obtemos o id do projeto e o id da versão do projeto e chamamos o método buildFeature, a partir daqui o processo de provisão é igual ao descrito anteriormente.

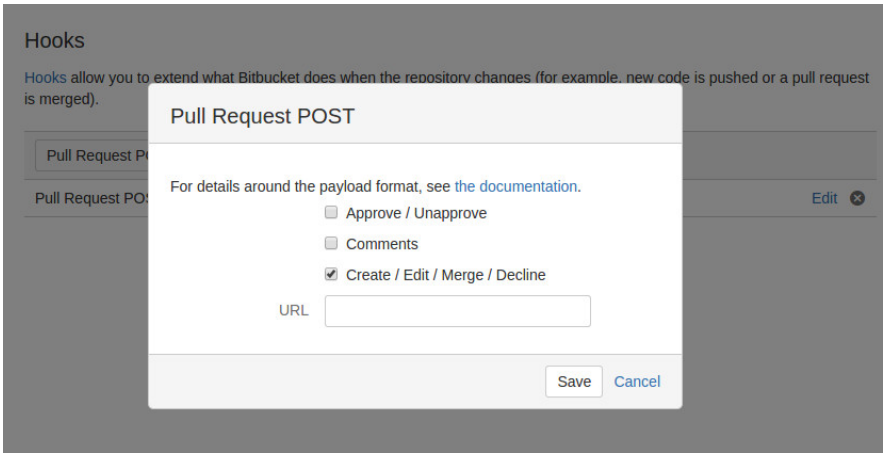



Figura 5.22: Diagrama sequência das operações durante um *pull request*

## 5.2 MODELO DE DADOS DE SUPORTE

A esquema da base de dados que suporta o sistema Presto (ver Figura 5.23), permite guardar informações relativas ao projeto, às diferentes versões associadas, detalhes sobre a provisão e informações específicas que suportam a orquestração dos ambientes de execução utilizando as diferentes ferramentas (LXC, Docker, Docker Compose e Ansible).

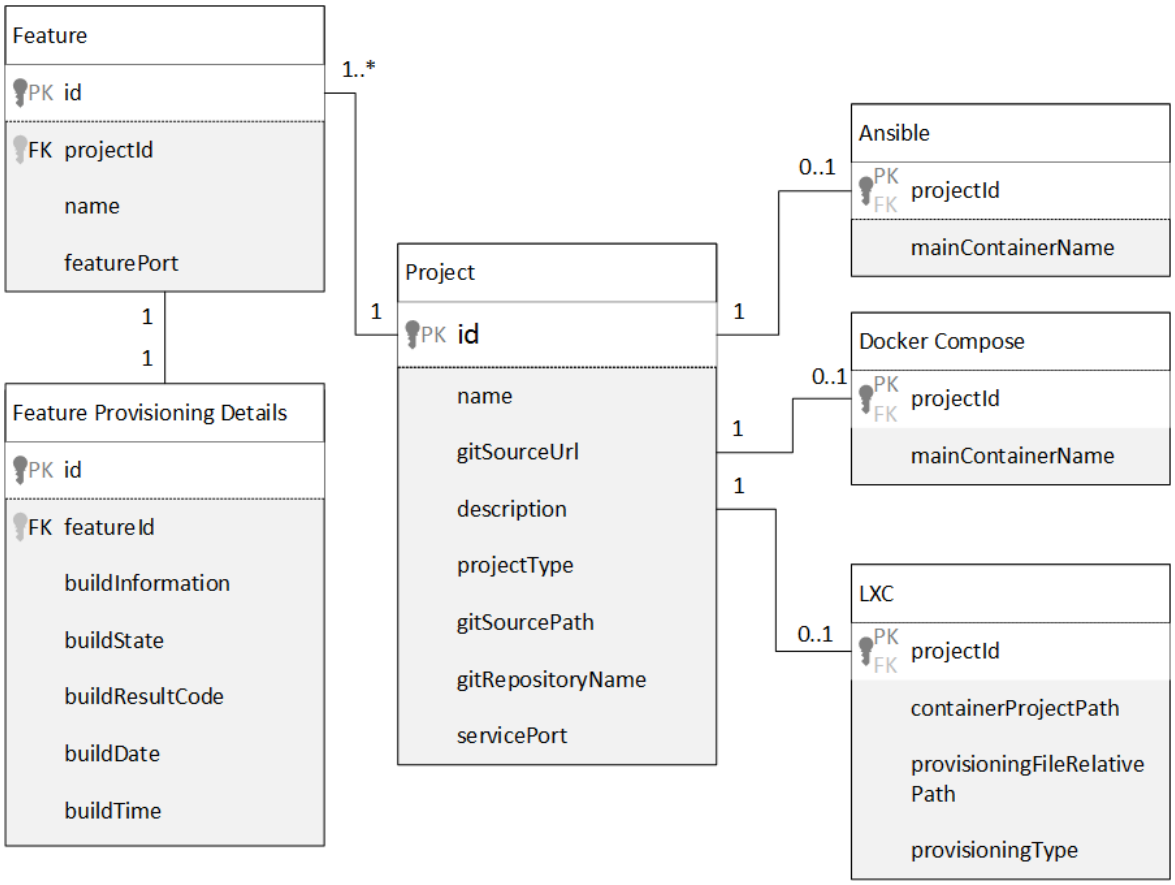



Figura 5.23: Diagrama da base de dados do sistema Presto.

A tabela projeto, tal como o nome indica guarda informações relativas ao projeto como o seu nome, uma descrição sumária, o tipo de ferramenta de orquestração, o endereço do repositório Git, o caminho da pasta onde estão os ficheiros do projeto, o nome do repositório e o porto do serviço que será utilizado para acedermos à versão da aplicação que for provisionada.

A utilização das ferramentas Docker Compose, Ansible e LXC requerem informações adicionais, relativamente às primeiras duas, é necessário saber qual o nome do *container* pelo qual é suposto acedermos à aplicação. Pode acontecer que os *containers* pertencentes ao mesmo meio de execução tenham os mesmos portos expostos, apesar de ser incomum é possível e por isso para além de identificarmos o porto também temos que identificar o *container*. A provisão com *containers* LXC requer mais alguma informação específica, é necessário saber a localização final dos ficheiros do projeto, qual a ferramenta que devemos utilizar para provisionar o *container* LXC e qual o caminho do ficheiro que será usado pela ferramenta de provisão.

Relativamente às versões de cada projeto que estão disponíveis para serem testadas é também importante guardar o seu nome e se o ambiente de *staging* estiver em execução, qual o porto do sistema base que está a ser utilizado. Esta informação é especialmente importante quando utilizamos o sistema de virtualização LXC, que não oferece esta informação.

Para cada versão é também importante guardar informações relativas à provisão, como por exemplo a data em que foi feito, o tempo que demorou, o estado em que se encontra (provisionada, não provisionada ou em provisão) e o resultado da provisão, este último permite verificar se a provisão foi executada com sucesso. Também é persistido um registo completo da provisão, isto para além de permitir ver todas as operações que foram efetuadas no meio de execução, permite em situações de erro obter informação mais detalhada sobre possíveis causas do problema.

## 5.3 INTERFACE COM O UTILIZADOR

O sistema Presto oferece duas formas de interação. A principal através do navegador, oferece todas as operações disponíveis desde a criação de um novo projeto, atualização das versões disponíveis ou inicialização dos ambientes de *staging*. A secundária através do terminal, oferece um conjunto mais limitado de opções.

### 5.3.1 LINHA DE COMANDOS

Através dos comandos do terminal é possível obter uma lista dos projetos e para um projeto específico obter a lista das diferentes versões que podem ser testadas. Estes dois comandos oferecem informação para que seja possível depois executar a provisão de uma dada versão do projeto, iniciar o ambiente de *staging* ou pará-lo. Um exemplo do output gerado por estes dois comandos é exposto na figura 5.24. A criação dos comandos, advém da necessidade de executar as operações de provisão de uma forma assíncrona, pois este é um processo que apesar de automatizado não é rápido o suficiente para que seja possível executa-lo de uma forma síncrona.

Esta necessidade fez com que fossem criados os restantes comandos que permitem a inicialização e paragem dos ambientes de *staging*, listar os projetos e respetivas versões. Para executar estas ações não é necessária a execução do servidor *web* e por isso, uma vez criados os projetos e se não houver

necessidade do desencadear automático através do *pull request*, o nosso sistema pode ser executado de uma forma independente da interface *web* se assim for preferível.

```
ubuntu@ip-10-2-0-89:/var/www/html$ php app/console sgqofi:list_projects
|Project ID |Project Name                    |   Project Type |
|         25 |Opencart Ansible               |        ansible |
|         27 |SWE LXC                        |            lxc |
|         28 |Magento Docker Compose         | docker-compose |
|         29 |SampleProject2DockerFile_Chef  |         docker |
|         30 |Opencart LXC                   |            lxc |
|         32 |Vital Responder                | docker-compose |
|         37 |Opencart Demo                  | docker-compose |
ubuntu@ip-10-2-0-89:/var/www/html$ php app/console sgqofi:list_features 37
|Feature ID |Feature Name                    |  Provisioning State |  Running State |Enviroment Address                |
|        103 |FrontPageRedesign              |         provisioned |        running |http://sgqofi.ubiprism.pt/11000   |
|        104 |NewDiscountsSystem             |       unprovisioned |    not running |----                              |
|        105 |awesomeFeature                 |         provisioned |    not running |----                              |
```

Figura 5.24: Utilização dos comandos que permitem listar os projetos e as diferentes versões da aplicação disponíveis para teste.

### 5.3.2 INTERFACE WEB

De forma a melhor mostrar as funcionalidades da interface *web* será utilizado o contexto do projeto Opencart.

A primeira tarefa que temos de fazer quando utilizamos o Presto é registar o projeto. O projeto Opencart utilizado é suportado por *containers* Docker, orquestrados pelo Docker Compose e provisionados pelos ficheiros Dockerfile. Para registar este projeto é por isso necessário definir o nome do projeto, uma descrição o endereço do repositório Git, o *container* do serviço que queremos aceder e o porto (figura 5.25).

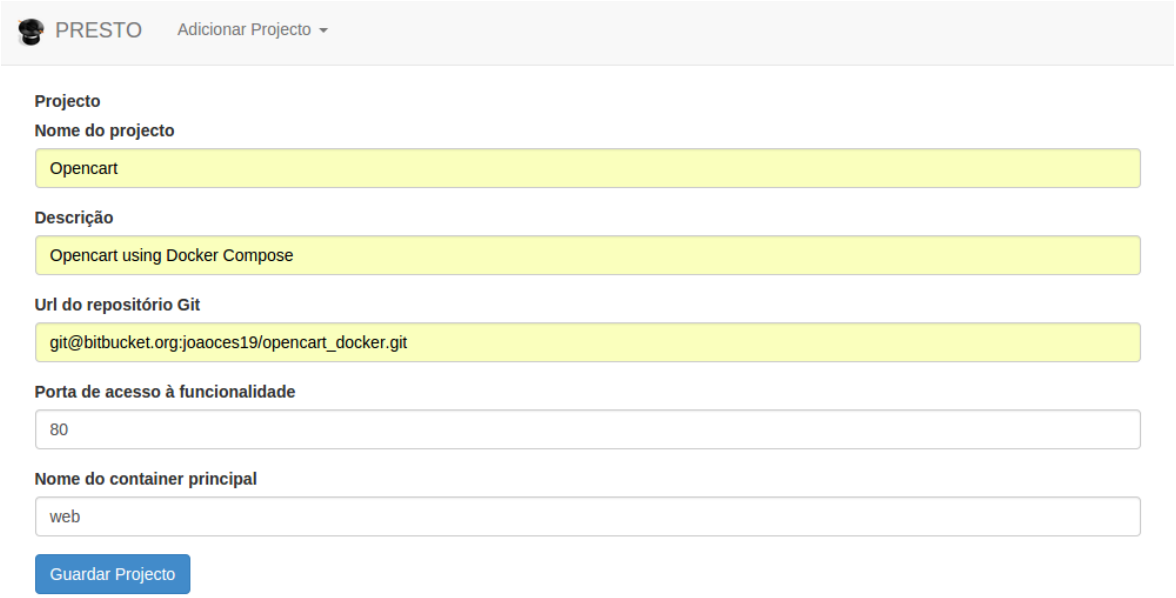



Figura 5.25: Página de registo do projeto.

A interface *web* é composta por uma página principal onde estão listados todos os projetos registados no Presto, para cada um deles é possível remover e aceder aos formulários que permitem editá-los. A partir desta página também podemos aceder aos formulários que permitem adicionar um novo projeto no nosso sistema.

Para cada projeto é também possível identificar se existem versões em que a provisão tenha falhado, em execução ou a serem provisionadas, tal como pode ser observado na figura 5.26. Para cada projeto também é possível identificar o sistema de virtualização utilizado.

No caso do projeto Opencart recentemente adicionado, pela figura 5.26 é possível observar que existe uma versão a ser provisionada e outra em execução.

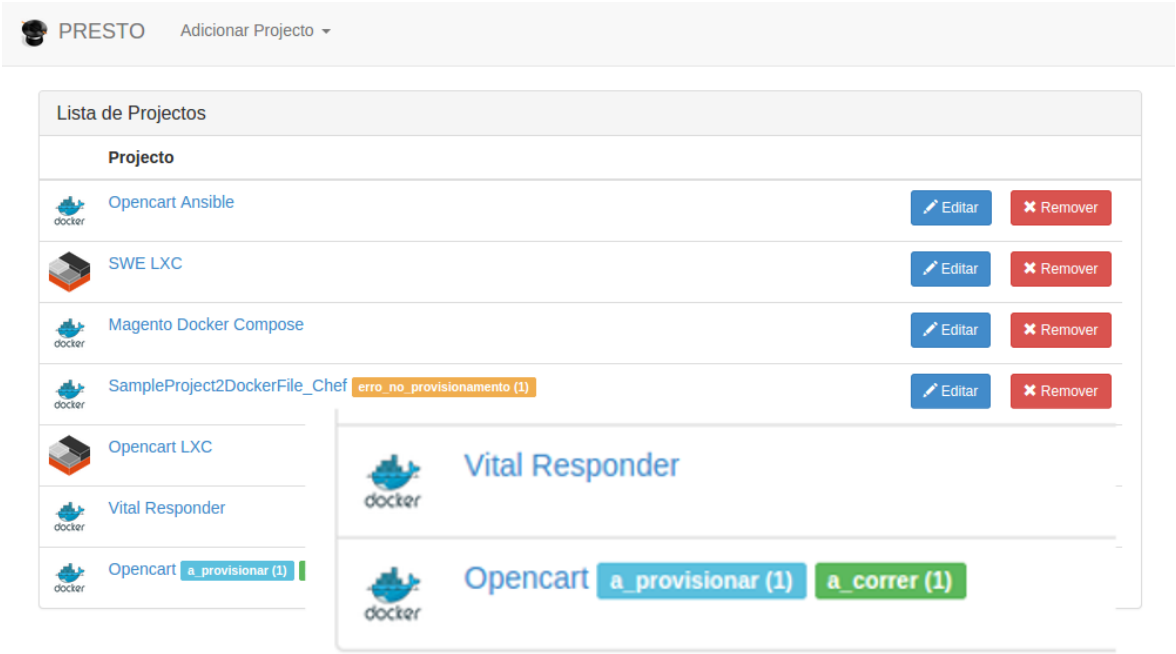



Figura 5.26: Página principal do sistema Presto.

Se selecionarmos um projeto específico na página principal somos levados para uma outra página que mostra todos os detalhes relativos ao projeto. Esta está dividida em duas partes como se pode ver pelas figuras 5.27 e 5.28. A primeira metade (5.27) mostra informações relativas ao projeto que são:

- O sistema de virtualização utilizado.
- O nome do projeto.
- A descrição do projeto.
- O endereço do repositório Git.

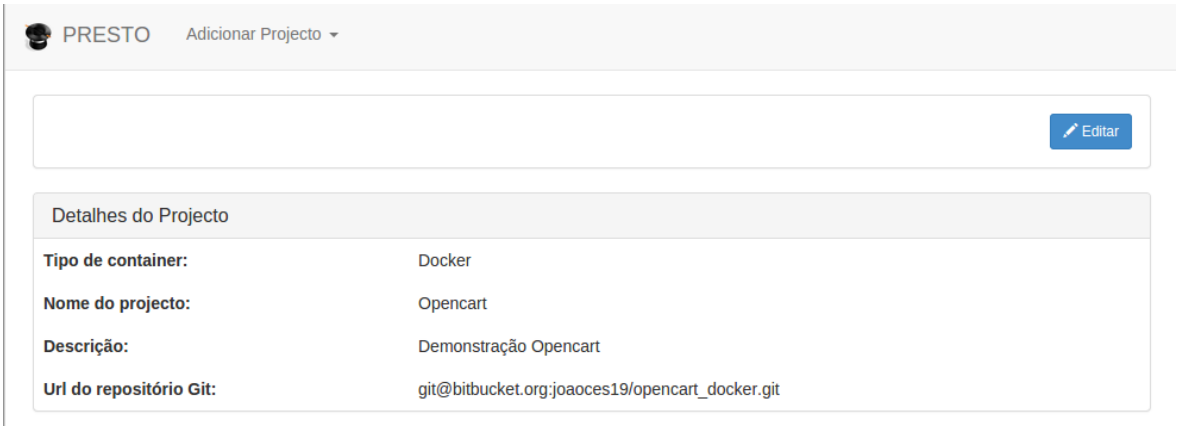



Figura 5.27: Página de detalhes do projeto, parte 1.

Na segunda metade da página podemos ver as diferentes versões do nosso projeto. Cada versão é identificada pelo nome da funcionalidade que pode ser testada. Esta lista permite identificar quais as versões do projeto provisionadas e situações de erro que possam ter ocorrido durante a provisão. Para cada versão pode ser iniciado o processo de provisão manualmente. Se essa versão já tiver provisionada o ambiente de *staging* pode ser executado ou parado (dependendo do estado em que está). Nesta página também é possível atualizar a lista de versões do projeto para que esta fique atualizada com as existentes no sistema de versionamento de código remoto.

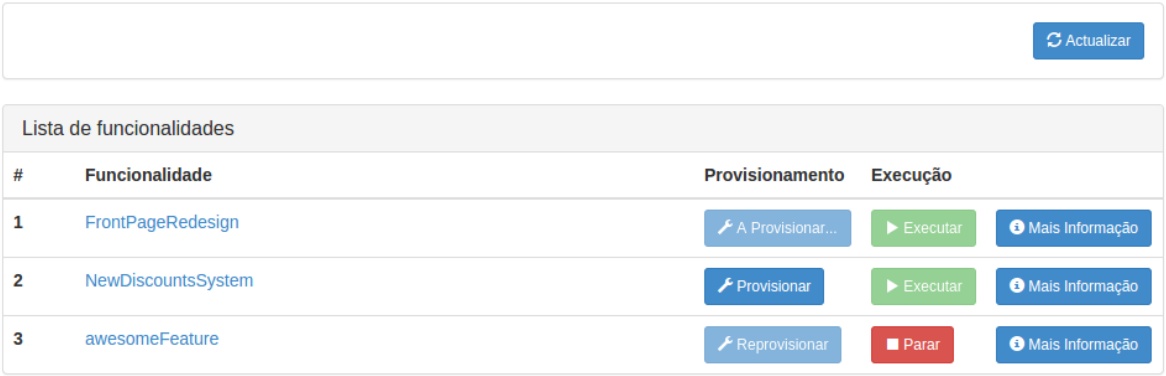



Figura 5.28: Página de detalhes do projeto, parte 2.

Nas duas últimas figuras anteriores (5.27 e 5.28) podem ser vistos os detalhes do projeto Opencart.

A partir da página anterior é possível aceder aos detalhes de provisão e do ambiente de *staging*. Esta página é divida em duas partes, representadas nas figuras 5.29a e 5.29b. A primeira é referente aos detalhes da provisão, onde é mostrada a seguinte informação:

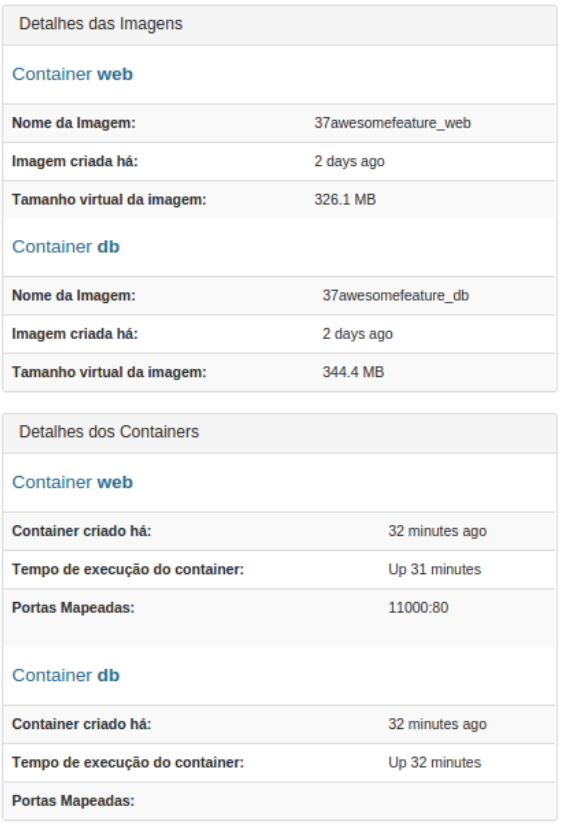
- A data do início da provisão.
- O tempo que demoraram a ser executadas as operações de provisão.
- O estado em que se encontra aquela versão do projeto.
- O registo detalhado da provisão do ambiente de *staging*, este permite identificar as operações efetuadas e caso tenha ocorrido algum erro, estas permitem detetar as causas desse erro.

E a segunda dedicada à descrição dos *containers* que compõem o ambiente de *staging*. As informações apresentadas quando o sistema de virtualização é o Docker são as seguintes:

- Imagens:
  - Nome da imagem.
  - Há quanto tempo foi criada.
  - Tamanho virtual da imagem.
- *Containers*:
  - Nome do *container*.
  - Há quanto tempo o *container* foi criado.
  - Há quanto tempo o *container* está em execução.
  - Portos mapeados.

No caso dos *containers* LXC a informação é mais limitada e resume-se ao nome do *container* e os endereços IPv4 e IPv6 quando está em execução.


(a) Detalhes do provisão.




(b) Detalhes do ambiente de *staging*.

Figura 5.29: Página de detalhes de uma versão do projeto.

Nas duas últimas figuras (5.29a e 5.29b) foram apresentados os detalhes de uma das versões do projeto Opencart.

### 5.3.3 INTEGRAÇÃO DO EVENTO DE PULL REQUEST

O desencadear do processo de provisão por *pull request*, pode ser feito através de qualquer repositório de código em que seja suportada a criação de *hooks*.

A título exemplificativo iremos demonstrar os passos necessários para criar este tipo de evento através do Bitbucket:

1. Através da página de definições do projeto acedemos à secção dos *hooks*.
2. Selecionamos o *hook* pretendido, neste caso queremos criar o *Pull Request POST*.
3. De seguida é apresentado um *pop-up* em que devemos indicar o serviço com que o Bitbucket irá comunicar.
4. Os próximos passos são repetidos ao longo do processo de desenvolvimento do produto:
   a) A partir da página referente aos ramos do projeto no Bitbucket, criamos um novo ramo.
   b) Quando é concluído o processo de desenvolvimento, criamos um *pull request* através da página do ramo.
   c) O Bitbucket comunica com o serviço indicado anteriormente e o processo de provisão é despoletado.

A integração do evento de *pull request* permite que o Presto possa proativamente provisionar o ambiente de execução para a nova funcionalidade, sem depender de ações do gestor de projeto.

## 5.4 INSTALAÇÃO DO SISTEMA PRESTO

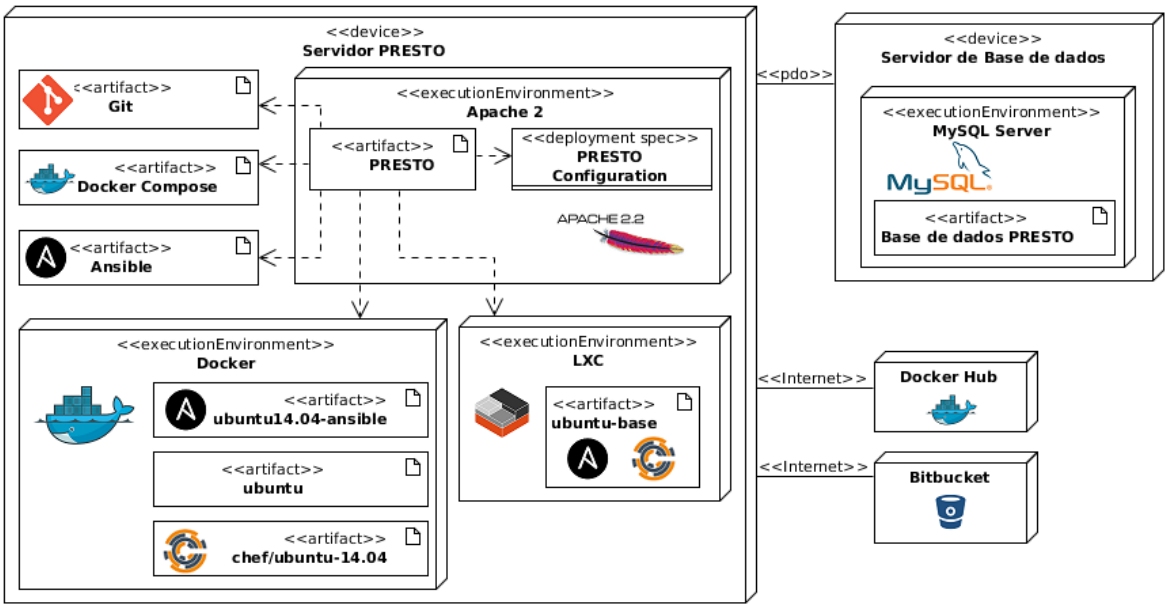



Figura 5.30: Diagrama de instalação do sistema Presto.

A vista de instalação representada pelo diagrama da figura 5.30, mostra um ambiente para a execução do nosso sistema.

O diagrama de instalação é formado por dois nós, um deles representa o ambiente onde estará instalada a base de dados e o outro onde será executado o nosso sistema.

Os únicos requisitos do nó responsável pela persistência é ter instalado o MySQL Server e ter a base de dados com o respetivo esquema criado.

Relativamente ao nó que executa o sistema Presto existem mais alguns requisitos que tem de ser satisfeitos. Primeiro tem que existir um servidor *web* como por exemplo, o Apache, capaz de executar a nossa aplicação, depois para que seja possível a utilização dos sistemas de virtualização é preciso instalar o Docker e o LXC. Relativamente ao Docker, a utilização do Chef e do Ansible implicará geralmente a utilização das imagens que já têm estas duas ferramentas instaladas, porém a transferência prévia destas não é fundamental, uma vez que quando a sua utilização for requisitada, o Docker fará a sua transferência automaticamente. O LXC não tem esta facilidade, e como foi explicado anteriormente quando criamos um novo ambiente de testes este *container* é criado a partir de um outro *container* que já tenha os sistemas de provisão do Chef e do Ansible instalados. Por isso, torna-se um requisito fundamental para o correto funcionamento a criação de um *container* onde estes dois últimos já estejam instalados.

Existem ainda mais alguns requisitos relativos a este nó; é necessária a instalação das ferramentas Docker Compose e Ansible que permitem a orquestração dos *containers* Docker. O Git, ferramenta que permite obter os ficheiros do repositório remoto, identificar as diferentes versões do projeto e trocar os ficheiros relativos a cada versão.

As configurações específicas relativas ao sistema Presto são descritas através do ficheiro YAML denominado sgqofi.yml. Os parâmetros possíveis de ser configurados são os seguintes:

| | |
|---|---|
| Pasta dos projetos | este parâmetro permite configurar o diretório onde os ficheiros dos projetos serão guardados. |
| Gama dos portos | configuração do intervalo dos portos do sistema base a utilizar. |
| Endereço do servidor | endereço do servidor, que depois em conjunto com o porto permite o acesso ao ambiente de *staging*. |
| Pasta do projeto Presto | o diretório onde a aplicação está instalada, necessário para que depois consigamos saber onde executar os comandos disponíveis no sistema Presto. |

Tabela 5.1: Ficheiro de configuração do sistema Presto.

capítulo 6

# Resultados e validação

## 6.1 utilização do sistema com soluções existentes

De forma a validarmos o trabalho desenvolvido ao longo desta dissertação iremos utilizar quatro projetos diferentes, um deles utilizando *containers* LXC e outros três utilizando os *containers* Docker.

O projeto utilizado para validar a utilização dos *containers* LXC foi o SWE que é uma aplicação *web* desenvolvida pela empresa Beubi e que tem como objetivo oferecer descontos personalizados. Atualmente na empresa Beubi o sistema de virtualização utilizado são os *containers* LXC, usados em conjunto com o Vagrant. O Vagrant permite abstrair o utilizador da complexidade subjacente a algumas operações que precisam de ser executadas para configurar o meio de execução, como o mapeamento de portos e a partilha de ficheiros e pastas entre o *container* e o sistema base, para além disso o Vagrant também tem suporte para a execução de alguns sistemas de provisão, sendo o Chef um deles e o que é utilizado atualmente pela Beubi.

Utilizando as receitas Chef previamente desenvolvidas pela equipa da Beubi precisámos de criar o ficheiro de configuração (solo.rb) que é utilizado para conseguirmos executar o comando chef-solo. Este processo é necessário pois o projeto está preparado para utilizar o Vagrant e estas configurações do Chef são descritas através do ficheiro Vagrantfile. Como o objetivo era apenas utilizar o LXC sem o Vagrant, foi necessário criar este ficheiro para que a provisão pudesse ser executada através do chef-solo.

O próximo passo foi adicionar o projeto à nossa plataforma: foi necessário definir o nome do projeto, o endereço do repositório Git, a localização final dos ficheiros do projeto dentro do *container*, a localização do ficheiro de configuração dentro da pasta do projeto e o sistema de provisão a utilizar, que neste caso é o Chef.

No sistema de versionamento de código remoto, definimos também o serviço da nossa plataforma que comunica com o Bitbucket para que os *pull requests* desencadeiem o processo de provisão.

Uma vez concluído o processo de configuração da aplicação na plataforma, podemos provisionar e lançar qualquer uma das versões disponíveis. Para além disso, a partir deste momento, a plataforma também será capaz de identificar *pull requests* destinados a este projeto e despoletados a partir do repositório de código remoto.

A utilização do Presto para provisionar automaticamente os ambientes de execução do projeto SWE foi conseguido sem alterações às práticas ou *core bases* já existentes.

Os projetos utilizados para validar a utilização dos *containers* Docker foram as plataformas Opencart e Magento, duas plataformas de *ecommerce*, *open source* e em constante desenvolvimento. O terceiro projeto utilizado foi o sistema Vital Responder, desenvolvido pelo IEETA, é uma plataforma que permite a monitorização de diversos indicadores fisiológicos e que avaliam possíveis situações de risco. Este é um projeto interessante pois depende de múltiplos serviços para funcionar e é executado num ambiente em que a maior parte das configurações ainda são realizadas manualmente.

Na tabela 6.1 são descritas as características de cada solução:

| | Opencart | Magento | Vital Responder |
|---|---|---|---|
| Linguagem de programação | PHP | PHP | Java |
| Servidor *web* | Apache 2 | Apache 2 | WildFly e Apache 2 |
| Base de dados | MySQL | MySQL | MongoDB |
| Numero de *containers* | 2 | 2 | 3 |

Tabela 6.1: Características dos projetos Docker.

O Opencart e o Magento são duas soluções bastante parecidas, ambas utilizam como linguagem de programação o PHP e ambas usam uma base de dados relacional MySQL, foram por isso criados dois *containers*: um que executa o servidor *web* Apache e outro que executa o MySQL.

O Vital Responder é uma solução mais complexa uma vez que a interface *web* (que recorre essencialmente a HTML, CSS e Javascript) é executada num servidor Apache, esta interface comunica depois com o servidor aplicacional WildFly através de uma API Rest que comunica por sua vez com a base de dados, MongoDB, responsável por persistir a informação.

Para os três projetos foi utilizada a ferramenta de orquestração Docker Compose e a provisão foi feita recorrendo aos ficheiros Dockerfile.

A empresa Beubi disponibilizou um servidor alojado na infraestrutura da Amazon e onde foram testados os projetos indicados anteriormente utilizando o Presto. Isto permitiu que a ferramenta ficasse disponível para ser testada e que os ambientes de *staging* criados pudessem ser acedidos por todos os interessados.

## 6.2 AVALIAÇÃO DO DESEMPENHO DA FERRAMENTA PRESTO

Na figura 6.1 podem ser observadas as principais fases do processo de criação do ambiente de *staging*. Todos estes passos são automatizados pelo sistema Presto, o objetivo nesta secção passa por comparar o processo manual com o processo automatizado pela nossa ferramenta.

O processo está dividido em quatro fases: inicialmente o *pull request* é desencadeado pelo programador, este é um processo manual e que não pode ser automatizado, o passo seguinte passa por obter os ficheiros do projeto da versão correta que queremos testar a partir do sistema de versionamento de código remoto. Os últimos dois passos são referentes à criação e provisão do ambiente de *staging* em si.

No processo manual, o tempo que cada etapa demora depende sempre de múltiplos fatores, como a experiência da pessoa, a qualidade da documentação, o tamanho do projeto, a quantidade de configurações necessárias e recursos disponíveis na máquina onde serão executadas as configurações.

De forma a termos algum termo de comparação iremos comparar o tempo de provisão automatizado pela nossa plataforma e o mesmo processo feito de forma manual. Para isso iremos utilizar o projeto do Opencart executado no ambiente Docker. No caso em que o processo é automatizado será criado um *pull request* no Bitbucket e o sistema Presto desencadeará logo a seguir todos os passos necessários utilizando o Docker Compose, os Dockerfiles e o Git. Relativamente ao processo manual, também iremos criar um *pull request* no sistema de versionamento de código mas todos os passos seguintes serão realizados de uma forma manual, também com o Docker como sistema de virtualização, mas todas as configurações serão aplicadas manualmente, partindo da imagem base do Ubuntu. O diagrama de atividades da figura 6.1 mostra de uma forma sumária as ações que vão ser executadas.

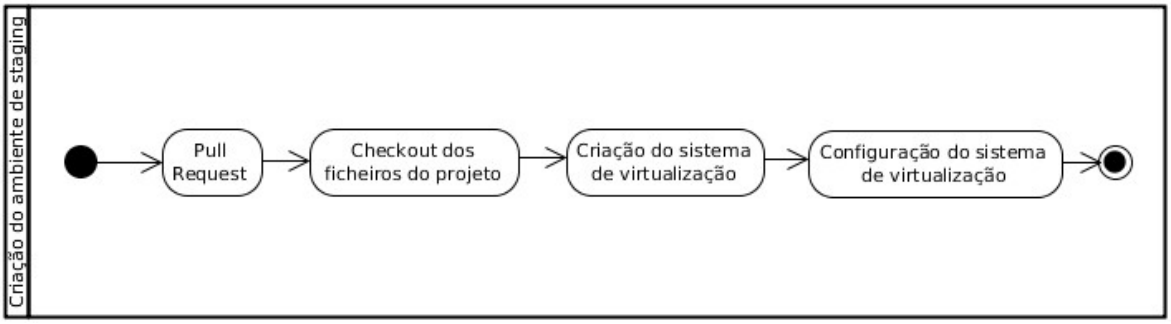



Figura 6.1: Principais passos para a criação do ambiente de *staging*.

Os resultados obtidos foram os seguintes:

| | Processo automatizado | Processo manual |
|---|---|---|
| *Pull Request* | 39 segundos | 39 segundos |
| Checkout dos ficheiros do projeto | 1 minuto e 28 segundos | 45 segundos |
| Criação do sistema de virtualização | | 2 minutos e 7 segundos |
| Configuração do sistema de virtualização | | 8 minutos e 51 segundos |

Tabela 6.2: Comparação entre o processo manual e o processo automatizado.

Como podemos observar pelos resultados apresentados na tabela 6.2 e apesar do número de configurações não ser muito elevado, existe uma diferença significativa entre os dois processos. Esta diferença tem tendência a crescer se considerarmos que com o aumentar do número de configurações a probabilidade da pessoa responsável cometer erros também é maior.

Estes testes foram executados no servidor da Amazon onde o Presto está instalado. A máquina tem as seguintes características:

| | |
|---|---|
| Instância | T2 Small |
| CPU | Intel(R) Xeon(R) CPU E5-2670 v2 2.50GHz |
| Memória | 2 GB |
| Disco | Amazon EBS magnético - 62.86 GB |

Tabela 6.3: Detalhes do servidor onde é executado o Presto.

## 6.3 AVALIAÇÃO DO DESEMPENHO COM DIFERENTES COMBINAÇÕES DE FERRAMENTAS

De forma a avaliarmos o desempenho das diferentes ferramentas de orquestração e formas de provisão que a nossa plataforma suporta fizemos uma série de testes em que medimos o tempo que as diferentes operações (provisão, início e paragem) demoraram a serem executadas.

O projeto de teste utilizado foi o Opencart, já descrito anteriormente. Para cada ferramenta de provisão foram criados os seus ficheiros de configuração específicos (*recipes/playbooks* - Chef/Ansible) e o Dockerfile que permite correr cada uma das formas de provisão, nos dois primeiros casos (utilizando o Chef e o Ansible) e no terceiro, configurando o *container* diretamente através do Dockerfile. Os mesmos ficheiros de configuração foram utilizados para provisionar os *containers* orquestrados pelo Ansible e pelo Docker Compose.

Também foi necessário criar os ficheiros de orquestração que suportam a execução das ferramentas Docker Compose e Ansible.

No caso do LXC não utilizamos nenhuma ferramenta de orquestração mas utilizámos duas formas de provisão através do Chef e Ansible, também anteriormente utilizadas na provisão dos *containers* Docker. As *recipes/playbooks* anteriormente criadas foram reutilizadas para provisionar os *containers* LXC.

A tabela 6.4 mostra todas as combinações realizadas. Foram realizados testes utilizando os dois sistemas de virtualização, Docker e LXC. Para os *containers* Docker foram utilizadas as ferramentas de orquestração Ansible e Docker Compose e para cada uma delas os *containers* foram provisionados através do Chef, Ansible e Dockerfile. Para os *containers* LXC estes foram executados diretamente através do comando `lxc-start` e por isso sem o auxílio de nenhuma ferramenta de orquestração. Neste caso também foram utilizadas como ferramentas de provisão o Chef e o Ansible.

| Provisão / Orquestração | Chef | Ansible | Dockerfile |
|---|---|---|---|
| Ansible | Docker | Docker | Docker |
| Docker Compose | Docker | Docker | Docker |
| Sem orquestração | LXC | LXC | – |

Tabela 6.4: Combinações realizadas entre as diferentes ferramentas.

Os testes foram realizados através de comandos Symfony2, com base nos anteriormente criados especificamente para a plataforma, com a diferença que todas as verificações, chamadas à base de dados e operações relacionadas com o versionamento dos ficheiros foram retiradas.

| Orquestração com Ansible utilizando *containers* Docker | | | |
|---|---|---|---|
| | Ansible & Dockerfile | Chef & Dockerfile | Dockerfile |
| Provisão | 290.29 s | 273.33 s | 79.88 s |
| Início | 3.71 s | 2.26 s | 2.25 s |
| Paragem | 21.86 s | 21.8 s | 21.78 s |

Tabela 6.5: Desempenho do Ansible com as diferentes formas de provisão

| Orquestração com Docker Compose | | | |
|---|---|---|---|
| | Ansible & Dockerfile | Chef & Dockerfile | Dockerfile |
| Provisão | 303.43 s | 277.55 s | 81.22 s |
| Início | 1.20 s | 1.33 s | 0.68 s |
| Paragem | 20.24 s | 20.22 s | 20.23 s |

Tabela 6.6: Desempenho do Docker Compose com as diferentes formas de provisão

| Orquestração com LXC | | |
|---|---|---|
| | Ansible | Chef |
| Provisão | 180.12 s | 194.26 s |
| Início | 0.83 s | 0.80 s |
| Paragem | 2.26 s | 2.33 s |

Tabela 6.7: Desempenho do LXC com as diferentes formas de provisão

Em termos de provisão, pelos resultados obtidos nas tabelas 6.5, 6.6 e 6.7 podemos constatar que os tempos de provisão entre as duas ferramentas de orquestração são muito idênticos e não existe uma discrepância que nos permita afirmar que uma ferramenta é claramente mais rápida que a outra. O Chef nos *containers* Docker foi ligeiramente mais rápido comparativamente ao Ansible, enquanto que nos *containers* LXC, o Ansible foi ligeiramente mais rápido comparativamente ao Chef.

Comparando a utilização das ferramentas de provisão nos *containers* LXC e Docker podemos observar que a sua utilização foi claramente mais eficiente no ambiente LXC. Porém, as mesmas configurações aplicadas através do Dockerfile nos *containers* Docker foram executadas em metade do tempo que as configurações executadas através de qualquer uma das ferramentas de provisão nos *containers* LXC.

Relativamente à orquestração do ambiente de execução da aplicação, o processo de inicialização e paragem dos *containers* LXC foi mais rápido que nos *containers* Docker. A inicialização dos *containers* Docker também foi mais rápida quando executada com a ferramenta Docker Compose em vez do Ansible, todavia esta discrepância é pouco significativa. A paragem do ambiente de execução foi idêntico na utilização das duas ferramentas (Ansible e Docker Compose).

capítulo 7

# Conclusão e trabalho futuro

O resultado desta dissertação é um sistema capaz de automatizar o processo de criação dos ambientes de *staging*, utilizando para isso os sistemas de virtualização LXC e Docker, as ferramentas de orquestração e provisionamento Chef, Ansible e Docker Compose e os ficheiros de configuração que são versionados juntamente com o restante código da aplicação.

O sistema Presto é capaz de reconhecer todas as versões de um projeto, passíveis de serem testadas analisando o sistema de versionamento, e desencadear automaticamente o processo de criação do ambiente de *staging* sempre que uma ação de *pull request* é criada no repositório remoto.

Utilizando as informações obtidas através dos comandos dos sistemas de virtualização e persistidas na base de dados conseguimos aferir que ambientes de *staging* estão criados, em execução ou parados. O Presto é capaz de detetar a existência de erros durante a provisão dos ambientes e apresentar essa informação de forma a que seja possível identificar mais facilmente a origem do problema.

Este sistema vem preencher uma lacuna existente nas ferramentas que auxiliam o ciclo de entrega contínua e visa resolver o problema de configuração de um ambiente de execução adaptado ao teste de funcionalidades isoladas. A nossa solução consegue criar ambientes leves, utilizando *containers*, provisionados automaticamente e acessíveis por toda a equipa de parceiros do projeto.

Devido à utilização de *containers* a criação e destruição destes ambientes é bastante expedita e o custo associado à sua execução é também menor, quando comparado com sistemas de virtualização convencionais.

Utilizando a nossa ferramenta o custo associado à manutenção e criação do ambiente de *staging* é reduzido e permite que múltiplas versões de múltiplos projetos possam ser acedidas ao mesmo tempo.

Devido à falta de soluções semelhantes a esta ferramenta foi difícil realizar comparações da solução desenhada por nós com outros sistemas. As funcionalidades incluídas no Presto, de forma integrada, são tipicamente suportadas em processos manuais e na utilização de *scripts* escritos pela equipa de desenvolvimento, de forma mais ou menos *ad-hoc*.

Anteriormente este processo de criação do ambiente de *staging* era executado através de uma pessoa responsável pelo mesmo, este necessitava que a mesma tivesse conhecimento sobre as diferentes ferramentas e que passos tinham que ser executados para criar esse ambiente. Atualmente com a utilização do sistema Presto este processo pode ser desencadeado a partir da interface *web* (selecionando

a versão do projeto que desejamos testar), a partir do terminal (executando apenas um comando), ou automaticamente quando é criado um *pull request* no sistema de versionamento de código remoto.

O sistema desenvolvido é também compatível com a utilização de múltiplas ferramentas de virtualização, orquestração e provisão, facilitando uma possível mudança de contexto na empresa. Suportar diferentes tipos de ferramentas permite a execução de uma variedade maior de projetos, com diferentes requisitos.

Apesar deste sistema cumprir todos os pressupostos da dissertação, existem alguns melhoramentos que podem ser aplicados e que permitirão a comercialização deste sistema como um produto.

À plataforma falta a gestão de utilizadores (atualmente não existe o conceito de utilizador). O desenvolvimento desta funcionalidade permitiria a atribuição de diferentes permissões para diferentes tipos de utilizadores do sistema. Poderia ainda fazer sentido a criação de diferentes ambientes de *staging* para a mesma versão do projeto, desta forma o ambiente de teste seria independente para cada pessoa responsável pelo teste.

Uma outra funcionalidade interessante seria a possibilidade dos diferentes utilizadores terem uma forma de comentar o trabalho desenvolvido, este tipo de interação poderia ainda ser integrado com uma ferramenta de gestão de projetos como o Jira.

Neste momento a única interação existente com o Bitbucket é quando um *pull request* é criado e desencadeia a provisão do ambiente de *staging* no sistema, porém o Bitbucket através da sua API, permite obter mais informações e expandir a interação existente. Seria útil obter detalhes sobre o *pull request* e apresentá-lo no Presto, o que ajudaria a identificar o alvo do teste, uma vez que apenas o nome da funcionalidade pode não ser suficiente para quem não está por dentro do desenvolvimento da mesma. Uma vez testada a funcionalidade, o *pull request* poderia ser aprovado ou rejeitado a partir do Presto.

Durante o teste de uma funcionalidade, ocasionalmente, torna-se necessário a introdução de dados que permitem a realização de testes, como por exemplo contas de utilizadores. Estes dados por vezes não estão contemplados no conjunto de dados com que a base de dados é inicializada e por essa razão faria sentido a possibilidade de reaproveitar estes dados de teste entre diferentes ambientes de *staging*.

# Referências


[1] K. Naik e P. Tripathy, *Software Testing and Quality Assurance: Theory and Practice*. Wiley-Spektrum; 1 edition (August 18, 2008), 2008, p. 648, ISBN: 978-0-471-78911-6.

[2] J. Loeliger e M. McCullough, *Version Control with Git: Powerful tools and techniques for collaborative software development*. O'Reilly Media; Second Edition edition (August 27, 2012), 2012, p. 456, ISBN: 978-1449316389.

[3] *Git Documentation*. endereço: `https://git-scm.com` (acedido em 16/12/2014).

[4] V. Driessen, *A successful Git branching model*. endereço: `http://nvie.com/posts/a-successful-git-branching-model/` (acedido em 16/12/2014).

[5] *Git Flow*, 2012. endereço: `http://joefleming.net/posts/git-flow/` (acedido em 20/04/2015).

[6] J. Humble e D. Farley, *Continuous Delivery*. 2010, p. 512, ISBN: 9780321601919.

[7] L. Chen, «Continuous Delivery: Huge Benefits, but Challenges Too», *IEEE Software*, vol. 32, nº 2, pp. 50–54, mar. de 2015, ISSN: 0740-7459. DOI: `10.1109/MS.2015.27`. endereço: `http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=7006384`.

[8] J. Gmeiner, R. Ramler e J. Haslinger, «Automated testing in the continuous delivery pipeline: A case study of an online company», em *2015 IEEE Eighth International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, IEEE, abr. de 2015, pp. 1–6, ISBN: 978-1-4799-1885-0. DOI: `10.1109/ICSTW.2015.7107423`. endereço: `http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=7107423`.

[9] R. Dua, A. R. Raja e D. Kakadia, «Virtualization vs Containerization to Support PaaS», em *2014 IEEE International Conference on Cloud Engineering*, IEEE, mar. de 2014, pp. 610–614, ISBN: 978-1-4799-3766-0. DOI: `10.1109/IC2E.2014.41`. endereço: `http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=6903537`.

[10] C. Costache, O. Machidon, A. Mladin, F. Sandu e R. Bocu, «Software-defined networking of Linux containers», em *2014 RoEduNet Conference 13th Edition: Networking in Education and Research Joint Event RENAM 8th Conference*, IEEE, set. de 2014, pp. 1–4, ISBN: 978-1-4799-6861-9. DOI: `10.1109/RoEduNet-RENAM.2014.6955310`. endereço: `http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=6955310`.

[11] *Linux Containers*. endereço: `https://linuxcontainers.org/lxc/getting-started/` (acedido em 05/04/2015).

[12] J. Gomes, J. Pina, G. Borges, J. Martins, N. Dias, H. Gomes e C. Manuel, «Exploring Containers for Scientific Computing», em *Exploring Containers for Scientific Computing*, I. Oliveira, J. Gomes, I. Campos e I. Blanquer, eds., Aveiro: Editorial Universitat Politècnica de València, 2014, p. 12, ISBN: 9788490482469.

[13] R. Rosen, *Linux Containers and the Future Cloud*, 2014. endereço: http://www.linuxjournal.com/content/linux-containers-and-future-cloud (acedido em 05/04/2015).

[14] W. Felter, A. Ferreira, R. Rajamony e J. Rubio, «An updated performance comparison of virtual machines and Linux containers», em *2015 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, IEEE, mar. de 2015, pp. 171–172, ISBN: 978-1-4799-1957-4. DOI: 10.1109/ISPASS.2015.7095802. endereço: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=7095802.

[15] M. Kerrisk, *Namespaces in operation, part 1: namespaces overview*, 2013. endereço: https://lwn.net/Articles/531114/.

[16] *Ubuntu manpages*. endereço: http://manpages.ubuntu.com (acedido em 05/04/2015).

[17] *Docker Documentation*. endereço: https://docs.docker.com/ (acedido em 15/12/2014).

[18] *Aufs*. endereço: http://aufs.sourceforge.net/aufs.html.

[19] *Docker Blog*. endereço: https://blog.docker.com (acedido em 15/05/2015).

[20] C. Anderson, «Docker [Software engineering]», *IEEE Software*, vol. 32, nº 3, pp. 102–c3, mai. de 2015, ISSN: 0740-7459. DOI: 10.1109/MS.2015.62. endereço: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=7093032.

[21] *Chef Documentation*. endereço: https://docs.chef.io/ (acedido em 15/02/2015).

[22] A. Vasiliev, *Cooking infrastruture by Chef*. endereço: http://chef.leopard.in.ua/.

[23] *Ansible Documentation*. endereço: http://docs.ansible.com/ (acedido em 25/04/2015).

[24] *Ansible is Simple IT Automation*. endereço: http://www.ansible.com/home (acedido em 25/04/2015).

[25] M. Fowler, *UML Distilled: A Brief Guide to the Standard Object Modeling Language (Object Technology Series)*. 2003, p. 208, ISBN: 978-0321193681.

[26] P. DuBois, *MySQL (5th Edition) (Developer's Library)*. Addison-Wesley Professional; 5 edition (April 12, 2013), 2013, p. 1176, ISBN: 978-0321833877.

[27] R. Nixon, *Learning PHP, MySQL, JavaScript, CSS & HTML5: A Step-by-Step Guide to Creating Dynamic Websites*. O'Reilly Media; 3 edition (June 16, 2014), 2014, ISBN: 978-1491949467.

[28] M. Noback, *A Year With Symfony: Writing healthy, reusable Symfony2 code*. Matthias Noback (September 4, 2013), 2013, p. 230, ISBN: 978-9082120110.

[29] K. Dunglas, *Persistence in PHP with the Doctrine Orm*. Packt Publishing (22 Dec. 2013), 2013, p. 114, ISBN: 978-1782164104.

[30] *Netstat*. endereço: http://linux.die.net/man/8/netstat (acedido em 07/04/2015).

[31] *Redir*. endereço: http://linux.die.net/man/1/redir (acedido em 07/04/2015).