Automação do Processo de Implantação de *Software* usando *Docker* e Microserviços - Um Estudo de Caso

Kássio Gomes Ouverney[1], Daves Martins[2]

[1]Instituto Federal de Educação, Ciência e Tecnologia do Sudeste de Minas Gerais – Campus Juiz de Fora – MG – Brasil

[2]Orientador - Instituto Federal de Educação, Ciência e Tecnologia do Sudeste de Minas Gerais – Campus Juiz de Fora – MG – Brasil

{kassio.ouverney@hotmail.com, davesmartins@gmail.com}

Abstract. *In many software development environments, the process of deploying the system in production has generally been performed manually. The manual deployment process can lead to errors, such as forgetting to upload an update without a database; errors like this generate delays by having long cycles in generating a new version of the application. This article aims to create a methodology for future implementation, a case of study, automation of the process of implementation of software applications based on integration and continuous delivery using the Jenkins tool, through a microservice oriented software development environment using container technology such as Docker.*

Resumo. *Em muitos ambientes de desenvolvimento de softwares, o processo de implantação do sistema em produção tem sido executado de forma manual. O processo de implantação manual pode levar a erros, como esquecer-se de subir uma atualização no banco de dados; erros como esse geram atrasos tendo ciclos longos na geração de uma nova versão da aplicação. Este artigo tem objetivo de criar uma metodologia para implementação futura, a um estudo de caso, da automação do processo de implantação de aplicações de softwares baseada na integração e entrega contínua utilizando a ferramenta Jenkins, isso através de um ambiente de desenvolvimento de software orientado a microserviços usando a tecnologia de container como a Docker.*

1. Introdução

Pesquisas *Institute for Advanced Architecture of Catalonia* (*IAAC*) têm mostrado que na maioria das organizações, o lado das operações de entrega de *software* têm sido contribuinte significativo para o atraso na entrega de *software*. Configurar o *hardware* para testar a compilação de desenvolvimento pode levar tempo variando de dias a semanas [01]. Ao mesmo tempo em que desenvolvedores desenvolvem e entregam novas funcionalidades, a equipe de operação tem o desafio de manter e de avaliar a estabilidade do *software* em produção. Desenvolvedores sempre querem que as mudanças estejam logo em produção, já a equipe de operação se desdobra para colocá-las, pois essas mudanças podem levar risco a estabilidade do sistema ou na sua implantação. Existe então uma divisão do time de desenvolvedores com o time de operações. Essa divisão, muitas vezes, atrasa a entrega de alta qualidade do sistema. Contudo, num mercado competitivo, torna-se fundamental entregar aplicações prontas ao mercado, aos usuários de forma confiável, rápida e com menos riscos.

Assim surge a cultura *DevOps*; uma cultura emergente em que desenvolvimento, testes, equipes de operações colaboram para entregar resultados de forma contínua e eficaz [02]. Inicia-se uma contínua necessidade de melhorar o ciclo de vida nas diferentes fases da aplicação e uma necessidade em diminuir o *Time To Market*, isto é,

necessidade em diminuir o tempo entre a análise e projeção do *software* e sua disponibilização ao mercado. Surgem então a integração contínua e entrega contínua.

A integração contínua detecta mudança que introduza problemas no sistema ou que não atenda aos critérios de aceitação especificados pelo cliente no momento em que foi introduzida. Equipes podem, então, corrigir o problema assim que ele ocorre. Quando essa prática é seguida, o *software* está sempre num estado funcional [03]. Já a entrega contínua é uma extensão natural da integração contínua; a entrega contínua é a grande contribuinte para a diminuição do *Time To Market*. Com ela, a empresa de tecnologia de informação se destaca ainda mais, não só pelo produto em si e pela qualidade, mas também pela velocidade que ela consegue levar o produto ao mercado [04].

Cada vez mais, as aplicações, os serviços precisam de estar mais escaláveis; o desenvolvimento orientado a serviços tem contribuído para tal. Esse modelo de desenvolvimento orientado a serviços, chamado de microserviços, trabalha de forma independente, permite ser executado por escala de serviços e não por aplicação, conforme adotado numa arquitetura monolítica; esta será explicada na subseção 3.2. A arquitetura de microserviços garante o funcionamento da aplicação mesmo em caso de falha num determinado serviço ou mesmo em caso de *bugs* [05].

Implantar a integração contínua e a entrega contínua utilizando desses pequenos serviços, com uma necessidade de mecanismos leves, publicação independente, escalabilidade e portabilidade, tornam-se um desafio para este trabalho. Através dessas práticas e dessa arquitetura poderão fornecer um ambiente ideal para a publicação desses serviços no que diz respeito à velocidade, a gestão de isolamento e ao ciclo de vida da aplicação como um todo, isso de forma contínua e integrada.

Foi realizado um estudo de caso no ambiente de desenvolvimento do sistema "iNtegra", que pertence ao núcleo de Recursos Computacionais do Instituto de Ciências Exatas da UFJF, em que o orientador deste trabalho é o gestor e um dos desenvolvedores do projeto. Diante do estudo feito, foram identificados problemas e possíveis melhorias a serem feitas quanto ao desenvolvimento e à entrega de *software*, são eles: para desenvolvimento do sistema e para provisionamento, isto é, para criar e configurar os ambientes para testes e para produção, atualmente o "iNtegra" utiliza da arquitetura monolítica, ou seja, utiliza de máquina virtual (*VM*), isso exige uma configuração considerável de *hardware* da infraestrutura; além disso, por se tratar de *VM*, caso surja um *bug*, toda a aplicação tem de ser retirada do ar deixando os usuários sem uso; outro problema relevante é que o processo desenvolvimento e de entrega de *software* é realizado de forma manual, isso pode resultar em vários problemas, diante dos passos repetíveis, que tornam esse processo cansativo e exaustivo, o desenvolvedor se esquece de realizar determinadas configurações e a aplicação é posta em produção com *bugs*; às vezes a aplicação é criada de forma instável, testes automatizados acusam *bugs*, porém por desatenção do desenvolvedor isso não é resolvido e a aplicação é posta no ar também com *bugs*; além disso, dos 04 desenvolvedores *in loco*, apenas 01 detém o conhecimento de provisionamento de ambientes, na sua ausência por quaisquer motivos, novas versões da aplicação não são publicadas; o último problema identificado foi que existem longos ciclos de entrega de novas versões da aplicação, entre cada ciclo existem muitas mudanças e ao colocar a aplicação em produção, as chances de *bugs* são maiores. O cenário e a metodologia atual de desenvolvimento do "iNtegra" são explicados na seção 2.

O objetivo deste trabalho foi de desenvolver uma metodologia, um modelo de desenvolvimento que sirva como base para implementação futura, no "iNtegra", de um servidor de integração contínua usando microserviços para provisionamento de ambientes da aplicação. Tem-se o propósito de automatizar o processo de desenvolvimento e de entrega de *software*, desde um *commit* até a aplicação estar em produção. A motivação é que a aplicação seja publicada de forma automatizada

eliminando passos repetitivos e suscetíveis a erros, que a cada novo *build* um novo servidor seja provido já instanciado com a aplicação configurada e instalada, e que a aplicação seja mais escalável evitando que ela seja retirada do ar quando surgir um *bug*.

O presente artigo é dividido da seguinte maneira: a seção 2 apresenta o cenário em que será aplicada a metodologia, seu atual funcional do processo de desenvolvimento de *software* e da entrega do mesmo em produção; a seção 3 apresenta os conceitos fundamentais que norteiam este trabalho - *DevOps*, a arquitetura de microserviços, a tecnologia *Docker*, a integração contínua e a entrega contínua, e o servidor de integração contínua, o *Jenkins*; a seção 4 apresenta a metodologia e a solução proposta; e por fim, a seção 5 apresenta as considerações finais.

2. O Cenário

O sistema "iNtegra" fornece uma conta de *e-mail* institucional para funcionários, professores e alunos; através do sistema também é possível criar listas de *e-mail*, calendários gerenciáveis para cada disciplina, criar grupos de estudos, criar diretórios online para compartilhamento de documentos, de vídeos e de imagens entre os usuários. Os professores conseguem disponibilizar listar de exercícios, trabalhos, textos e planilhas. Por outro lado, os alunos conseguem entregar seus trabalhos. É de fácil uso e de boa experiência com o usuário, pois possui interfaces amigáveis. E existe uma sincronização com o SIGA (Sistema de Gestão Acadêmica) utilizado pela UFJF e uma integração com o *Google*, provendo todos os serviços disponíveis pelo *Google*.

A equipe que mantém o "iNtegra" é composta por 04 funcionários públicos, um dos desenvolvedores é o orientador do artigo e responsável pela gestão do projeto, e 2 alunos bolsistas. Os desenvolvedores têm as responsabilidades de desenvolver, gerenciar, implantar, manter a documentação do sistema, testes, manutenção do sistema e infraestrutura dos servidores.

Para manter a aplicação ou os módulos da mesma no ar, utiliza-se de apenas uma máquina virtual que é instalada num servidor central, nela são instalados e configurados os devidos servidores, e instaladas e configuradas as dependências da aplicação para seu correto funcionamento, isso tudo de forma manual. Toda a aplicação é provisionada nessa máquina virtual.

Adentrando mais na metodologia de desenvolvimento do sistema, as tecnologias usadas para manter todo o "iNtegra" são:

- Sistema operacional: *Ubuntu Server*.
- Ferramenta de desenvolvimento: *Netbeans*.
- Linguagem de desenvolvimento: a linguagem usada é a *Java*. Utiliza-se de *framework* tais como: *Spring*, *JPA*, *CXF* entre outras.
- Servidor *Web*: *Tomcat*.
- Ferramenta de compilação: *Maven*.
- Linguagem de marcação e de estilo: *HTML5*, *JavaScript* "*JQuery*" e o *CSS*. Utiliza-se o *framework Bootstrap*.
- Banco de dados: *MySQL* e *Oracle*.
- Controle de versão: *Subversion*.
- Controle de Tarefas: *Redmine*.

3. Referencial teórico para a solução

Considerando que cada vez mais as empresas buscam a excelência na qualidade do *software* visando a otimização dos recursos, redução no prazo de entrega e instabilidades causadas pelas mudanças e erros frequentes nos sistemas em desenvolvimento, torna-se fundamental ter o conhecimento de práticas e de ferramentas que muito podem auxiliar a evitar problemas e a alcançar os objetivos que as organizações de *T.I.* visam.

3.1. DevOps

Alguns ambientes de desenvolvimento de *software* ainda utilizam de metodologia tradicional de desenvolvimento. Há os analistas de negócios que especificam as regras de negócios da funcionalidade a ser desenvolvida, os desenvolvedores que a codificam, os testadores que testam a nova funcionalidade e validam seu correto funcionamento e o time de operação (administradores de redes, de banco, etc.), que recebe o *software* desenvolvido e o coloca rodando num ambiente para uso. Com o uso de metodologias ágeis de desenvolvimento (*SCRUM* ou *XP*), as fábricas de *softwares* começaram a ter resultados mais rápido e em menor tempo. Essa mudança começou a gerar uma demanda muito grande e inesperada pela equipe de operação; a equipe passou a ter um desafio ainda maior de pôr o sistema em produção e ainda garantir sua estabilidade. Conforme HUMBLE e FARLEY (2014) afirmam que, as equipes de desenvolvimento e de negócios se tornaram mais eficientes com entregas mais rápidas em um período de tempo menor. Porém, toda essa mudança gerou, para a equipe de operação, uma demanda de novos sistemas e de *builds* muito maior do que antes, resultando, muitas vezes, em uma taxa muito elevada de erros, fazendo que, muitas vezes, o código desenvolvido e testado não fosse implantado com rapidez e com confiabilidade necessária.

Inicialmente houve conflitos entre desenvolvedores e a equipe de operação, afinal sempre houve uma barreira cultural e organizacional entre essas equipes [06], bem como:

- Temor: equipes têm receios de compartilhar seus conhecimentos para suas atividades não serem confrontadas e não perderem poder;
- Times separados: em alguns ambientes há times que sempre defenderam seus interesses individuais;
- Falta de padronização no processo de desenvolvimento: os times, trabalhando de maneira separada, tendem a desenvolver sua sintaxe, seu idioma próprio trabalhando em cima de seus problemas, ao invés de desenvolverem um idioma único e geral e que todos se ajudem em prol de um único objetivo.

Com o crescimento contínuo da base de clientes e do alcance do mercado, tornou-se essencial que os processos de entrega interno da organização fossem alinhados com a expectativa de negócios e que fossem otimizados na melhor medida possível. Tempo e recursos sempre foram restrições críticas em qualquer ambiente de negócios. Com essas restrições, esperava-se que as empresas reagissem mais rapidamente às necessidades do mercado com alto nível de qualidade. Nenhuma organização pode dar ao luxo de viver com atividades manuais, propensas a erros e repetidas no ciclo de vida de entrega de *software*. Com o tempo as equipes de projeto identificaram essa precisa necessidade de negócios e passaram a adotar *DevOps* para otimizar seus processos [01].

DevOps é um movimento dentro de engenharia de *software* que busca trazer desenvolvedores de *software* e pessoal de operações em estreito alinhamento, para garantir a harmoniosa tarefa e transição suave de artefatos (aplicações) do projeto através de processos interoperáveis e através de ferramentas. Deve-se notar que os

autores estão discutindo *DevOps* no contexto dos sistemas e processos internos do projeto, além da definição tradicional de implantação e transição entre o desenvolvimento e a equipe de operações. Embora este seja o contexto tradicional do *DevOps*, processos de desenvolvimento de *software* interativos podem fazer uso dos mesmos conceitos e de ferramentas de *DevOps* para permitir a entrega interna contínua de *software*, sistemas e todos os outros artefatos de projeto gerados [07].

Cada empresa de *T.I.* vai aplicar a cultura *DevOps* e adotar princípios da forma que lhe convier e lhe atender. Como já se sabe, é uma cultura que tem como objetivo a comunicação e a colaboração entre as equipes de desenvolvimento e de operação ajudando elas a trabalhar juntas e de forma efetiva. Dentro dessa cultura existe um conjunto de práticas que se tornaram conhecidas e que acabaram por gerar confiabilidade nos ambientes de desenvolvimento, bem como a *pipeline* de implantação, a integração contínua e entrega contínua. Os dois últimos serão abordados na subseção 3.4.

O processo de implantação confiável e repetível é fundamental para a implantação do *DevOps* e isso se faz através da automação criando um *pipeline* de entrega confiável. Uma *pipeline* de implantação é, em essência, um processo automático de entrega de *software*. Isso não implica que não há interação humana com o sistema durante o processo de entrega; ela garante que passos complexos e passíveis de erro sejam automatizados, passíveis de repetição e confiáveis [03]. A *pipeline*, geralmente, estrutura-se a partir de um *commit* feito num servidor de controle de versão, o *build* é feito de forma automática (testes são executados e a aplicação é gerada) e, por fim, há a entrega da aplicação diretamente no ambiente do usuário, isso também de forma automática. Sua implementação varia muito entre projetos. Usá-la como um padrão pode acelerar a criação do processo de compilação e implantação. Seu alvo, sobretudo, é modelar o processo de compilar, implantar, testar e entregar a aplicação. Ela garante que cada mudança passe por esse processo independentemente de forma tão automatizada quanto possível.

3.2. Microserviços

Diante da velocidade que as tendências tecnológicas forçam as arquiteturas das aplicações a evoluir, essas tendências geraram um grande desafio às organizações no tocante ao desenvolvimento de *software* sustentável. Pode-se apontar que a reutilização de componentes de *software* em projetos futuros é um assunto de grande importância no meio especializado, pois tal procedimento pode muito contribuir com o aumento de produtividade, logo reduzindo os custos de manutenção dos *softwares*. A manutenção de um sistema consiste na correção de erros e na implementação de novas funcionalidades. O custo da manutenção de um *software* pode chegar a 70% exigindo técnicas de *software* que privilegiem a extensibilidade, reusabilidade e que suportem o desenvolvimento sistemático de *software* de forma a garantir correção e robustez [18]. Manter uma aplicação no ar de forma escalável e estável mesmo quando surge um *bug* ou outra falha qualquer tem sido um enorme desafio nos últimos anos.

Atualmente, há diversos padrões comumente difundidos para o desenvolvimento de aplicações corporativas, visto que as fases do desenvolvimento não são atividades triviais. A arquitetura monolítica é um padrão amplamente usado para o desenvolvimento de aplicações corporativas. Esse padrão funciona razoavelmente bem para pequenas aplicações, pois o desenvolvimento, testes e implantação de pequenas aplicações monolíticas são relativamente simples [19]. Porém, para aplicações grandes, complexas e muito acopladas, a arquitetura monolítica se torna um obstáculo ao desenvolvimento e à implantação, além de dificultar a entrega contínua e limitar a adoção de novas tecnologias. Para aplicações complexas, que necessitem de alta performance computacional e disponibilidade ininterrupta é mais interessante utilizar

uma arquitetura de microserviços, que divide a aplicação em um conjunto de serviços leves e coesos.

Conforme NEWMAN (2015), microserviços são pequenos serviços autônomos que trabalham juntos e que, nos últimos dez anos, os sistemas distribuídos se tornaram mais refinados, evoluindo de aplicações monolíticas de código pesado para microserviços independentes. Cada serviço é executado num único processo independente, podendo ser implementados como um sistema isolado, uma plataforma como serviço, ou como um próprio processo do sistema operacional.

Antes de adentrar à arquitetura de microserviço, é importante compará-la com a arquitetura monolítica que é construída como um único bloco [19]. As estruturas monolíticas executam em único processo e qualquer mudança no sistema envolvem criação e implantação de uma nova versão da aplicação, isso significa que a cada mudança todo o bloco é suspenso causando indisponibilidade dos outros serviços e dificultando a implantação de novas versões. Para cada linha de código ou biblioteca atualizada todos os times de desenvolvimento precisam de estar cientes da modificação, isso diminui a autonomia dos times podendo interferir nas entregas continuas do projeto. À medida que aplicações monolíticas crescem em estrutura, sua complexidade e dificuldade de gerenciamento aumentam proporcionalmente, causando dificuldade na integração de novos membros nas equipes de desenvolvimento; dependendo da tecnologia utilizada fatores como curva de aprendizado e atualizações de ferramentas de terceiros afetam diretamente o ciclo de desenvolvimento do *software* [20].

A arquitetura de microserviços, por sua vez, propõe a decomposição de sistemas monolíticos em suítes de serviços autônomas, com criação, gerenciamento e implantação independentes entre si. Um grande benefício de microserviços em relação à estrutura monolítica é o isolamento de falhas [20]. Nas estruturas monolíticas se uma funcionalidade falha toda a estrutura fica indisponível, na arquitetura de microserviços se um serviço falha, este é isolado rapidamente, enquanto os outros permanecem em execução, garantido a disponibilidade da aplicação no ar e tornando fácil e rápida a correção de erros.

Outro ponto de fundamental importância da arquitetura de microserviços é o gerenciamento de serviços. Segundo NEWMAN (2015), a gerência do serviço é completamente separada de outros serviços, permitindo que cada estrutura seja construída utilizando diferentes tecnologias e linguagens de programação, usando próprio modelo de dados e garantindo a heterogeneidade dos serviços. Isso possibilita que as equipes de desenvolvimento sejam pequenas, autônomas e mais produtivas, focando nas regras de negócio do projeto e escolhendo pela melhor estrutura para seus serviços, consequentemente se obtém menos linhas de código escritas e novos profissionais podem ser incluído mais facilmente aos projetos. Os microserviços por serem independentes permitem a adoção de novas tecnologias mais facilmente, diminuindo o impacto das mudanças. Caso ocorra uma falha ao experimentar uma nova tecnologia, pode-se descartar o uso e voltar à versão anterior sem afetar todo o projeto. Essa opção é bastante crítica em estruturas monolíticas, nessas estruturas todas as mudanças impactam em todas as funcionalidades, podendo causar problemas de execução e de indisponibilidade.

A figura 1 representa o esquema gráfico da arquitetura de aplicações monolíticas e de microserviços.

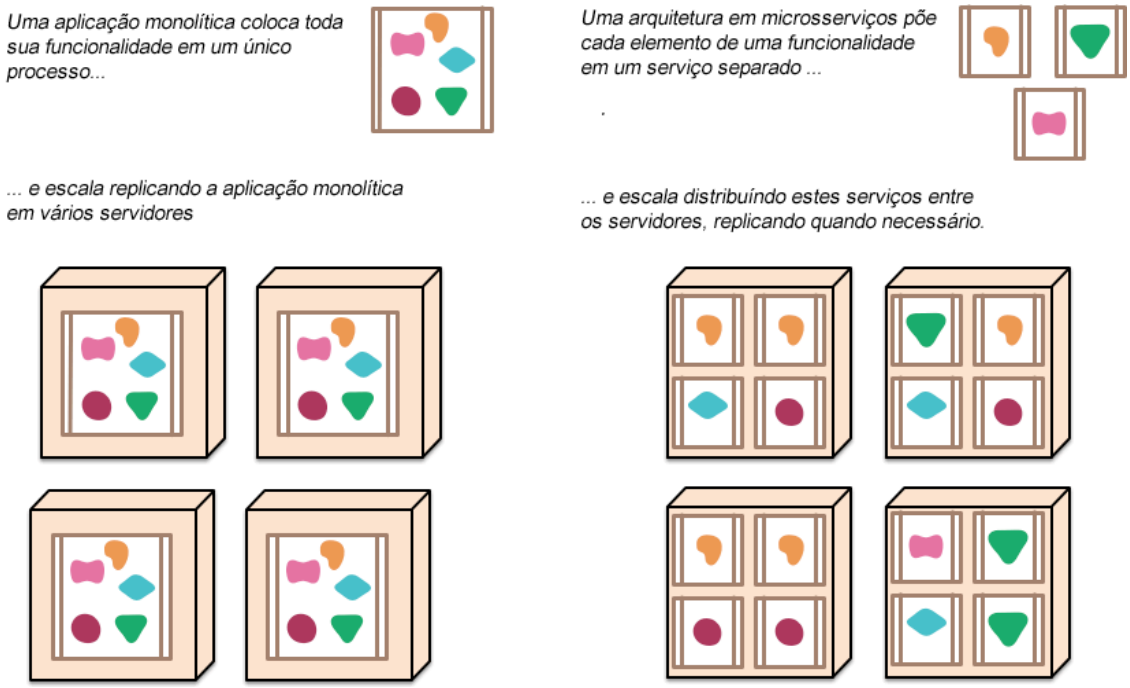

Figura 1. Aplicações monolíticas e microserviços. Fonte: (FLOWER e LEWIS, 2014).

Portanto a arquitetura de microserviços visa reduzir a inércia, fazendo escolhas que favoreçam ao *feedback* e a mudanças rápidas, reduzindo as dependências entre as equipes. Para eliminar a complexidade acidental deve-se substituir os processos complexos, desnecessários e principalmente as integrações também demasiadamente complexas com o propósito de privilegiar o desenvolvimento [20].

3.3. *Docker*

Docker é uma tecnologia de virtualização de servidores; ela foi projetada para ajudar a fornecer aplicativos mais rapidamente usando uma plataforma de virtualização em *containers*, cercada por um conjunto de ferramentas e de fluxos de trabalho que ajudam os desenvolvedores a implementar e gerenciar aplicativos mais facilmente [23].

A ideia do uso do *Docker* é prover a cada *build* um novo servidor já instanciado com a aplicação instalada. Com o uso de microserviços, cada serviço disponível será instanciado em uma máquina isolada não interferindo no funcionamento dos demais serviços.

Conforme SANDOVAL (2015), a plataforma *Docker* se constitui de vários elementos, dentre eles, cabe destacar três elementos principais para seu funcionamento - Imagens *Docker*, *Dockerfiles* e *containers Docker*:

- Imagens *Docker*: são a base de todos os *containers*. Fornecem um sistema de arquivos somente leitura que contém um sistema operacional base ou um servidor base qualquer. As imagens do *Docker* são facilmente fornecidas por vários fornecedores e contêm versões despojadas do sistema operacional. O sistema de arquivos não é, no entanto, baseado no sistema de arquivos do servidor *host*. Elas são auto-suficientes.

- *Dockerfiles*: são planos desenvolvidos pelo desenvolvedor para criar imagens *Docker* personalizadas. É possível no *Dockerfile* adicionar uma aplicação, definir variáveis de ambiente para o *container* a ser criado, definir portas *Transmission Control Protocol* (*TCP*) que serão expostas fora do *container* e

executar comandos dentro do *container* durante o tempo de compilação do *Dockerfile*. O processo de construção de uma imagem envolve a execução de cada instrução dentro de um *Dockerfile* em sua própria camada e, no final, mesclando essas camadas em uma imagem final.

- *Containers Docker*: são instâncias de tempo de execução de imagens. Pode-se iniciar muitos *containers* com base na mesma imagem. Cada *container* possui um identificador exclusivo fornecido pelo *daemon Docker* ou o identificador pode ser definido pelo usuário. Se um ponto de entrada for definido para o *container*, esse processo será iniciado na inicialização. Se nenhum ponto de entrada for definido, pode-se definir um quando o *container* for iniciado. Um *container* só tem acesso à sua própria instância do sistema de arquivos em camadas. O *container* obtém uma camada de leitura e de gravação quando instanciado. Se um *container* precisa editar um arquivo que faz parte de uma das camadas inferiores somente leitura, esse arquivo é copiado para a camada de leitura e gravação de nível superior. O *container* é capaz de editar o arquivo, porém o arquivo não é persistente de volta para a imagem base. Isso fornece isolamento para cada *container*. Vale ressaltar que os *containers* compartilham camadas somente no sentido de que podem ser modificadas em um único artefato. Esse artefato é a imagem criada ao criar a imagem a partir do *Dockerfile*. Cada *container* obtém uma cópia da imagem em tempo de execução.

Em comparação com as máquinas virtuais, o *Docker* compreende apenas a aplicação e suas dependências. Ele é executado como um processo isolado no espaço do usuário no sistema operacional hospedeiro, compartilhando o *kernel* com outros *containers*. Em máquinas virtuais (*VM's*) a aplicação virtualizada inclui não apenas a aplicação, binários e bibliotecas necessárias, mas também um sistema operacional inteiro [19]. Dessa forma, a *Docker* usufrui de isolamento, de alocação, de benefícios e de recursos das *VM's*, entretanto é muito mais portátil e muito mais eficiente do que as *VM's* [21].

A figura 2 representa a comparação do esquema da máquina virtual e do *container Docker*.

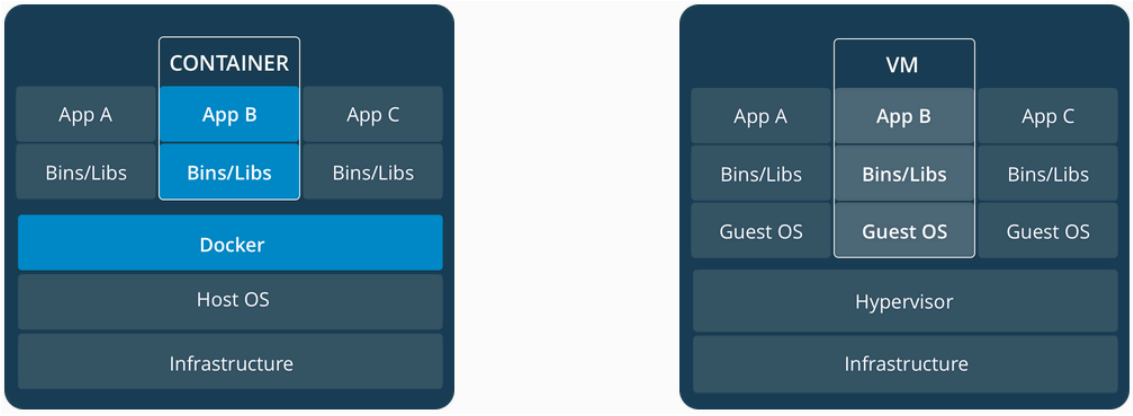

Figura 2. Máquina Virtual x *Container Docker*. Fonte: (*DOCKER INC*, 2013).

Portanto, *Docker* permite que aplicativos sejam montados rapidamente a partir de componentes e elimina o atrito entre os ambientes de desenvolvimento, controle de qualidade e de produção. Como resultado, a *T.I.* pode enviar mais rápido e executar o mesmo aplicativo, sem alterações, em *laptops* ou centro de dados implantados em *VM's* [22].

3.4. Integração continua e entrega contínua

A integração contínua (*IC*) foi mencionada pela primeira vez pelo Kent Beck em seu livro *Extreme Programming Explained* publicado em 1999 [03]. Ela nasceu junto à metodologia de desenvolvimento ágil *Extreme Programming* (*XP*). A *IC* defende a necessidade de serem realizadas pequenas mudanças nos *commits* e nos testes ao invés de grandes e longas mudanças, para que haja redução do risco de *bugs* e facilidade na depuração do código ao mesmo tempo [08].

A *IC* começou como a automação da fase de desenvolvimento apenas, em breve a revolução cobriu a fase de teste da equipe de qualidade de *software* e a implantação em vários ambientes da equipe de operação [09], envolvendo todo o ciclo de vida de um produto e introduzindo um novo conceito posteriormente chamado de Entrega Contínua (*CD*).

IC é uma prática de desenvolvimento de *software* em que membros de uma equipe integram seu trabalho frequentemente, geralmente cada pessoa integra pelo menos diariamente [10]. Com ela, os desenvolvedores conseguem ter um rápido e contínuo *feedback* de determinada funcionalidade desenvolvida, além de poder visualizar o histórico do projeto desenvolvido por outros [03].

Nos ambientes de desenvolvimento de *software* em que se utiliza de metodologias tradicionais, a integração é realizada apenas após o término do desenvolvimento de cada módulo; na integração contínua a integração é feita continuamente sempre que é feito um *commit* ou quando é agendada. A cada integração é gerado um novo *build* automatizado, incluindo a execução dos testes. Isso contribui a fim de que haja menos erros de integração, tornando a manutenção menos custosa, e ao final do processo, os desenvolvedores recebem uma notificação acerca do fracasso ou sucesso como resultado da integração [10].

A *IC* exige que a aplicação seja compilada novamente a cada mudança feita e que um conjunto abrangente de testes automatizados seja executado. É fundamental que, se o processo de compilação falhar, o time de desenvolvimento interrompa o que está fazendo e conserte o problema imediatamente. Assim, o objetivo dela é manter o *software* em um estado funcional o tempo todo [03]. Os defeitos, então, são descobertos mais cedo no processo de entrega do módulo desenvolvido, e tornam-se muito mais barato consertá-los, o que representa ganhos de tempos e de custos.

A integração contínua é uma prática que tem sido muito usada nas organizações, no entanto há variação de como é definida e usada nas organizações. Para usá-la, existem algumas práticas e alguns princípios que são usados por algumas organizações [11]. Essas práticas, que na verdade é o ciclo de desenvolvimento utilizado pela *IC*, trazem inúmeros benefícios, são eles:

- *Build* automatizado: o *build* é composto por várias tarefas como compilação, testes, empacotamento, geração de artefatos e outras. O processo de *build* é rodado toda vez que é gerada uma nova versão [12]. Com a criação de *builds* automatizados pode-se utilizar de *scripts* caseiros e de ferramentas como *Apache Ant e Apache Maven* [13].

- Uso de controle de versão: todos os itens de configuração do projeto são armazenados em um único repositório (*scripts*, códigos, testes, *scripts* de compilação, banco de dados, etc.).

- Elaboração e execução de testes próprios: automatizar testes locais antes de integrar o código desenvolvido.

- Execução de *commits* com frequência no repositório: a integração contínua melhora a comunicação permitindo que desenvolvedores tomem conhecimento

do código desenvolvido por outros para uma maior resolução e rastreabilidade de problemas.

- Execução de um conjunto de testes automatizados e abrangentes: garantir que a aplicação foi bem-sucedida na compilação não garante que ela está funcional, é necessário um conjunto de testes automatizados [03].
- Uso de servidor de integração: o objetivo desse servidor é integrar as novas versões geradas no repositório. Geralmente ele permite a configuração de *builds* para que o processo de integração seja executado. Depois da execução é gerado um relatório em forma de *log* que é enviado aos desenvolvedores como forma de *feedback* [03].

Através desse contínuo *feedback*, isto é, através do resultado obtido, seja positivo ou negativo, tem-se e busca-se sempre a melhoria do processo, e os desenvolvedores conseguem acompanhar se a integração obteve sucesso ou falha, conseguem acompanhar também a qualidade do código gerado.

A figura 3 representa o esquema gráfico da arquitetura de integração contínua.

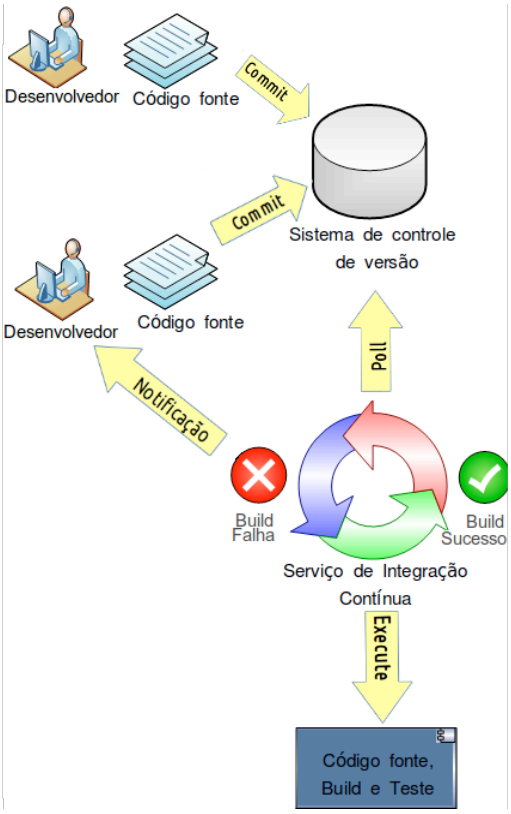

Figura 3. Arquitetura da Integração Contínua. Fonte: (BATHIE, 2010).

A entrega contínua (*CD*), por sua vez, é uma prática que garante a entrega de *software* da equipe de desenvolvedores para o ambiente de produção em um processo confiável, previsível, visível e o mais automatizado possível, com riscos quantificáveis e bem entendidos [03]. Ou seja, ela permite a repetição de processos de forma automatizada, interligando as atividades de desenvolvimento, desde o versionamento do código até a implantação em ambiente de produção.

A entrega contínua permitiu que as equipes de Desenvolvedores, de Testes (Qualidade de *Software*) e de Operação trabalhassem em estreita colaboração e abraçassem pontos chaves de *CD* [15], bem como: colaboração em todas as equipes envolvidas no ciclo de vida do produto; e automação extensiva do processo de entrega.

Implementar *CD* significa ser capaz de encadear as diferentes etapas envolvidas na *pipeline* de implantação entre as equipes e automatizar sua execução desde o *checkout* do código e os testes unitários, até a análise de código estático e os testes de desempenho [03].

Com a entrega contínua é possível entregar novas versões de *software* para produção, diminuindo o *gap*, isto é, diminuindo a lacuna entre a ideia de desenvolver o *software* e a disponibilização do *software* ao usuário final, através da automação de todo o sistema de entrega (*build*, implantação, teste e liberação) [16]. As técnicas e práticas de entrega contínua reduzem o tempo e o risco associados à entrega de novas versões do *software* para os usuários, permitindo assim o aumento de *feedback* e a colaboração entre desenvolvedores, testadores e o pessoal de operação responsáveis pela entrega [03].

A entrega contínua de *software* tem como objetivo: entregar *software* de uma maneira mais rápida e frequente, adicionando valor ao negócio e obtendo *feedback* o mais rápido possível; aumentar a qualidade e estabilidade do *software*; reduzir o risco de uma versão desenvolvida realizando e executando testes em ambientes de testes e ambientes semelhantes ao de produção; reduzir o desperdício e aumentar a eficiência no processo de desenvolvimento; e entregar e manter o *software* em um estado de pronto em que se pode implantar o mesmo de acordo com sua necessidade [17].

Logo, a entrega contínua é o passo final do *pipeline* de implantação. Aqui a *pipeline* possui um destaque, pois elas existem para criar um processo confiável, automatizado e passível de repetição, para que mudanças cheguem à produção o mais rápido possível; a *pipeline* existe para ajudar a criar *software* de mais alta qualidade usando um processo de mais alta qualidade, reduzindo assim, o risco de entregas [03].

3.5. *Jenkins*

Jenkins é um servidor de integração contínua, de automação autônoma e de código aberto, que pode ser usado para automatizar todo tipo de tarefas, como construção, teste e implantação de *software*. O *Jenkins* pode ser instalado através de pacotes de sistema nativos, via *containers Docker*, ou mesmo executado de forma independente por qualquer máquina com o *Java Runtime Environment* (*JRE*) instalado [25].

Esse servidor foi criado para atingir basicamente dois objetivos [26]:

- Construir e testar projetos de *softwares* continuamente. A ferramenta permite fácil implementação da Integração Contínua dos sistemas, tornando mais fácil aos desenvolvedores a integração de suas alterações do projeto e permite aos usuários obterem a construção atualizada do sistema;
- Monitorar as execuções das tarefas, através de agendador de tarefas e de *e-mails*; este monitoramento pode ser utilizado também para as tarefas que são executadas de forma remota. *Jenkins* mantém o resultado dessa execução remota e torna fácil para a equipe ser notificada quando algo de errado ocorrer.

Segundo BOAGLIO (2016), um *job* é uma tarefa que o *Jenkins* deve fazer. Geralmente, tem alguns parâmetros de execução, juntamente com alguns procedimentos que podem ser feitos antes ou depois de sua execução. Ou seja, é a construção, uma execução de um *job*, por exemplo, o procedimento de montar um pacote, que pode envolver *download*, complicação ou testes da aplicação, e, portanto, a criação do artefato. O *build* pode ser considerado como uma instância de um *job* e possui diferentes status podendo ser um *build* com sucesso, um *build* falho ou um *build* instável.

Ainda segundo BOAGLIO (2016) explica que, a integração do *Jenkins*, com os diversos tipos de servidores e sistemas é feita por meio de *plugins*, não só para melhorar

o funcionamento existente, mas também para criar um que não existe. Os *plugins* enriquecem mais o funcionamento do servidor *Jenkins*.

O processo de automatizar o *build* surge de forma a beneficiar o desenvolvimento do dia a dia. Tendo em vista que todo o projeto, todo o código-fonte está num repositório, num servidor de controle de versão, *Jenkins* faz o *download* do projeto e de suas dependências para que, diante das configurações específicas num *job*, *plugins* possam controlar todo o código-fonte e possam determinar o que será feito com o projeto; além disso testes são executados no momento da geração do artefato; e tendo terminado o *build*, são gerados relatórios de testes do projeto e das classes do projeto relatando quais testes passaram e quais testes não passaram.

Cabe destacar que, mesmo o *build* falhando ou sendo instável, é possível configurar que os desenvolvedores sejam notificados via *e-mail*, assim, quaisquer problemas existentes na execução, após tomar conhecimento do mesmo, é possível rapidamente e de forma controlada, realizar as devidas correções antes que outro membro da equipe crie novas funcionalidades e gere mais problemas podendo impactar o projeto.

4. Metodologia e Solução Proposta

Os testes foram realizados com uma aplicação na qual é um sistema *web* em que o professor de uma disciplina cadastra seu aluno e esse registro vai a um banco de dados. Foram criados três *containers*, um para o Sistema de Gerenciamento de Banco de Dados *MySQL* contendo o banco de dados; outro para a *api* em que terá uma máquina com o sistema operacional *Ubuntu* instalado contendo o *MySQL*-cliente, o *Java* instalado e o arquivo de *Java* contendo bibliotecas, dependências e configurações; e um terceiro *container* para a aplicação *web* em si contendo o *Tomcat* e o artefato (a aplicação *web*). Esses três *containers* foram interligados.

Primeiramente foram criados os *Dockerfiles* para cada *job*. Cada um foi colocado na raiz de seu projeto no servidor de controle de versão *Subversion*. A figura 4 representa o *Dockerfile* do banco de dados *MySQL*-servidor.

Dockerfile

```
FROM mysql:5.7.9
MAINTAINER kassio.ouverney@hotmail.com

RUN apt-get update && \
    apt-get upgrade -y --force-yes && \
        apt-get install -y  nmap && \
    apt-get clean

#Adiciona o banco ao docker
ADD script.sql /docker-entrypoint-initdb.d/script.sql
#Senha para o banco
ENV MYSQL_ROOT_PASSWORD=0123
```

Figura 4. *Dockerfile* do banco *MySQL*-servidor. Fonte: (O autor).

Quando foi instanciado o *container* desse *Dockerfile*, a máquina já foi criada com o *MySQL*-servidor instalado e já adicionado o banco e passado sua senha.

Já para o *job* da *api*, a figura 5 representa o seu *Dockerfile* configurado.

Dockerfile ×

```
FROM ubuntu
MAINTAINER kassio.ouverney@hotmail.com
RUN apt-get update && \
 apt-get upgrade -y && \
 apt-get install -y --allow-unauthenticated software-properties-common nmap mysql-client && \
 add-apt-repository "deb http://ppa.launchpad.net/webupd8team/java/ubuntu xenial main" && \
 apt-get update && \
 echo oracle-java8-installer shared/accepted-oracle-license-v1-1 select true | /usr/bin/debconf-set-selections && \
  apt-get install -y --allow-unauthenticated oracle-java8-installer && \
  apt-get clean
#Adiciona o arquivo de java (bibliotecas, dependências e configurações) ao container futuro
ADD target/api-0.0.1-SNAPSHOT.jar /home/api.jar
ENTRYPOINT ["java", "-jar", "/home/api.jar"]
```

Figura 5. *Dockerfile* da *api*. Fonte: (O autor).

Quando foi instanciado o *container* desse *Dockerfile*, a máquina já foi criada com o sistema operacional *Ubuntu* instalado e já instalado também o *MySQL*-cliente, o *Java*, as bibliotecas e as dependências - o arquivo de *Java .jar* - já adicionado.

Para o último *job*, o da aplicação *web*, a figura 6 representa o seu *Dockerfile* configurado.

Dockerfile ×

```
FROM tomcat:8.0
MAINTAINER kassio.ouverney@hotmail.com
RUN apt-get update && \
    apt-get upgrade -y --force-yes && \
        apt-get install -y  nmap links2 && \
    apt-get clean
ADD target/webDev-1.0-SNAPSHOT.war /usr/local/tomcat/webapps/webDev.war
CMD ["catalina.sh","run"]
```

Figura 6. *Dockerfile* da aplicação *web*. Fonte: (O autor).

Quando foi instanciado o *container* desse *Dockerfile*, a máquina já foi criada com o servidor *Tomcat* instalado e adicionado a ele a aplicação *web*.

A ferramenta para compilação, implantação, teste e tarefas de versão utilizada para os projetos foi a *Maven*. Esta foi escolhida devido à sua importância de suporte para o gerenciamento automático de bibliotecas *Java* e de dependências entre projetos, que geralmente, é um dos problemas mais complicados em grandes projetos *Java*.

Foi configurado os três *jobs*. A figura 7 representa os *jobs* na tela inicial do *Jenkins*.

Delivery Pipeline Plugin | Tudo | +

S	W	Nome ↓	Último sucesso	Última falha	Última duração	
		api	4 minutos 11 segundos - #24	14 dias - #17	57 segundos	
		banco	4 minutos 22 segundos - #4	N/D	3,4 segundos	
		webDev	3 minutos 6 segundos - #18	14 dias - #11	14 segundos	

Figura 7. Tela inicial do *Jenkins* com os *jobs* configurados. Fonte: (O autor).

Os *jobs* tiveram seus *builds* estáveis e realizados com sucesso estando com o status azul; pode acontecer de as vezes, um teste não passar, com isso o *job* ficaria com o status amarelo tendo seu *build* instável, mesmo assim a aplicação seria compilada e gerada normalmente. Contudo não é o caso, pois todos os *builds* ficaram estáveis.

Antes de criar e configurar a *pipeline* de implantação para ligar os *jobs* entre si, foi preciso configurar cada *job*.

O primeiro *job* da *pipeline* foi o "banco"; este é o banco de dados *sql* em que mantém o registro dos alunos cadastrados; nele foi configurado para o mesmo consultar o servidor de controle de versão de 30 em 30 minutos; caso tenha tido qualquer

alteração no projeto, ao finalizar esse intervalo, automaticamente o *Jenkins* consulta o servidor de controle de versão e roda o *job* e gera uma nova versão do projeto; além disso, esse agendamento configurado apenas no primeiro *job* faz executar a *pipeline* e roda todos os *jobs* nela configurados. Nos demais *jobs* não foi preciso configurar o agendamento, pois uma vez executada a *pipeline* a partir do primeiro *job*, o *Jenkins* saberá o *job* que teve alteração, se sim, gera uma nova versão da aplicação em questão. Esse intervalo de consulta ao servidor de controle de versão pode ser também definido e agendado em horas ou em dias.

Em seguida, ainda nesse primeiro *job*, foi configurado para pós *build*, executar em *shell* em que foram executados comandos via *Docker* para criar a imagem do *Dockerfile* do banco e em seguida para gerar o *container* da imagem gerada. A figura 8 representa essa execução.

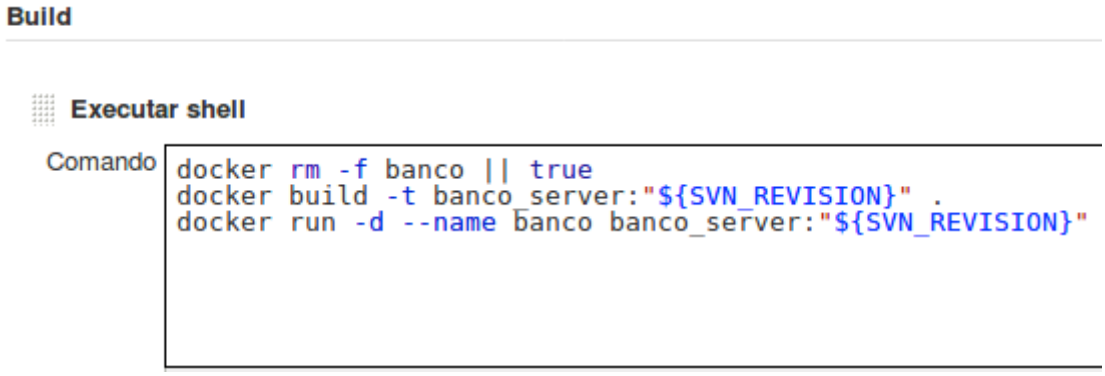

Figura 8. Pós *build* - Executar em *shell* comandos via *Docker*. Fonte: (O autor).

Já o segundo *job* é o "*api*". Para a correta configuração da *pipeline*, a partir do segundo até o último *job*, foi preciso configurar em cada *job*, o *job* que foi executado anteriormente e ainda configurar que o atual *job* só será executado se o anterior for apenas tido como estável, ou mesmo se o anterior estiver instável ou falhar. A figura 9 representa essa configuração.

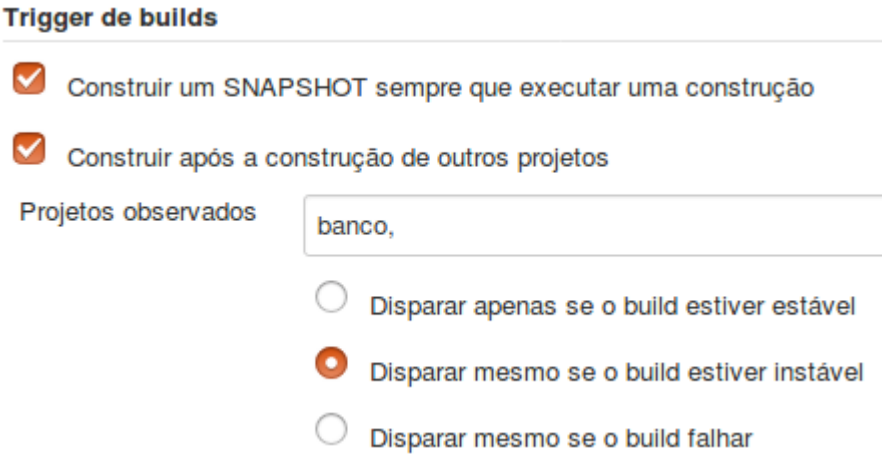

Figura 9. Requisito para a criação da *Pipeline*. Fonte: (O autor).

Ainda no segundo *job*, foi também configurado para pós *build* para executar em *shell* via *Docker*. A figura 10 representa essa execução.

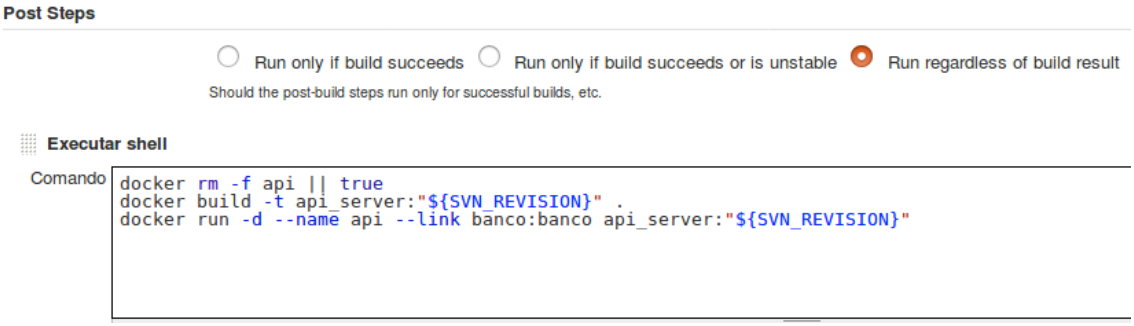

Figura 10. Pós *build* - Executar em *shell* comandos via *Docker*. Fonte: (O autor).

Nessa configuração também foi criada a imagem do *Dockerfile* e gerado o *container*, agora para a *api*; o *container* do banco de dados *sql* (o gerado pelo *job* anterior) foi *linkado* ao *container* recém-criado. Foi criado esse *link* para que fossem trafegadas informações entre os *containers* de forma segura.

Já para o terceiro *job* que é a aplicação *web* em si foi configurado da seguinte maneira no pós *build*. A figura 11 representa essa execução.

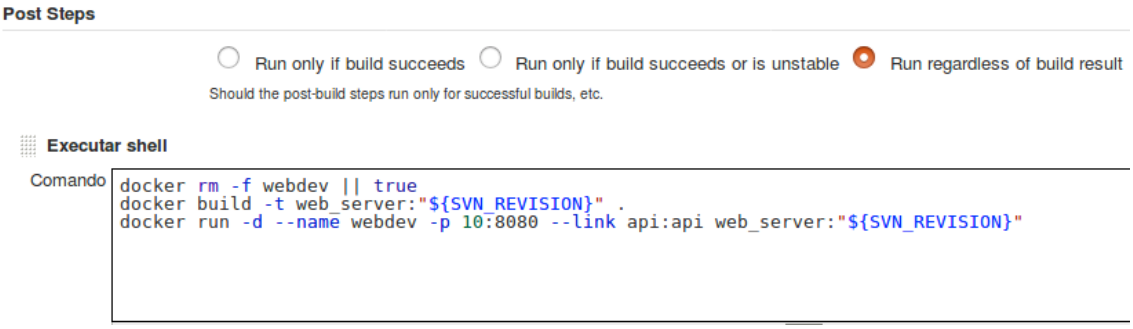

Figura 11. Pós *build* - Executar em *shell* comandos via *Docker*. Fonte: (O autor).

Nessa configuração foi também criada a imagem do *Dockerfile* e gerado o *container*, agora para a aplicação *web* em si e o *container* foi exposto à porta 10 para acesso no navegador *web*. Esse *container* por sua vez foi *linkado* ao *container* da *api*.

Entende-se melhor aqui a ideia dos microserviços, isto é, cada aplicação está num pequeno serviço e cada um desses serviços se interligam. Cabe então destacar e reafirmar que se houver um *bug* na aplicação do banco de dados, por exemplo, mesmo no momento em que ocorre a correção do *bug*, a aplicação toda continuará a funcionar, pois o serviço da aplicação *web* está em funcionamento.

Uma vez configurados os *jobs*, foi criada e configurada a *pipeline* de implantação representada na figura 12. Foi utilizado o *plugin* "*Delivery pipeline*" para criação da mesma.

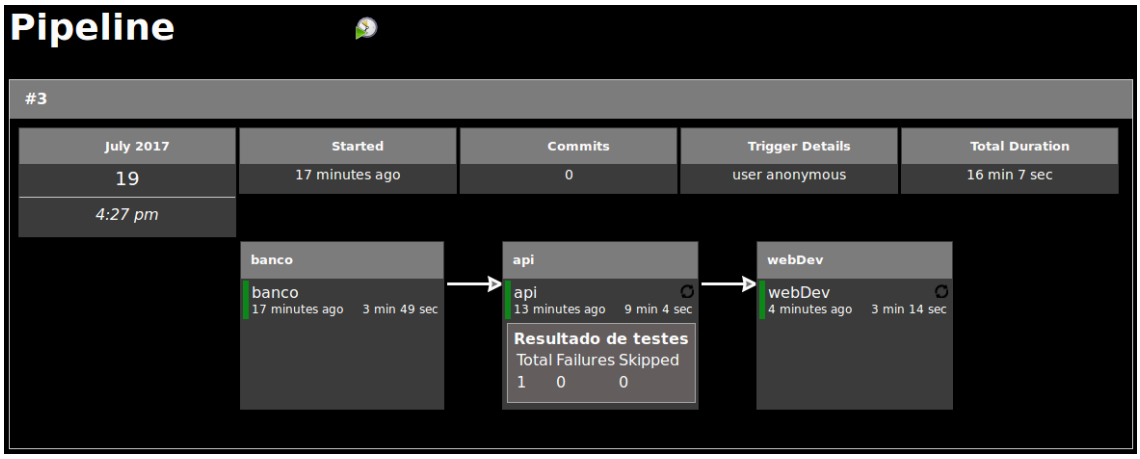

Figure 12. *Pipeline* de Implantação. Fonte: (O autor).

Através dela é possível visualizar a cadeia dos *jobs* em sequência. Quando a execução da *pipeline* terminou foi possível visualizar o resultado dos testes do projeto do *job* e caso algum *job* viesse a falhar, seria possível também reexecutá-lo separadamente a partir da própria tela. Consoante a figura 12, todos os *jobs* tiveram seus *builds* estáveis e realizados com sucesso. Caso algum *job* viesse a ter seu *build* instável ou falho devido a falhas em testes, o desenvolvedor seria imediatamente notificado por *e-mail* através de um relatório, assim se conseguiria saber em qual classe houve as falhas de testes, a correção seria feita rapidamente, seria feito um novo *commit* no servidor de controle de versão e o *build* desse *job* poderia ser executado separadamente ou poderia ser deixado para rodar no próximo agendamento da *pipeline*.

A primeira execução da *pipeline* durou cerca de 16 minutos. Já na segunda execução em diante durou menos tempo, cerca de 1 minuto. A primeira execução dura mais tempo, pois para executar o *build* e gerar a aplicação, o *Jenkins* faz o *download* das dependências do projeto em questão, e após gerar o artefato (a aplicação) com sucesso, diante da configuração no pós *build*, a imagem do *Docker* baseada no *Dockerfile* é criada e é feito o *download* de tudo que foi configurado no *Dockerfile*, e o *container* é gerado com a aplicação já configurada e já em execução. Já nas execuções seguintes da *pipeline*, não foi preciso de que o *Jenkins* fizesse o *download* das dependências do projeto, pois todas as dependências já estavam no *cache* local do *Maven*; além disso, a imagem que o *Dockerfile* se baseou, seu *download* já tinha sido feito, para o servidor local, quando foi criada pela primeira vez a imagem em que o *container* se baseou, assim a execução dura menos tempo.

Uma vez que a execução da *pipeline* terminou, tem-se as aplicações em funcionamento dentro de seus respectivos *containers*. A figura 13 representa os *containers* em execução.

```
kassio@kassio-Dell-System-XPS-L702X:~$ docker ps
CONTAINER ID        IMAGE               COMMAND                  CREATED             STATUS              PORTS                    NAMES
4ff1d8115b72        web_server:119      "catalina.sh run"        6 minutes ago       Up 6 minutes        0.0.0.0:10->8080/tcp     webdev
c4c37a4130ef        api_server:120      "java -jar /home/a..."   9 minutes ago       Up 9 minutes                                 api
2b0d62d00a3a        banco_server:112    "/entrypoint.sh my..."   18 minutes ago      Up 18 minutes       3306/tcp                 banco
```

Figure 13. *Containers* em execução. Fonte: (O autor).

Por fim, a aplicação *web* foi executada e acessada com sucesso sendo representada pela figura 14.

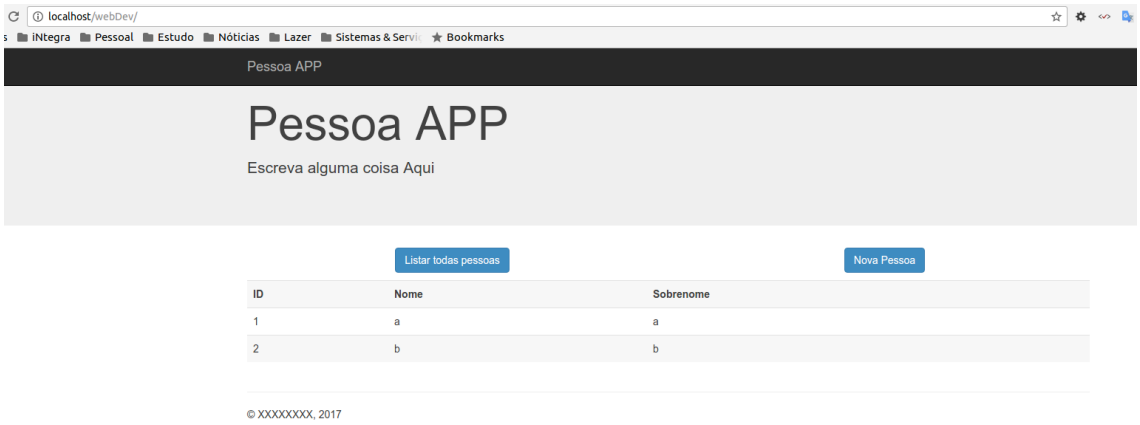

Figure 14. Aplicação *web*. Fonte: (O autor).

5. Considerações finais

A metodologia criada e usada neste trabalho se mostrou viável e confiável. Foi apresentada ao Daves Martins, que é o gestor e um dos desenvolvedores do projeto "iNtegra" e o mesmo aprovou. O processo de implantação do sistema já pode ser automatizado.

Com o uso do servidor de integração contínua e de *containers*, foi possível resolver os problemas elencados na introdução. Com essa metodologia, não mais precisaria de uma alta configuração de *hardware* da infraestrutura para provisionar ambientes, pois os *containers* são muito leves; caso surja um *bug* num dos *containers* não mais a aplicação toda tem de ser retirada do ar, a aplicação se torna mais estável e escalável; o processo de desenvolvimento e de entrega de *software* se tornou automatizado, fazendo que esse processo não dependa mais exclusivamente de uma só pessoa para ser executado e proporcionou mais qualidade à aplicação; por fim, essa metodologia força a adoção da prática de se ter mais *deploys* da aplicação, isso faz que o intervalo entre os ciclos de entrega da aplicação seja menor e faz que a chance de haver *bugs* seja também menor.

A utilização da integração contínua pode ser considerada como uma nova técnica de desenvolvimento de *software* e também pode estar relacionada à qualidade de *software*, ou seja, pode estar relacionada a evolução de padrões e à qualidade nos processos. E com a utilização de *containers* tornou o sistema mais escalável e portátil.

Referências

[01] M. Virmani. Understanding DevOps & bridging the gap from continuous integration to continuous delivery. Fifth International Conference on the Innovative Computing Technology (INTECH 2015).

[02] M. Soni. End to End Automation on Cloud with Build Pipeline: The Case for DevOps in Insurance Industry, Continuous Integration, Continuous Testing, and Continuous Delivery. 2015 IEEE International Conference on Cloud Computing in Emerging Markets (CCEM).

[03] HUMBLE, J.; FARLEY, D. ENTREGA CONÍNUA: COMO ENTREGAR SOFTWARE DE FORMA RÁPIDA E CONFIÁVEL. Porto Alegre: Bookman, 2014. 464 p.; 25cm.

[04] A. Valentina. Continuous Delivery with Jenkins: Jenkins Solutions to Implement Continuous Delivery. 2015 IEEE/ACM 3rd International Workshop on Release Engineering.

[05] Durães, Ramon. Introdução ao conceito de Microservices. 2015. Disponível em:< http://www.ramonduraes.net/2015/05/10/introduo-ao-conceito-de-microservices/>. Acesso em: 13 mai. 2017.

[06] HUTTERMANN M. DevOps for Developers: Integrate Development and Operations, The Agile Way. 2012.

[07] COIS, C; YANKEL, J; CONNELL, A. Modern DevOps: Optimizing software development through effective system interactions. Professional Communication Conference (IPCC), 2014 IEEE International.

[08] K.Beck and M.Flower, Planning extreme programming. Addison Wesley, 2000

[09] C.Le Clerc, "Pre-requisites for a successful enterprise Continuous Delivery implementation", Betanews, Janeiro 2015. Disponível em:< http://betanews.com/2015/01/13/pre-requisites-for-a-successfulenterprise-continuous-delivery-implementation/>. Acesso em: 27 mai. 2017.

[10] FOWLER, Martin. CONTINUOUS INTEGRATION. 2006. Disponível em:< http://martinfowler.com/articles/continuousIntegration.htm>. Acesso em: 27 mai. 2017.

[11] STAHL D.; BOSCH J. Modeling Continuous Integration Practice Differences in Industry Software Development. 2013.

[12] SHORE, James; WARDEN, Shane.O'Reilly. Sebastopol - The Art Of Agile Development - 2008.

[13] GARCIA, Francisco A. Integração Contínua: da teoria à prática. Java Magazine. DevMedia. Rio de Janeiro, 2013.

[14] BATHIE, Mark. CONTINUOUS INTEGRATION: BASIC OVERVIEW AND BEST PACTICES. 2010. Disponível em:< http://blogs.collab.net/cloudforge/continuous-integration-overview-best-practice/>. Acesso em: 27 mai. 2017.

[15] M. Fowler, "Continuous Delivery", martinfowler.com, 2013. Disponível em:< http://martinfowler.com/bliki/ContinuousDelivery.html>. Acesso em: 27 mai. 2017.

[16] DUVALL M. P. Continuous Delivery Patterns and AntiPatterns in the Software LifeCycle. 2011.

[17] WOOTTON B. Preparing for Continuous Delivery. 2013.

[18] BIGONHA, Roberto S. Programação Modular em C++ Qualidade de Software, 2008. Disponível em:< homepages.dcc.ufmg.br/~mtov/pmslides/1sp/qualidade.pdf>. Acesso em: 01 jun. 2017.

[19] LEWIS, James; FLOWER, Martin. Microservices, 2014. Disponível em:< http://martinfowler.com/articles/microservices.html>. Acesso em: 01 jun. 2017.

[20] NEWMAN, Sam. Building Microservices. Sebastopol: O'Reilly, 2015.

[21] DOCKER INC. What is Docker. Disponível em:< https://www.docker.com/whatisdocker>. Acesso em: 24 jun. 2017.

[22] O.S.Tezer. How To Install and Use Docker: Getting Started. Disponível em:< https://www.digitalocean.com/community/tutorials/how-to-install-and-use-docker-getting-started>. Acesso em: 24 jun. 2017.

[23] D. Jaramillo. Leveraging microservices architecture by using Docker technology. SoutheastCon 2016.

[24] DOCKER INC. What is a Container. Disponível em:< https://www.docker.com/what-container>. Acesso em: 24 jun. 2017.

[25] JENKINS. Jenkins Documentation. Disponível em:< https://jenkins.io/doc/>. Acesso em: 24 jun. 2017.

[26] KAWAGUCHI, Kohsuke. Who's Koshuke? (2014). Disponível em:< http://kohsuke.org/about/>. Acesso em: 28 jun. 2017.

[27] BOAGLIO, F. (2016). Jenkins: Automatize tudo sem complicações. ISBN: 978-85-5519-153-4

[28] Sandoval, Robert. A Case Study in Enabling DevOps Using Docker. Disponível em:< https://repositories.lib.utexas.edu/bitstream/handle/2152/32322/SANDOVAL-MASTERSREPORT-2015.pdf>. Acesso em: 01 ago. 2017.