DevOps

UNIDADE 02

Não é só Git: sistemas de controle de versão

Lembrando dos trabalhos em grupo do ensino médio

É provável que você não queria se lembrar dos trabalhos em grupo aqui, mas acho que é um bom exemplo para explicar o porquê de o Git existir. Vamos usar como exemplo um trabalho em grupo que deve ser feito no Word. O projeto consiste em vários capítulos, gráficos, referências e anexos. Todos no grupo tem que contribuir e vocês decidem dividir o trabalho em seções.

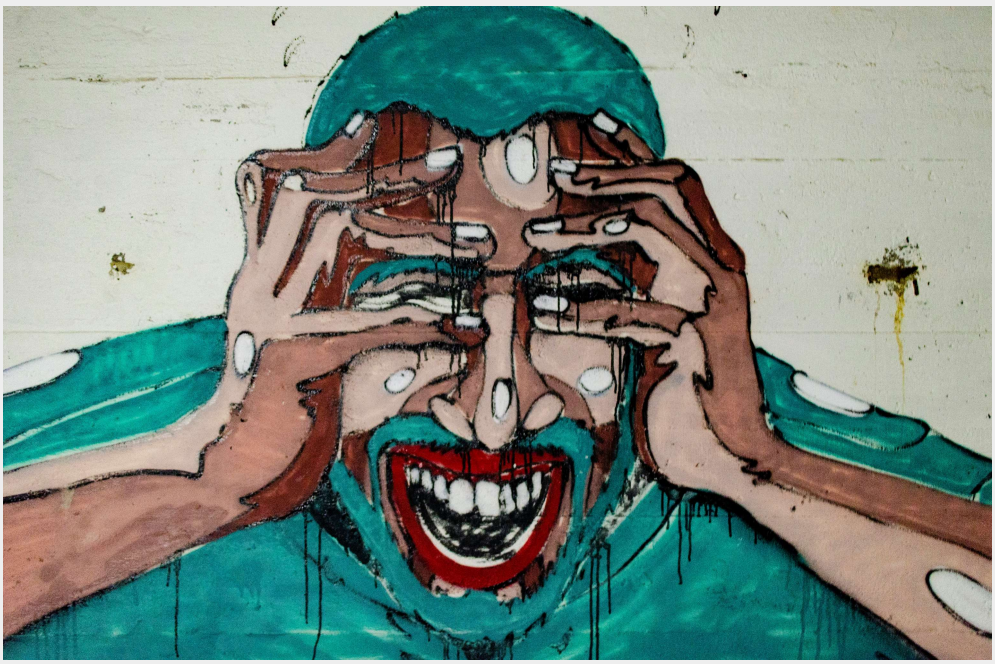
Inicialmente, vocês tentam enviar versões do trabalho por *e-mail*, mas rapidamente se deparam com confusões. Quem tem a versão mais recente? Quem corrigiu aquele erro no capítulo 3? E se duas pessoas editarem a mesma seção ao mesmo tempo?

Rapidamente, seu *e-mail* está inundado com versões como TrabalhoFinal_v1.docx, TrabalhoFinal_v2_revisado.docx, TrabalhoFinal_v3_Joao_editou.docx, e assim por diante.

Além disso, quase na data final da entrega da atividade, alguém acidentalmente sobrescreve o **que outra pessoa fez**, perdendo, desse modo, várias horas de esforço. Já pensou no estresse que isso seria, né?

Você, ao perceber que alguém estragou todo o trabalho que você fez por várias horas.

#ParaTodosVerem



Fonte: @the_meaning_of_love / Unsplash

Agora, e se tivesse um sistema mágico, como um **Google Docs avançado**, no qual todos pudessem fazer as suas contribuições em partes distintas do trabalho ao mesmo tempo, e sem ter o risco de um atrapalhar o outro? Cada mudança é registrada com detalhes: quem fez, quando fez e o que foi alterado. Se alguém cometer um erro, você pode facilmente voltar a uma versão anterior, ou até comparar o que mudou. Se duas pessoas editarem a mesma parte simultaneamente, o sistema avisa e permite a vocês decidirem qual versão manter, ou até combinar ambas. Interessante, né?

Esse Google Docs avançado é como se fosse um **sistema de controle de versões (*version control system*, ou VCS)**, mas para projetos de *software*. Ele garante que todos os colaboradores possam trabalhar juntos, mantendo um histórico detalhado de todas

as alterações. Em vez de se perderem em um mar de versões de documentos e correções, você possui uma linha clara de progresso e colaboração. Sabe quem ajudou com o que, e em qual momento do tempo.

DICA

Isso é especialmente interessante para as pessoas que interagem com a construção e manutenção de sistemas de TI. Ao contrário de um único documento, essas pessoas trabalham com centenas ou milhares de arquivos ao mesmo tempo e, para ajudar, não é uma equipe de um trabalho de ensino médio de três ou quatro pessoas, mas sim dezenas (ou centenas) de pessoas ao mesmo tempo dentro de uma empresa.

Entendendo melhor os sistemas de controle de versões (VCS)

Continuando com as nossas analogias, vamos usar um exemplo de um grande livro feito por vários escritores. Isso é comum em trabalhos científicos e em áreas como o direito, por exemplo. Nessa analogia, cada pessoa que trabalha na criação e desenvolvimento de *software* em uma empresa é como se fosse uma escritora trabalhando num grande livro em parceria com outros escritores. E cada capítulo desse livro é como se fosse um projeto ou funcionalidade diferente de um *software*.

Em alguns dias, a nossa escritora escreve páginas e mais páginas. Em outros dias, ela decide reescrever ou corrigir algumas partes. No processo, ela pode querer voltar a versões anteriores do seu trabalho para verificar ou retomar algo que estava melhor antes. Seria um desastre se ela tivesse apenas um manuscrito e, ao fazer uma mudança, perdesse tudo o que foi escrito anteriormente.

É aí que entra o **controle de versão**. Cada vez que a nossa escritora termina de escrever algo significativo, ela pode salvar uma **versão** do seu trabalho na máquina. Se no futuro ela quiser voltar a um ponto anterior, basta selecionar a versão correspondente e... *voilà*! O livro está de volta àquela versão.

> *Ou seja: o segredo (ou vantagem) do controle de versões é possibilitar que várias pessoas trabalhem na mesma coisa ao mesmo tempo, e sem que um atrapalhe (muito) o trabalho do outro. Isso é bem importante em desenvolvimento de software, já que várias pessoas desenvolvem diferentes funcionalidades ou correções dentro de um mesmo sistema.*

Vamos pensar em um outro exemplo, agora voltando ao caso do iFood que usamos na semana passada. Partamos da premissa de que você resolveu criar uma versão totalmente repaginada do aplicativo do iFood, feito do zero. Esse aplicativo possui várias telas, por exemplo, uma de pagamento, uma com o pedido em andamento, e uma com os restaurantes listados. Além disso, precisamos trabalhar em outros códigos que não necessariamente o cliente verá, mas que são imprescindíveis para o negócio. Isso inclui códigos para realizar o cadastro de clientes, pagamentos por cartão, e registro de pedidos. E, como você deve imaginar, vai precisar de uma equipe grande para cuidar de tudo isso.

Sistemas de controle de versão mais comuns

Agora, empresas com uma preocupação minimamente aceitável com a segurança dos seus códigos usam algum tipo de sistema de controle de versão. Vou explicar brevemente alguns deles e, ainda, algumas analogias usando sistemas bancários, beleza?

Git

Este é o sistema de controle de versão mais usado nas empresas. Ele é distribuído, o que significa que cada desenvolvedor tem uma cópia completa do histórico do repositório em seu computador.

Curiosidade: a mesma pessoa que criou o Linux (que também é a base do Android) também criou o Git.

Ele poderia ser comparado com uma instituição financeira moderna, como o Nubank. Cada cliente (desenvolvedor) tem uma conta bancária completa em seu celular (uma cópia do repositório). Eles podem revisar seu histórico de transações, fazer pagamentos e outras atividades sem precisar ir ao banco. De tempos em tempos, as informações que estão no celular são sincronizadas com o banco para garantir que todos os registros estejam alinhados.

DICA

Se você já ouviu falar em Git, também deve ter ouvido falar sobre o **Github**. Agora, qual é a diferença entre eles? Se o Git é uma tecnologia, o Github (ou competidores como o GitLab, Bitbucket e outros) são soluções que implementam aquela tecnologia. Isto é: se o Git fosse um chinelo, o Github/GitLab/Bitbucket e outros seriam marcas como Havaianas, Ipanema, Rider, entre outros. Ficou mais claro?

Subversion (SVN)

Antes do Git existir, o SVN era muito popular. E, diferentemente do Git, ele é centralizado. Isso significa que existe um repositório central e os desenvolvedores baixam e enviam alterações para ele.

Como uma analogia, ele é parecido com um banco tradicional. Existe apenas uma agência na cidade (repositório). Os clientes (desenvolvedores) precisam ir ao banco para ver seu histórico completo de transações ou fazer qualquer atividade significativa. Eles podem ter um pedaço de papel guardado em casa contendo as suas transações (cópia de trabalho), mas o registro completo está apenas no banco.

Mercurial

Assim como o Git, o Mercurial é distribuído. Ele tende a ser um pouco mais simples em termos de interface do usuário.

Ele pode ser comparado com algum tipo de novo banco por aplicativo. Ele possui menos funcionalidades e é mais simplificado, mas essa simplicidade pode ser desejável em alguns casos específicos.

Team Foundation Version Control (TFVC)

Oferecido pela Microsoft como parte de uma solução chamada Azure DevOps. É centralizado, semelhante ao SVN.

Como uma analogia, ele é parecido com um banco tradicional e específico para pessoas jurídicas (PJ). Não é feito para todas as pessoas, e possui algumas funcionalidades bem específicas. Não é necessariamente pior ou melhor do que os outros, mas possui aplicações e casos de uso específicos.

Semelhanças e diferenças

Elas possuem o mesmo objetivo: rastrear quem mudou o quê (e quando), e manter um histórico de versões. Isso ajuda muito na colaboração entre os membros de uma equipe e em momentos que as alterações devem ser revertidas se algo der errado (o que é muito, muito comum em TI). O quadro abaixo pode nos ajudar a melhor entender os prós e contras de cada solução:

Aspecto	Git	Subversion (SVN)	Mercurial	TFVC (Team Foundation Version Control)
Modelo de controle	Distribuído	Centralizado	Distribuído	Centralizado
Desempenho	Alta *performance* em operações locais	Desempenho adequado, mas pode ser mais lento que Git e Mercurial	Alta *performance* em operações locais	Desempenho adequado, mas pode ser mais lento que Git e Mercurial

Uso corporativo	Amplo uso em projetos *open-source* e corporativos	Usado em ambientes corporativos, especialmente legados	Usado em projetos que preferem simplicidade e desempenho	Amplamente usado em ambientes corporativos com outras ferramentas Microsoft
Ferramentas e integrações	Muitas ferramentas e integrações disponíveis	Boa quantidade de ferramentas, mas menos que o Git	Menos ferramentas que o Git, mas suficientes	Forte integração com ferramentas Microsoft (Azure DevOps, Visual Studio)
Popularidade	Muito popular	Menos popular que o Git atualmente	Popularidade média	Menos popular; usado em ambientes corporativos Microsoft
Instalação e configuração	Necessita de instalação e configuração locais	Simples, instalação no servidor	Necessita instalação e configuração local	Configuração integrada com Azure DevOps
Comunidade e suporte	Grande comunidade e muito suporte	Comunidade menor que Git, mas ainda grande	Comunidade menor que Git, mas ainda ativa	Suporte da Microsoft, comunidade menor
Controle de permissões	Controle de permissões por repositório	Controle de permissões detalhado por diretório	Controle de permissões por repositório	Controle de permissões detalhado por diretório
Plataformas cuportadas	Multiplataforma	Multiplataforma	Multiplataforma	Principalmente Windows, integração

				com Azure DevOps

Atualmente, a esmagadora maioria dos desenvolvedores trabalham com Git no mercado de trabalho. Aliás, também é comum ensinarmos Git dentro dos cursos de TI nas Universidades. Logo, também trabalharemos com Git dentro desta disciplina.

Trabalhando com Git

Agora, como uma equipe trabalha com Git? O primeiro conceito que precisamos esclarecer é o de **repositório (ou repo):** este é o local seguro onde os desenvolvedores de *software* guardam todos os códigos e outros recursos adicionais de que o aplicativo possa precisar. Dependendo do caso, isso pode incluir coisas como o ícone do aplicativo e algumas documentações de código internas.

O que acha de trabalharmos juntos em um exemplo com um repo? Primeiro, trabalharemos em um exemplo puramente visual. Depois, com os comandos.

Boas práticas com Git

Existe uma diferença entre **trabalhar com Git e trabalhar com qualidade no Git**. Afinal, se você busca crescer na carreira é importante que você saiba como fazer as coisas direito, não é? Logo, gostaria que você seguisse as dicas abaixo quando estiver trabalhando com Git. Isso não vale somente para esta disciplina, mas também para a sua carreira, beleza?

1. **Não atrapalhe os outros**
    a. As pessoas gostam de pessoas legais. Seja uma pessoa legal, e tente facilitar o trabalho dos outros. Quanto mais você ajuda os outros ao criar um código de fácil entendimento, melhor para você.
2. ***Commits* pequenos e frequentes são melhores do que *commits* grandes e espaçados**
    a. Pense em cada ***commit*** como sendo um vídeo curto do TikTok ou do YouTube. Cada episódio (*commit*) deve contar uma parte pequena da história, sendo mais fácil encontrar a parte do vídeo que possui aquilo que estamos procurando.
    b. *Commits* pequenos e frequentes ajudam a rastrear mudanças e identificar problemas com mais facilidade.
3. **Garanta que as suas mensagens de *commit* sejam claras e descritivas**
    a. As mensagens de *commit* funcionam como se fossem títulos de capítulos em um livro.
    b. Portanto, elas devem ser claras e descritivas para que qualquer um possa entender o que foi alterado sem precisar ler todo o código.

4. **Utilize *branches* para funcionalidades e correções**
    a. Pense nos *branches* como galhos de uma árvore (afinal, ***branch*** significa **galho**, mesmo em português).
    b. Cada *branch* deve representar uma nova funcionalidade ou correção, permitindo trabalhar de forma independente do resto do código até que você esteja pronto para integrá-lo (*merge*) ao *branch* principal (*master* ou *main*).
5. **Garanta que o *branch* principal seja sempre confiável**
    a. O *branch* principal (chamado de *main*" ou *master* em repositórios mais antigos) deve estar sempre pronto para produção. Isto é: ele sempre deve conter uma versão confiável para consumo dos usuários.
    b. Por isso, evite fazer *commits* diretos nele: sempre trabalhe em um *branch* separado e use *pull requests* (PRs) com um processo de revisão de código para garantir que apenas códigos de qualidade sejam integrados.
6. **Sincronize seu trabalho frequentemente com o repositório remoto**
    a. Mantenha-o sincronizado com o repositório remoto para evitar conflitos.
    b. Portanto, procure sempre puxar (*pull*) e enviar (*push*) mudanças com frequência (ao menos diariamente) ao repositório remoto.

Pull requests

Lembra que comentei que os repositórios geralmente possuem um *branch* principal, e que geralmente ele é chamado de ***main*** ou de ***master***? Então: para compartilharmos a responsabilidade entre mais de uma pessoa (ou equipe) é importante termos um processo de **revisão do trabalho**. Errar é humano, e é comum acabarmos nos esquecendo de alguma coisa no código, ou de introduzirmos algum tipo de erro ou *bug* no código. Quando pedimos a ajuda de outras pessoas para revisar o nosso trabalho **antes de um *merge* no *branch principal***, o risco de erros diminui bastante.

E é aí que entram os ***pull requests*** (ou PRs). Um PR é um mecanismo que permite aos indivíduos contribuírem para projetos abertos ou colaborativos, sugerindo mudanças que podem ser revisadas e discutidas antes de serem aceitas no *branch* principal. Trabalhar com PRs é algo muito legal porque ajudam a equipe ao:

- **Facilitar na colaboração**, já que qualquer pessoa pode contribuir para um repositório independentemente de ser um membro oficial da equipe. A diferença aqui é que as pessoas que estão no projeto são responsáveis por validar essas alterações.
- **Garantir a qualidade**, já que as mudanças podem ser revisadas, discutidas e testadas antes de serem integradas.

- **Ajudar muito na documentação e auditoria**, já que cada PR tem uma descrição e discussão associada, criando um histórico rastreável do porquê e como as decisões foram tomadas.

DICA

Mantenha poucos arquivos na sua PR (menos de 10, se possível). Isso ajuda muito no processo de revisão, uma vez que os revisores precisarão olhar menos arquivos.

Cheat Sheet

Cheat Sheet e Soluções Remotas: GitHub, GitLab e BitBucket

DICA

Está com dificuldades para entender Git? Sugerimos a leitura do Git – Guia Prático.

Configuração Inicial

```
# Configurar nome de usuário
git config --global user.name "Seu Nome"

# Configurar email
git config --global user.email "seuemail@example.com"
```

Criando um Repositório

```
# Inicializar um novo repositório Git
git init

# Clonar um repositório existente
git clone URL_DO_REPOSITORIO
```

Trabalhando com o status dos arquivos

```
# Verificar o status dos arquivos
git status

# Adicionar arquivos para staging
git add NOME_DO_ARQUIVO

# Adicionar todos os arquivos para staging
git add .
```

Trabalhando com Commits de Mudanças

```
# Fazer commit das mudanças
git commit -m "Mensagem do commit"
```

```
# Commitar todos os arquivos modificados e adicionar uma
mensagem
git commit -am "Mensagem do commit"
```

Branching e merging

```
# Listar branches
git branch

# Criar um novo branch
git branch NOME_DO_BRANCH

# Mudar para um branch específico
git checkout NOME_DO_BRANCH

# Criar e mudar para um novo branch
git checkout -b NOME_DO_BRANCH

# Mesclar um branch no branch atual
git merge NOME_DO_BRANCH

# Deletar um branch
git branch -d NOME_DO_BRANCH
```

Sincronização com Repositório Remoto

```
# Adicionar um repositório remoto
git remote add origin URL_DO_REPOSITORIO

# Verificar repositórios remotos
git remote -v
```

```
# Puxar mudanças do repositório remoto
git pull origin BRANCH_NAME

# Enviar mudanças para o repositório remoto
git push origin BRANCH_NAME
```

Visualizando o histórico

```
# Ver o histórico de commits
git log

# Ver o histórico de commits com diffs de um arquivo
específico
git log -p NOME_DO_ARQUIVO

# Ver um gráfico do histórico de commits
git log --graph --oneline --all
```

Desfazer mudanças

```
# Desfazer mudanças no arquivo antes do commit
git checkout -- NOME_DO_ARQUIVO

# Resetar staging area para o último commit
git reset HEAD NOME_DO_ARQUIVO

# Resetar para um commit específico (permanente)
git reset --hard ID_DO_COMMIT
```

Comandos úteis

```
```

```
# Ver diferenças entre arquivos
git diff

# Ver diferenças entre o staging e o último commit
git diff --staged

# Stash mudanças temporariamente
git stash

# Aplicar mudanças stashed
git stash apply

# Listar stashes
git stash list
```

Conclusão

Chegamos ao final da nossa segunda semana, na qual vimos os sistemas de controle de versão (*version control system*, ou VCS). Durante esta semana, exploramos como os VCS são fundamentais para qualquer prática de DevOps, permitindo a múltiplos desenvolvedores colaborarem no mesmo código, rastrearem mudanças e reverterem facilmente para versões anteriores quando necessário. Sem um VCS robusto, seria impossível gerenciar o desenvolvimento de *software* em escala, garantir a integridade do código e manter a produtividade da equipe. Vimos como o uso de um VCS é a espinha dorsal de uma integração contínua eficaz, já que é ela que permite o desenvolvimento e a entrega contínua de *software* de alta qualidade. Quer dizer: não dá para falar sobre CI/CD sem um VCS, como o Git.

Embora o Git seja o sistema de controle de versão mais popular e amplamente utilizado hoje, ele não é a única opção. Há outros sistemas de controle de versão que as empresas usam, como o Mercurial, o Subversion (SVN) e o TFVC (também conhecido por algumas pessoas como TFS). Nesta disciplina, focamos principalmente o Git devido à sua popularidade e ampla adoção na indústria. No entanto, é essencial reconhecer a existência e o valor desses outros sistemas. Ignorar essas alternativas seria ocultar o conhecimento para você e, ainda, limitar o seu potencial quanto às práticas de DevOps.

FREEMAN, E. **DevOps para leigos**. Rio de Janeiro: Editora Alta Books, 2021.

Referências

KIM, G.; BEHR, K,; SPAFFORD, G. **O projeto fênix**. Rio de Janeiro: Editora Alta Books, 2020.

KIM, G.; HUMBLE, J.; DEBOIS, P.; WILLIS, J. **Manual de DevOps**. Rio de Janeiro: Editora Alta Books, 2018.