GERÊNCIA DE CONFIGURAÇÃO

Sumário

1. Introdução

1.1 Contexto

Um dos maiores desafios do desenvolvimento de soluções é realizar entregas frequentes, no prazo mais curto possível, com elevado nível de qualidade e a custos adequados, conforme a perspectiva do cliente. A questão central passa a ser transformar uma ideia, problema ou oportunidade, em solução efetiva entregue o mais rápido possível.

Há várias questões que funcionam como anti-padrões ou fatores que limitam a entrega rápida de soluções, dentre as quais podem ser citadas [Humble]:

- Processo manual de construção, empacotamento e implantação;
- Implantação em um ambiente de produção apenas após o fim do desenvolvimento;
- Longos ciclos de desenvolvimento com correspondentes ciclos de *feedback*;
- *Feedback* falho em todo processo;
- Gestão de configuração manual, especialmente dos diversos ambientes, como o de produção;
- Funções segregadas em silos.

Alguns princípios norteadores são fundamentais para acelerar os ciclos de entrega [Humble]:

- Crie um processo de liberação de software repetível e confiável;

- Automatize quase tudo;
- Mantenha tudo sob controle de versões;
- Promova *feedback* constante;
- Se machuca, faça mais frequentemente, e antecipe a dor.
- Construa qualidade embutida;
- Feito significa liberado;
- Todos são responsáveis pelo processo de entrega;
- Melhoria contínua.

Para encarar os desafios citados, ao longo dos anos várias abordagens têm sido adotadas tais como integração, liberação e implantação contínuas. A figura a seguir mostra a interligação desses conceitos, por meio de um pipeline de liberação:

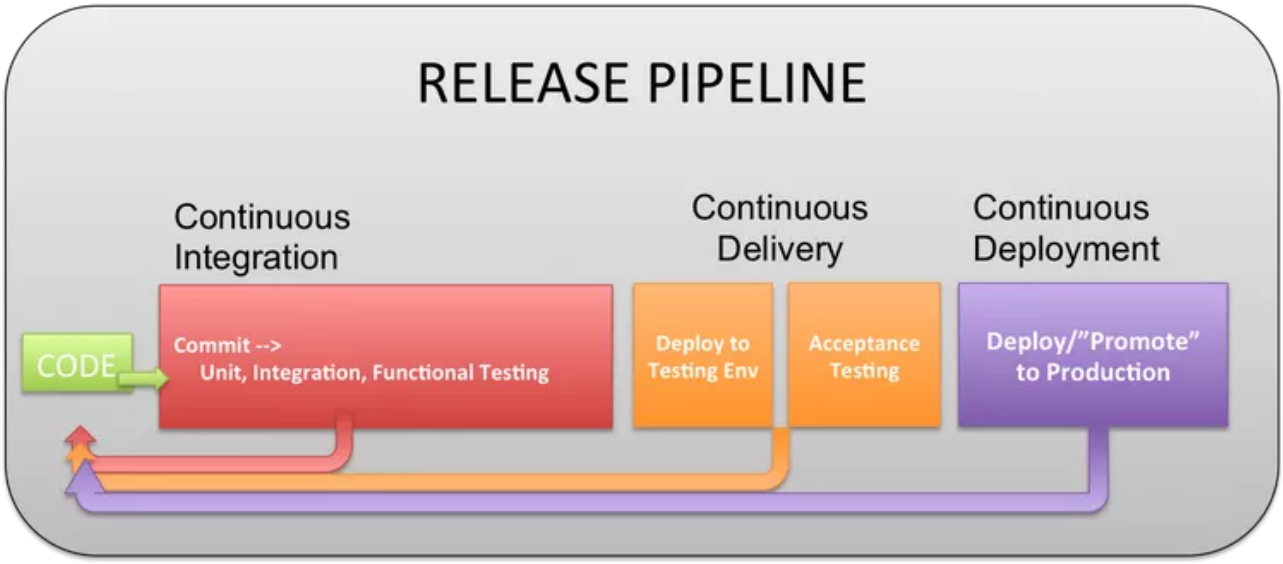

Figura 1 – Integração, liberação e implantação contínuos [Riley]

1.2 Integração contínua

Integração contínua é uma prática que tem como objetivo permitir que o software esteja em um estado de trabalho válido o tempo todo [Duvall].

A integração contínua requer que a cada vez que alguém comita qualquer mudança, a aplicação toda é construída (build) e um conjunto abrangente de testes automatizados é executado. Se a build ou os testes falham, o time de desenvolvimento interrompe o que está sendo feito para corrigir o problema imediatamente.

Com a integração contínua o software é posto à prova a cada nova mudança – e todos sabem o momento em que quebra e podem corrigir imediatamente.

A figura a seguir apresenta os principais componentes de um sistema de integração contínua.

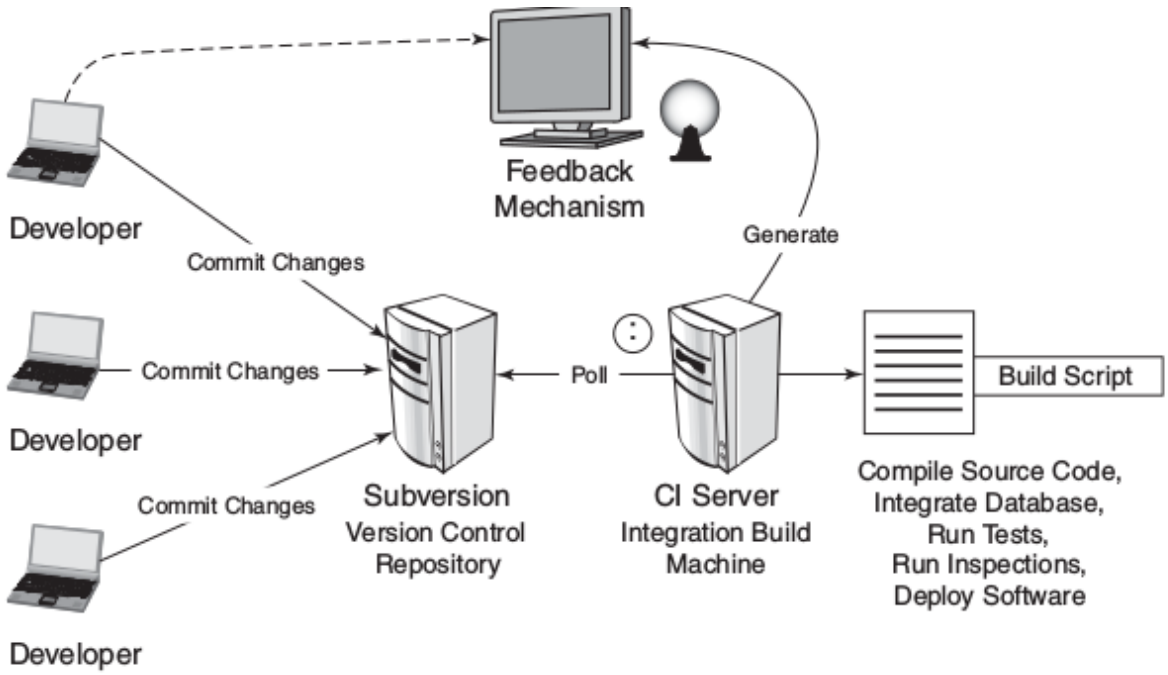

Figura 1 - Componentes de um sistema de Integração contínua [Duvall]

Os seguintes elementos são necessários para a integração contínua [Humble]:

- Controle de versões;
- Build automatizado;
- Acordos do time.

Processo geral de integração contínua [Humble]:

1. Verificar se o processo de build está rodando. Se estiver, aguarde o término. Caso haja falhas, corrija;
2. Uma vez concluído o processo de build e aprovado nos testes, obtenha atualizações do código no seu ambiente de desenvolvimento, a partir do controle de versões;
3. Execute o script de build e rode os testes localmente;
4. Faça o checkin no controle de versões;

5. Espere o servidor de integração contínua rodar o build com suas mudanças;
6. Caso ocorra erros, corrija-os imediatamente;
7. Mova para a próxima tarefa.

Alguns aspectos são pré-requisitos para a integração contínua [Humble]:

- Faça checkin regularmente;
- Crie uma abrangente suíte de testes automatizados;
- Mantenha curtos os processos de build e teste;
- Gerencie o ambiente de trabalho.

Algumas práticas essenciais são essenciais para a integração contínua [Humble]:

- Não faça checkin de uma build quebrada;
- Sempre execute todos testes localmente antes de fazer o commit;
- Espere os testes de commit passarem para avançar;
- Nunca vá para casa com uma build quebrada;
- Sempre esteja preparado para reverter a uma versão anterior;
- Defina um timebox antes de reverter;
- Não comente testes falhos;
- Assuma a responsabilidade por todas quebras resultantes de seus commits;
- Adote TDD.

1.3 Liberação contínua

A liberação contínua (continuous delivery) é uma abordagem que tem com objetivo transformar em rotina a entrega de soluções [Humble]. O conceito central para viabilizar a liberação contínua é o pipeline de implantação.

O plpeline de implantação [Humble] diz respeito a uma implementação automatizada do processo de build, deploy, teste e release. As fundações do pipeline estão no processo de integração contínua e trata, em essência, os princípios da integração contínua levados a sua conclusão lógica. O propósito do plpeline de implantação é triplo:

- Tornar cada parte do processo de build, deploy, teste e release, visível para todos envolvidos, apoiando a colaboração.
- Melhorar o feedback de forma que problemas são identificados e resolvidos o quanto antes no processo.
- Habilitar o time a fazer o deploy e release de qualquer versão do software em qualquer ambiente desejado por meio de um processo completamente automatizado.

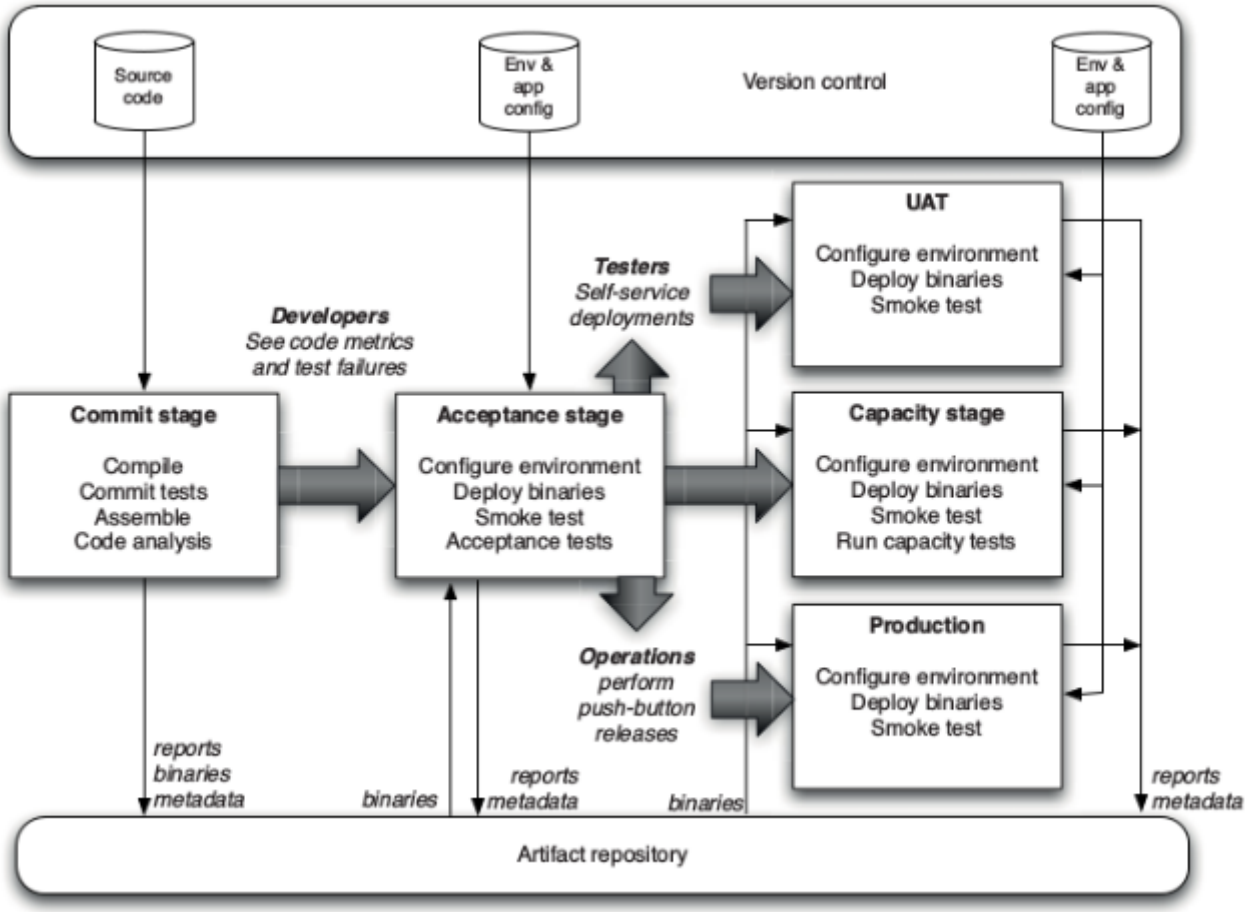

Figura 2 - Exemplo de plpeline de implantação [Humble]

Em linhas gerais, o plpeline de implantação funciona da seguinte forma [Humble]:

- Cada mudança feita na configuração, código fonte, ambiente ou dados, dispara a criação de uma nova instância do pipeline;
- Um dos primeiros passos no pipeline é criar binários e instaladores;
- O resto do pipeline executa uma série de testes nos binários pra provar que podem ser liberados;
- Cada teste a que a release candidata passa dá mais confiança de que essa combinação particular de código binário, informação de configuração, ambiente e dados irá funcionar;
- Se a release candidata passar em todos testes, ela pode ser liberada.

Para uma implementação efetiva, o pipeline de implantação traz uma série de práticas [Humble]:

- Construir binários apenas uma vez (corolário: publicar o mesmo build em vários ambientes);
- Fazer o deploy da mesma forma em todos ambientes (ex.: usar um arquivo de propriedades para cada ambiente);
- Fazer testes fumaça nos deployments;
- Fazer o deploy em um ambiente próximo ao de produção;
- Fazer cada mudança propagar pelo pipeline instantaneamente;
- Parar o processo caso falhe qualquer estágio do pipeline.

1.4 Implantação contínua

Implantação contínua segue a liberação contínua e implanta automaticamente no ambiente de produção todas as mudanças aprovadas nos testes automatizados [Riley].

O maior benefício obtido com a implantação contínua diz respeito a uma redução do tempo de entrega (lead time) com duas grandes consequências:

• Para cada funcionalidade, depois de ser desenvolvida, um retorno de investimento antecipado, o que reduz a necessidade de grandes investimentos de capital;

• Avaliações antecipadas dos usuários em cada nova funcionalidade, que permite testes para estabelecer quais das várias possibilidades são preferidas pelos clientes.

A implantação contínua tem custos adicionais, à medida que lida com instrumentação para garantir que cada nova funcionalidade não resulta em defeitos e também demanda infraestrutura que permite realizar backup de novas funcionalidades quando um defeito não é interceptado nos testes automatizados.

1.5 Gerência de configuração

Para assegurar que esses desafios sejam alcançados, é fundamental contar uma estrutura adequada para gerenciar os incontáveis produtos de trabalho que compõem uma solução, de forma a prover um suporte adequado a práticas como integração, liberação e implantação contínuos.

A gerência de configuração refere-se ao processo pelo qual todos artefatos relevantes para o projeto e a relação entre eles, são unicamente identificados, armazenados, recuperados e modificados [Humble].

A gerência de configuração tem o propósito de “estabelecer e manter a integridade de todos os produtos de trabalho de um processo ou projeto e disponibilizá-los a todos os envolvidos” [Softex].

Da perspectiva da gerência de configuração, a implantação de uma solução envolve aspectos como [Humble]:

- Provisionar e gerenciar o ambiente (configuração de hardware, software, infraestrutura e serviços externos);
- Instalar a versão adequada da aplicação;
- Configurar a aplicação, incluindo dados e estados requeridos.

No ciclo de desenvolvimento, a gerência e configuração trata dos seguintes aspectos [Humble]:

- Obter os pré-requisitos para gerenciar o processo de construção (build), implantação (deploy), testes e liberação (release) da aplicação, colocando tudo sob controle de versões e gerenciando dependências;
- Gerenciar a configuração da aplicação;
- Gerenciar a configuração de todo ambiente – software, hardware e infraestrutura de que o sistema depende, os princípios por trás do gerenciamento de ambientes, do sistema operacional ao servidor de aplicação, bancos de dados e outros softwares.

A gerência de configuração permite responder questões como as seguintes [Humble]:

- É possível reproduzir exatamente qualquer dos ambientes, incluindo a versão do sistema operacional, seu nível de atualizações (patch), configurações de rede, pilha de software, a aplicação implantada nesse ambiente e a configuração desses elementos?
- É fácil fazer uma mudança incremental em cada um dos itens de trabalho individualmente e fazer a implantação das mudanças em qualquer, e todos, ambientes?
- É fácil ver cada mudança que ocorre em um ambiente em particular e rastrear de volta para ver exatamente qual foi a mudança, quem fez e quando?
- É possível atender satisfatoriamente todas as regulações de compliance a que o sistema é sujeito?
- É fácil para cada membro do time obter a informação que precisa e fazer as mudanças que precisa fazer? Ou a estratégia de gerência de configuração trilha o caminho da eficiência na implantação, levando a aumento no tempo de entrega e redução do feedback?

A gerência de configuração pode ser organizada em áreas funcionais [Aiello]:

- Gerenciamento de código fonte;
- Engenharia de build;
- Configuração de ambiente;
- Controle de mudanças;
- Engenharia de release;
- Deployment.

Sob a perspectiva de desenvolvimento, a Gerência de Configuração é dividida em três sistemas principais [Softex]:

- Controle de mudanças: armazenamento de informações das solicitações de modificação, relato aos envolvidos e contabilização da situação da configuração;
- Controle de versões: permite que os itens de configuração sejam identificados e evoluam de forma distribuída e concorrente;
- Gerenciamento de construção: automatiza o processo de transformação dos diversos produtos de trabalho que compõem um projeto em um sistema executável propriamente dito;
- Gerenciamento de liberação e entrega: estrutura baselines selecionadas para liberação, conforme necessário, para a execução função de avaliação e revisão da configuração.

2. Controle de mudanças

O propósito do controle de mudanças é gerenciar cuidadosamente todas mudanças nos ambientes produtivos. Uma parte do esforço diz respeito a coordenação e outra trata o gerenciamento das mudanças que irão impactar todos sistemas do ambiente.

Princípios do controle de mudanças [Aiello]:

- Mudanças devem ser planejadas e não serem apenas eventos de última hora;
- Mudanças devem ser entendidas, incluindo seus impactos;
- Autoridade e aprovações para mudanças devem ser estabelecidas e obtidas apropriadamente;
- Procedimentos para mudança emergentes devem ser estabelecidos para cobrir incidentes emergentes;
- Controle de mudanças deve acessar e confirmar que todo processo de gerência de configuração seja seguido.

A implementação mais comum do controle de mudanças é essencialmente um guardião que previne liberações não autorizadas de serem promovidas para produção ou outro ambiente. Há ainda um controle de mudanças prévio, que trata as mudanças pretendidas, sua revisão, antes de serem feitas, e autorização para fazer as mudanças [Humble].

É essencial ser capaz de gerenciar o processo de fazer mudanças nos ambientes. Um ambiente de produção deve ser completamente “locked down”, ou seja, não pode ser permitido a ninguém fazer mudanças no ambiente sem passar por um processo de mudanças [Humble].

Uma mudança deve ser testada antes de ir a produção e mantida sob controle de versões. Uma mudança no ambiente deve ser vista como uma mudança no software, portanto, sujeita ao processo de build, deploy, test and release [Humble].

3. Controle de versões

3.1 Sistema de controle de versões

Conforme [Softext], “o sistema de controle de versões permite que os itens de configuração sejam identificados, segundo estabelecido pela função de identificação da configuração e que eles evoluam de forma distribuída e concorrente, porém disciplinada. Essa característica é necessária para que diversas solicitações de modificação efetuadas na função de controle da configuração possam ser tratadas em paralelo, sem corromper o sistema de Gerência de Configuração como um todo”.

Algumas premissas são fundamentais para um sistema de controle de versões efetivo:

- Mantenha tudo no controle de versões;
- Faça check in regular para o ramo principal de desenvolvimento;
- Use mensagens de commit significativas.

Uma das bases do controle de versões define que tudo deve ser incluindo [Humble]:

- Documentação de requisitos;
- Documentação técnica;
- Scripts de teste;
- Casos de teste automatizados;
- Configurações de rede;
- Scritps de configuração;

- Scripts de deployment, criação, upgrade, downgrade e atualização de banco de dados;
- Scripts de pilha de configuração de aplicação;
- Bibliotecas e programas para compilação, build, deploy, release e depuração, dentre outros.

Deve ser possível para um novo membro de time sentar-se em uma nova estação de trabalho, fazer checkout do projeto a partir de um repositório de controle de versões e executar um comando único para fazer o build e deploy da aplicação em qualquer ambiente acessível, incluindo o ambiente local.

Também deve ser possível ver qual build das várias aplicações estão implantadas em cada ambiente, e de quais versões vieram.

Alguns conceitos são centrais para o controle de versões [Aiello]:

- Baseline: a criação de uma baseline consiste na identificação das versões exatas de produtos de trabalho para uma release específica. Baselines devem ser imutáveis;
- Gerenciamento de variações (branching): uma característica importante do gerenciamento de código-fonte é a habilidade de suportar múltiplas variações do mesmo código. Criar essa variação no código geralmente é feito por meio da criação de linhas de desenvolvimento, usualmente chamadas de branches;
- Merging: consiste em mesclar novamente para um ramo, usualmente o ramo principal, algum trabalho desenvolvido em uma variação, usualmente uma branch;
- Changeset: são uma forma conveniente de agrupar uma ou mais modificações ao código base. Fazer checkin de um changeset, também chamado de commiting, é usualmente uma transação atômica, o que

significa que todo conjunto de mudança é comitado com sucesso ou completamente revertido.

Alguns princípios do gerenciamento de código fonte [Aiello]:

- Código é resguardado e nunca pode ser perdido;
- Código é transformado em baseline, sinalizando um marco específico ou outro ponto no tempo;
- Gerenciamento de variações no código deve ser fácil com adequada ramificação;
- Código modificado em uma variação (branch) pode ser mesclado de volta no ramo principal (trunk) ou outra variação;
- O processo de gerenciamento de código-fonte deve ser repetível, ágil e enxuto;
- O gerenciamento de código-fonte deve prover rastreabilidade de todas mudanças.

3.2 Versionamento semântico

O versionamento semântico é um esquema de versionamento em que os números de versões e a forma como são modificados embute o significado da ação sobre os correspondentes produtos de trabalho e o porque da mudança [Preston-Werner].

O propósito dessa abordagem é tratar o "inferno de dependências" que pode existir entre os diversos componentes de uma solução. Em sistemas com várias dependências, liberar novos pacots pode se tornar trabalhoso

facilmente. O problema do "inferno de dependências" ocorre quando há dificuldade em rastrear as combinações entre componentes.

Estrutura geral do versionamento semântico. Dado um número de versão incremental sua formação segue a regra:

Alguns aspectos da especificação:

- Um número normal de versão deve seguir a forma X.Y.Z, onde X, Y e Z são inteiros não negativos e não devem conter zeros à esquerda;
- Uma vez que um pacote modificado é liberado, seu conteúdo não deve ser modificado;
- Versões maiores começando com zero (0.Y.Z) são para desenvolvimento inicial;
- Versões maiores começando com 1 (1.0.0) definem a API pública;
- Versões com correções Z (x.y.Z | x > 0) devem ser incrementadas apenas se forem introduzidas apenas correções de bug;
- Versões menores Y (x.Y.z | x > 0) devem ser incrementadas se novas funcionalidades foram introduzidas na API pública;
- Versões maiores X (X.y.z | x > 0) devem ser incrementadas se houver mudança na API pública.

4. Construção

4.1 Engenharia de construção

O propósito da engenharia de construção (build) é ser capaz de compilar confiavelmente e ligar (link) o código-fonte com o executável binário no tempo mais curto possível [Aiello].

Inclui a identificação da compilação exata e dependências de tempo de execução (runtime) e quaisquer outros requisitos técnicos, incluindo compilador e dependências.

Princípios da engenharia de construção [Humble]:

- Builds são entendidas e repetíveis;
- Builds são rápidas e confiáveis;
- Cada item de configuração é identificável;
- O código-fonte e dependências de compilação podem ser facilmente determinadas;
- Build pode ser montada uma única vez e implantada (deployed) em qualquer lugar;
- Anomalias da build são identificadas e gerenciadas de forma aceitável;
- A causa de quebra de build é rápida e facilmente identificada e corrigida.

Conceitos centrais da engenharia de construção:

- Identificação de versões: Da mesma forma que deve ser possível identificar uma baseline no código-fonte, deve ser possível identificar a versão exata de qualquer item criado pelo processo de construção (build), incluindo binários e outros arquivos.
-

- Gerenciamento de dependências: Além do código-fonte, toda compilação e dependências de tempo de execução (runtime) devem ser entendidas e controladas. Isso significa que os scripts de build devem definir todas variáveis de ambiente requeridas e confirmar que todas dependências da build estão no local correto toda vez que a build for executada.
- Build independente: Uma das melhores formas de evitar erros caros é ter cada construção de release de forma independente.

4.2 Dependências

As dependências externas mais comuns na aplicação são bibliotecas de terceiros e componentes e módulos desenvolvidos por outros times. Os primeiros costumam ser mais estáveis, mudando com menor frequência que os últimos.

As bibliotecas externas usualmente vêm na forma de binários. As bibliotecas podem ser baixada da internet ou armazenados localmente. De qualquer forma, é imprescindível que o sistema de build especifique exatamente qual a versão de bibliotecas externas estão em uso.

Já com os componentes, o trabalho pode começar com um processo de build monolítico, criando binários ou instaladores para toda a aplicação de uma única vez. Entretanto, se o sistema cresce ou há componentes de que vários projetos dependem, pode ser importante considerar dividir os builds dos componentes em pipelines separados. Se isso for feito, é importante ter dependências de binários entre os pipelines ao invés de dependência de código fonte. Além da recompilação ser menos eficiente, há o risco de criar um artefato diferente de outro já testado.

4.3 Scripts

Há um conjunto de princípios e práticas para o gerenciamento de scripts de build e deployment:

- Crie um script para cada estágio do pipeline de implantação;
- Use uma tecnologia apropriada para realizar o deploy da aplicação;
- Use o mesmo script para realizar o deploy em todos ambientes (inclusive do desenvolvedor);
- Use as ferramentas de empacotamente do sistema operacional;
- Garanta que seu processo de deployment seja idempotente.

Algumas orientações gerais no controle de scripts:

- Sempre use caminhos relativos;
- Elimine passos manuais;
- Construa rastreabilidade dos binários para o controle de versões;
- Não inclua binários no controle de versões como parte da build;
- Tests targets não devem fazer o build falhar;
- Restrinja a aplicação com testes fumaça de integração.

5. Liberação e implantação

5.1 Engenharia de liberação

O propósito da Engenharia de liberação (release management) é criar e manter um processo repetível para empacotar uma release que inclui uma forma clara de identificar cada componente da release. Geralmente, o empacotamento é uma função automatizada que inclui criar um identificador imutável que é embutido no pacote da release [Aiello].

Princípios da Engenharia de liberação [Aiello]:

- Release deve ser prontamente identificável como um identificador de versão imutável;
- Release deve ser empacotada com todas dependências incluídas;
- Empacotamento da release deve ser automatizado para evitar falhas humanas;
- Gerenciamento de release deve ser rápido e confiável para facilitar desenvolvimento iterativo;
- Deve haver um mecanismo para conduzir uma auditoria de um pacote liberado para verificar todos itens de configuração;
- O conteúdo de uma release deve ser bem entendido, incluindo a rastreabilidade para os requisitos;
- O gerenciamento de release deve ser uma fonte de informação sobre o status de todas releases, idealmente por meio de um dashboard de gerenciamento de releases.

5.2 Implantação

Deployment é o último passo no processo de promoção de código. O propósito do processo de deployment é promover uma release para um ambiente alvo sem a ocorrência de problemas [Aiello].

Princípios de deployment [Aiello]:

- Promover uma release ou removê-la deve ser um processo confiável e o mais simples quanto possível;
- Promover uma release ou removê-la deve ser completamente rastreável com um log de auditoria de todas mudanças;
- Deve haver um processo bem estabelecido procedimento para verificar as versões da uma release em um ambiente;
- O processo de deployment deve ser continuamente revisado e melhorado.

5.3 Configuração de ambiente

O propósito da configuração do ambiente é sempre apontar para os recursos corretos em tempo de execução. Configuração do ambiente refere-se a identificação, modificação e gerenciamento das dependências de interface necessárias para uma build progredir com sucesso entre ambientes.

Envolve o gerenciamento do produto compilado e dependências de tempo de execução (runtime) que podem mudar frequentemente a medida que o código é promovido do desenvolvimento para testes e para produção. Envolve ainda gerenciar os ambientes geralmente designados desenvolvimento, teste, integração e produção.

Princípios do controle de configuração do ambiente [Aiello]:

- Dependências do controle de configuração são identificadas e bem entendidas;
- Ambientes podem ser interrogados sobre seu estado corrente;
- Código-fonte pode ser construído uma vez com a configuração do ambiente mudando antes do deployment.
- Configurações do ambiente podem ser mudadas em uma forma controlada e previsível.
- Configurações do ambiente podem ser documentadas e entendidas pelas partes.

Informação de configuração pode ser usada para mudar o comportamento do software em tempo de build, deploy ou runtime. O gerenciamento da configuração deve ser feito para garantir consistência entre componentes, aplicações e tecnologias. A informação de configuração também deve ser submetida a gerenciamento e testes.

6. Estudo de caso

A vinícola Concha Y Oro é uma empresa especializada na produção e exportação de vinhos, com atuação de mais 50 anos no mercado e uma carteira abrangente de clientes em todo mundo.

Para suportar grande parte de seus processos de negócio, a empresa possui uma solução tecnológica contendo os principais componentes tecnológicos:

- Plataforma de desenvolvimento: Java;
- Servidor web: Apache Tomcat;
- Servidor de gerenciamento de banco de dados: Mysql.

O processo de trabalho ainda é bastante rudimentar, não havendo práticas nem infraestrutura que suportem o processo de gerência de configurações.

Os desenvolvedores realizam seus trabalhos localmente nas respectivas estações de trabalho e copiam os artefatos resultantes em um diretório compartilhado na rede interna da empresa. O controle de versões é feito a partir da réplica dos diretórios contendo os artefatos, sendo que cada diretório resultante deve seguir um padrão de nomenclatura dos diretórios, conforme as mudanças são feitas.

O processo de compilação é feito de forma totalmente manual, apenas com o uso dos recursos básicos disponíveis como o compilador `javac`. Todas as bibliotecas necessárias para a compilação estão disponíveis no mesmo diretório e estrutura de armazenamento do código-fonte.

O processo de geração de build é feito de forma manual, por meio do empacotador `jar`. Não há armazenamento dos arquivos resultantes, sendo gerado um novo arquivo toda vez que é necessário realizar alguma publicação.

O processo de implantação também é feito de forma manual com os produtos de trabalho sendo copiados pelo desenvolvedor a partir de sua estação de trabalho para o servidor.

Após alguns estudos, decidiu-se por adotar algumas práticas ligadas à gerência de configuração, além de uma infraestrutura composta por:

- Controle de versões: Git;
- Gerenciamento de build: Maven;
- Repositório de artefatos: Nexus;
- Automação do ciclo de desenvolvimento: Jenkins.

No desenvolvimento deste estudo de caso, serão realizadas atividades em cinco etapas distintas e incrementais:

- Realizar compilação, build e deploy de forma manual.
- Adotar controle de versões.
- Adotar solução para processos de compilação, build e deploy.
- Adotar armazenamento de arquivos.
- Adotar automação do ciclo de desenvolvimento.

Os roteiros detalhados encontram-se nos anexos deste documento.

Referências

[Aiello]

Aiello, Bob, Sachs, Leslie. Configuration Management – Best Practices. Addison Wesley, 2011.

[Duvall]

Duvall, Paul; Matyas, Steve; Glover, Andrew. Continuous Integration: Improving Software Quality and Reducing Risk. Addison Wesley, 2007.

[Humble]

Humble, Jez; Farley, David. Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation. Addison-Wesley, 2011.

[Preston-Werner]
Preston-Werner, Tom. Semantic Versioning 2.0.0. Disponível em https://semver.org/. Acessado em 01/02/2018.

[Riley]

Riley, Cris. I Want to do Continuous Deployment. Disponível em https://devops.com/i-want-to-do-continuous-deployment/. Acessado em 01/02/2018.

[Softex]

Softex. MpsBr - Guia de Implementação – Parte 2. Softex, 2016.

Anexo I – Instalação do ambiente

1. Instalação

- Obter o Docker for Windows do site https://www.docker.com/docker-windows.
- Instalar seguindo as orientações.
- Iniciar o serviço Docker acionando o atalho na área de trabalho.
- Testar a instalação, executando o container de boas vindas. Acessar CMD e executar o comando:

```
docker run hello-world
```

2. Configurações

- Selecionar o serviço Docker a partir da barra de tarefas no canto inferior direito.

- Acionar o botão direito no ícone do serviço, item de menu Settings.

- Configurar o item General, habilitando os dois primeiros itens.

- Configurar o item Shared Drivers.

- Configurar o item Advanced, definindo o número de CPUs e memória (recomendação 50% da memória total).

Alguns comandos úteis no contexto deste material:

Testar a instalação do docker	`docker run hello-world`
Listar contêineres em execução	`docker ps`
Inspecionar contêiner	`docker inspect <identificador do contêiner>`
Iniciar serviços	`docker-compose up -d`
Iniciar serviços (forçando a construção dos contêineres)	`docker-compose up -d –build`
Parar contêineres do serviço	`docker-compose down`
Excluir contêineres	`docker rm $(docker ps -a -q)`
Excluir contêineres (forçando)	`docker rm $(docker ps -a -q) -force`

Limpeza geral	`docker container prune -f`
	`docker image prune -f`
	`docker volume prune -f`
	`docker network prune -f`

Anexo II – Configuração do projeto

1. Obtenção do projeto

- Obter projeto a partir do endereço: http://github.com/valuedriven/curso-gco.
- Selecionar comando "Clone or Download".
- Descompactar arquivo curso-gco-master.zip na área de trabalho.
- Acionar botão Iniciar da área de trabalho.
- Informar CMD no campo de busca.
- Acessar diretório do projeto:

```
cd Desktop/curso-gco
```

2. Inicialização do container e banco de dados

- Executar Docker-compose:

```
docker-compose up -d
```

- Inicializar banco de dados:

```
docker-compose exec db mysql -u root -psecret conchayorodb < bancoscript/conchayorodb.sql
```

3. Acessar as aplicações web

- Acionar aplicativos por meio do navegador web:

```
Tomcat: http://localhost:8080
Nexus: http://localhost:8081
Jenkins: http://localhost:8082
```

Anexo III - Processo de trabalho manual

Iniciar serviços

```
docker-compose up -d
```

1. Crie a estrutura de diretórios para compilação e empacotamento

```
md .build\lib
md .build\META-INF
md .build\script
md .build\WEB-INF\classes\META-INF
md .build\WEB-INF\lib
```

2. Copie os produtos de trabalho necessários para o empacotamento

```
cp src\main\webapp\WEB-INF\web.xml .build\WEB-INF
cp src\main\webapp\index.html .build\
cp src\main\webapp\produto.html .build\
cp src\main\webapp\lib\angular.js .build\lib
cp src\main\webapp\script\produto.js .build\script
cp src\main\resources\META-INF\persistence.xml .build\WEB-INF\classes\META-INF
cp bibliotecas\* .build\WEB-INF\lib
```

3. Compile o código-fonte utilizando o compilador javac

```
docker-compose run java javac -verbose -cp 'bibliotecas/*'  -d '.build/WEB-INF/classes'
src/main/java/br/com/conchayoro/entity/Produto.java
```

```
src/main/java/br/com/conchayoro/persistence/ProdutoDao.java
src/main/java/br/com/conchayoro/service/ProdutoService.java
```

4. Gere o arquivo de publicação (war) com o utilitário jar

```
docker-compose run java jar cvf conchayoro.war -C .build .
```

(atente para o fato de que o ponto final faz parte do comando)

5. Acesse a interface gráfica do container web

```
http://localhost:8080
```

6. Implante a aplicação no container

Acionar o comando "Manager App".

Informar "manager", "manager" para usuário e senha.

Na seção Deploy, War do Deploy, acionar o comando "Escolher arquivo".

Selecionar o arquivo conchayoro.war.

Acionar o comando "Deploy".

7. Acesse a aplicação por meio do navegador

```
http://localhost:8080/conchayoro
```

Processo de trabalho manual - Atividades complementares

1. Mudança

Um cliente solicitou que o link “Controlar Produtos” na página principal seja alterado para “Manter Produtos”.

Altere o arquivo index.html na pasta “fonte”.

Repita todo processo realizado nesse roteiro.

Imagine agora que o cliente mudou de ideia e deseja voltar atrás na mudança. Não é necessário proceder os ajustes. Apenas analise a situação.

2. Identificação de produtos de trabalho

Descreva todos produtos de trabalho, hardware e software, utilizados na execução do roteiro. Faça uma análise breve de cada produto identificado.

3. Análise do processo

Identifique o máximo de questões que podem impactar a entrega rápida de soluções.

Para cada questão, descreva o risco potencial e formas de resolver.

Avalie o processo de trabalho a luz dos princípios norteadores são fundamentais para acelerar os ciclos de entrega, apresentados no capítulo 1.

Anexo IV – Controle de versões

Iniciar serviços

```
docker-compose up -d
```

Ao final de cada comando, verifique a situação do repositório por meio do comando:

```
docker-compose run --rm git status
```

1. Defina as variáveis de ambiente para o projeto

```
docker-compose run git config --global user.name "concha.yoro"
docker-compose run git config --global user.email  "user@conchayoro.com"
```

2. Crie o repositório para o projeto

```
docker-compose run --rm git init
```

Navegue na estrutura de diretórios criada (diretório .git).

3. Inclua produtos de trabalho no repositório

```
docker-compose run --rm git add ambiente\* bancoscript\* bibliotecas\* documentacao\* roteiros\* src\* docker-compose.yml Jenkinsfile pom.xml README.md .env
```

4. Efetive a inclusão de produtos de trabalho no repositório

```
docker-compose run git commit -m "Versão inicial do projeto"
```

5. Crie uma tag para a situação atual do repositório

```
docker-compose run --rm git tag -a v1.0.0 -m "Versão inicial"
```

6. Crie um novo ramo no repositório

```
docker-compose run --rm git checkout -b release-1
```

Liste os ramos disponíveis no repositório

```
docker-compose run --rm  git branch
```

7. Retorne para o ramo principal de desenvolvimento

```
docker-compose run --rm git checkout master
```

Liste os ramos disponíveis no repositório

```
docker-compose run git branch
```

Controle de versões - Atividades complementares

1. Mudança

Repita a alteração proposta no roteiro anterior. Para o procedimento, utilize o ramo principal de desenvolvimento (master). Ao final da alteração, crie uma nova marcação da mudança efetivada.

2. Nova ocorrência

Imagine que o time está desenvolvendo uma nova versão do produto. Durante o processo, surge um erro na versão atualmente em produção.

Discuta como o tratamento do erro deverá refletir no repositório e processo de trabalho paralelo.

Identifique quais seriam as estratégias para o tratamento do cenário.

4. Análise de princípios

Analise os procedimentos adotados neste roteiro com o propósito e princípios de controle de versões detalhados no capítulo específico acima neste documento.

5.Tipos de produtos de trabalho

Analise todo o conteúdo disponível no diretório do projeto. Para cada tipo de produto de trabalho (código-fonte, banco, bibliotecas etc.), descreva se e como deve ser colocado sob controle de versões e com qual ferramenta.

Anexo V - Automação de tarefas

1. Compile o projeto

```
docker-compose run --rm maven mvn clean --settings
ambiente/maven/settings.xml
```

```
docker-compose run --rm maven mvn compile --settings
ambiente/maven/settings.xml
```

2. Gere o arquivo de implantação

```
docker-compose run --rm maven mvn package --settings
ambiente/maven/settings.xml
```

3. Instale o produto gerado no repositório local

```
docker-compose run --rm maven mvn install --settings
ambiente/maven/settings.xml
```

4. Faça a instalação do arquivo no repositório de artefatos

```
docker-compose run --rm maven mvn deploy --settings
ambiente/maven/settings.xml
```

6. Faça a implantação no servidor web

```
docker-compose run --rm maven mvn tomcat7:deploy --settings ambiente/maven/settings.xml
```

Caso já haja uma versão instalada, utilize a tag redeploy

```
docker-compose run --rm maven mvn tomcat7:redeploy --settings ambiente/maven/settings.xml
```

7. Faça a publicação no servidor de qualidade

```
docker-compose run --rm maven mvn sonar:sonar -Dsonar.host.url=http://sonar:9000 --settings ambiente/maven/settings.xml
```

Automação de tarefas - Atividades complementares

1. Sistema de gerência de configurações

Reelabore o diagrama da figura 1, mapeando agora as ferramentas usadas durante o curso.

2. Sistema de controle de versões

Analise os procedimentos adotados neste roteiro com o propósito e princípios de controle de versões detalhados no capítulo específico.

3.Tipos de produtos de trabalho

Analise os produtos de trabalho resultantes deste roteiro. Para cada tipo de produto de trabalho (código-fonte, banco, bibliotecas etc.), descreva como estão alocados e em quais ferramentas.

Avalie este roteiro com o roteiro anterior (git) e aponte quais artefatos tiveram seu armazenamento modificado.

Anexo VI – Repositórios com Nexus

Criação de repositórios

Acesse o nexus por meio do navegador web: http://[IP]: 8081.

Acione o comando “Sign in”, no canto superior direito.

Informe os valores: User= admin e Password: admin123.

Acesse o menu de configurações, disponível na barra superior, canto superior esquerdo, ícone de engrenagem.

No painel da lateral esquerda, selecione a opção “Repositories”.

No painel central, acione o comando “Create Repository”.

Na seção “Select Recipe”, escolha a opção Maven2 (group).

Preencha os campos Name e Group.

Acione o comando ”Create Repository”.