02

Bibliografia e links externos

Devido a sua importância, a **Integração Contínua** é um tópico bastante discutido na literatura e na web. Seguem algumas fontes que usamos para criar este curso:

- Livro: Continuous Integration, de Paul M. Duvall
- Artigo da ThoughtWorks: Continuous integration (https://www.thoughtworks.com/pt/continuous-integration)
- Artigo do Martin Fowler: Continuous Delivery (https://martinfowler.com/bliki/ContinuousDelivery.html)
- Série de artigos da Caelum:
    - Branches e integração contínua: o problema de feature branches (https://blog.caelum.com.br/branches-e-integracao-continua-o-problema-de-feature-branches/)
    - Integração Contínua - Builds rápidos com Grids e paralelismo (https://blog.caelum.com.br/integracao-continua-builds-rapidos-com-grids-e-paralelismo/)
    - Integração contínua: deploys e aprovações sem dor de cabeça para o cliente (https://blog.caelum.com.br/integracao-continua-deploys-e-aprovacoes-sem-dores-de-cabeca-para-o-cliente/)
- Artigo no CodeBetter: Check in Dance (http://codebetter.com/jeremymiller/2005/07/25/using-continuous-integration-better-do-the-check-in-dance/)

01

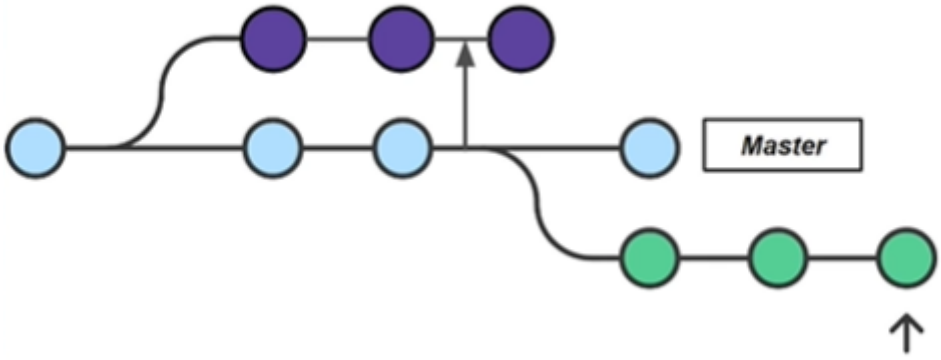
O que são Branching Models?

Transcrição

Discutimos o que é a integração contínua, os problemas que ela busca resolver e suas vantagens enquanto prática de desenvolvimento. Além disso, apresentamos alguns modelos de repositório: Multi e Mono-repo.

Nesta aula estudaremos as ramificações ou *branching models*. De certa forma,o código deveria ter uma linha principal e todas as alterações realizadas devem refletir, por meio de commits, em seu master ou linha principal.

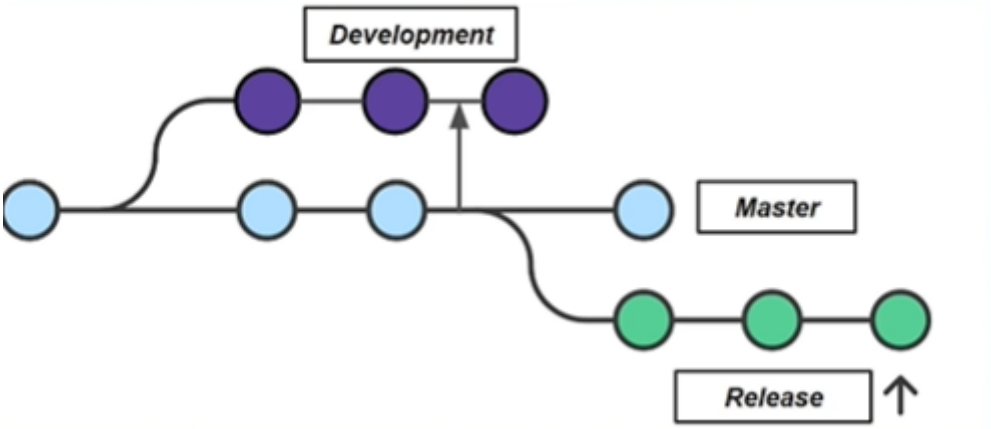
No momento em que começamos a desenvolver uma nova funcionalidade, pode fazer sentido que se saia master para focar nessa nova característica. Esse processo é chamado de ramificação, o fluxo separado do código principal.



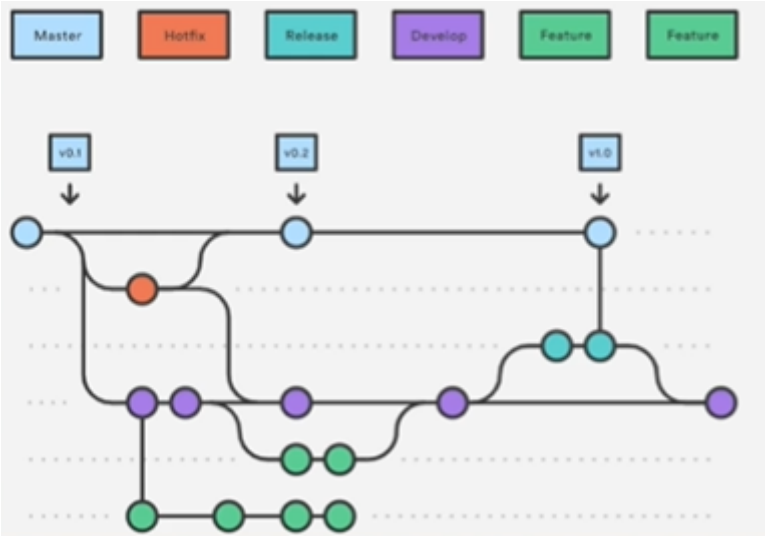
Não se trata de um processo novo, mas por que tocar neste tópico? O Git é a ferramenta mais popular no que envolve as práticas de integração contínua, e com ele a criação de ramos no código é menos custosa.

Criar os ramos com SVS, por exemplo, eram muito mais difíceis de realizar. Com o Git elas são mais leves, rápidas e fáceis de criar, o que reconduz estratégias de projeto.

Poderíamos pensar que existe uma ramificação para o desenvolvimento outra apenas para os releases como homologação.



Podemos ir muito além, existem modelos mais complexos: para cada feature existirá um ramo específico.



Foram criadas por diferentes empresas e equipes semânticas específicas para o uso de modelos de ramificação. Um dos mais populares é o Git flow, mas teremos ainda o one flow, pull request flow, gitlab flow e assim por diante.

Devemos lembrar que o nosso master deve representar o core, o estado principal da aplicação. No momento em que criamos um branch, você se afasta do seu core. Muitas vezes isso é confortável para o desenvolvedor, porque ele atua em ambiente isolado e pode testar ações com mais autonomia.

Mas qual era a ideia de integração contínua? Alterar de maneira contínua o repositório principal. Martin Fowler declara:

> "Everyone commits To the Mainline every day."

A regra da integração continua é antecipar os problemas. A princípio, o branch pode ser um perigo, um ampliador de risco. Existem alguams regras que podemo minimizar esse risco:

- commits simples e releaseble, orientados à uma tarefa
- branches de vida curta, margens mais simples
- estratégia combinada pela equipe

Devemos lembrar que muitos branches geram mais burocracia, e apresentar suas desvantagens. Na próxima aula analisaremos algumas estratégias e

modelos específicos.

03

Comparando Modelos

Transcrição

Discutimos o que são as ramificações, e é chegado o momento de comparar os modelos populares que existem e elencar suas vantagens e desvantagens.

Alguns branching models famosos são:

- **Temporários (branches locais)** São branches localizados apenas na máquina local e deverão se extiguir, são utilizados para organizar fluxos de trabalho e depois realizar o commit.

- **Feature Branches** São utilizados para implementar funcionalidades ou orientar tarefas.

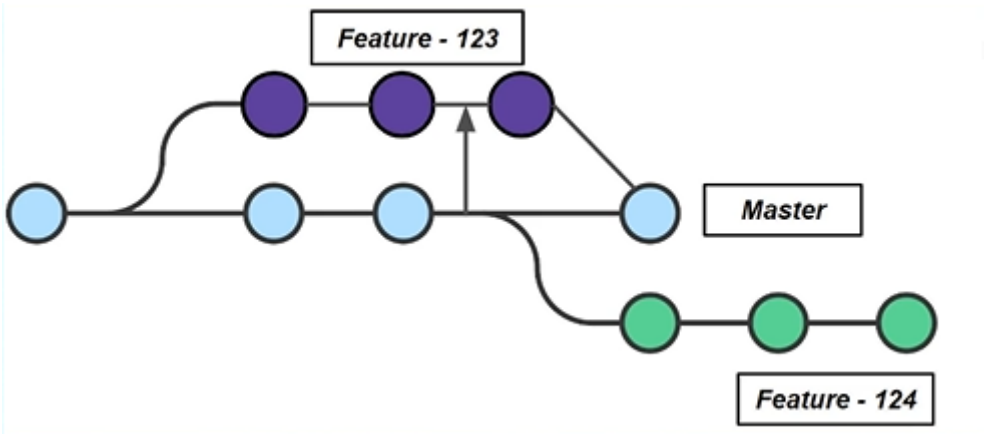
- **Historical Branches (master e develop)** As alterações ficam organizadas em commits baseados na cronologia no caso de um projeto de software.

- **Environment Branches (Staging e Production)**] Existem branches que são baseados no ambiente, isto é, em que espaço é realizado o deploy.

- **Maintenance Branches (Release e Hotfix)** Temos, ainda, os branches de manutenção do sistema.

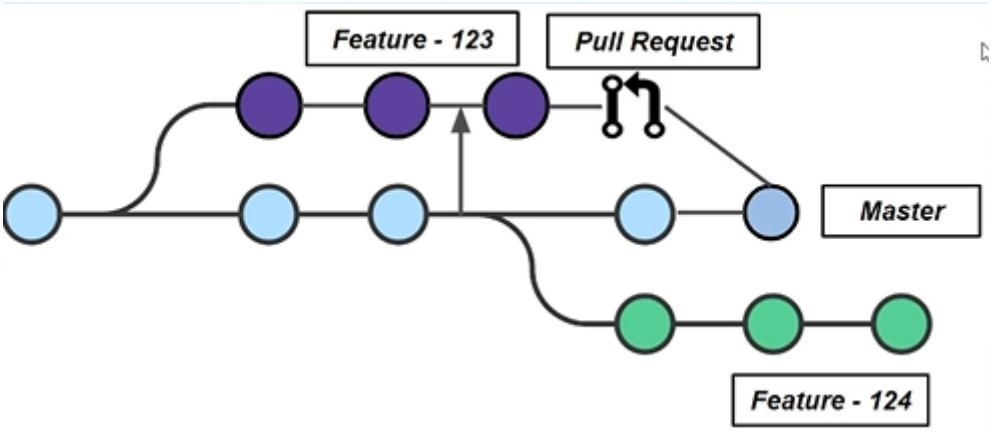
Estudaremos cada um deles com mais detalhes. O primeiro modelo é o mais simples, menos sofisticado de todas as ramificações, mas reduz a distância de um branch para o master até o mínimo possível. Nesta página (https://trunkbaseddevelopment.com/) encontramos algumas referências de como utilizar essta metologia.

O modelo future branch workflow, já mencionado no curso, faz parte da realidade de mais desenvolvedores. A ideia é simples: precisamos implementar uma nova funcionalidade e então criamos uma ramificação para este fim.



Uma vez implementado, ele será comitado ao master e então teremos duas fontes de rebase, e que ainda deverão ser discutidas.

Algo comum - inclusive utilizado aqui na Alura - é combinar o future branch work flow com o pull request.



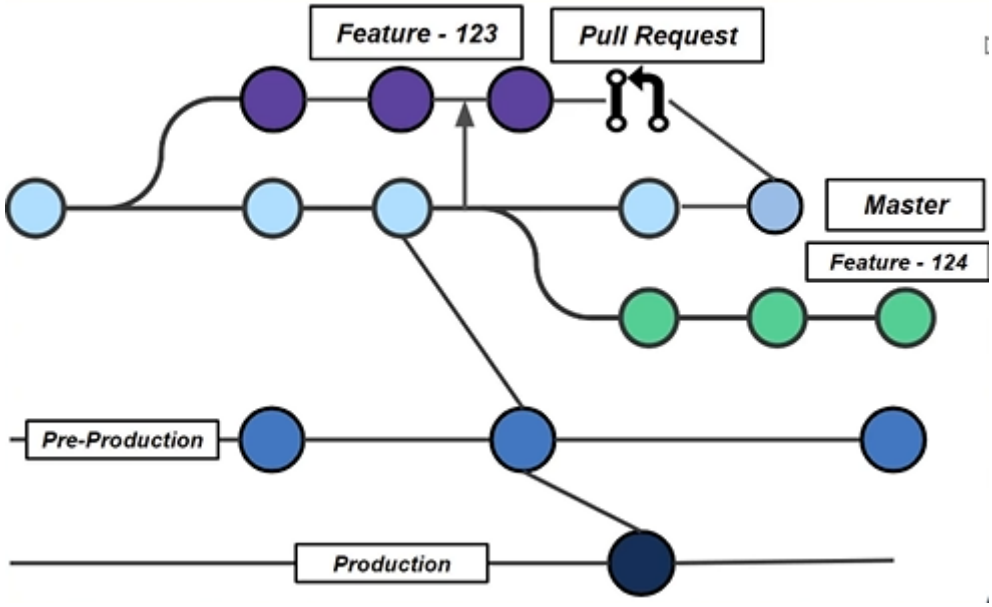
A partir de um recurso do Git, fazemos a implementação, mas não podemos - ou não queremos - fazer o merge diretamente no master. Neste caso, utilizamos o pull request. Uma fez que a feature estiver finalizada, o pull request notifica todos os envolvidos da equipe, e então será avaliado o merge com o master.

Contudo, lembramos que a ideia da integração contínua são commits diários no repositório principal, e o pull request não deixa de ser uma barreira.

Esse processo é muito comum em projetos open source, em que outras pessoas podem clonar o código e enviar pull requests, e neste caso é um processo que faz sentido. Contudo, em um ambiente de equipe temos uma barreira.

O pull request dialoga com a prática de code review, em que os desenvolvedores conversam e discutem sobre a estrutura do código e fazem tomadas de decisão. Mas o code review independe do pull request. Esse fluxo de trabalho é chamado de Github Flow.

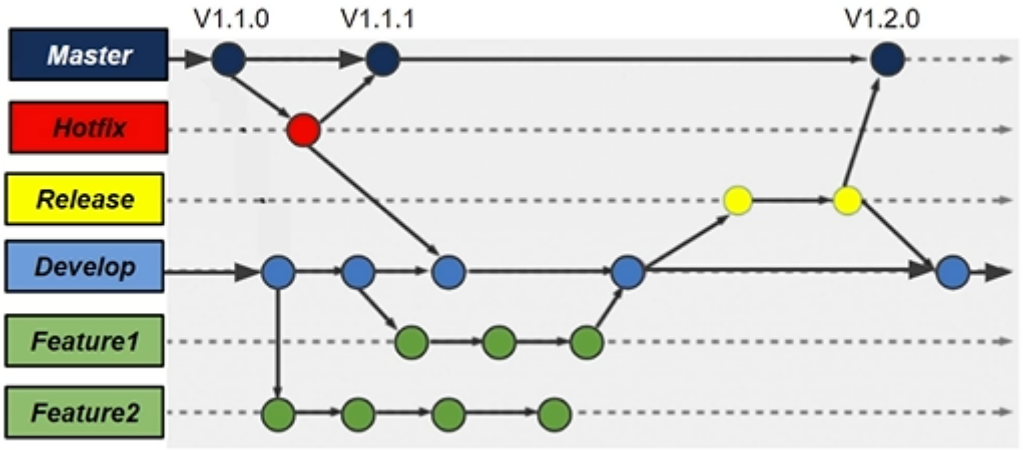
Outro modelo de ramificações é o Gitlab Flow, que apresenta as mesmas características principais do Github Flow, contudo possui mais ramificações, chamados environment branches.



Eventualmente, não podemos realizar o deploy integral de tudo que está no master de uma vez. Precisaremos fazer um deploy planejado antes de inserir a aplicação em produção.

Novamente, esse processo gera mais complexidades e mais dificuldades, mas em alguns casos esse processo é necessário.

O mais popular - e mais complexo - modelo de trabalho é o Git Flow. Obviamente, apesar da popularidade do modelo, os problemas de gerar muitos intermédios são os mesmos.



O branch que representa o histórico de trabalho é o "develop", e os features são integrados e ao final ocorre o commit. Se em algum momento desejamos criar uma nova versão, criaremos um novo branch a partir de um commit.

Uma vez que a nova versão foi liberada e revisada, se libera uma comitação no master, que represesenta os releases em produção.

Se surge algum problema em tempo de produção, é criado uma nova ramificação e implementar o Hotfix e então lançar uma nova versão no master.

Portanto existem vários caminhos que o código percorre. Não sabemos dizer qual é o trunk mais atualizado: master, release ou develop, afinal possuem uma autonomia que pode gerar confusão em algum momento, além de criar uma maior complexidade no sistema de automação.

Essa não é uma prática da integração contínua, como é declarado em vários artigos sobre o assunto.

Esses são os modelos mais comuns de work flow no mercado. A complexidade amplia gradativamente como foi apresentado na aula.

05

Branch by abstraction

Transcrição

Discutimos bastante o que são branches e quais estratégias de workflow se baseiam essa lógica. Em alguns momentos, foi mencionado que idealmente devemos utilizar menos ramificações quanto for possível, pois cada novo afastamento do trunk principal aumenta fatores de risco.

Uma nova funcionalidade pode demorar para ser implementada, então muitas vezes faz sentido termos um feature branch para desenvolver até a nova versão estar estável. Contudo, até nestes casos existem metodologias que evitam a criação de branches de vida longa. O nome dessas metodologias são **Feature Flags e Branch by Abstraction***.

O que é o Feature Flag? Suponhamos uma nova funcionalidade em nosso projeto que terá um tempo longo de implementação. Contudo, não queremos criar uma nova ramificação para esse processo, queremos trabalhar diretamente com o master ainda que o código não esteja completo.

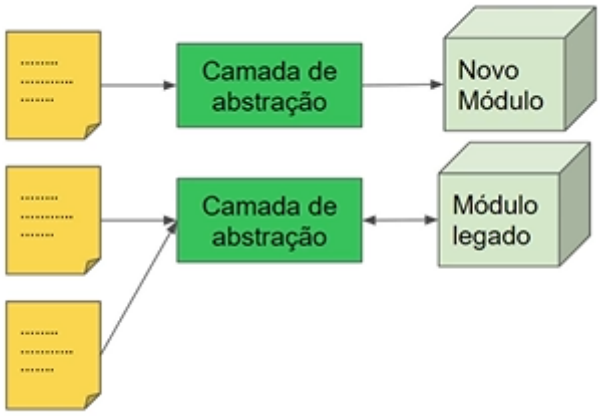
Anteriormente, comentamos que cada commit deve ser releasable, isto é, pode ser publicado. Existe uma maneira de trabalhar sem branches: a feature flag.

O código é inserido no master, mas ele não é visível para a equipe. O Feature flag server também para testar funcionalidades, por exemplo.

Branch by Abstraction, apesar do nome, não envolve a criação de uma nova ramificação. Temos um módulo ligado, uma parte da aplicação utiliza uma biblioteca antiga e precisa ser substituída. Esse é um processo lento, e muito elementos precisam ser alterados.

O primeiro passo é introduzir uma abstração no código principal, isto é, uma camada intermediária para isolar o código que utiliza o módulo, portanto todas as chamadas deverão passar pela camada de abstração. Essa camada pode ser uma interface, várias ou mesmo uma classe que realiza delegações.

Uma vez que é aplicada essa técnica de desacoplamento, podemos gradativamente fazer a re-implementação. Podemos utilizar um módulo legado para o que é de fato utilizava o módulo antigo.



Com o tempo, o módulo antigo fica em desuso e pode ser suprimido completamente.

No mundo ideal, todas as features tem uma granulidade suficiente para não precisar necessitar de um branch de vida longa, mas como nem sempre o ideal é possível, foram criadas essas técnicas.

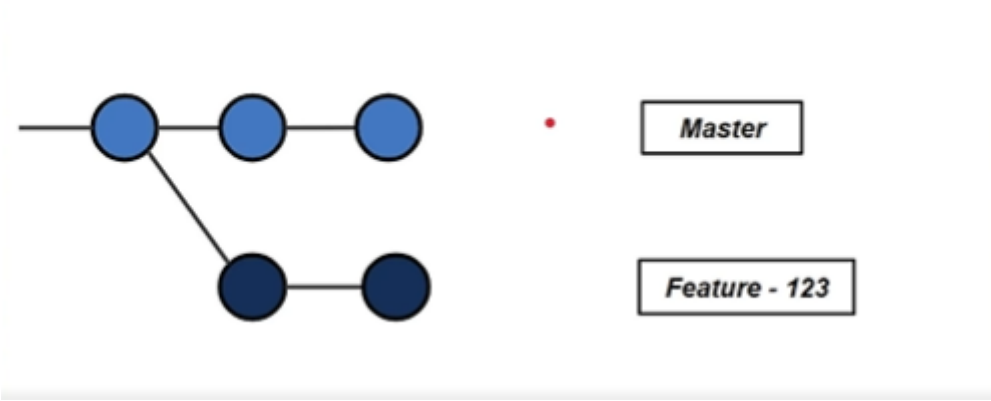
06

Merge e Rebase

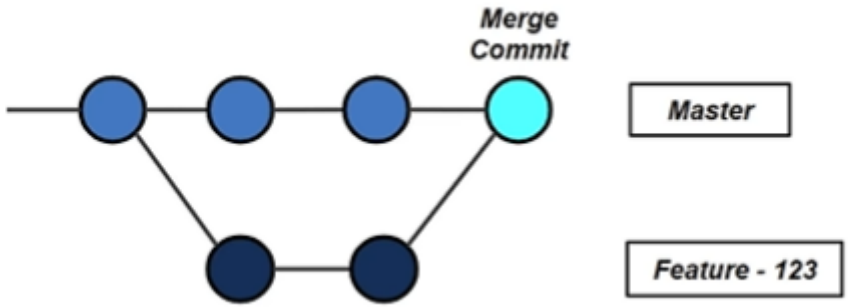
Transcrição

Conversamos sobre branches, work flows e como evitar ramificações de vida longa. Nesta aula, atacaremos um novo problema em que deveremos optar por merge ou rebase.

Mas quais são as diferenças entre merge e rebase? Temos um master e um feature branch, baseado no primeiro commit do master e as duas ramificações evoluiram ao mesmo tempo. Em algum momento, o desenvolvedor decide enviar as atualizações para o master, isto é, realizar a sincronização.



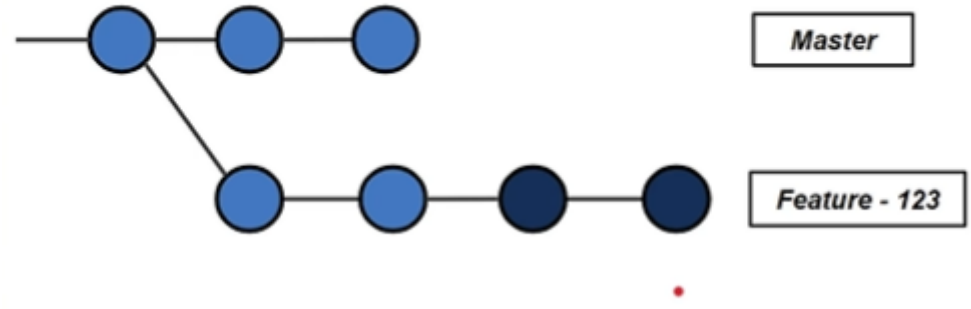
O comando clássico para essa situação é o merge, e então ocorre o chamado "merge commit", cria-se um novo commit que representa esse momento de sincronização.



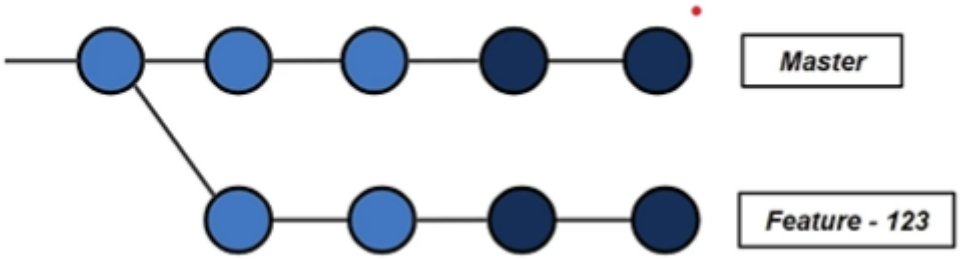
Há desenvolvedores que não gostam desse processo, afinal trata-se de um commit que simplesmente representa um evento e que seu estado é baseado nas alterações realizadas na feature branch e na commit do master. Dessa maneira já não temos uma linha histórica muito interessante no desenvolvimento do projeto, o que pode gerar confusões.

Outra maneira de sincronizar o branch é pelo uso do rebase. Neste caso, a ideia é que se mude a base do commit, e então as modificações são aplicadas nessa nova base.

Dessa maneira, temos um histórico diferente de trabalho, e é por isso que o rebase deve ser aplicado apenas em máquina local.



A mudança então pode ser enviada de fato ao master e novamente temos um histórico linear.



Quando este processo estiver concluído, podemos inclusive excluir a feature branch.

09

Para saber mais: Fontes externas

Seguem alguns links externos das estratégias mencionadas:

- https://trunkbaseddevelopment.com/ (https://trunkbaseddevelopment.com/)
- https://www.atlassian.com/git/tutorials/comparing-workflows/feature-branch-workflow (https://www.atlassian.com/git/tutorials/comparing-workflows/feature-branch-workflow)
- https://guides.github.com/introduction/flow/ (https://guides.github.com/introduction/flow/)
- https://docs.gitlab.com/ee/topics/gitlab_flow.html (https://docs.gitlab.com/ee/topics/gitlab_flow.html)
- https://danielkummer.github.io/git-flow-cheatsheet/index.pt_BR.html (https://danielkummer.github.io/git-flow-cheatsheet/index.pt_BR.html)

Bom estudo :)

01

Self testing

Transcrição

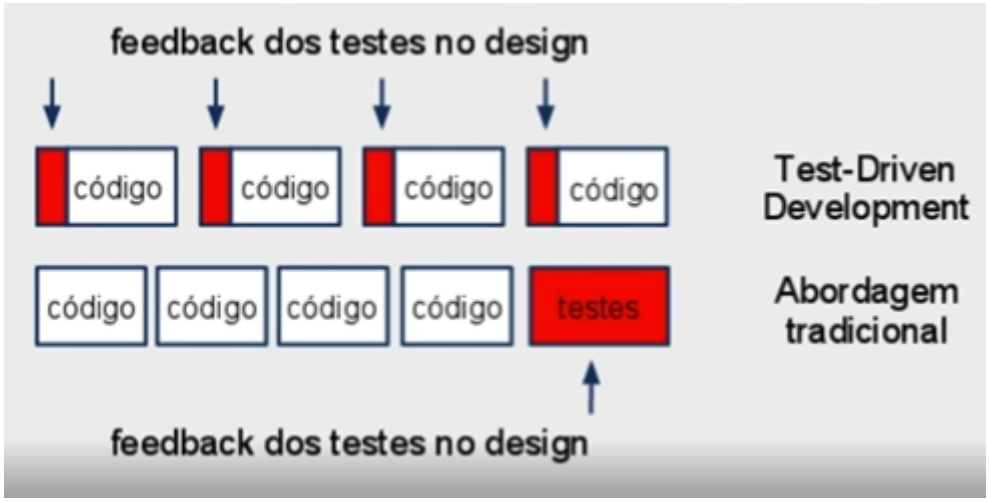
Estudamos os branchs, os work flows e concluímos que dentro da prática da integração contínua, devemos nos afastar o mínimo possível do nosso trunk master principal.

Com as alterações que realizamos o tempo todo em nosso software, como podemos garantir a qualidade do código? Testes. No caso da integração contínua, precisaremos utilizar **testes automatizados**. O ideal é que a cada alteração, seja realizado um novo teste automatizado, para termos certeza de nenhum problema será gerado.

- Testes fazem parte da construção do software
- Devem ser realizados antes do commit
- TDD pode ajudar neste processo
- Desempenho bom em testes

Os testes demorados podem ser uma barreira para a integração contínua, por isso precisamos ficar atentos.

Abordaremos o **TDD** (Test Drive Development). Os testes em integração contínua são sobre feedback do software, como a maioria dos métodos ágeis.



Existem diversas categorias e níveis de testes automatizados, aqui descaremos três: unit tests, integration tests e functional tests.

os uninit tests verificam unidades, como métodos e funções dentro do software. São os testes mais rápidos, baratos de escrever e sua manutenção é simples.

Os testes de integração são de um nível mais alto, e testam a relação de elementos, como por exemplo um banco de dados e o software. A realização destes testes é mais lenta, afinal possuem um outro grau de complexidade.

Testes de um nível ainda maior, são os functional tests, que testam o sistema completo e garante a correção de funcionalidades no ponto de vista do cliente.

O que é importante pensarmos é no tempo de execução de testes que teremos. Os testes de unidade existem desde o início do projeto, qualquer commit deveria ser acompanhada por um teste.

É comum que o desenvolvedor que queria concluir um projeto rapidamente deixe de fazer testes para otimizar o tempo. Como resolver esse impasse? Antes do commit, devemos executar todos os testes, embora saibamos que isso é em um plano ideal, e muitas vezes desnecessário dependendo da modificação que foi realizada. Até mesmo executar todos os testes unitários pode ser complicado.

Uma técnica comum é executar o que chamamos de **smoke tests**. Na prática, trata-se de uma seleção de testes que garantem que as funcionalidades mais importantes do sistema estejam operando corretamente. Esses testes avaliam um conjunto menor de elementos, por isso são mais rápidos, e dessa maneira teremos a garantia de que o software está operante em sua estrutura básica. Depois disso, podemos aplicar todos os testes e garantir uma varredura maior de erros.

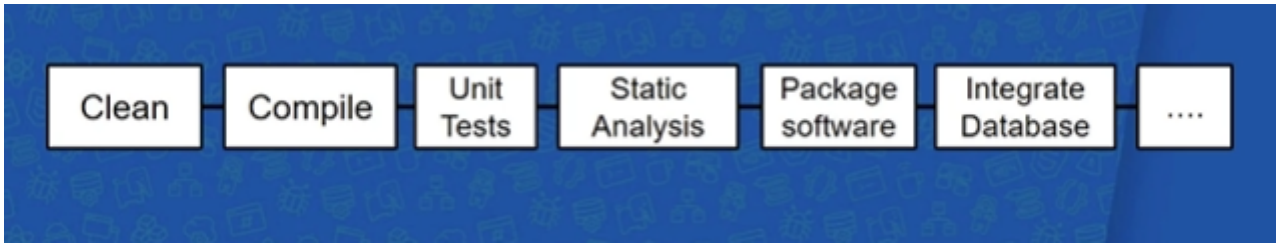
Em resumo, devemos observar a categoria de cada teste; em ambientes diferentes fazer escolhas de desempenho e que melhor atendam nossa demanda; aplicar boas práticas de testes (testes isolados, legíveis, expressivos); realizar testes na parte de build e adquirir feedbacks o mais rápido o possível.

03

Build automatizado

Transcrição

Discutimos sobre testes automatizados e a importância dessa prática na lógica da integração contínua. Apenas executar testes durante a integração do commit não basta, precisamos também construir nosso software, que segue algumas etapas clássicas.



Primeiro realizamos os smoke tests para garantir as funcionalidades essenciais estejam operantes. Então podemos incluir uma análise estática para garantir que o commit coninua baseado nos padrões estabelecidos, e então poderemos criar o primeiro artefato do build.

Para realizar essas etapas de maneira contínua, devemos automatizar o máximo possível esses processos. Existem ferramentas específicas para isso, no mundo Java existe o Gradle e Maven por exemplo.

Em resumo:

- Build a cada commit
- Automatização
- Build independente da IDE
- Todo conteúdo necessário armazenado no repositório

A metáfora do "integrate button" nós é útil agora. Devemos ter um comando simples para realizar builds rápidos. A depender do projeto, é possível que nem todas as fases sejam realizadas.

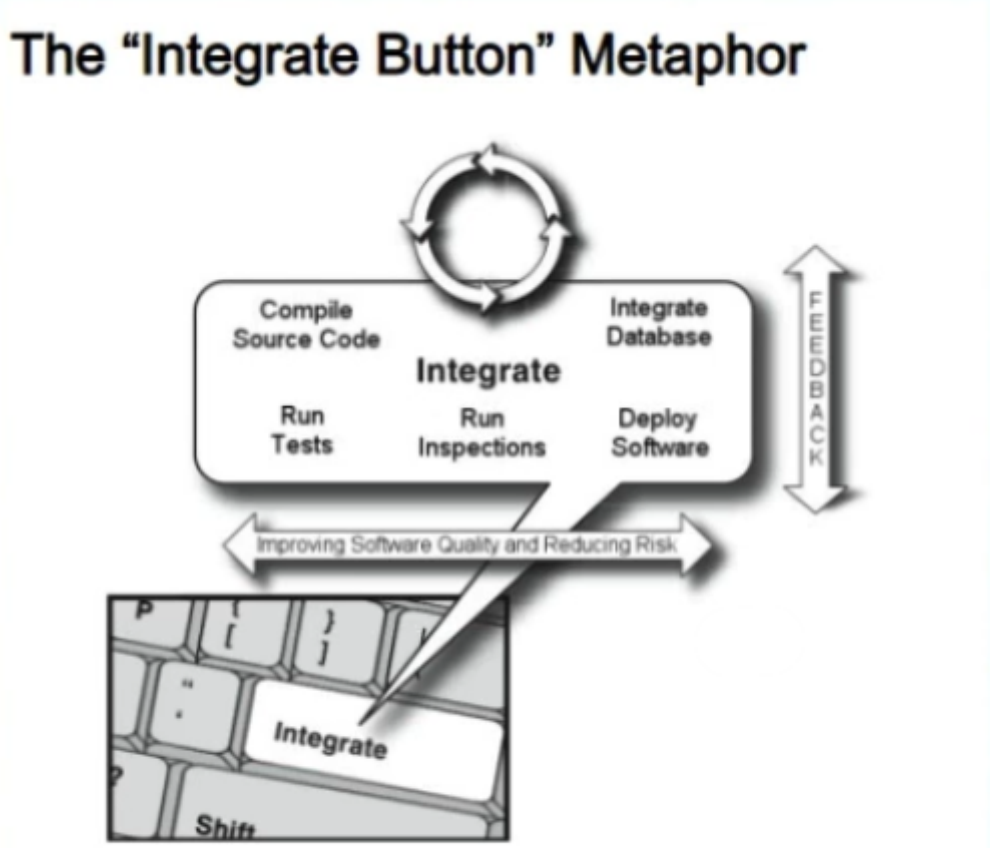


Figure 2: The “Integrate Button” Metaphor

Source: Duvall et al. (2007)

Quando falamos em um buil rápido, podemos utilizar as ideias de XP Programming, em que o build não deve durar mais de 10 minutos. Otimizar o build é importante, e nestes casos métricas ajudam, e elas podem ser adquiridas por meio das ferramentas de automação que comentamos anteriormente.

Em resumo:

- Verifique a fase de testes e analise o código
- Verifique a ordem dessas fases
- Verifique a infra do build system
- Use cache

- Use staged build/pipeline

Não devemos fazer builds periódicos, e eles devem ser independentes da IDE, todo conteúdo necessário para o desenvolvimento deve estar no repositório.

Precisamos, ainda, definir a organização dos artefatos que forem criados pela aplicação, em que locais serão armazenados os scripts e relatórios, por exemplo. Além disso, devemos usar um build machine, afinal pode ser que na máquina local não seja possível executar todas as etapas do build. Nos aprofundaremos mais no assunto na próxima aula.

06

Para saber mais: Algumas ferramentas

As ferramentas de automação e construção variam muito pois são especificas da linguagem e plataforma mas também variam na área de uso como desenvolvimento web, mobile ou data science. Além disso, existem ferramentas especificas para uma etapa de build como *teste* ou *deploy*. Enfim, a lista abaixo representa apenas alguns exemplos.

Ferramentas de automação e construção, separado por linguagem/plataforma:

- Java: Ant, Maven, Gradle
- Front-end: Gulp, Grunt, Webpack
- .NET: MSBuild
- Ruby: Rake
- Kotlin: Gradle
- Clojure: Leiningen
- C/C++: CMake/Make
- PHP: Composer

E alguns frameworks famosos da área de teste:

- Selenium (automação do navegador)
- Cucumber (testes de aceitação)
- Postman e SoapUI (testes de API)
- JMeter (stress tests)
- JUnit, xUnit, PHPUnit (automação de testes)
- entre muitos outros frameworks e bibliotecas

Para o *configuration management e provisioning* podemos mencionar:

- Ansible
- Puppet
- Chef
- Salt
- Terraform (cloud)

O *configuration management / provisioning* é sobre a instalação e manutenção da maquina.

Na Alura temos cursos específicos para a maioria das ferramentas.

01

Servidor de integração contínua

Transcrição

Discutimos sobre testes automatizados e builds automatizados. Idealmente, cada desenvolvedor deverá realizar os testes adequados para garantir que não haja quebras e então passar as informações para o repositório central. Mas quem pode garantir que o desenvolvedor de fato cumprirá esse processo?

Existem casos por questão de tempo em que não podemos realizar todos os testes necessários ou o projeto é tão complexo que não pode ser testado integralmente em uma máquina local.

Em algum momento precisamos encontrar um local que realmente será capaz de testar todo o projeto, e este é o **CI Daemon** ou server de integração. Esse servidor irá garantir que a cada modificação seja realizado um novo teste, então o repositório é sempre checado e builds contínuos são uma realidade.

A prática de integração contínua não especifica que o CI Daemon é necessário, mas hoje em dia podemos considerar como uma obrigação nos projetos. Contudo, só porque utilizamos um server desas natureza estamos praticando integração contínua, uma ação é realizada por pessoas.

O CI Daemon realiza builds contínuos e notifica os desenvolvedores se alguma alteração se deu corretamente ou não. Normalmente esses softwares já possui um test board que oferecem os dados do build, então as informações e relatórios são de fácil acesso.

Existem alguns serviços no mercado que podem ser instalados localmente, o **Jenkins** é um exemplo. Existem também serviços na nuvem já associados. É

importante que apenas que o servidor realize buidls a cada commit e notifique os desenvolvedores de maneira clara e funcional.

Alguns servidores oferecem o private build, que possibilita o desenvolvedor a fazer uma observação do repositório e testa-lo antes da produção, com um build mais completo.

Mas e se ocorre a notificação de um erro? Como devemos proceder?

03

Para saber mais: Servidores do mercado

Listamos abaixo alguns servidores de integração disponíveis no mercado. Isso **não é uma lista ordenada** por popularidade e algum outro critério.

Alguns servidores são *opensource*, outros não, alguns são pagos ou podem ser alocados na nuvem e outros só existem para nuvem ou instalação local.

Em geral, não existe uma *bala de prata* e a melhor ferramenta é aquela que te serve bem:

- Jenkins (https://jenkins.io/ (https://jenkins.io/))
- GoCD (https://www.gocd.org/ (https://www.gocd.org/))
- Bamboo (https://www.atlassian.com/br/software/bamboo (https://www.atlassian.com/br/software/bamboo))
- Travis CI (https://travis-ci.org/ (https://travis-ci.org/))
- Team City (https://www.jetbrains.com/teamcity/ (https://www.jetbrains.com/teamcity/))
- Circle CI (https://circleci.com/ (https://circleci.com/))
- Gitlab (https://about.gitlab.com/product/continuous-integration/ (https://about.gitlab.com/product/continuous-integration/))
- AWS Code Pipeline https://aws.amazon.com/codepipeline/ (https://aws.amazon.com/codepipeline/)
- Azure (https://azure.microsoft.com/pt-br/services/devops/server/ (https://azure.microsoft.com/pt-br/services/devops/server/))

entre outras possibilidades!

04

Build quebrado

Transcrição

No último vídeo falamos sobre servidores de integração, e paramos no ponto em que o um erro é notificado no build. É importante lembrar - e devemos pensar também no conceito de entrega contínua - que o software funcional é a principal métrica que devemos nos atentar.

É tarefa de toda uma equipe resolver um problema no build o mais rápido possível, afinal não poderá ser realizados novos commits até que o build esteja estável e confiável novamente.

> "Nobody has a higher priority task than fixing the build." -Kent Beeck

Caso o build não possa ser corrigido em um intervalo de tempo curto, o commit que gerou o problema deve ser desfeito para o que o projeto continue, e este é um ponto fundamental.

O que mais envolve a integração contínua? Conversaremos sobre o deploy nas próximas aulas.

01

Certificação de CI

Transcrição

Avançamos bastante em nosso curso sobre integração contínua, então vamos checar nossos conhecimentos em uma "certificação". A ideia da certificação é refletir sobre a sua própria equipe, e responder as próximas três perguntas.

1. Você "comita" diariamente o seu código no "mainline" do projeto?

Sabemos que integração contínua exige esse fazer, pois devemos construir o projeto de maneira fragmentada e com feedback rápido.

2. Build e testes rodam automatizados e trazem confiança que o software está correto?

Alguns projetos possuem os testes automatizados, mas não realizam a quantidade suficiente para garantir a confiabilidade do sofware. O build pode falhar, mas a princípio os testes trazem a confiança para realizar os commits.

3. Quando um build quebra, a equipe o conserta rapidamente? Tempo ideial de 10 minutos.

Quando um bug é detectado a equipe deve se movimentar e fazer a resolução desse problema sua maior prioridade.

02

O que é entrega contínua?

Transcrição

Discutimos até aqui os processos da integração contínua: repositórios, servidor de integração, relatórios, notificações, artefato de build e assim por diante. Contudo, será que só essas práticas são o suficiente no desenvolvimento de softwares?

Observemos o famoso manifesto ágil:

> "Working software over comprehensive documentation"

Um softawere funcional é mais importante do que uma documentação enorme. Mas o que significa um software funcional?

Novamente observaremos outra citação do manifesto ágil:

> "Our highest priority is to satisfy the customer through early and continuous delivery of valuable software."

A prioridade mais alta é deixar o cliente satisfeito, e isso é feito por meio de **entrega contínua**. Isto é, um software funcional é aquele usado pelo cliente com suas novas features de maneira confiável. A integração contínua é uma parte fundamental para se chegar à entrega contínua.

Ao revisitarmos a metáfora do botão de integração em que tudo é simples e automatizado, devemos ter um outro botão de "release", isto é, de deploy.

Qual alteração que chega em nosso trunk principal e pode entrar em produção? Obviamente existem decisões de negócio sobre lançar novas features, como esperar datas comerciais importantes. Mas na visão técnica, devemos saber quais são as modificações aplicáveis.

A entrega contínua é um assunto para um próximo curso, mas sobrevoaremos alguns conceitos e suas principais características.

Dentro da arquitetura da aplicação, é anexado um novo requisito funcional: "deployability". Esta característica deveria ser pensada desde o início, pois devemos entregar com frequência o software funcional.

Um deployment pipeline é um fluxo em que o artefato passa e cada etapa incrementa e agrega mais segurança ao se aproximar do ambiente de produção.

Deploys de baixo risco são aqueles experimentais, contínuos, atualizações pontuais acompanhadas. Devemos separar a ideia de "deploy" e "publicação".Otimizar para resiliência é prevenir erros. No contexto de integração e entrega contínua temos o deploy contínuo.

As equipes de desenvolvimento normalmente possuem divisões, como as pessoas do QA,deploy e operações. As tarefas são delegadas para os subgrupos durante o projeto. Equipes separadas que mal se comunicam dificultam o andamento do trabalho, aumentam a possibilidade de problemas e atrasam os deploys.

A entrega contínua também exige uma mudança no comportamento e na cultura da empresa: as pessoas precisam trabalhar integradas.

04

Sobre DevOps

Transcrição

Discutimos sobre as equipes que trabalham divididas, com muros materiais imateriais. A velocidade de equipes pode ser medida por meio de tarefas que são implementadas, essa é uma ideia.

Por outro lado, a equipe de operações valoriza a estabilidade, afinal cada alteração é uma possível causa de problemas. O operacional deseja controlar o ambiente de produção. Temos diferenças importantes na forma de trabalho: um preza velocidade em novas funcionalidades e outro estabilidade.

As equipes precisam trabalhar em conjunto para conseguir chegar à entrega contínua e ter um softaware funcional em ambiente de produção.

DevOps é um movimento cultural que fisa a integração e otimizar o processo de aprendizagem entre os integrantes da equipe. DevOps não é um título de cargo, mas uma visão de organização de trabalho que visa criar um pipeline veloz, seguro e integrado.

Precisamos, claro, de ferramentas que irão facilitar as integrações e monitoramento, mas é muito mais um movimento culturao do que uma aparelhagem técnica.

07

Conclusão

Transcrição

Parabéns! Chegamos ao final do nosso curso de integração contínua em que aprendemos muito sobre boas práticas dentro dessa metodologia de trabalho.

Estudamos os sistemas de controle de versão e organização de repositórios: um projeto por repositório ou vários dentro do mesmo.

Caracterizamos a integração contínua, sua essência e benefícios de commits melhores sempre no trunk principal para evitar o risco de problemas no projeto.

Conhecemos algumas estratégias de ramificação, sempre com a visão de que quando afastamos o commit do trunk principal não utilizamos integração contínua.

Testes e builds automatizados são obrigatórios! Se iremos alterar o tempo todo nossa base de código precisamos realizar testes de unidade e outros níveis para garantir a qualidade do softaware.

Dentro dessa lógica, é necessário um servidor de build contínuo. O repositório sempre estará monitorado e a equipe notificada caso haja erros.

Por fim, chegamos à entrega contínua e DevOps em que devemos otimizar e garantir a qualidade do projeto também por mundaças culturais dentro da empresa. Equipes integradas e que realizam trocas úteis para produzir o melhor produto e satisfazer o cliente.

Não pare por aqui e continue estudando!