=== tutorial - end

Pro próximo exemplo, uma coisa muito comum é você começar a trabalhar numa coisa, aí você dá uma divagada, resolve codar outra coisa não relacionada, só que agora você ficou com duas coisas não relacionadas que precisa comitar. Vamos pro caso mais simples. Onde você criou arquivos relacionados a uma coisa e criou outros arquivos que são de outra coisa diferente.

A maioria das pessoas faz um git add . (ponto) e comita tudo junto. Tá errado! Commits devem ter só uma "coisa" de cada vez. Um commit pra uma pequena funcionalidade. Um commit pra correção de bugs. Um commit pra documentação. Meio difícil definir o que é essa "coisa" mas usa o bom senso, pra variar. Não faz sentido você misturar um arquivo CSS de uma tela com uma linha nova de README descrevendo infra, por exemplo.

=== tutorial - ini

Vamos criar uns arquivos css qualquer. E vamos criar um arquivo README por exemplo. E vamos criar mais algumas coisa só pra ficar mais realista. A coisa mais simples é dar git add e o nome do css e fazer o commit só do css. Depois fazer git add separado do readme e um commit só pra ele. Quando você tem poucos arquivos esse é o jeito mais simples. Mas normalmente você se empolga, ou tá usando algum gerador de código de algum framework e fica com um tanto de arquivo.

Eu sei, é um saco digitar git add pra caaaada arquivo pra organizar o commit. Digamos que você começou fazendo a besteira de fazer git add . (ponto) agora todos os arquivos estão misturados no stage e se você der git commit vai tudo pro mesmo commit. Ainda tem como consertar. Se for só um arquivo que você quer tirar do stage basta fazer git reset -- (traço traço duas vezes) e o nome do arquivo. Pronto, se der git status, veja que ele saiu do stage.

Mas como eu falei, faz de conta que são uma dúzia de arquivos e você deu git add . de tudo. Novamente, a preguiça reina e você tá quase ligando o foda-se e fazendo commit de tudo. Calma! Ainda dá pra consertar fácil. Digite git add -i que é de novo opção interativa. O git vai te abrir esse menuzinho e você pode usar a opção 3 que é revert. Ele vai listar os arquivos que estão no stage e você pode só digitar o número dos arquivos que não quer comitar agora. No final dá enter pra voltar pro menu anterior.

Digamos que você olha e fala, puuuts, tinha um arquivo que eu tirei que era pra estar lá mesmo na real. Então você tem a opção 2 de update. É o oposto, agora você digita o número do arquivo que quer colocar no stage. Dá enter no final, e no menu principal digita q pra dar quit e sair. Pronto. Vamos dar git status e olha só, é como se tivéssemos dado git add manualmente arquivo por arquivo. Agora sim podemos fazer o commit correto e fazer git add -i de novo pra ir adicionando arquivos pro próximo commit. Super simples.

Mas, como eu falei, esse é o caso simples. E se no mesmo arquivo temos modificações que um é pra ir num commit e outro é pra ir só em outro commit? Vamos fazer um arquivo README qualquer com um conteúdo aleatório pra servir de base. (...) Pronto. Agora vamos fazer 3 modificações. Agora eu queria que a primeira modificação e a terceira fossem um commit só. E a segunda modificação fosse um commit separado. Vamos digitar git diff pra ver o patch completo. Olha só, se eu der git add e comitar vai tudo no mesmo commit. E agora?

Não se preocupem, pra isso tem git add -p. A idéia é que o git vai perguntar um hunk de modificação de cada vez. Veja só, como as modificações estão próximas ele me pergunta se quer adicionar elas juntas pro stage. Se eu quisesse bastaria digitar y de yes, ou se eu quiser pular eu digito n pra não e esse tanto de opção aqui embaixo se eu der enter ou interrogação, ele vai explicar todas as opções que eu tenho. Vale a pena explorar cada uma delas depois, num repositório local. Mas no meu caso eu não quero adicionar tudo junto, quero dividir. Pra isso eu digito s de split, e veja só ele dividiu o hunk mais uma vez e me deu só a primeira parte! E é isso que eu quero. Então vamos dar yes. O próximo hunk é o que eu quero separado, então digito “no” pra pular, e a última eu também quero no mesmo commit, então dou yes. Se eu quisesse parar no meio, só digitar q de quit.

Se eu der git status vemos que parte do arquivo README está no stage e parte não está. Se eu der git diff --staged pra ver só o que vai no próximo commit, veja que está só o hunk que eu escolhi. Vamos criar o commit. Pronto, e agora só sobrou a segunda modificação que eu posso adicionar no stage e criar o segundo commit. (...) Entenderam? Essa parte pode ser meio complicada da primeira vez então recomendo assistir de novo. Mas resumindo, eu tinha modificações num mesmo arquivo que eu queria separar em commits diferentes. Pra isso usamos git add -p. Aliás esse p é de patch.

== tutorial - end

Trabalhar com o git tem sempre 3 estágios. O primeiro estágio acontece onde chamamos de working tree que é o diretório normal de trabalho com seus arquivos pra você editar e tudo mais. Pra criar um commit você precisa ir estacionando as modificações num lugar intermediário, esse lugar chamamos de stage. O Git não rastreia arquivos, ele rastreia conteúdo. Hunks e patches, lembra que eu falei no episódio passado? Podemos adicionar todas as modificações de um ou mais arquivos com os comandos git add ou git add interativo -i ou adicionar um hunk de cada arquivo de cada vez usando git add -p.

Aliás, como o Git rastreia conteúdo e não diretórios, por isso que diretórios vazios nunca entram num repositório git e por isso muitos frameworks reservam diretórios vazios colocando um arquivo vazio escondido como .gitkeep só pro diretório ter alguma coisa e com isso conseguir adicionar no Git. Uma vez que eu tenho tudo que eu quero no stage, posso empacotar com o comando git commit, onde você cadastra uma mensagem e o git vai terminar adicionando o resto dos metadados pra grudar seu commit como a nova cabeça do grafo, com a etiqueta de HEAD.

Mas mesmo depois que você já comitou, ainda dá pra voltar atrás. Você pode fazer git reset tanto hard pra apagar tudo sem dó nem piedade. Ou git reset soft pra tirar os commits do grafo e jogar as modificações de volta pro stage. Vamos fazer isso pra você ver.

=== tutorial - ini

Digamos que temos alguns commits. Vamos fazer git reset --soft HEAD~2 pra apagar 2 commits. Essa sintaxe de HEAD e tilzinho pode ser mais complicado se você tiver merges no meio do caminho, então recomendo pesquisar e estudar como isso funciona na documentação oficial, mas pra maioria dos casos serve associar que esse número é o número de commits pra trás. === tutorial - end

Por exemplo, se você quiser reverter um commit de merge que aponta pra outros 10 commits, ele é virar um galho paralelo no histórico. Mas basta usar HEAD~1 que ele vai matar só o commit do merge que é quem liga os outros commits e você some com os 10. Isso não é tão intuitivo então é melhor você praticar.

=== tutorial - ini

De qualquer forma, demos reset soft. Se dermos git log veja que os commits desapareceram, mas você não perdeu nada. Se der git status veja que está tudo no stage, como se você tivesse acabado de dar git add. Nesse ponto, como já fizemos antes, podemos dar git commit e empacotar tudo num commit. Mas e se alguns desses arquivos eu não quero no próximo commit? Pra isso podemos fazer git reset -- (traço traço) e o nome do arquivo. Ou só git reset -- e ele vai tirar tudo do stage. Olha no git status como mudou e eles saíram do stage. A partir desse ponto podemos fazer git add -i ou git add -p como já expliquei antes.

Mas digamos que tô estressado e quero só marretar tudo pro lixo. Então eu vou fazer git reset --hard porque eu sei que só tem lixo aqui mesmo ... e puta que pariu! tinha um arquivo que eu não podia perder!! e agora???

Calma!! Se você tiver o SHA-1 do último commit tá tudo certo. E por sorte demos um git log aqui em cima, então temos o SHA-1 e podemos fazer o equivalente a um undelete. Mesmo se eu não tivesse ainda ia dar, só ia dar um pouquinho mais de trabalho pra achar o commit certo. Como eu disse antes o Git é um banco de dados de grafos. Quando demos o reset só mudamos a etiqueta, ou referência do branch master dois commits pra trás. Os commits em si não foram apagados, eles foram só marcados pra serem apagados. Aliás, pro tipo: sempre evite apagar dados, melhor sempre marcar como apagado. De temos em tempos o Git roda um garbage collector, que você pode executar manualmente com o comando git gc, que seria o equivalente a esvaziar a lixeira no Windows.

Mas não queremos rodar o GC agora. Vamos primeiro tirar nossos commits da lixeira. Pra isso podemos rodar o comando git reflog -b (pra criar um novo branch) teste xxxxx (com o SHA-1). Aproveitando, note que normalmente usamos só um pedaço do começo do SHA-1 em vez do SHA-1 inteiro porque como raramente eles colidem, só a versão curta é suficiente pra achar o commit. E pronto! Se dermos git log veja que o master continua apontando 2 commits pra trás, mas acabamos de anexar a etiqueta de teste pro commit antigo.

Se eu quiser recuperar de volta pro master, basta dar git checkout master pra voltar pro master, e como estamos trabalhando só localmente sem dar push ou pull de servidor, podemos fazer um git rebase teste. E se dermos git log de novo veja que o branch master está apontando pro mesmo commit que o branch teste. Se listarmos os arquivos, pronto, tá tudo recuperado! Agora podemos respirar fundo e reescrever esses commits sem perder algo importante. Ah sim, e pode apagar esse branch temporário com git branch -d teste. Pronto, tudo limpo.

=== tutorial - end

Tudo que eu mostrei até agora são comandos que você pode e deve usar no seu dia a dia pra manter o histórico dos seus commits limpos. Muita gente, especialmente iniciantes e amadores, ligam o foda-se pra isso. Eu mesmo sou culpado de já ter dado push de muito commit ruim então não posso julgar tanto. O correto é deixar limpo. Repositório limpo, código limpo, tudo isso faz parte de você se tornar um profissional. Se for um espaço compartilhado, demonstra sua falta de respeito com as pessoas ao redor.

E se você é um tech lead ou seja lá que posição que é responsável por mergear pull requests dos outros, é parte do seu trabalho rejeitar pull requests se eles estiverem sujos e ruim. Senão pra que você serve? Rejeite sumariamente pull requests que estão quebrados, incompletos, não tem testes, estão bagunçados com coisas que não deviam estar misturados, com commits sem mensagens decentes e descritivas. Tudo isso faz parte do protocolo de boa vizinhança.

“Ah mas estamos com pressa, os prazos tão apertados, foda-se se tá um pouco bagunçado ou um pouco quebrado, depois a gente arruma”. Não, depois que alguma coisa é mergeada na master, ninguém nunca mais vai arrumar. Faz direito da primeira vez. Sujeira se acumula, e ninguém nunca vai limpar. O que vai acontecer, como sempre acontece, é que lá na frente essa sujeira vai aparecer. E quando a produtividade cair, número de bugs aumentar, mais pessoas novas entrarem na equipe com todo mundo copiando os maus maus hábitos, é tarde demais. É muito difícil, se não impossível, mudar maus hábitos depois. Então enforce bons hábitos no começo.

Dez anos atrás tinha um excelente programador na minha equipe. E ele gostava de mentorar os outros desenvolvedores. Eu não tenho certeza mas eu acho que todo mundo podia dar push direto na master, ainda não era tão comum usar Pull Requests como fluxo de trabalho. Ele basicamente apagava o commit da master se o novo código estivesse sem testes. A regra dele era muito simples "código sem teste não tem lugar no repositório". E ele tinha razão. No começo todo mundo reclama, depois sabe o que acontece? Todo o código está com cobertura de teste. E a produtividade não cai. Mais do que isso, ela aumenta, porque no futuro não acumulou débito técnico.

Débito técnico daria um outro episódio inteiro, mas resumindo é qualquer coisa que você sabe que devia fazer, mas acha que vai te deixar mais devagar, então você escolhe pular. Você cria uma dívida. Só que como o nome diz, uma hora você tem que pagar essa dívida. E quanto mais demora mais os juros vão aumentando. Vai ficando cada vez mais caro, mais e mais rápido. É uma escolha, e às vezes realmente compensa assumir uma pequena dívida, basta ter consciência de pagar os juros sem reclamar depois.

Mudando de assunto, falando em projetos dos outros, vai uma outra dica. Tô parecendo vitrola quebrada, mas na documentação no site oficial do Git tem um capítulo sobre migração de projetos pra Git. Eu mencionei que mesmo se você estiver num projeto subversion, ainda assim pode usar Git. Pra fazer isso primeiro você vai instalar o suporte a subversion do Git.

=== tutorial - ini

Vamos voltar pro Ubuntu e instalar o pacote apt install git-svn. Todas as distros tem algum pacote similar. Feito isso vamos clonar o projeto do próprio subversion, que obviamente está em subversion.

git svn clone ....

Isso vai demorar um tempão, então vai almoçar. Quando voltar você vai ter um clone do repositório em Git. A partir daqui, você pode simplesmente pode dar git remote pra adicionar um repositório vazio do Github por exemplo, e dar push pra lá. E a partir desse ponto, pode jogar fora o repositório subversion. Todo o histórico está salvo.

=== tutorial - end

Mas, digamos que você não possa fazer isso porque sei lá porque motivo a empresa exige que precisa comitar no servidor de subversion mesmo. Não tem problema. Localmente tudo que eu expliquei no tutorial até agora vai funcionar igualzinho. É um repositório Git normal. De tempos em tempos você pode fazer git svn rebase pra puxar os últimos commits do subversion. E quando você tiver terminado basta agora fazer um git svn dcommit (observe o d antes da palavra commit). Me repetindo de novo, diferente de um commit de Git que é só local commit de subversion é pra mandar pro servidor. Aqui o git svn vai pegar os commits que você trabalhou localmente no Git e criar commits de subversion.

A graça disso é que outros desenvolvedores de subversion, quando derem svn update, vão baixar seus commits como se você nunca tivesse usado Git. A diferença é que você vai trabalhar muito melhor que todo mundo, porque você pode fazer branches locais pra experimentar coisas e trabalhar aos poucos fazendo commits pequenos, e só no final pode fazer squash dos seus branchs e criar commits limpos pra mandar de volta pro servidor legado. Era o que eu fazia em 2007. E digamos que em vez de subversion você tá usando Mercurial, Bazaar, Perforce ou mesmo o antigo TFS. Tem opções similares pra todos eles. Olhem a documentação no site oficial.

Uma última dica importante que muitos iniciantes fazem errado. Digamos que você é um desenvolvedor front-end. Digamos que você receba arquivos de Photoshop dos designers. E como é de costume; vira e mexe os designers ficam mandando novas versões dos photoshops. Uma hora vocês perdem noção de qual era a versão mais nova. E ninguém chega num consenso de qual era a versão do design certo pra mexer. Pra facilitar, alguém tem a grande idéia de criar uma pasta chamada photoshop na raíz do projeto e vocês ensinam o designer a usar Git também. Daí sempre que tiver versões novas, todo mundo recebe no próximo git pull e ninguém mais se confunde. Ótima idéia ….

Só que isso tá errado se você não souber o que é LFS. Lembrem do que eu ensinei no episódio anterior. Ferramentas como diff, patch e o próprio Git tem algoritmos muito bons pra lidar com hunks e patches de texto. Porém, photoshop é um arquivo binário. Não existe nenhuma forma eficiente de se fazer patches de arquivos binários. Se você tentar mergear binários sem saber a estrutura interna delas, a chance de dar problemas é praticamente garantido. Portanto o Git não faz deltas de binários, ele grava o binário inteiro. Aliás, ele grava o arquivo inteiro pra cada modificação.

Pra entender o problema pensa assim. Digamos que o photoshop original tinha 200 Mb. Você adiciona ele no Git. O designer vai mexendo, e todo dia ele cria um novo commit. E ele faz isso por 5 dias só. Em cada commit vai os 200Mb inteiro de cada versão, porque como eu disse, não tem como tirar delta de binário. Ao final de 5 dias seu repositório tem 5 vezes 200 Mb ou 1 Gb. Agora todo novo desenvolvedor que fizer um git clone vai ter que puxar 1 Gb, sendo que o código mesmo, no mesmo repositório, são poucos megas. E normalmente você só precisaria do último photoshop do designer.

Se for arquivos pequenos como jpgs ou pngs tudo bem, mas qualquer coisa muito maior que isso precisa ser tratado diferente. Isso é uma das arestas que ainda não é exatamente a melhor solução, mas pra casos assim, onde você precisa armazenar grandes binários, como photoshops, videos, audio, instaladores, você tem que lembrar de fazer da seguinte maneira:

=== tutorial - ini

Em Ubuntu, primeiro você precisa instalar apt install git-lfs. Pronto, agora no diretório do projeto você inicializa com git lfs install. Finalmente você tem que dizer pro Git que tipos de arquivos você quer jogar no LFS. Pra isso você faz git lfs track "\*.psd" por exemplo. Isso vai criar um arquivo chamado .gitattributes com essa condição.

Agora você pode começar a dar git add de arquivos com extensão .psd. Vamos adicionar alguns psds, dar git add, criar um commit e até aqui, tudo igualzinho com era antes. Do seu ponto de vista de desenvolvedor, não parece que mudou nada.

=== tutorial - end

O que está acontecendo agora é que no commit não vai mais anexado o arquivo binário inteiro. Em vez de jogar o binário no diretório .git/objects, que eu ensinei no episódio anterior, ele vai jogar em outro diretório, o .git/lfs. E no commit em si, em vez de ir armazenado o binário inteiro, vai só um texto indicando onde tá o arquivo. Pense como se fosse um link simbólico num Linux. Assim o commit fica super leve, sem o bináriozão pesando nele. A vantagem disso é que agora quando você fizer um novo git clone do zero, não vai mais vir todo o 1 GB com todas as versões de photoshop que o designer comitou. Vai vir só o último e você pode pedir versões antigas só se precisar.

A desvantagem disso é que instalar e configurar o suporte de LFS num servidor de Git seu pode ser um pouco chatinho de gerenciar. Se você estiver usando GitHub ou GitLab ou outra plataforma web, esse trabalho deve ser mais transparente. Na dúvida, sempre use git lfs se tiver arquivos binários grandes no seu projeto.

Porém, no caso do seu projeto ser basicamente só binários grandes e pouco ou nenhum texto, eu diria que o Git não é a melhor solução. Não só git, nenhum versionador feito pra código. Imagina uma agência de design ou uma produtora de vídeo. Ninguém usa Git num lugar desses. Se você precisa lidar com binário, o fluxo de trabalho é completamente diferente. Não tem como dois designers trabalharem ao mesmo tempo, em máquinas separadas, no mesmo arquivo de Illustrator. Não existe pull request de edição de vídeo de Premiere Pro. Não tem como fazer patches num arquivo de áudio de LogicPro. Trabalho de mídia, pela própria natureza, sempre vai ser um fluxo em cascata, onde a próxima etapa só pode ser iniciada quando a anterior já terminou. E sempre que precisa voltar passos é uma dor de cabeça mesmo.

Por isso mesmo, não é incomum ver servidores de arquivos em rede de alta velocidade, rede de 10 gigabit, com servidores mais parrudos, com gigabytes de RAM pra fazer buffer, e soluções bem caras como os famosos Jellyfish da LumaForge, cujos modelos mais baratos como o de 32TB móvel, começa em mais de 9 mil dólares. E 32TB é bem pouco espaço na real. O modelo servidor, que começa com 80TB não sai por menos de 30 mil dólares. Tá vendo porque um Mac Pro de 10 mil dólares não é caro pra lugares como esses?

Voltando ao assunto. Pra poucos binários como designs de photoshop, video em mp4 de tutorial do seu produto, manuais em PDF e coisas assim, use o git lfs. Mas, digamos que você não sabia disso, e já tem um repositório com um monte de binário, e já tá sentindo que tá lento, todo mundo que faz git clone fica esperando um tempão baixar tudo pela internet. E na cidade que você mora sequer tem suporte a internet de fibra com mais banda. E agora?

O problema de um repositório de versionamento é que por princípio, o binário agora está anexado num commit e esse binário é usado também pra gerar o SHA-1 que identifica o commit. Lembra? Se eu tirar o binário vai necessariamente mudar o SHA-1. Qual o problema disso? Lembra que o Git é um DAG, ou seja um grafo, onde os nós são cada commit apontando pro commit anterior? E como um commit aponta pro anterior? Fácil, armazenando nele o SHA-1 desse commit anterior. Ou seja, se o SHA-1 do commit anterior mudar, o SHA-1 de Todos os outros commits que apontam pra ele mudam também.

Portanto, sim, é possível arrancar fora um arquivo do Git, mas pra fazer isso precisamos literalmente reescrever toda a história até o ponto onde o binário foi adicionado pela primeira vez. Se fizer isso, todo mundo que trabalhava no repositório antigo vai precisar dar um git clone novo tudo outra vez. Se você tá no ponto onde seu repositório tem gigas de tamanho e tá ficando cada vez pior, só adicionar o git lfs não vai adiantar porque o lfs vai ficar ativo só pra novos commits. Mas os antigos vão continuar no repositório.

Entendido isso, vamos aos finalmentes. A situação tá feia, a equipe concorda que vale a pena todo mundo parar de trabalhar um pouco pra limpar o repositório. Lembre de avisar todo mundo pra ver se deram push de tudo. Uma solução é usar o recurso de filter-branch do Git mas ele é muito lento. Não, em vez disso existe uma ferramenta feita em java chamada BFG. Eu gosto dele só pelo nome. Eu não sei se foi essa a inspiração mas qualquer um que tenha jogado Doom sabe o que é um Big Fucking Gun.

Vou deixar o link nas descrições abaixo. Mas resumindo, primeiro você instala Java na sua distro. Como cada um tem seu jeito, veja nas documentações pra cada sistema operacional. Num ubuntu seria um apt install openjdk. Agora você baixa o .jar neste link. Pra executar um java, se você não sabe é rodar o comando java -jar com o caminho pra onde você baixou o .jar.

Agora você precisa fazer um git clone --mirror do repositório gigante. Tem que ser com --mirror porque quando você acabar e der git push ele vai reescrever todo o repositório. Mais ou menos como se você fizesse um git push com force.

Com o BFG você pode escolher remover arquivos específicos pelo próprio nome. Mas se forem muitos, você pode escolher remover todos os blobs maiores que um certo tamanho por exemplo. Digamos que todos os seus videos tenham 100mb ou mais, então use esta opção pra remover por esse critério.

Tem um outro caso de uso pro BFG, se por acaso em vez de arquivos grandes na realidade você fez a besteira de iniciante que todo mundo faz, e comitou arquivos de configuração com senhas hardcoded de coisas de produção. Se você não sabia disso, deixa eu explicar: JAMAIS comite coisas com senha num repositório. Mesmo se você apagar do arquivo depois o histórico vai continuar lá. Se você fez a monumental besteira de comitar senhas no Git, agora você precisa obrigatoriamente atirar o BFG pra reescrever a história também.

Lembram que eu já expliquei que o git não apaga pra sempre seus commits? O BFG vai gerar novos commits e reapontar todas as referências de branchs e tudo mais corretamente, mas os commits antigos vão continuar na lixeira. Pra apagar definitivamente você vai usar o comando git gc. O procedimento completo é primeiro rodar um git reflog pra expirar tudo que você não vai precisar e depois rodar o git gc assim. (imagem)

Feito isso, é só dar um git push digamos pro repositório principal no Github. Agora você pode apagar essa cópia local onde rodou o BFG e todo mundo pode dar git clone de novo do novo histórico. E sim, o procedimento não é exatamente rápido nem trivial, mas também a idéia é que se você fizer certo como eu expliquei até agora, você não devia nunca precisar rodar algo assim. O BFG é pros casos raros onde você fez merda e agora temos que limpar a merda. Se isso fosse Pulp Fiction, o BFG seria Winston The Wolf e você é o Vincent ou o Jules tendo que limpar a merda dentro do carro.

E falando em sujeira, vamos falar rapidamente sobre monorepos. Esse é um daqueles assuntos que arrisca ser controverso, mas vou tentar simplificar pra vocês. Se você nunca ouviu falar dessa palavra, se não me engano começou com o Twitter. Mas o que interessa é o seguinte. Digamos que você é uma empresa de tecnologia gigante, mais especificamente uma com poucos produtos muito populares e centenas de desenvolvedores internos. Como num Twitter. Pior ainda, digamos que esse produto é composto por dezenas de bibliotecas e microserviços. Imagine o pesadelo logístico.

Você tem dezenas de pull requests aparecendo toda hora em dezenas de repositórios de código. Alguns deles podem acabar quebrando as integrações entre os diversos componentes. Num exemplo hipotético simples, digamos que num dos repositórios você atualize uma biblioteca de ... parsing de JSON por conta de uma melhoria de performance. Mas digamos que em outro repositório você precise voltar uma versão pra trás dessa biblioteca porque a versão nova causa um erro de compatibilidade com seu código. E agora imagine um terceiro repositório que importa o código desses dois repositórios, cada um dependendo da mesma biblioteca em versões diferentes. Fodeu.

Agora imagine situações como essa acontecendo na mão de centenas de desenvolvedores, toda hora, todos os dias. Imagine a dificuldade que é no final empacotar tudo junto pra gerar um deploy coerente. Milhões de linhas de código, divididos em dezenas ou centenas de repositórios, recebendo modificações o dia inteiro pela mão de centenas de desenvolvedores, sendo vários desses times distribuídos geograficamente e separados até por fusos horários diferentes, e no final precisa empacotar uma versão de produção sem bugs. Ah, e isso tem que acontecer com bastante frequência, tipo uma vez por semana ou mesmo uma vez por dia. Ah sim, e o deploy acontece em centenas de máquinas espalhadas em dezenas de data centers pelo mundo.

E pra piorar imagine 20 anos de histórico de código acumulado no caso de alguém como Google. Falando em Google, no caso dele estamos falando de mais de um bilhão de linhas de código. Mais de 25 mil engenheiros. E quase 20 anos de histórico num mono repositório gigante com todo o código da empresa que hoje já passou dos 80 terabytes de tamanho. Só de código! Esse é o tipo de desafio que empresas com Twitter, Google, Facebook, Netflix, e outros gigantes enfrentam todos os dias.

Nesse contexto, é muito fácil de imaginar que uma das técnicas que pode ajudar é reunir todos os projetos que compõe o produto num único repositório. Mais do que isso, vendorizar, ou seja, baixar as dependências de terceiros e também comitar junto com o código; em vez de instalar bibliotecas só durante o processo de deployment em produção. Por que você acha que o Go, que é a linguagem criada pelo Google, não tem um gerenciador de dependência tradicional como um NPM de Javascript? Porque o Google prefere manter todas as dependências estáticas junto com o código do projeto. Eles também preferem compilar todas as dependências estaticamente no mesmo binariozão, de forma que não precise depender de quase nada instalado no sistema operacional. Essa é a necessidade deles, e um Go da vida foi feito primariamente pra resolver os problemas específicos deles, não necessariamente os seus.

Casos como Google, Twitter ou Facebook são obviamente pontos fora da curva. Você pode contar nos dedos quantas empresas tem o mesmo nível de problemas. Agora vem o ponto principal: você não é o Google! Desculpa, as chances da sua empresa ser o próximo Twitter ou o próximo Facebook é uma fração de uma fração de uma fração de um porcento, o que em matemática nós diríamos que é o limite tendendo a zero.

Quando um engenheiro de uma dessas empresas apresenta em palestras e eventos ou posts de blog sobre o que eles estão fazendo, esses insights servem na maior parte das vezes só pra outras empresas de porte similar. Portanto se você é uma empresa com uma equipe de 10 ou 20 pessoas, com 10 ou 20 repositórios, e só meras dezenas de milhares de linhas de código, fazendo deployment em produção uma vez por quinzena; então quase nada da escalabilidade de um Google se aplica a você. Se você é menor que isso, pode pular esse assunto por um bom tempo, ele não te interessa.

Uma das coisas mais prejudiciais que alguém pode fazer é adotar uma tecnologia ou uma técnica baseado na premissa errada de “se o Google usa, deve ser bom e devemos usar também”. Não, essa premissa não faz o menor sentido, porque o correto deveria ser “se o Google usa, qualquer empresa de mesmo porte e com os mesmos problemas, talvez se beneficiem e talvez valha a pena tentar”. Você tem o porte do Google? Mais importante, você tem o dinheiro que o Google investe? Lógico que não. Então não, você nunca, jamais, deve usar alguma coisa meramente porque você é fanboy de uma marca. Fanboyismo é o oposto de engenharia. Engenharia é encontrar a solução do tamanho certo pro problema certo em mãos, dentro das restrições do seu contexto. Nem mais, nem menos.

Agora tem um problema similar que vale a pena colocar em perspectiva. Eu mencionei no episódio anterior que o desenvolvimento da kernel do Linux não está no GitHub. Um dos motivos é que o Linus Torvalds jamais vai confiar num terceiro pra manter a cópia mestra do kernel.

(torvalds - google)

Mas tirando isso, seria bem difícil usar o fluxo de trabalho e interface gráfica ou mesmo as apis do GitHub. Assim como no caso do monorepo, daria pra ficar horas listando item a item todos os prós e contras e explicar tecnicamente os efeitos pra cada caso particular. Em vez disso quero que você leiam os links que vou deixar nas descrições abaixo e por hoje quero listar só os pontos que eu acho importantes.

A kernel do Linux é um dos projetos open source mais antigos em atividade hoje, e com maior quantidade de código, e com a maior quantidade de voluntários colaborando ao redor do mundo. Dezenas de empresas, de dezenas de setores, dependem do Linux. A kernel em si é tecnicamente um monorepo. Mais corretamente ela é um monotree. Diferente de projetos dentro de um Google, onde a maior parte dos repositórios é centralizada e todo mundo clona de um único lugar. No caso da kernel, existe o repositório do Linus, só que o único desenvolvedor que trabalha nesse repositório é o próprio Linus.

Cada distro como Ubuntu, Fedora ou outros tem seus próprios repositórios principais, que não necessariamente clonam direto do Linus. Cada fabricante de hardware que desenvolve drivers tem repositórios diferentes. Mais importante, diversos produtos estão trabalhando em versões diferentes da kernel. Por exemplo, na própria Canonical que faz o Ubuntu, você tem o Ubuntu LTS que é o Long Term Service e o Ubuntu com versões mais novas dos componentes. A idéia do LTS é que ele atualize o menos possível pra não ficar quebrando compatibilidade o tempo todo.

Porém, digamos que seja descoberto um problema grave de segurança num driver na kernel nova. E esse bug afeta várias versões da kernel pra trás. O bug é descoberto no Fedora primeiro. Então a correção é feita no repositório deles. O patch da correção começa a ser distribuída em mailing lists. Cada empresa ou comunidade com um repositório tem seu mailing list. A forma de distribuir pra tanta gente é literalmente usar o recurso de CC ou carbon copy de e-mail, e assim distribuir o patch pra diversos lugares diferentes.

Pense por um momento num GitHub. Se cada empresa tem uma organização e cada um tem um fork. É como se você tivesse um jeito de postar o mesmo pull request pra diferentes repositórios em diferentes organizações. Coisa que na interface gráfica atual, não dá pra fazer. Na interface web você só devolve pull request pro repositório de quem fez fork. Mas o problema não acaba aí. Não quer dizer que esse patch vai funcionar em todas as versões e diferentes variações de cada organização. Agora os responsáveis precisam testar, muita discussão pode acontecer. Numa mailing list essa discussão é um Reply All pra todo mundo. Num GitHub seriam as Issues em cada repositório, só que no GitHub as issues são atreladas a um repositório. Como que o cara da RedHat que tem repositório próprio vai ficar respondendo nas Issues dele e nas Issues de outras organizações como da Canonical, sobre o mesmo problema? De novo, a interface gráfica atual não tem como resolver esse problema.

Mais do que isso, cada área de interesse dentro do kernel, driver de rede, o X, o systemd e assim por diante tem uma lista de mantenedores. Eles que em última instância vão aprovar ou reprovar uma modificação em seu subsistema. No caso do Linux, se eu quiser achar quem é o responsável pra receber o tal patch de segurança, eu procuro no arquivo chamado MAINTAINERS que tem na raíz do projeto. No caso de um GitHub, cada mantenedor talvez teria um fork do projeto na sua conta do GitHub. Mas não existe uma forma fácil de achar quais dos forks são de mantenedores ativos e quais dos forks são de colaboradores. De novo, a interface gráfica atual não resolve esse problema.

Em última instância, até depois dos outros repositórios já terem aplicado o patch e deployado, a versão mestra pode acabar aplicando o patch depois dos outros, porque o Linus vai ser mais cuidadoso em aceitar os patches, talvez até rejeite e mande modificar. Mas enquanto isso a RedHat ou Canonical precisam fechar o bug pros seus clientes, independente do Linus ter aprovado o patch ou não. Vai ter muitas idas e vindas em cada patch, e não existe a obrigação de que os forks vão esperar pra dar git fetch do Linus só depois dele ter dado merge no repositório dele.

O fluxo de trabalho num GitHub envolve fazer forks dos repositórios, discutir Issues na interface gráfica do repositório, e receber colaborações via pull requests vindos de outros forks. O fluxo de trabalho da kernel do Linux envolve dezenas de repositórios distribuídos que são forks do repositório do Linux, com discussões acontecendo simultaneamente em dezenas de mailing lists, e esses e-mails servindo como canal de distribuição de dezenas de patches de código que são automatizados de diversas maneiras diferentes por cada empresa ou mantenedor. Uma das primeiras coisas que é diferente no fluxo é que no GitHub só tem como fazer pull request de volta pro repositório de onde se fez fork. Na kernel você quer distribuir os patches pra múltiplos repositórios de uma só vez. Além de pode discutir o assunto em paralelo entre diversos repositórios ao mesmo tempo.

É por essas e várias outras razões que, hoje, não tem como suportar o fluxo de trabalho da kernel no GitHub ou GitLab ou qualquer um que siga esse mesmo modelo e tenha essa escala. O fluxo deles segue uma árvore, onde o principal é a raíz e ele vai expandindo numa única direção pra cada galho. No caso do Linux é uma rede, um mesh, onde não existe direção definida, cada nó da rede pode se comunicar com qualquer outro nó da rede, sem um que coordena ou bloqueia essa comunicação. O Git foi originalmente feito com essa estrutura totalmente distribuída em mente. Hoje em dia nós simplificamos de uma forma mais centralizada, como num GitHub, pra ficar mais usável pra maioria dos mortais.

Em resumo, sim, se o GitHub ou GitLab resolverem criar uma estrutura de projeto de múltiplas árvores do mesmo repositório separados pra mantenedores diferentes com uma forma mais simples de unificar a cooperação entre repositórios, tanto do ponto de vista de Issues e Pull Requests distribuídos, talvez, bem talvez, um projeto como Linux poderia viver nessas plataformas. Mas seria um tanto de funcionalidade extra pra atender um nicho muito pequeno de projetos. Pouquíssimos projetos tem os requerimentos da kernel do Linux, por isso não vejo isso acontecendo tão cedo.

Pros colaboradores do kernel, tirando alguns subgrupos que podem trabalhar mais isoladamente; digamos, equipes de drivers ou algo assim, a grande maioria ainda vai ser mais produtiva com simples mailing lists e distribuindo patches mesmo, sem ficar na dependência de nenhuma plataforma controlando a comunicação.

Em ambos os casos, os monorepos do Google e Facebook ou o monotree da kernel do Linux. Nenhum é o nosso caso comum, meu ou seu. Pra maioria esmagadora dos projetos, o GitFlow do GitHub, com um repositório principal, diversos forks, e pull requests que voltam pro repositório principal, e um único lugar pra registrar Issues, ainda é o caso de uso mais produtivo.

Você como desenvolvedor, antes de perder muito tempo tentando achar que deveria usar monorepos porque é fã do Google ou que deveria usar fluxos de monotrees porque é fã do Linus, primeiro pára de fazer commits sujos com mensagens de “fixing” pra lá e pra cá e escreve código decente. Eu posso estar sendo injusto em generalizar, mas vou fazer isso mesmo assim: normalmente quem é fanboy demais e fica só falando em “deveríamos, deveríamos” é o primeiro que faz o trabalho mal feito, pela metade, sujo. Antes de pensar em utopias, primeiro limpa seu quadrado aí, acaba as coisas que começa, e escreve código limpo pra variar. Depois sai cagando regra por aí.

E por hoje é isso aí! Pelo menos pelo futuro próximo eu acho que falei o grosso do que eu tinha pra falar sobre Git. E como eu disse no primeiro episódio de Conhecimentos Gerais, Git é obrigatório saber. E Git é uma ferramenta que à medida que você vai usando e explorando, mais surpresas ele vai revelando. Não é uma coisa que você aprende num workshop de poucas horas e já acha que sabe tudo. Eu diria que se você realmente gosta de programação, Git vai te manter fascinado por um bom tempo. Comece no site oficial, leia a documentação oficial, é lá que se começa. Pare de perder tempo com divagações desnecessárias, ponha a mão na massa e escreva código limpo, pra variar. Se ficaram com dúvidas, não deixem de mandar nos comentários abaixo, se curtiram o vídeo deixem um joinha, assinem o canal e não deixem de clicar no sininho pra não perder os próximos episódios. A gente se vê, até mais.