As quatro áreas: fluxo de trabalho básico Revisitando o básico Bem-vindo ao segundo módulo do Mastering Git. Bom te ver de novo. No módulo anterior, eu disse que, para entender um comando Git, devemos fazer algumas perguntas. Como esse comando move as informações pelas quatro áreas? E como isso afeta o repositório? Neste segundo módulo, examinaremos os comandos básicos do fluxo de trabalho do Git, os que você já conhece, adicionar, confirmar, finalizar a compra, mover e arquivos restantes, os trabalhos. Somente, examinaremos esses comandos através do filtro dessas duas perguntas. Portanto, mesmo que você já conheça esses comandos, isso pode ser uma novidade. Vamos começar imediatamente. Movendo dados para a direita Você já conhece o fluxo de trabalho básico do Git. Edite um arquivo, prepare-o e confirme o arquivo. Vejamos esses comandos novamente, mantendo as duas perguntas em mente. Para começar, editarei os dados na minha área de trabalho no arquivo menu.txt. Eu amo a culinária indiana, acho incrível, então vamos adicionar algo indiano, ou pelo menos indiano internacional. Lá, temos frango tikka masala no menu agora. Então, mudei o arquivo de menu e isso mudará sua cor no diagrama também. Eu vou fazer laranja. E o status nos diz que o arquivo foi alterado, mas ainda não está preparado. É vermelho. Agora eu quero copiar o arquivo atualizado da área de trabalho para o índice. Você sabe o comando que faz isso, é git add. Aí está você. O arquivo foi copiado da área de trabalho para o índice, substituindo a versão anterior do mesmo arquivo. Agora, o status git me diz que o arquivo é modificado e preparado, e o git diff não vê diferença. A área de trabalho e o índice estão alinhados. Mas se eu comparar o índice e o repositório, posso ver as alterações. Eles estão prontos para levá-lo ao próximo commit. Então vamos fazer isso, git commit. E assim que confirmamos, o arquivo atualizado é copiado do índice para o repositório e agora tudo está alinhado novamente. Portanto, no decorrer dessas poucas operações, movemos os dados da esquerda para a direita, por assim dizer, da área de trabalho para o índice e para o repositório. E agora, como resultado, as três áreas contêm exatamente os mesmos dados. Não há diferença alguma. Estamos no estado limpo novamente. Observe que o último comando, commit, fez mais com o repositório do que apenas copiar este arquivo. Ele também criou um novo commit e outros objetos. Atualizou o ramo atual, todas as coisas que não estou mostrando nesta imagem aqui. Portanto, a confirmação move os dados e também altera o repositório. Na verdade, é um dos comandos mais importantes que alteram o repositório. Por outro lado, o add apenas move os dados e não toca no repositório. Movendo dados para a esquerda Então, vimos dois comandos que movem dados da esquerda para a direita, da área de trabalho para o índice que foi adicionado e do índice para o repositório que foi confirmado. Uma pergunta rápida para você. Você consegue pensar em um comando que move os dados na outra direção, do repositório para a área de trabalho e o índice? Há um que é realmente comum. Entendi? Sim, é checkout. O Google Checkout faz duas coisas essencialmente. No repositório, ele move a referência do cabeçalho, geralmente para outra ramificação, para alterar a confirmação atual. E a segunda coisa que faz é obter dados do novo commit atual e copiar esses dados do repositório para a área de trabalho e o índice. Portanto, ele muda o repositório primeiro e move os dados depois. Ok, isso provavelmente foi difícil de seguir. Deixe-me mostrar como isso acontece, e tudo deve ficar claro. Temos alguns ramos aqui. Vamos olhar para o ramo liso. Eu posso ver as diferenças entre esse ramo e nosso ramo mestre atual pelo filtro diff. Em lisa, o arquivo de menu é diferente e outros arquivos no diretório de receitas também são diferentes. Então, o que acontece se fizermos o checkout do ramo lisa? Aconteceram algumas coisas. A primeira coisa que você provavelmente já sabe sobre isso é olhar para esses arquivos no repositório. Há uma parte de confirmação nos seus dados e esta é a confirmação atual. É o commit atual, porque existe um ramo apontando para ele, que é mestre, e o cabeçalho está apontando para esse ramo. Bem, a primeira coisa que aconteceu quando fizemos o check-out é que a cabeça foi movida para o ramo liso, assim. Portanto, o commit atual foi alterado. Onde estamos agora, os arquivos são diferentes. Eu mostrei isso no diagrama usando cores diferentes para os arquivos. Como sempre, tomei alguns atalhos para tornar o diagrama mais simples. Eu ignorei muitos objetos no repositório, como os arquivos no diretório recipes, alguns commits de mesclagem e similares. Simplifiquei o diagrama para torná-lo menos ocupado, mas, caso contrário, essa é uma aproximação suficientemente boa para expressar meu argumento. Ou seja, não são os dados no repositório realmente alterados, é apenas a cabeça alterada. Portanto, a confirmação atual foi alterada, e estamos vendo dados diferentes. Este é um atalho. Quando dizemos que os dados no repositório foram alterados, na verdade queremos dizer que o commit atual foi alterado, então estamos analisando dados diferentes no repositório. A partir de agora, tomarei esse atalho como garantido, tanto no meu idioma quanto em meus diagramas. E tudo bem, essa foi a primeira coisa que o checkout fez: mudou de cabeça. O segundo, o último passo, foi copiar os dados atuais do repositório para a área de trabalho e o índice, como este. Portanto, agora todas as três áreas têm o mesmo conteúdo e estamos no status limpo novamente. Portanto, o checkout altera o repositório porque move o ponteiro para dentro e copia os dados do repositório para a área de trabalho e o índice. Ok, chega de falar sobre checkout, vamos voltar para o ramo principal. Então vamos ver. Vimos três comandos fundamentais: adicionar, confirmar e finalizar a compra. Você já conhecia esses comandos. Mas agora respondemos a duas perguntas fundamentais sobre cada uma delas. Primeiro, como esse comando move os dados pelas áreas e, segundo, o que esse comando faz no repositório? Por enquanto, tudo bem. Removendo arquivos Agora vamos falar sobre algo um pouco menos óbvio, removendo arquivos no Git. Estamos começando do status de limpeza com um diagrama limpo. Vamos ignorar os arquivos e diretórios existentes. Vou criar um novo arquivo. Tenho notas de direitos autorais para o meu projeto de livro de receitas. Vamos preenchê-lo, não quero que ninguém roube meu precioso livro de receitas. Lá, agora temos um novo arquivo na área de trabalho. Se eu solicitar um status, vejo esse arquivo marcado como não rastreado. Isso significa que o arquivo está na área de trabalho, mas não no índice ou no repositório. O Git ainda não sabe o que fazer com isso. Estou planejando confirmar esse arquivo, então usarei o git add para copiá-lo no índice. E agora o status diz que o arquivo é novo, o que significa que o arquivo está na área de trabalho e no índice, mas ainda não está no repositório. Ainda não é uma parte oficial do projeto. Se eu o confirmar agora, o arquivo será copiado para o repositório, as três áreas serão alinhadas e eu estaria no status limpo novamente. Mas e se eu mudasse de idéia? E se eu quiser remover o arquivo do índice? Digamos, quero manter esse arquivo na área de trabalho, mas não o quero mais no índice. Talvez eu queira confirmar outra coisa primeiro e voltar a esse arquivo mais tarde. Em outras palavras, quero voltar à situação em que estava antes da última estatística. Eu quero o arquivo na área de trabalho, mas não no índice. Agora, usei o comando add para copiar o arquivo para o índice, para que você possa pensar que eu posso usar o comando remove do git, rm, para remover o arquivo do índice. Remova sons como o oposto de add, certo? Bem, infelizmente, não é isso que remove. Ou melhor, essa não é a única coisa que remove. Se eu o usei sem nenhuma opção, o remove tentará excluir o arquivo da área de trabalho e do índice. Isso pode ser bastante destrutivo, portanto, o remove possui um tipo de recurso de segurança incorporado, que você pode ver se eu enviar este comando. O que aconteceu é que o git percebeu que o arquivo que estou removendo não está no repositório. Então, o git está basicamente me dizendo: olhe, eu não vejo esse arquivo no histórico do projeto; se eu removê-lo da área de trabalho e do índice, ele desaparecerá para sempre. Você tem certeza que deseja fazer isso? E isso me dá algumas opções. Eu posso forçar a remoção, o que é como dizer sim, eu sei o que estou fazendo, basta excluir este arquivo e esquecê-lo. E eles também têm a opção de remover o arquivo apenas do índice, mas não a área de trabalho com --cached. É isso que queremos que faça, então vamos fazê-lo. Lá, o arquivo ainda está na minha área de trabalho, mas eu o removi do índice. Ou seja, cancelei o teste do arquivo. Portanto, para recapitular, no Git, remover sem argumentos não é o oposto de add. Adicionar apenas altera seu índice, enquanto a remoção simples altera seu índice e sua área de trabalho. Você precisa usar a opção --cached para fazer o trabalho de remoção ser o oposto de adicionar. Vale a pena mencionar, porque é um pouco contra-intuitivo. Remove foi bastante desconcertante para mim na primeira vez em que o usei, ou nas primeiras vezes na verdade. Agora que falei sobre remover, deixe-me excluir o arquivo de direitos autorais definitivamente. Pensei, não acho que alguém queira roubar meu livro de receitas de qualquer maneira. Lá, removi o arquivo da área de trabalho e estamos com o status limpo novamente. Ok, é isso sobre a remoção de arquivos. Agora, vamos dar outro passo para um tópico um pouco mais complexo. Renomeando arquivos Vimos o que acontece quando você remove arquivos, agora vamos falar sobre mover e renomear arquivos. Mover e renomear é realmente a mesma coisa, certo? Renomear um arquivo é como movê-lo no lugar, se desejar, movê-lo para outro nome no mesmo diretório. Então, mostrarei o que acontece quando você renomeia arquivos aqui e o move é exatamente o mesmo mecanismo. Vamos começar com um status limpo, como de costume, com as três áreas que contêm os mesmos dados. Vou visualizar apenas um dos nossos arquivos, menu.txt. Eu quero renomear este arquivo e dar uma extensão de remarcação em vez de uma extensão de texto sem formatação. Estou planejando torná-lo um arquivo formatado de remarcação no futuro. Vou fazer isso na minha área de trabalho primeiro, lá. Então, agora o arquivo na minha área de trabalho mudou. Fiz amarelo na foto para mostrar que mudou. É como um arquivo completamente diferente. O que acontece se eu solicitar o status git? O status é um pouco confuso aqui. O Git pode ver que há um arquivo na área de trabalho que não está no índice, esse arquivo amarelo aqui. Por isso, diz que este arquivo é novo, não sei, não é rastreado. E também, o Git pode ver outro arquivo que está no índice, mas não na área de trabalho, esse arquivo laranja aqui. Então ele diz, esse arquivo está no índice, não está na área de trabalho, você o excluiu, certo? Bem, não, não exatamente. Como posso dizer ao Git, veja, esse é realmente o mesmo arquivo, apenas com um nome diferente? Bem, a boa notícia é que não preciso fazer isso. Eu posso apenas copiar todas as alterações da área de trabalho para o índice. Primeiro, deixe-me adicionar um novo arquivo. Lá, foi adicionado. E agora vamos cuidar do arquivo laranja adicionando esse também. Admito que isso parece estranho. Estou adicionando algo da área de trabalho ao índice, mas esse item não está na área de trabalho em primeiro lugar. Bem, lembre-se do que realmente significa adicionar. Isso significa copiar esses dados da área de trabalho para o índice. Portanto, se esses dados não forem nada, como neste caso, o Git simplesmente substituirá os dados no índice com nada, o que significa que removerá esses dados do índice. Então agora temos os mesmos dados na área de trabalho e no índice. Todas as nossas mudanças foram realizadas. E se pedirmos o status, surpresa! Veja isso, Git já entendeu o que está acontecendo. Ele comparou o conteúdo do arquivo na área de trabalho e o índice com o conteúdo dos arquivos no repositório. E percebeu que o arquivo laranja e o arquivo amarelo têm o mesmo conteúdo, portanto, eles devem ter o mesmo arquivo com um nome diferente. Isso é bastante inteligente e funciona tanto para renomear quanto para mover. Fica ainda melhor, na verdade o Git é mais inteligente do que isso. Na maioria dos casos, entende que você está renomeando ou movendo um arquivo, mesmo que você altere o conteúdo do arquivo ao mesmo tempo. Apenas diz que esses dois arquivos aqui parecem bastante semelhantes, deve ser o mesmo arquivo. Agora é claro que você pode confundir o Git se se esforçar muito. Por exemplo, se você move um arquivo e também altera a maior parte do conteúdo ao mesmo tempo, mas eu argumentaria que, se você chegar a esse ponto, talvez esteja fazendo muitas coisas ao mesmo tempo. Talvez considere mover e alterar o arquivo em dois comandos separados. Para mudanças simples, juntamente com movimentos, simplesmente funciona. Ah, a propósito, eu não fiz minhas alterações ainda. Vamos fazer isso. Agora, o arquivo foi renomeado no repositório e estamos no status de limpeza novamente. Acabei de lhe dizer que o Git rastreia sua mudança ou renomeação automaticamente, então tecnicamente você não precisa da operação de movimentação, certo? Ainda assim, existe essa operação, mas é apenas um comando de conveniência. Basicamente, faz as mesmas coisas que fizemos. Ele move o arquivo na área de trabalho e atualiza o índice, apenas o faz em uma única captura. Veja, eu não preciso renomear o arquivo primeiro e adicionar o arquivo antigo e o renomeado ao índice posteriormente. Eu fiz tudo isso com um único git mv. Francamente, eu pessoalmente nem uso muito o git mv, prefiro seguir as etapas manualmente e deixar o Git descobrir o que aconteceu. Ok, pouco antes de sair deste módulo, permita-me confirmar o que temos para reverter o efeito da confirmação anterior e deixar tudo no status limpo novamente. Lá! Um resumo rápido Vamos resumir o que vimos neste módulo rapidamente. Aqui está como os comandos básicos afetam as três principais áreas do Git. Adicionar dados de cópias da área de trabalho ao índice, isso não afeta o repositório. A confirmação copia os dados do índice para o repositório e também cria objetos adicionais no repositório, particularmente um novo commit, e move as referências no repositório. O checkout copia os dados do repositório para a área de trabalho e o índice e move a referência principal no repositório. Remover exclui arquivos da área de trabalho e do índice, mas não toca no repositório. E mv, a operação de movimentação, move o arquivo na área de trabalho e também atualiza o índice. Ele não toca no repositório, e você pode realmente ignorá-lo e executar as mesmas etapas que ele faz com os outros comandos básicos. Lá, cobrimos todos os conceitos básicos. Agora vamos avançar e aprofundar-se no território avançado. As quatro áreas: git reset Entendendo a redefinição Bem-vindo ao terceiro módulo no Master Git. Agora, estamos começando a falar sobre o que diferencia o usuário ocasional do Git de um usuário competente do Git; portanto, esteja pronto. Falaremos sobre um dos comandos mais úteis do Git, que também parece ser um dos mais difíceis de entender, o git reset. Eu tenho que admitir que por muito tempo fiquei nervoso com a redefinição. Usei-o para alguns casos de uso comuns, mas nunca senti que realmente entendia o que estava fazendo. Eu também sabia que a redefinição era uma operação potencialmente destrutiva; isso também me deixou bastante desconfortável com isso. Mas, como muitas outras coisas no Git, depois de entender como esse comando funciona, você se perguntará como ele se sentiu tão difícil em primeiro lugar. Por que uma redefinição é realmente um comando tão confuso? Uma razão é que, antes de entender uma redefinição, você precisa entender algumas outras coisas sobre o Git. Você precisa estar familiarizado com o funcionamento das ramificações e com a área de trabalho, o índice e o repositório. E se você não estiver familiarizado com isso, será difícil entender a redefinição. A boa notícia é que você está familiarizado com essas coisas agora, então está tudo pronto. A outra razão pela qual a redefinição parece confusa é que ela tem muitos casos de uso. Ou seja, você pode usá-lo de maneiras diferentes por razões muito diferentes. Portanto, se você procurar maneiras de fazer as coisas com o Git e vir a redefinição repetidamente, com pequenas variações, os resultados parecerão muito diferentes, dependendo da exata situação. Então você acaba pensando que deve ser um comando muito complexo. Bem, não é. Vamos ver como isso funciona. Vou lhe dizer o que a redefinição faz primeiro e depois veremos como essas operações podem se tornar úteis. Iniciarei nossa discussão sobre redefinição com um pequeno questionário. Quantos comandos Git você conhece que movem um ramo? Pense nisso por um momento. Você certamente conhece alguns; quatro deles, no mínimo. Você pode listá-los? Ok, seu tempo acabou. Os comandos que você conhece provavelmente movem uma ramificação porque criam novas confirmações e movem a ramificação atual ao mesmo tempo em que criam a nova confirmação. Então, commit é o óbvio. Ele cria uma nova confirmação e move a ramificação atual para apontar para a nova confirmação. A mesclagem também cria um commit na maioria dos casos, e também move o branch atual para parte do novo commit. E a rebase também faz algo semelhante. Ele cria novas confirmações copiando confirmações existentes nesse caso e move a ramificação atual para apontar para uma das novas confirmações. E também, o git pull obtém novas confirmações de um controle remoto e atualiza as ramificações local e remota. E talvez você saiba mais um pouco. Todas essas operações movem ramificações. No entanto, nenhum deles é uma operação especializada para mover uma ramificação. Todos eles movem ramificações nesses três como um efeito colateral da criação de novas confirmações ou da retirada remota. Você pode se perguntar se existe uma operação mais especializada, uma operação que explique explicitamente a movimentação de uma ramificação. Bem, isso é um reset. A coisa mais importante que uma redefinição faz, o primeiro passo de uma redefinição, por assim dizer, é apenas isso: move uma ramificação. Geralmente o ramo atual, o ramo para o qual a cabeça está apontada. Você escolhe uma confirmação, diga esta confirmação aqui, e a redefinição move o ramo atual para essa confirmação, para que a confirmação agora seja a confirmação atual. Observe que a redefinição não move a cabeça. A cabeça ainda está apontando para o mesmo ramo em que estava apontando antes, mas o próprio ramo está se movendo, de modo que a cabeça está seguindo o caminho. Se você olhar apenas para o repositório, é tudo o que a redefinição faz; move uma ramificação para apontar para o commit específico. A parte que você pode achar confusa, no entanto, não é o que uma redefinição faz no repositório. É o segundo passo, o que uma redefinição faz nas outras duas áreas principais: a área de trabalho e o índice. E a redefinição faz coisas diferentes, dependendo de suas opções. Se você der a opção --hard, a redefinição copiará os dados do novo commit atual para a área de trabalho e o índice. Com a opção --mixed, redefinir copia dados do novo commit atual para o índice, mas deixa a área de trabalho em paz. Esta é a opção padrão; portanto, se você não der nenhuma opção para redefinir, será uma redefinição mista. E, finalmente, a opção --soft significa não tocar em nenhuma das áreas, basta mover o ramo e pular completamente a etapa dois. Então, é isso que a redefinição faz. Primeiro, ele move a ramificação atual para alterar também a confirmação atual. E segundo, opcionalmente, copia os arquivos e diretórios do novo commit atual para a área de trabalho e o índice. Isso é tudo o que realmente faz. Ok, você pode dizer, como isso é útil? Como você realmente usa uma redefinição na prática? E é aí que fica interessante porque não há uma resposta única para essa pergunta. Lembra quando eu lhe disse que uma redefinição tem muitos casos de uso? De fato, dependendo do seu caso de uso, convém usar uma redefinição com opções diferentes. Vejamos alguns exemplos práticos. Um exemplo de redefinição Vou precisar fazer algumas alterações no repositório antes Eu mostro a você como a redefinição funciona. Desta vez, vou desenhar o repositório um pouco maior do que as outras duas áreas, porque vamos nos concentrar no que está acontecendo lá em primeiro lugar; então, precisamos de espaço na imagem. Ok, estamos no ramo mestre no status limpo, portanto as três áreas contêm os mesmos dados. Vou desenhar o arquivo menu.txt e o diretório de receitas em cada área. No entanto, quando se trata do repositório, deixe-me desenhar mais alguns detalhes. Aqui está o commit atual. Ele contém os dados e também tem o commit anterior como pai, e também desenharei o branch atual, master, que está apontando para o commit atual. E, finalmente, aqui está a referência principal, apontando para o ramo atual. Agora, agora temos uma visão completa do repositório. Não está realmente completo, como de costume, estou simplificando aqui. Existem arquivos adicionais no diretório de receitas que eu não desenhei. Não desenhei o diretório do projeto raiz e assim por diante. Se eu tentar espremer tudo nesse diagrama, ele ficará bastante confuso, mas esse é o material essencial que precisamos para seguir esta explicação redefinida. Ok, então esta é a nossa situação inicial. Agora imagine que ouvi falar desse chef da moda na Dinamarca, que está tendo muito sucesso com seu novo prato de assinatura, lulas com geléia de morango. Quero adicionar este prato ao menu, por isso vou adicioná-lo ao menu. Lá, morangos, legal. Então, olhe para o diagrama. Acabei de alterar o arquivo de menu na área de trabalho. Fiz laranja para mostrar que mudou. Agora vamos prepará-lo e apenas copiei o arquivo atualizado para o índice. E agora vamos cometer. Veja o que acontece no repositório quando eu confirmar. Git cria um novo commit. Ele move a ramificação principal para parte desse novo commit, e o novo commit contém a nova versão do arquivo de menu e a mesma versão do diretório de receitas. Vamos passar por todo esse processo mais uma vez. No meu projeto de livro de receitas, sempre que adiciono uma nova receita, também devo ter um arquivo na pasta de receitas que contém os ingredientes e os outros detalhes sobre a receita, então deixe-me adicionar um arquivo com alguns ingredientes. Agora, esse arquivo está aqui na área de trabalho. Como não estou desenhando todos os arquivos, simplifico para mostrar a alteração na pasta de receitas. Eu também fiz laranja. E eu o adiciono ao índice e o comprometo. E, novamente, o Git cria uma nova confirmação, atualiza a ramificação principal e a nova confirmação faz referência à versão mais recente do menu.txt e do diretório de receitas. Agora imagine que recebo uma ligação de um amigo em quem confio que realmente tentou cozinhar lulas com geléia de morango, e ela garante que elas são terríveis. Ela definitivamente não os recomenda. Aparentemente, ótimos chefs às vezes podem exagerar um pouco na experimentação e, bem, para encurtar a história, mudei de idéia; Não quero mais lulas de morango no meu livro de receitas. Infelizmente, até agora não tenho um, mas dois commits que fazem referência a lulas de morango. Lamento como esses commits. Eu gostaria que eles fossem embora agora. Como eu posso fazer isso? Bem, uma maneira de fazer isso é usando uma redefinição. Eu tive um commit aqui, dois commits atrás, este. Ele ainda contém as versões amarelas dos meus arquivos antes de começar a fazer as lulas. Gostaria que esses fossem os mais recentes commit no master, como se os dois últimos commits nunca tivessem acontecido. É aí que uma redefinição se torna útil. Posso anotar o hash desse commit e pedir ao Git para redefinir o ramo atual. Isso moveria o ramo de volta no tempo, por assim dizer. Mas e as opções para redefinir? Deve ser uma redefinição física, uma redefinição suave ou uma redefinição mista? Bem, após a redefinição, quero ter as versões amarelas dos arquivos, não apenas no repositório, mas também no índice e na área de trabalho. Quero estar alinhado, quero estar limpo novamente; portanto, essa deve ser uma redefinição definitiva. Lembre-se --hard copia os arquivos do repositório para as outras duas áreas. Vamos fazer isso e ver o que acontece. Primeiro, o Git move o ramo atual para o commit anterior, que agora se torna o novo commit atual. E a cabeça segue adiante. Segundo, como essa é uma redefinição definitiva, o Git copia o conteúdo da nova confirmação atual, as versões amarelas dos arquivos, para o diretório ativo e o índice. E terceiro, como os dois commits relacionados ao squid estão inacessíveis agora, eles não têm nenhum ramo apontando para eles. Eles serão coletados como lixo. Então, estamos de volta aonde estávamos antes de todo esse fiasco de lulas de morango, tudo com uma rápida reposição. Mais exemplos de redefinição Então, acabamos de ver uma maneira de usar uma redefinição para mover uma ramificação, um caso de uso específico. O caso de uso é que eu quero reverter todo o projeto para o estado em que estava no commit anterior. Fizemos isso com uma reinicialização total. Mas essa não é de forma alguma a única razão para usar uma redefinição. Se eu tivesse que dar exemplos por todos os motivos em que posso pensar, teria que falar sobre redefinição para um módulo inteiro; e tenho quase certeza de que existem mais alguns motivos em que não consigo pensar, então vou mantê-lo breve. Vou apenas dar alguns exemplos muito rápidos. Então, começando com um status limpo, vamos nos concentrar no arquivo de menu. Suponha que eu estivesse experimentando esse arquivo. Por exemplo, eu o edito e adiciono uma nova linha para uma receita de churrasco. Vou encenar esta edição, aqui estamos. Eu mudo o arquivo na área de trabalho, e isso também mudou no índice. Agora, e se eu mudar de idéia agora e quiser limpar o arquivo do palco? Talvez eu queira confirmar outra coisa primeiro e voltar a essas alterações mais tarde. Em outras palavras, quero manter as alterações na minha área de trabalho, mas quero remover todas as alterações do índice. No índice, quero a mesma versão dos arquivos que estão no repositório. Como faço isso? Anteriormente, vimos uma maneira de fazer isso, usando o comando rm --cash. Isso funciona, mas não é a única maneira de fazê-lo. De fato, se você ler atentamente as mensagens do Git, quando perguntei o status, o próprio Git sugeriu um comando diferente para o nosso arquivo de teste. Sugeriu o uso de redefinição. Como exatamente? Bem, a ideia é redefinir o que chamo de cabeça. Isso significa que estamos movendo a ramificação atual para a consolidação apontada pelo cabeçalho, mas a ramificação atual já está apontando para a consolidação por definição, portanto, nesse caso, a redefinição apenas move a ramificação. Se desejar, ele o move para o mesmo local onde já está, para que não o mova. É como pular completamente o primeiro passo na redefinição, o passo em que ele move o ramo. O que acontece depois disso? Lembre-se do segundo passo da redefinição. O Git move os dados do repositório para a área de trabalho no índice. Nesse caso, não especificamos o tipo de redefinição que queremos, portanto o Git seguirá o padrão; e o padrão é uma redefinição mista. Lembre-se do que uma redefinição mista faz; move dados da confirmação atual para o índice, mas não para a área de trabalho. E o resultado é desestabilizar todas as alterações. Você também pode desestabilizar um único arquivo nomeando o arquivo que foi redefinido. Agora, e se eu decidir jogar tudo fora? Não quero mais cometer essa alteração. Afinal, um churrasco não é para a sua receita. Eu deveria ter receitas separadas para todas as coisas diferentes em um churrasco. Então, por enquanto, eu só quero excluir minhas alterações e voltar ao status de limpeza. Eu posso fazer isso com um reset de cabeça dura. Uma redefinição do cabeçote rígido não move a ramificação porque ainda é uma redefinição do cabeçote, mas copia os dados do repositório para o índice e a área de trabalho. Substitui tudo o que há lá. Este é um comando popular, mas também um comando que eu uso com cuidado, porque é um comando destrutivo. De fato, é uma das maneiras mais fáceis de perder dados no Git. Você está dizendo explicitamente: eu não ligo para todas as coisas na minha área de trabalho. Por favor, apenas defina-o como o commit atual. E se eu enviar, tudo na área de trabalho e o índice serão substituídos. Eu poderia continuar com exemplos de como redefinir, redefinições suaves e duras, redefinições de cabeça, mas você já recebeu a mensagem agora. Existem muitos tipos diferentes de redefinições que você pode usar em diversas circunstâncias. O que você precisa lembrar são os dois passos. Primeiro, mova um ramo, possivelmente movendo-o no lugar, mas em geral movendo-o para um commit específico. Segundo, copie os dados da confirmação atual para o índice, a área de trabalho, ambos ou nenhum, dependendo do tipo de redefinição que você está fazendo. E como não quero renomear este curso Mastering Git Reset, acho melhor parar por aqui. As quatro áreas: mais ferramentas The Stash Olá novamente de Paolo. Neste último módulo sobre as quatro áreas, veremos algumas ferramentas que não encontraram espaço nos módulos anteriores. Vamos fazer alguns comentários sobre conflitos de mesclagem, trabalhando com arquivos individuais e a filosofia geral do Git. Mas primeiro, finalmente vamos falar sobre a quarta área, o estoque. No início deste treinamento, eu disse a você que existem quatro áreas em que o Git armazena seus dados, mas me concentrei apenas em três delas. Eu ignorei o estoque completamente. Isso ocorre porque os comandos que vimos até agora têm algum tipo de efeito na área de trabalho, no índice e no repositório, mas nenhum deles afeta o stash. De fato, existe apenas um comando que afeta o stash e, como você o nomeou, git stash. Então, se você quer que algo aconteça no estoque, você precisa ser muito explícito. E este é realmente o ponto de venda do estoque. É todo seu, ainda mais do que a área de trabalho. Os dados no stash não mudam, a menos que você realmente queira. Vamos ver como você pode usar o esconderijo. Eu começaria do status claro, como de costume. E por enquanto vou desenhar apenas o arquivo de texto do menu aqui. Agora, vamos criar a nova receita no diretório de receitas, guacamole. Esse arquivo deve conter uma receita ou pelo menos uma lista de ingredientes, mas pensarei nisso mais tarde. Por enquanto, estou deixando em branco e encenando. Neste projeto, sempre que adiciono uma receita, também devo adicioná-la ao arquivo de menu. Deixe-me fazer isso. Lá. assim agora temos um menu atualizado na área de trabalho e o novo arquivo na área de trabalho e no índice. Agora, imagine que enquanto estou trabalhando nesta nova receita, sou interrompido por qualquer motivo. Eu preciso fazer algum trabalho em outro ramo, por exemplo. Quero me concentrar nesse outro trabalho, mas não quero que minha receita de guacamole meio cozida atrapalhe, então é um bom momento para usar o estoque. Posso guardar todas as minhas alterações no estoque e elas permanecerão lá com segurança até que eu decida voltar à receita de guacamole. Eu armazeno o status atual com o git stash save, ou apenas o git stash, geralmente uso o formulário abreviado. E eu também uso essa opção, include-untracked. Isso significa também esconder arquivos que ainda não são rastreados. Não faz diferença neste caso. Não temos arquivos não rastreados. Mas, por padrão, o git stash apenas ignora arquivos não rastreados. Eu, pessoalmente, não gosto muito do padrão, então uso essa opção sem sequer pensar nela normalmente. E aqui está o que acontece. O Git pega todos os dados da área de trabalho e o índice que não está no commit atual no repositório e prepara todos esses dados no stash. E então, também verifica o commit atual. Então, agora estamos alinhados com o commit atual. Estamos no estado limpo novamente. Nossos arquivos foram retirados da área de trabalho e do índice. Eles realmente não se foram, é claro, ainda estão no esconderijo. Eu posso ler o conteúdo do stash com a lista de stash. Ali está, todo o meu trabalho semi-acabado, cuidadosamente embalado em um único elemento oculto. Agora podemos ver o ponto do esconderijo. O estoque é como uma prancheta para o seu projeto. É o lugar onde você armazena coisas que precisa reservar por um tempo. E é uma área de transferência múltipla. Você pode ter quantos elementos quiser. Cada elemento é rotulado com informações sobre a confirmação mais recente para facilitar a identificação e também obtém um ID de série. No momento, só há um elemento, por isso é chamado stash @ 0. O próximo seria stash @ 1. Agora que minhas coisas estão escondidas, eu poderia mudar para o outro ramo e fazer coisas nele, criar novas confirmações, qualquer coisa. Meu trabalho pela metade em guacamole permanece no esconderijo. E depois de fazer todo esse trabalho, posso recuperar as coisas que escondi. Você usa o stash apply para mover dados do stash para a área de trabalho no índice. E você pode dar o nome de um elemento stash. Como não vou dar um nome, ele se aplica ao elemento mais recente por padrão, que é stash @ 0. E aí estamos, todos os nossos dados na área de trabalho e o índice está de volta onde estava quando eu o escondi. Eu posso terminar o trabalho. Eu posso preparar o arquivo do menu, adicionar alguns ingredientes à receita do guacamole. Aqui, boas coisas vegetarianas. Mmm, ficando com fome. E vamos preparar essas atualizações também e confirmar a coisa toda. Agora, agora que nossos dados estão seguros no repositório, não preciso mais desses dados no stash. Vamos limpar todo o estoque. E aí está, esse foi o fluxo de trabalho básico relacionado ao armazenamento em cache. Há mais informações sobre o estoque, se você quiser ir mais fundo, mas o que você acabou de ver é o essencial, uma área de transferência para o seu projeto. Resolução de conflitos Outra coisa que devo mencionar neste treinamento são conflitos de mesclagem. Eu sempre falo sobre fusões como se nada pudesse dar errado ao fazer uma mesclagem, mas na vida real você pode ter conflitos, e mesmo que eu os chamei de conflitos de mesclagem, eles também podem ocorrer durante outras operações, como rebotes. Agora, você provavelmente já sabe como resolver conflitos básicos de mesclagem e, nesse caso, pode pular este clipe, se desejar. Mas, para ser completo, quero ter um exemplo muito rápido de um conflito aqui, porque este tópico também tem algumas conexões com as quatro áreas. Então, vamos editar a receita de guacamole a partir de dois ramos separados e depois tentar mesclá-los. Primeiro, vou criar um novo ramo chamado tomate e verificá-lo. E neste ramo, vou adicionar um pouco de tomate à receita. Guacamole é melhor com tomate, desde que você não exagere. E vamos cometer. Lá. Agora, vou voltar ao ramo principal e também modificarei a receita. Neste ramo vou adicionar cebola. Essa também é uma boa maneira de torná-la mais saborosa. E eu comprometo. OK. Agora, vamos tentar mesclar o ramo de tomate em nosso ramo mestre atual. E assim que fazemos isso, temos um conflito. E, de fato, se eu solicitar o status, ele me informa que há um caminho não mesclado, um arquivo em que os dois lados da mesclagem o modificaram. O Git é muito bom em resolver esse tipo de problema, mas, neste caso, os dois lados adicionaram a linha no final do arquivo, portanto, precisamos alterá-la na mesma posição. O Git não pode decidir qual dessas mudanças ocorre primeiro ou se uma dessas alterações deve substituir a outra. Então, ele mudou o arquivo na área de trabalho para nos fornecer informações suficientes para resolver o conflito. Essa é a primeira maneira pela qual um conflito de mesclagem afeta as quatro áreas. Isso resulta em um arquivo modificado na área de trabalho. Antes de analisar esse arquivo com mais detalhes, apenas uma pergunta. Como o comando git status sabe que estamos no meio de uma mesclagem e que há um conflito nesse arquivo? Bem, ele sabe porque o comando mesclar criou alguns arquivos no diretório .git aqui que sinalizam que há uma operação de mesclagem em andamento. E eles também contêm informações sobre o que está sendo mesclado entre outras coisas. Portanto, por exemplo, o arquivo MERGE\_HEAD é uma referência como HEAD, apenas MERGE\_HEAD é uma referência temporária. Ele dura apenas o tempo necessário para concluir a mesclagem. E está apontando para a ponta do galho que estamos fundindo, o tomate comprometido. Normalmente, nem vemos essas coisas porque o Git conclui a mesclagem por si só, mas agora estamos no meio de uma mesclagem e podemos cancelar a mesclagem inteira ou corrigir os conflitos e seguir em frente. Alguns conflitos podem ficar bastante complicados, mas esse não é um deles. Temos uma situação fácil aqui. Se editarmos esse arquivo, podemos ver que o Git está mostrando os dois lados da mesclagem. Marcou-os claramente para nós, para que possamos identificá-los. O lado marcado como HEAD é o commit atual e o lado marcado como Tomate é o ramo do tomate. Agora, podemos editar manualmente esse arquivo e decidir como deve ser a aparência do arquivo mesclado. Aqui, vou manter o tomate e a cebola e colocar o tomate por cima. Agora que resolvemos o conflito, precisamos contar ao Git porque ele ainda não sabe que o solucionamos e, de fato, o status ainda vê um conflito aqui. E essa é outra conexão com as quatro áreas. Preciso usar o índice para dizer ao Git que resolvi o conflito. Sabemos que o índice informa ao Git quais informações serão inseridas no próximo commit, e isso ainda é verdade aqui, mas nesse caso o significado do comando add é um pouco mais específico. Isso significa que este arquivo está pronto para ser confirmado porque o editei e resolvi os conflitos que ele continha. Portanto, agora podemos finalmente avançar e concluir a mesclagem. Tudo o que precisamos fazer é dizer ao Git para cometer. E aqui estamos, a mesclagem está concluída, está no log. E é assim que você resolve conflitos de mesclagem manualmente. Novamente, você provavelmente já sabia tudo isso, certo? Mas é interessante no contexto das quatro áreas, porque mostra que os mesmos comandos do Git, como adicionar ou confirmar, significam coisas sutilmente diferentes, dependendo do contexto. Lembre-se deste ponto, porque mencionaremos novamente em breve. Mas primeiro, há um último recall que quero mencionar. Trabalhando com caminhos Passei muito tempo neste treinamento falando sobre confirmações. E, de fato, geralmente trabalhamos com arquivos apenas até o ponto em que os colocamos em uma confirmação e, a partir de então, tendemos a trabalhar com confirmações o tempo todo. Quando você redefine, redefine para uma confirmação. Quando você faz o check-out, faz o check-out de um commit e assim por diante. Mas confirmações são bastante grosseiras. Lembre-se, uma confirmação é como uma captura instantânea de todo o seu projeto em algum momento. Às vezes, você precisa trabalhar com algo menor que todo o seu projeto. Você precisa trabalhar com um único arquivo ou um único diretório. Vejamos um exemplo envolvendo o arquivo de menu e o arquivo README. Editarei o menu e tornarei tudo em maiúsculas com um pouco de magia do vim. E também farei uma alteração semelhante ao arquivo README. E agora deixe-me organizar todas essas mudanças. Agora, digamos que não tenho mais certeza dessas mudanças. Digamos que eu goste da alteração no README, mas ainda não tenho certeza sobre a alteração no menu, portanto, quero desagrupar o arquivo do menu, mas não o arquivo README. No módulo anterior, vimos uma maneira de desagrupar um arquivo. Você pode redefinir a cabeça, mas da última vez fizemos em todo o conteúdo do índice. Se você ler a mensagem de status com atenção, verá que também pode fazer isso em um único arquivo. Então, vamos fazê-lo. Não especificamos que essa redefinição de cabeçalho seja rígida; por padrão, é uma redefinição de cabeçalho mista. E lembre-se do que uma redefinição de cabeçalho misto faz. Ele não move a ramificação atual porque é uma redefinição de cabeçalho, mas copia dados da confirmação atual para o índice. Não muda a área de trabalho. Normalmente, essa redefinição copia todos os dados da confirmação atual para o índice, mas, neste caso, estamos dizendo que queremos redefinir apenas um arquivo, o arquivo de menu. Portanto, apenas o arquivo de menu é copiado. E, de fato, é que o arquivo de menu, e somente o arquivo de menu, não foi testado. O README ainda está preparado. Brilhante. Exatamente o que queríamos, certo? Agora, vamos dar um passo adiante e também remover as alterações de menu da área de trabalho. Você pode pensar que, para fazer isso, poderíamos usar a mesma instrução, só que desta vez a reinicialização do cabeçalho será difícil. Mas não, isso não funciona. O Git se recusa a redefinir um cabeçalho rígido com um caminho. Isso parece um pouco inconsistente, mas é assim. Em vez disso, a maneira mais comum de reverter um único arquivo ou diretório em uma área de trabalho sem tocar em outros arquivos é usar o checkout. Olhe para este checkout estranho aqui. Normalmente, o checkout move o Referência HEAD no repositório, geralmente para uma ramificação e, em seguida, copia todos os arquivos do repositório para a área de trabalho e o índice. Nesse caso, nosso checkout não moverá HEAD. O checkout copiará apenas as coisas do commit atual no repositório para a área de trabalho e o índice. E isso será feito apenas para o arquivo que especificamos. Então, perdemos todas as alterações nesse arquivo. O menu volta ao local onde estava na confirmação mais recente, exatamente o que queríamos. Observe que o check-out com um arquivo ou caminho em geral não é tão inofensivo quanto um check-out regular. Na verdade, é uma das operações mais destrutivas que você pode fazer no Git. Nem recebemos um aviso. Nós destruímos todas as nossas alterações no arquivo de menu de maneira rápida e irrecuperável. Portanto, lide com esse commit com algum cuidado. Portanto, agora o único arquivo que me resta, modificado e preparado, é o README. E eu posso finalmente cometer. E esse foi mais um exemplo da flexibilidade no Git. Usamos os mesmos comandos que já usamos nas redefinições e caixas anteriores, apenas com opções diferentes e obtivemos resultados sutilmente diferentes na melhor das hipóteses, e às vezes resultados muito diferentes. E como essa não é a primeira vez que falo sobre a flexibilidade do Git, vamos discutir esse ponto de uma maneira mais geral. Git é uma caixa de ferramentas Considere uma caixa de ferramentas, apenas uma caixa de ferramentas comum, uma chave de fenda, martelos, esse tipo de coisa. As ferramentas são especializadas de certa forma, certo? Cada ferramenta possui recursos específicos, mas ao mesmo tempo você pode executar muitos trabalhos diferentes com cada ferramenta. Você pode usar um martelo para plantar uma unha ou remover uma unha ou endireitar um pedaço de metal dobrado, por exemplo. E você também pode usar ferramentas diferentes para realizar trabalhos semelhantes, se for criativo. Se você precisar remover um prego, faça isso com um martelo ou use um alicate, por exemplo. Então, você tem um monte de ferramentas e muita flexibilidade na maneira de usá-las. E essa é a metáfora que tenho para você aqui. Git é uma caixa de ferramentas. Não é um programa único que faz coisas. Nem mesmo uma coleção de utilitários corresponde literalmente aos seus casos de uso. É uma caixa de ferramentas porque você possui ferramentas genéricas, como redefinir ou efetuar checkout, e, como em uma caixa de ferramentas, pode usar a mesma ferramenta para trabalhos diferentes. Pense em git reset, por exemplo. Você pode usá-lo para desestabilizar um arquivo. Você pode usá-lo para limpar seu diretório de trabalho. Você pode usá-lo para remover as principais confirmações do seu histórico e assim por diante. Ou git add. Você pode usar o git add para informar ao Git sobre um novo arquivo, colocar um arquivo modificado no próximo commit ou sinalizar que você resolveu os conflitos em um arquivo durante uma mesclagem. E você também pode usar ferramentas diferentes para trabalhos semelhantes. Se você deseja desestabilizar um arquivo, o Git não fornece um comando com o nome, eu não sei, de estágio. Não existe esse comando. Para desagrupar um arquivo, você precisa entender qual é o primeiro índice e, depois disso, desagrupar o arquivo com a redefinição ou usar remove --cached, por exemplo. Usamos os dois comandos para desestabilizar um arquivo no histórico. Git não diz, aqui está este comando. É o comando que você deve usar para desestabilizar um arquivo. O Git fornece o conjunto de ferramentas básico e, em seguida, cabe a você escolher a ferramenta certa. Essa é uma abordagem de folga muito exclusiva, o que não é surpreendente, considerando que o Git foi concebido por Nils Torvalds. E essa abordagem torna o Git um pouco mais desafiador do que outros sistemas de controle de versão, é verdade. Mas também o torna muito, muito poderoso. E isso encerra a primeira parte deste treinamento. Nesses quatro primeiros módulos, exploramos essa caixa de ferramentas do Git e, em particular, nos concentramos em ferramentas que movem dados pelas quatro áreas do Git. Algumas dessas ferramentas também têm outros efeitos no repositório, em particular. Pense em comandos como confirmar ou redefinir a si próprio. Também descrevemos esses efeitos, mas nosso foco estava na movimentação de dados pelas áreas. Agora que você conhece essas ferramentas de embaralhamento de dados, podemos mudar para outro conjunto de ferramentas, que se concentra quase exclusivamente no repositório. História: Explorando o Passado Tornando-se um cirurgião de história Olá novamente de Paolo Perrotta. Este é o primeiro de dois módulos sobre como trabalhar com o histórico do seu projeto. Este módulo, Explorando o Passado, trata de explorar a história e dar sentido a ela. E no próximo módulo, falaremos sobre como mudar o histórico. Lembro-me do treinamento que assisti uma vez sobre o Vim, o editor de texto e o professor disse que usar o Vim parece ser um cirurgião de texto. Eu acho que é uma boa metáfora se você gosta ou não de Vim em particular. Algumas tecnologias são tão precisas e exatas que, quando você as utiliza, parecem um pouco como fazer uma cirurgia. Você faz exatamente o que deseja nos dados que deseja, e usa ferramentas um pouco assustadoras, talvez, elas cortam com facilidade, mas também são muito poderosas. O Git também se sente assim, especialmente quando você está trabalhando com história. Portanto, este módulo e o próximo são sobre se tornar um cirurgião de história, se você desejar. E como o seu histórico faz parte do seu repositório, pararemos de falar sobre as outras áreas de armazenamento e focaremos apenas no repositório. Apenas uma observação sobre essa parte do treinamento, prometo que não me demorarei nos detalhes técnicos desse treinamento, que me concentraria no modelo e na maneira de pensar no Git, em vez de nos comandos, nas opções e iguais. Bem, nesses dois módulos, vou relaxar um pouco essa abordagem. Trabalhar com história é principalmente uma coisa técnica, então vou me concentrar um pouco mais nos comandos aqui. Vamos mergulhar. Uma confirmação por qualquer outro nome Quando falo em explorar o histórico do seu projeto, falo principalmente de confirmações, portanto, nos referiremos a muitas confirmações nesses dois módulos, e há muitas maneiras de se referir a uma confirmação no Git. Vamos levar apenas alguns minutos para conversar sobre isso. Aqui está o nosso repositório de livros de receitas novamente. Vou fazer o checkout de um dos ramos existentes, esse ramo aqui chamado nogood, é um dos ramos do treinamento anterior. Não importa o que contém exatamente. Estou apenas procurando por um monte de commits antigos aqui. Posso usar o git log para examinar o histórico, mas o problema é que o log git padrão não é muito útil quando você está tentando entender um histórico complexo, porque esmaga tudo em uma única lista. Portanto, é difícil entender as ramificações e as fusões e entender o que realmente acontece. Então, vou usar o git log com algumas opções. A opção de gráfico fornece uma boa estrutura semelhante a um gráfico, na qual eu posso ver como a consolidação e ramificação são confirmadas. E a opção de decoração mostra as referências posicionais, como ramos da cabeça. E, finalmente, vou formatar o log para que cada confirmação tenha apenas uma linha. Lá, linda. Podemos ver a estrutura do repositório agora, todas as referências, incluindo caminho, ramificações, ramificações remotas, os trabalhos. Somente este comando pode quase substituir uma ferramenta GUI para a maioria das minhas inits. Agora, deixe-me copiar essas informações para um diagrama aqui, para que permaneçam visíveis enquanto eu continuo trabalhando na linha de comando. Ok, digamos que eu quero ver informações sobre o commit atual. Quero saber quais alterações foram introduzidas por esse commit, a data do commit e assim por diante. Eu posso usar o comando git show para fazer isso. Claro, eu preciso dizer para qual commit eu quero olhar. Como posso me referir a este commit? A maneira óbvia é usar seu hash, como fizemos no passado, ou melhor, os primeiros caracteres do hash, assim. Aí está, informações detalhadas sobre esse commit. No entanto, o uso de hashes nem sempre é a maneira mais fácil ou prática de se referir a uma confirmação. Outra maneira de fazer isso é dar ao Git o nome de uma referência que está apontando para o commit. O ramo nogood, por exemplo, está apontando para este commit, para que eu possa usá-lo para me referir ao commit. Além disso, HEAD está apontando para o ramo que está apontando para o commit. Este é o commit atual no momento, então eu também posso me referir a ele através do HEAD. Até aí tudo bem, então nada de novo, certo? Já fizemos todas essas coisas neste treinamento. Mas agora, e se quisermos referenciar esse outro commit, o segundo último? Não há referência apontando para ele, então parece que nossa única opção é usar seu hash. No entanto, existem outras maneiras. Posso começar em HEAD e adicionar um sinal de intercalação, assim. O sinal de intercalação significa o pai confirmar, então agora estou pedindo o pai confirmar. Bingo. E se eu quiser me referir a esse commit aqui, posso usar dois pontos de intercalação. Isso significa ir para o pai do pai da cabeça. E também posso dizer exatamente a mesma coisa usando um sinal de til seguido de um número. Você pode lê-los como ir para a cabeça e depois voltar para confirmações. Isso é útil, especialmente se você quiser dizer, digamos, o décimo commit antes da cabeça e não quiser digitar 10 pontos de intercalação. Você pode apenas dizer o til 10. Essa sintaxe é boa se cada confirmação tiver exatamente um pai, mas será interrompida assim que você confirmar com vários pais, como esse commit de mesclagem aqui. E se eu quiser me referir a este commit, por exemplo? É o segundo pai do segundo commit antes do head. Nesse caso, eu posso direcioná-lo com esta sintaxe. Vamos ver. Comece da cabeça, volte duas confirmações e escolha o segundo pai. Voila. Existem outras maneiras ainda mais sofisticadas de se referir a confirmações. Alguns deles são surpreendentemente sofisticados. Apenas para um exemplo rápido, pedi ao Git para me mostrar onde estava a cabeça há um mês. Isso é legal, né? No entanto, não vou entrar em mais detalhes, não precisamos deles. Quando se trata de referenciar commits, acho que temos informações suficientes para sobreviver. Vamos continuar. História Forense Agora que você sabe como se referir a um commit, vamos finalmente falar sobre como você pode explorar como esses commits estão conectados, seu histórico. Existem alguns comandos úteis para isso, um deles é a culpa do git. Isso mostra de onde as linhas do arquivo são provenientes. Vamos ver quem mudou o arquivo apple\_pie e quando. Aqui estão todas as linhas no arquivo Para cada linha, você pode ver a confirmação mais recente em que essa linha foi alterada. O sinal de intercalação aqui significa que este é o primeiro commit que adicionou esta linha ao repositório. Portanto, essas linhas estão no arquivo desde a primeira vez que o arquivo foi adicionado ao projeto e todas as outras linhas foram alteradas ou adicionadas em algumas confirmações secretas. E aqui eu posso ver a data e os autores dessas mudanças. O autor sempre foi eu, mesmo que eu aparentemente tenha mudado meu nome na minha configuração do Git depois de criar o arquivo. Outro comando útil é o git diff. Já usamos o git diff neste treinamento para comparar o conteúdo de duas áreas, como o repositório e o índice, mas você também pode usá-lo para comparar outras coisas. Por exemplo, dois commits. Vamos ver as diferenças entre o commit atual e os dois commit anteriores. Ok, esse é o único arquivo que foi alterado entre os dois commits. Uma linha foi adicionada e outra linha foi modificada. E, é claro, você pode usar ramificações para referenciar confirmações, portanto, uma técnica comum é usar git diff para comparar duas ramificações. Isso mostra todas as alterações entre os dois ramos. Isso é muito útil. Comparar ramificações é realmente útil, especialmente antes de mesclar coisas. Vimos culpa do git e diff do git; no entanto, o comando mais importante de longe quando você está explorando o histórico é um comando que você já conhece, git log. Navegando no log O log do Git é o comando mais útil para explorar o histórico de um projeto. Já vimos algumas opções: gráfico, decoração e on-line. Mas há muito mais opções para o git log, uma enorme quantidade de opções de fato. O log do Git é sem dúvida o comando Git mais complicado. É super poderoso. Vou dar alguns exemplos. Portanto, você não deve memorizar esses exemplos; estou apenas dando uma demonstração rápida de algumas coisas que o git log pode fazer. Para começar, você pode obter o diff de detalhes de cada confirmação no log, assim. Assim, você pode ver exatamente quais alterações foram introduzidas no commit. Também é muito colorido. Mais uma opção, você pode filtrar as confirmações. Por exemplo, eu só quero ver os commits que contêm as maçãs de seqüência de caracteres em suas mensagens, e há apenas um deles. Você pode até solicitar todas as confirmações que adicionaram ou removeram a palavra maçãs de qualquer arquivo com -G maiúsculas. Também usarei a opção de patch aqui para mostrar quais linhas foram impactadas exatamente por essas alterações. Há, um, dois, três confirmações que adicionaram ou excluíram esta palavra. E também posso ver as alterações específicas e, em todos os casos, elas incluem as maçãs das cordas. Como alternativa, também existe um comando git grep que é útil para esse tipo de pesquisa de texto em todo o histórico. Confira se desejar. O log do Git também pode visualizar um intervalo específico de confirmações. A maneira mais fácil de fazer isso é dizer, por exemplo, git log -N, o que significa mostrar apenas as últimas confirmações de N no log. Você também pode usar dois pontos para expressar um intervalo como este. Isso significa que mostra as confirmações de cinco confirmações antes de ir para o pai da cabeça. Eu costumava achar essa sintaxe de intervalo levemente confusa porque você especifica a confirmação mais antiga primeiro aqui, mas na saída padrão do log git, as confirmações são revertidas, portanto, você vê a confirmação mais antiga na parte inferior. Mas, eventualmente, eu me acostumei. Esse recurso de organização de confirmação é particularmente útil se você precisar comparar duas ramificações. Este é um cenário comum. Isso não é como um diff. É um tipo diferente de comparação. Um diff pode comparar os arquivos em dois ramos, aqui queremos comparar o histórico de dois ramos. É mais fácil entender com um exemplo. Por exemplo, o ramo atual é o ramo não bom. Digamos que eu queira listar os commits que estão no ramo principal, mas não no ramo bom. Então você pode ler isso como ir de nogood a master e me mostrar todos os novos commits que você vê, e aqui estão eles. Se mesclarmos o mestre agora, esses são os commits que obteríamos. Essa comparação de ramificações é realmente útil, eu faço isso o tempo todo. Ok, acho que você já entendeu a ideia. O log do Git tem uma enorme quantidade de opções para mostrar seu histórico de várias maneiras diferentes para filtrar confirmações, formatá-las e assim por diante. Novamente, eu definitivamente não espero que você se lembre desses recursos apenas porque passei alguns segundos apresentando-os. Ainda assim, não quero gastar mais tempo analisando todas essas opções. No final, você provavelmente me odiaria. O objetivo deste módulo é que, se você souber que esses recursos existem, sempre que precisar deles, eles estarão a uma rápida pesquisa do Google. Então, vamos recapitular. Neste módulo, vimos algumas maneiras de nos referir a commits com sintaxes especiais, como o sinal de intercalação e o til, e vimos alguns exemplos de comandos que você pode usar para explorar histórico, git diff, git blame e log git. Aqui nesta lista, mostrei exemplos de alguns dos argumentos que estamos usando. Ah, não vamos esquecer o Git Show. Foi isso. Este módulo foi rápido, essencialmente um aquecimento para o próximo módulo, que oferece coisas realmente interessantes, como mudar seu histórico. História: Reparando erros A regra de ouro Olá de Paolo. Este é o segundo módulo deste treinamento que fala sobre cirurgia de história. No módulo anterior, vimos alguns comandos para explorar o histórico do seu projeto. Neste, veremos como editar o histórico do seu projeto. Agora é claro que você já está editando o histórico do seu projeto de uma maneira. Toda vez que você confirma, por exemplo, você adiciona algo ao histórico. E toda vez que você usa um rebase, está mudando seu histórico. Neste módulo, vamos dar alguns passos além disso e examinar algumas ferramentas cirúrgicas avançadas. Todas essas ferramentas são realmente úteis em situações de emergência, como quando você deseja corrigir um erro. É por isso que chamo esse módulo de Reparando erros. No entanto, essas ferramentas também são úteis em circunstâncias menos dramáticas, como quando você está limpando as confirmações locais antes de enviá-las para um servidor remoto, por exemplo. Uma nota importante antes de começarmos. Se você assistiu ao meu treinamento Como o Git Works, e mesmo se não assistiu, provavelmente se lembra de uma regra geral sobre o git rebase. Você nunca deve refazer as confirmações compartilhadas. Ou seja, depois de enviar essa confirmação para um repositório compartilhado, a partir desse momento, você deve evitar rebasar essa confirmação. Isso ocorre porque um rebase é um comando que altera o histórico. Ele copia confirmações antigas para novas confirmações. As novas confirmações podem parecer iguais às antigas, mas na verdade são objetos diferentes no banco de dados. E, como resultado, se você refazer o commit de outras pessoas em seus repositórios, poderá criar muita confusão. Você pode introduzir conflitos que não podem ser corrigidos sem muitos ajustes manuais. Então, em geral, você não deve fazer isso. Bem, isso não se aplica apenas a rebase. A maioria das técnicas que veremos neste módulo muda o histórico, portanto, assim como a rebase, essas técnicas não devem ser usadas no histórico compartilhado. Essa regra é tão importante que eu chamo de regra de ouro do Git. Está tudo bem quando você altera seu próprio histórico local, mas alterar o histórico compartilhado não vai lhe render nenhum amigo em sua equipe. Portanto, neste módulo, veremos alguns truques emocionantes e poderosos, mas sempre lembre-se da regra de ouro, ok? E com isso fora do caminho, vamos abrir o pacote de truques. Alterando a confirmação mais recente A primeira maneira de mudar a história que eu vou lhe mostrar também é a mais fácil. Trata-se de corrigir o último commit. Estamos no projeto do livro de receitas novamente, no ramo principal, e vamos adicionar uma nova receita ao menu, Salada Caesar. Deixe-me encenar essa mudança e cometer. E aqui está a minha mensagem de confirmação. Lá, está no log agora. Agora, depois de me comprometer, percebo que ainda não terminei o trabalho. As regras do livro de receitas dizem que sempre que tenho uma receita no menu, também preciso do arquivo correspondente na pasta de receitas. Eu não tenho esse arquivo para Caesar Salad. No mínimo, preciso de um arquivo de espaço reservado rápido para isso. Vamos criar esse arquivo e preenchê-lo com alguns ingredientes. Apenas um espaço reservado por enquanto, apenas alguns ingredientes básicos. Aqui estamos. Provavelmente uma salada incompleta, mas ei, você realmente não pode dar errado com salada, certo? E eu encordoo. Então, agora estou corrigindo o problema, mas não quero corrigi-lo apenas criando mais um commit. Isso não pareceria certo. Eu terminaria com dois commits separados, onde o primeiro commit ainda teria um item de menu sem uma receita correspondente. Esse é um estado inconsistente para o meu livro de receitas. É o equivalente ao código que não compila no repositório. E o segundo commit resolveria isso, mas, em vez disso, gostaria que minha história fosse mais limpa do que isso. Gostaria de voltar e corrigir meu último commit e adicionar esse arquivo ao último, para que eu tivesse apenas um commit, e esse seria bom e limpo. Eu posso fazer isso alterando o commit. Basta um argumento de alteração adicional ao meu comando de confirmação. Lá. Não estou criando um novo commit do zero. Em vez disso, estou alterando a confirmação mais recente. E agora, esse commit incluirá o menu modificado e o arquivo de receita mais recente. E também posso aproveitar esta oportunidade para fazer outras alterações que deseje neste commit. Por exemplo, a mensagem aqui, deve estar no tempo presente. Esse é o estilo recomendado para as mensagens do Git, então deixe-me alterá-lo. Feito. E eu salvo e saio desse arquivo de mensagens. E vamos ver o que está acontecendo com o diagrama. Veja o commit atual. Essa é a confirmação que estamos alterando. O Git não pode realmente mudar esse último commit, certo? Commits são imutáveis. O que o Git está realmente fazendo quando eu termino de alterar esse commit é que ele copia o commit atual para outro commit que também inclui todas as minhas emendas, o novo arquivo e a mensagem editada. Este é um objeto novo em folha com um novo hash. Em seguida, o Git move a ramificação atual para apontar para o novo commit. E o commit antigo acabará sendo coletado no futuro enquanto o novo commit permanecer lá. Então, alterando um commit é uma operação que muda a história. É como uma pequena recuperação. Nesse caso, alterei um commit que ainda não havia compartilhado. Eu nunca o enviei para um repositório compartilhado, então não há problema em alterá-lo. E aqui estamos nós. Se olharmos para o log, o commit mudou. Esse novo commit possui a nova mensagem que eu corrigi e, se observarmos os detalhes, podemos ver que ele inclui o menu atualizado e o novo arquivo de receita. E isso é commit --amend. É bastante útil porque sempre que você deseja consertar algo, é mais frequente que haja algo no commit mais recente. Na verdade, eu teria vergonha de admitir quantas vezes altero meus commits. Então, bom recurso. Mas é útil apenas para o commit atual. Você não pode corrigir um commit anterior como esse. E se você quiser corrigir um problema que aconteceu há alguns commits atrás? Rebase interativo Acabei de mencionar esta convenção no meu projeto de livro de receitas que diz para cada linha do menu, como Torta de maçã aqui, que você deseja um arquivo correspondente no diretório de receitas, como apple\_pie.txt aqui. Bem, como se viu, eu fiz isso errado no passado, duas vezes. Existem dois itens no menu, cheesecake e frango tikka masala, que não possuem arquivos correspondentes na pasta de receitas. E se eu quiser corrigir esses erros do passado? Se os erros ocorreram no último commit, eu poderia corrigi-los. Mas esse não é o caso, infelizmente. Vamos usar a culpa do git para ver quando essas linhas foram adicionadas ao menu. E agora vamos olhar para o log. Então, aqui está o commit que adicionou frango tikka masala ao menu. Foi há alguns commits atrás. E aqui está o commit que adicionou o cheesecake. Este aconteceu no passado. Um dos primeiros commits. Ambos os commits violaram as regras. Eles adicionaram itens ao menu sem adicionar receitas correspondentes no diretório de receitas. Agora temos que ter um pouco de cuidado. Veja a posição das ramificações remotas aqui no log, em particular a ramificação principal remota de origem. Este repositório é compartilhado, certo? Eu o clonei no projeto de livro de receitas no GitHub. Aqui é onde está o ramo principal no GitHub, ou pelo menos onde foi a última vez que nos comunicamos com o projeto no GitHub. Portanto, esses dois commits estão errados, mas esse commit aconteceu após a última vez que eu me comuniquei com o GitHub, enquanto esse outro commit aconteceu muito antes disso. Já foi compartilhado. Então, eu não quero mudar o commit do cheesecake por causa da regra de ouro. Eu vou decidir viver com isso por enquanto. Em vez disso, vou corrigir o commit tikka masala, que ainda não é compartilhado. Ainda é um commit local no meu próprio repositório, para que eu possa alterá-lo. Então, para encurtar a história, aqui está o plano. Vou corrigir esse commit para que ele inclua um novo arquivo de receita que corresponda a esse item de menu, tikka masala. Vamos fazer isso em duas etapas. Primeiro, criarei esse arquivo de receita e o confirmo como um commit totalmente novo e, segundo, alterarei o histórico do projeto para que esse commit e o commit tikka masala antigo sejam agrupados em um único commit. A primeira parte é fácil. Vamos criar uma nova receita. Confessarei minha ignorância sobre os detalhes aqui. Não lembro de cor como fazer frango tikka masala, então vou colocar um espaço reservado aqui. Na verdade, eu cozinhei tikka masala no passado e acho que foi muito bom. Meus amigos indianos provavelmente discordariam disso, mas ei. Boa. Está feito, vamos encenar e confirmar. E essa foi a primeira parte. Agora, vamos finalmente chegar ao ponto importante. Preciso editar minha história e fazer uma cirurgia séria nela. Como eu posso fazer isso? Antes de mostrar, deixe-me abrir um espaço para um diagrama e também deixe-o um pouco maior e mais legível. Ok, é aqui que eu mostro um dos comandos mais poderosos do git. E, estranhamente, é apenas um sabor diferente de um comando que você já conhece, rebase. Esqueça o que você sabe sobre o rebase padrão, no entanto. Se você fizer o rebase interativo, ou simplesmente -i para abreviar, o rebase deixará de ser um rebase normal e se tornará uma super poderosa faca suíça de edição de história com esteróides. Este é um dos truques estranhos do vocabulário do Git. A palavra rebase não significa apenas rebase, ela evoluiu para significar histórico de alterações. E, neste caso, a maneira como estamos mudando a história não tem quase nada em comum com uma recuperação regular. Preciso de mais um argumento para que isso funcione, e esta é a referência a um commit. Eu posso usar o último commit que foi compartilhado aqui, aquele apontado pelo branch master de origem remota. Isso significa, deixe-me editar o histórico desse commit, excluído, em diante. Então, eu vou editar essas linhas da história aqui. Estamos prontos? Vamos fazer isso. E aqui estamos nós, no editor de texto, realizando uma rebase interativa. Temos uma lista de confirmações aqui. E não se confunda, a ordem dos commits é o oposto da ordem de log do menos recente para o mais recente. Aqui está o que é o rebase interativo. O que estamos fazendo aqui é essencialmente escrevendo um programa de computador. Aqui está o programa. O programa é executado nos commits atuais do histórico, e a saída do programa é um histórico totalmente novo. A primeira palavra em cada linha do programa é uma instrução que se aplica a um commit e diz ao git o que fazer com esse commit específico. No momento, cada linha é uma escolha. Isso significa, basta aceitar este commit. Portanto, se executássemos o programa como está agora, o Git apenas comporia o novo histórico escolhendo todos esses commits, um após o outro, o que significa que o novo histórico seria exatamente como o histórico que tenho agora. Nada mudaria. Mas queremos mudar essa história, então vamos mudar o programa. Por exemplo, antes mesmo de olhar para o commit tikka masala, veja este commit aqui. É uma confirmação válida, mas tem uma mensagem estranha que realmente não corresponde à convenção do Git. Não sei o que estava pensando quando escrevi isso. Eu quero mudar esta mensagem. E se você ler os comentários aqui embaixo, verá que a instrução para alterar a mensagem de um commit é reformulada. Então, deixe-me alterar esta instrução de escolha para reformular. Observe que ainda não estou reformulando a mensagem. Isso acontecerá mais tarde quando o programa for executado. No momento, estou escrevendo o programa aqui. O quê mais? Sim, a coisa do tikka masala, certo? Essa foi a razão pela qual fizemos uma rebase interativa em primeiro lugar. Primeiro, quero alterar a ordem desses commits. Vou cortar esse commit aqui, o mais recente, aquele em que adicionamos a receita do frango tikka masala e colá-lo logo após o commit mais antigo que adicionou o tikka masala ao menu. E também quero compactar esse commit e o commit anterior juntos, e torná-los um único commit. E enquanto estou aqui, deixe-me fazer algo ainda mais radical. Temos alguns commits sobre guacamole aqui. E você deve se lembrar que esses commits envolvem ramificações e mesclagens, e acho que isso é desnecessário. Eu gostaria de esmagar todas essas coisas de guacamole em um único commit. Portanto, temos duas instruções de squash seguidas. E é isso, o programa está concluído. Vamos sair e salvar. E nesse ponto, o git começa a executar este programa. Agora, farei algo mágico que só pode ser feito quando você estiver editando um screencast. Vou diminuir o tempo. Assim que o rebase interativo está começando, deixe-me congelar o terminal no local por um momento. Copiarei o programa de rebase interativo aqui. Vamos começar na primeira instrução. É uma escolha. Então o Git está apenas escolhendo esse commit. E então, a segunda instrução é uma abóbora, então o Git precisa esmagar esses dois commits juntos. E isso requer uma decisão minha. E, de fato, se você olhar para o terminal, o Git acabou de interromper o rebase interativo e está me perguntando: eu tenho dois commits separados e você deseja que eu os agrupe em um commit, mas qual deve ser a mensagem de commit desse novo commit ? Eu posso ver as duas mensagens aqui das confirmações originais, além de linhas de comando e linhas vazias que são ignoradas. Vou escolher apenas uma dessas mensagens como a nova mensagem. E essa outra linha vai embora. E assim que eu salvo, o Git esmaga os dois commits juntos, criando um commit totalmente novo. Ok, eu congelei o tempo novamente. A terceira instrução é uma escolha, então o Git pode escolher essa próxima confirmação e adicioná-la ao histórico. Na verdade, ele não pode literalmente adicionar esse commit ao histórico porque esse commit inclui um link para seu pai, e o git não pode mudar esse link sem alterar o commit inteiro, e os commit são imutáveis, portanto, mais uma vez, um commit totalmente novo, um cópia do commit existente. A propósito, este passo não requer nenhuma intervenção minha, então o Git apenas escolhe e passa para a próxima instrução. Em seguida, temos nossa reformulação, e o Git para novamente. Agora está me pedindo para alterar a mensagem de confirmação. É disso que se trata a reformulação. Ok, vamos fazer isso. Até agora, temos a sequência de instruções mais complicada até agora, uma escolha seguida de squash e, em seguida, outra squash. Isso significa que esses três commits devem ser compactados todos juntos. Isso não parece difícil. Parece ser o mesmo que esmagar dois commits juntos, mas desta vez temos um problema. Git para e reclama de um conflito. Que conflito é esse? Bem, se você se lembra do que aconteceu alguns módulos atrás, naquela época, tínhamos uma mistura da receita de guacamole que resultou em um conflito, e resolvemos o conflito manualmente. Agora, o que estamos fazendo é nos livrar dessa fusão, incluindo a resolução do conflito. Estamos compactando tudo em um único commit. Portanto, temos que rever o antigo conflito novamente e resolvê-lo novamente, desta vez por uma questão de rebase interativa em vez de mesclagem. E mais uma vez, vou abrir a receita de guacamole, colocar tomate e cebola na linha, para que o conflito e depois eu possa continuar a re-reformulação, como o Git está sugerindo aqui. Oh, desculpe. Esqueci de dizer ao Git que resolvi o conflito. E, assim como na mesclagem, posso fazer isso adicionando o arquivo ao índice. E agora eu continuo novamente. Lá. Agora resolvemos os conflitos e finalmente estamos esmagando esses três commits juntos éter. Escolhemos uma mensagem de confirmação e continuamos. E depois disso, temos a instrução 8, que é apenas uma escolha. E 9, que também é uma escolha. Portanto, o Git pode copiar esses commits para o novo histórico. E agora que esta última operação está concluída, o Git pode mover a ramificação atual para o novo histórico e deixar o histórico antigo para o coletor de lixo. E nós terminamos. O rebase interativo está concluído. Se eu olhar o registro agora, posso ver minha história totalmente nova. Nós exercemos muito poder aqui. Você pode fazer muitas coisas com rebotes interativos. Vimos reordenar confirmações, espremer confirmações, reformulá-las, mas você também pode remover confirmações apenas excluindo-as do programa e até dividir uma confirmação em várias confirmações menores. Os rebotes interativos são realmente poderosos e ainda fáceis de fazer, como acabamos de ver. Na verdade, eles são tão fáceis que eu os faço parte padrão do meu fluxo de trabalho. Aqui está o que eu faço. Quando estou trabalhando em um projeto, comprometo-me cedo e com frequência. Eu comprometo, comprometo, comprometo o tempo todo, a cada poucos minutos. E a maioria dos meus commits está meio quebrados. Eles têm mensagens de confirmação temporárias, talvez títulos na mensagem. Eu não me importo muito com isso. Ao confirmar o tempo todo, tenho certeza de que, se eu cometer um erro, posso voltar imediatamente a rastrear para um estado anterior do meu código. É como ter uma operação de desfazer sempre disponível enquanto escrevo código. Então, quando estou feliz com o estado do meu código e me sinto pronto para compartilhá-lo, normalmente antes de ir para a origem, paro e faço uma nova recuperação interativa. Eu limpo minha história. Refatoro minha história, por assim dizer. É como a refatoração que muitos de nós fazemos no nosso código depois de executá-lo e antes de o comprometermos para que ele seja agradável, limpo e pronto para produção. Bem, você pode fazer o mesmo com sua história do git. Você esmaga vários commits juntos, reformula as mensagens deles, talvez mude a ordem deles. Você o torna agradável e depois pressiona. Adoro usar rebase interativo assim, para corrigir erros, é claro. É uma ferramenta muito flexível. O Reflog Dissemos que, sempre que você faz algo que muda o histórico, como uma re-interação interativa, por exemplo, ou mesmo algo tão simples quanto alterar uma confirmação, o Git precisa copiar informações de confirmações antigas para novas confirmações. Os novos commits podem parecer com os antigos, mas não são os mesmos objetos. E os antigos commits são deixados para trás. Geralmente, eles são inacessíveis. Não há mais ramificação ou tag apontando para eles, portanto eles permanecerão no banco de dados de objetos por um tempo até que o Git finalmente decida coletar lixo. Agora, e se eu mudar de idéia e quiser recuperar um desses objetos? Esta não é uma situação comum, mas às vezes pode acontecer. Por exemplo, e se eu fizer uma nova recuperação interativa e excluir uma confirmação por engano? Agora, esse commit e seus dados associados não estão mais no histórico. Sei que eles ainda estão em algum lugar no banco de dados de objetos, mas não conheço mais os hashes deles, então não posso recuperá-los. Então, o que eu faço agora? Bem, a boa notícia é que existe uma maneira muito fácil de recuperar os hashes de objetos abandonados. Toda vez que a referência é movida em um repositório, o Git registra essa movimentação. Por exemplo, quando você faz checkout de uma ramificação, está movendo a referência HEAD, então o Git está registrando isso. Vamos dar uma olhada no ramo de espaguete. E então vamos fazer o checkout master novamente. Lá, mudei a referência HEAD duas vezes. O Git registrou esses movimentos em algo que é chamado de log de referência ou reflog, abreviado. E eu posso olhar para o reflog com git reflog, e então posso dar o nome de referência. Vamos olhar para a cabeça. Aqui estamos. Veja as duas primeiras linhas do reflog. Ele nos diz que as duas últimas alterações no HEAD foram do mestre para o espaguete e depois do espaguete para o mestre novamente. E, se você continuar lendo, poderá ver todas as alterações no HEAD que aconteceram quando eu fiz o rebase interativo mais cedo, e ainda mais cedo quando alterei a confirmação mais recente, quando fiz check-out dos ramos, quando criei novos commits, quando redefinii ramifica alguns módulos atrás, e assim por diante. E essa informação vem com os hashes dos objetos que o HEAD estava apontando. Portanto, por exemplo, esse commit aqui não está mais no histórico, foi alterado e substituído pelo seguinte commit. Mas até que o lixo seja coletado, seu hash permanece no reflog, portanto ainda podemos analisá-lo referenciando o hash diretamente ou usando esta sintaxe aqui, o que significa a 15ª posição anterior do HEAD. Aí está. E se você pode vê-lo, pode recuperá-lo. Por exemplo, você pode colocar um ramo nele e não será mais um commit abandonado. Apenas para esclarecer, as informações no reflog são estritamente locais. Esse reflog pertence a este repositório e somente a ele. Se eu clonar este repositório novamente em outro diretório, obteria um reflog diferente. Mas quando se trata deste repositório, toda vez que o HEAD se move, o Git o registra aqui. E o mesmo vale para outras referências, como o ramo principal. Lá. Veja isso se movendo por aqui, desde o momento em que clonei este repositório do GitHub. Foi quando o ramo mestre local foi criado. E é isso sobre o comando reflog. Espero que isso faça você se sentir um pouco mais seguro quando estiver usando o Git. Nos módulos anteriores, vimos algumas operações que são verdadeiramente irreversíveis e destroem dados, mas essas operações geralmente destroem dados na sua área de trabalho e talvez no índice. Quando se trata do repositório, geralmente você pode recuperar todos os objetos que você deixou para trás graças ao reflog. Revertendo confirmações Eu insisti muito neste módulo sobre o fato de que mudar a história é algo com o qual você deve ter cuidado, para não mudar uma história compartilhada e assim por diante. Talvez eu tenha sido um pouco confuso sobre isso. Mas eu quero fechar o módulo com um comando que é completamente seguro a esse respeito. Não destrói dados, não altera o histórico existente. Ainda é sobre fixar o histórico, mas faz isso sem alterar nenhum commit. Apenas cria novos commits. Primeiro, vamos para outro ramo, o ramo lisa, e vamos ver o log. Você deve se lembrar deste commit aqui. É antigo, um dos primeiros commit neste projeto. E está errado no sentido de que não respeita as regras do projeto. Se olharmos para ele, podemos ver que ele adicionou um novo item ao menu, cheesecake, mas não adiciona uma receita correspondente ao cheesecake na pasta de receitas. Veja, nenhuma receita de cheesecake aqui. Tivemos o mesmo problema para o commit do tikka masala e o corrigimos com um rebase interativo. Mas naquela época, decidimos não tocar nesse outro commit porque ele foi compartilhado. Ele veio antes da atualização mais recente da ramificação lisa remota aqui. Eu quero resolver essa situação para sempre. Vamos recapitular. Quero que cada item do menu tenha uma receita correspondente, mas, neste caso, eu tenho o item, mas não a receita. Por fim, posso finalmente adicionar a receita do cheesecake ou removê-lo completamente do menu. E depois de pensar nisso, decidi que preferia fazer o segundo. Quero excluir o cheesecake do menu. Este commit está errado. Quantas maneiras temos de excluí-lo? A primeira maneira seria a opção nuclear, faça uma grande reformulação interativa baseada no primeiro commit da história e remova esse commit que adiciona o cheesecake enquanto você faz o rebase interativo. Isso é exatamente o que eu não quero fazer, no entanto, porque criaria uma linha inteira de novos commits, uma nova história e mudaria o histórico compartilhado. Então não, não vamos fazer isso. Segunda maneira de fazê-lo, edite o menu ou remova esta linha do menu e crie uma nova confirmação. O commit antigo e quebrado permanece na história, mas pelo menos é corrigido mais tarde. Eu posso fazer isso. Mas, em vez de fazê-lo manualmente, por assim dizer, editando manualmente esse arquivo, posso pedir ao git para fazer isso automaticamente com um comando chamado git revert. Tudo o que tenho a fazer é dizer, olhe, eu quero reverter esse commit aqui. O que acontece é que o git cria automaticamente um novo commit que contém exatamente o oposto das mudanças no commit original, o que é muito útil se as mudanças não forem apenas uma linha, como neste exemplo trivial, mas digamos, centenas de linhas de código. Posso até reverter vários commits de uma só vez. Mas, neste caso, é apenas um pequeno commit de uma linha. E, de fato, tenho um novo commit aqui com uma boa mensagem que explica exatamente o que está acontecendo. E a única coisa que ele contém é o contrário do commit do cheesecake. Remove essa linha. A propósito, a razão pela qual eu fiz isso no ramo lisa e não no ramo mestre é que tentei fazer essa reversão no mestre enquanto preparava esse treinamento, mas tive um conflito lá porque o arquivo de menu foi alterado. muito no mestre desde que esse antigo commit confirma que mesmo a remoção de uma única linha causou um conflito. Então, eu não queria mostrar mais uma solução de conflito. No interesse de manter esse exemplo o mais simples possível, fiz isso em outro ramo. E agora posso mais uma vez verificar o mestre de caixa, onde essa reversão nunca aconteceu, e simplesmente viver com esse cheesecake no menu. Realmente não importa, espero ter feito o meu ponto. Mais uma coisa. Eu lhe disse que a reversão reverterá as alterações de uma maneira que é completamente segura no sentido de que não toca em confirmações existentes, apenas adiciona novas confirmações. Mas esse comportamento tem um problema. Tudo o que a reversão pode fazer é reverter seus dados escrevendo os dados opostos, mas não pode reverter suas alterações estruturais. Por exemplo, se você tentar reverter uma mesclagem, a reversão poderá remover todos os dados que foram adicionados pela mesclagem, mas não poderá remover a consolidação da mesclagem. A mesclagem ainda está lá. E isso pode causar algumas situações confusas, especialmente se você tentar mesclar novamente depois disso. Agora, estamos saindo um pouco do escopo deste treinamento, por isso não entrarei em detalhes aqui. Vou apenas mencionar esta questão rapidamente y como um aviso geral. Tenha cuidado ao reverter mesclagens. É um caso especial e você deve estar ciente de como lidar com isso especificamente. Em outras palavras, não pense em reverter como se fosse uma operação de desfazer genérica no Git. De fato, quanto mais próximo o Git estiver de uma operação de desfazer é provavelmente uma redefinição, quando você o usa para mover um ramo de volta para onde estava no passado. A reversão é muito mais estreita em escopo. Tudo o que ele faz é escrever uma nova confirmação com novos dados, que é o oposto dos dados existentes. É isso. Neste módulo, focamos em como alterar seu histórico e corrigir erros. Parabéns, você é um cirurgião de história do Git agora. Este módulo está repleto de informações, mas se você olhar para trás, introduzimos apenas alguns comandos ou, às vezes, novas opções para os comandos que já conhecíamos. Git commit --amend para modificar o commit mais recente, git rebase -i para fazer rebotes interativos, git reflog para acessar o reflog e corrigir erros aparentemente irrecuperáveis, e git revert para criar novos commit que revertem alterações nos commit anteriores. E essa é uma ferramenta pequena, mas poderosa, configurada para editar o histórico. Assim, podemos incluir esses dois módulos no histórico do projeto agora e passar para o último módulo do nosso treinamento. Encontrando seu fluxo de trabalho Fluxos de trabalho e dor E bem-vindo ao módulo final do Mastering Git. Até agora neste treinamento, fizemos uma suposição implícita. Assumimos que você estava trabalhando sozinho em seu próprio repositório Git e nunca estava compartilhando coisas com outros repositórios. Às vezes, prestamos atenção ao mundo exterior, como onde dissemos regra de ouro, não mudamos as coisas que você compartilhou com outras pessoas, mas principalmente ignoramos outras pessoas e outros repositórios como se eles não existissem. Mas é claro, o ponto principal do Git para a maioria das pessoas é que você o usa para compartilhar informações com outros desenvolvedores. E você tem muitas opções para compartilhar as informações. De fato, o conjunto de ferramentas Git é tão poderoso e flexível que você tem muitas opções às vezes, e isso pode causar algumas dores de cabeça quando várias pessoas se reúnem para trabalhar em um projeto compartilhado. Para evitar essas dores de cabeça, você e sua equipe precisam tomar algumas decisões sobre como trabalhar juntos e três coisas em particular. Primeiro, você deve decidir que tipo de modelo de distribuição adotar. Quantos repositórios você possui e como eles interagem? Você talvez tenha um repositório compartilhado que seja visível para todos os desenvolvedores? Talvez muitos repositórios compartilhados. Todos os desenvolvedores podem enviar suas confirmações para o repositório compartilhado ou alguns desenvolvedores têm acesso somente leitura e assim por diante? Além disso, você precisa tomar decisões em alguns ramos. Para quais ramos você tem no seu projeto e para que os usa? Qual ramo você usa para integrar seu trabalho ao trabalho de outros desenvolvedores? De qual ramo você usa para compactar um release? E similar. E, finalmente, você precisa definir restrições mais gerais para todos que contribuem para o projeto. Essas são todas as regras adicionais que realmente não pertencem ao modelo de distribuição ou ao modelo de ramificação, como por exemplo, quando você recebe um monte de novas confirmações de um controle remoto, você deve mesclar essas confirmações no seu próprio repositório ou deve você os rebase? Você pode enviar seu código para um controle remoto se as tarefas forem interrompidas ou deve garantir que o código esteja estável e funcionando antes de enviar? E tantas outras decisões possíveis. Portanto, modelo de distribuição, modelo de ramificação e restrição. Essas três coisas juntas definem o que você pode chamar de fluxo de trabalho distribuído, e é sobre isso que trata este módulo de trabalho, fluxos de trabalho distribuídos. Como você usa o Git na prática em um projeto do mundo real? Este módulo é um pouco diferente do restante do treinamento. Os fluxos de trabalho são mais um tópico social do que um tópico técnico; portanto, você não terá muitas informações técnicas aqui. O que você obterá é um vocabulário para discutir fluxos de trabalho com outras pessoas, como um conjunto de padrões. De certa forma, você pode pensar neste módulo como padrões de fluxo de trabalho do Git. Para entender este módulo, você precisa saber algumas coisas sobre o Git distribuído. Não muito, apenas o básico. Em particular, espero que você saiba o que é controle remoto e esteja familiarizado com o conceito de enviar para um controle remoto e puxar de um controle remoto. Se você já trabalhou em um projeto Git distribuído, mesmo em seu próprio projeto no GitHub, já possui esse conhecimento. Se você não tiver certeza sobre esses tópicos, poderá atualizá-los assistindo ao último módulo do treinamento Como o Git Funciona. Vamos começar a falar sobre fluxos de trabalho. Modelos de distribuição A primeira coisa que você deve decidir ao configurar um fluxo de trabalho Git, a decisão que influencia a maioria das outras decisões na linha, é o seu modelo de distribuição. O modelo de distribuição mais simples é provavelmente o modelo ponto a ponto. Imagine que você é um desenvolvedor em uma equipe de três. Todos esses repositórios contêm o mesmo projeto, é claro, foram originalmente clonados um do outro, mas divergem à medida que os desenvolvedores se comprometem com suas próprias cópias do repositório. O truque aqui é que cada desenvolvedor pode ver os repositórios de outros desenvolvedores como controles remotos. Portanto, se você é Nick neste diagrama e sabe que Ralph tem alguns novos commits que deseja, basta sair e inserir essas alterações no seu repositório, e se você sabe que há coisas novas no repositório de Jane, então você também pode fazer essas mudanças e os outros desenvolvedores podem fazer o mesmo, para que as mudanças se espalhem assim. Agora, é verdade que o modelo ponto a ponto é simples, mas não é necessariamente fácil, especificamente porque nenhum repo é mais importante que os outros. Essa é a definição de ponto a ponto, certo? E isso pode se tornar um desafio quando você deseja fazer algo tão simples quanto liberar o projeto, por exemplo. Então você precisa decidir de qual repositório liberar, e cada repositório pode conter coisas ligeiramente diferentes, por isso pode ser difícil decidir qual é o certo, por assim dizer, e se isso pode ser difícil para três desenvolvedores, pode ser muito mais difícil com quatro ou cinco. Por esse motivo, a menos que seu projeto seja literalmente duas pessoas em uma sala, você provavelmente desejará tomar algumas decisões adicionais antecipadamente. Em particular, você pode decidir que um dos acordos é especial. É abençoado, como eles dizem. Esse repositório abençoado é acessível a todos no projeto, tanto para puxar quanto para empurrar, para que você possa confirmar dados em seu próprio repositório local, mas também possui um controle remoto que está apontando para o repositório abençoado e pode enviar dados para esse controle remoto . A maioria dos desenvolvedores chama essa origem remota, por convenção, porque esse é o nome padrão que o Git atribui ao controle remoto do qual você clona o projeto. O repositório abençoado geralmente é um repositório simples, o que, em termos do Git, significa que é apenas o repositório sem uma área de trabalho ou um índice. Ninguém está trabalhando diretamente nessa máquina. É usado apenas para compartilhar dados e, talvez, para hospedar a máquina de compilação, o sistema que executa os testes de unidade, pacotes, versões e assim por diante. Então, deixe-me colocar um robô aqui para significar que este é o repositório de máquinas de construção compartilhada, não um repositório em que um humano está trabalhando dia após dia. Agora, neste modelo, você não acessa os repositórios de seus colegas de equipe e vê os dados deles, como no modelo ponto a ponto. Em vez disso, você se preocupa apenas com os dados no repositório abençoado. Esse é o estado oficial do projeto, por assim dizer. Portanto, todo mundo extrai seus dados do repositório abençoado, e o desenvolvedor que possui novos commit, como Ralph neste caso, deve enviar esses commit para o repositório abençoado, para que o restante da equipe possa obtê-los. Tudo é centralizado. Na verdade, você pode simplesmente chamar o repo abençoado de servidor, e o modelo em si é frequentemente chamado de modelo centralizado. Essencialmente, o modelo centralizado é o mesmo que as pessoas usam com sistemas de versão não distribuídos, como o subversion ou o Team Formation Server. Atualmente, a maioria das empresas substitui esses sistemas de versão. Eles mudam para o Git por vários motivos, mas não necessariamente mudam o modelo de distribuição. Muitos deles ainda usam um modelo centralizado por pares. Há ainda outro modelo que é uma reviravolta no modelo centralizado, em que você ainda possui repositórios de desenvolvedores e ainda possui um repositório abençoado central, mas a maioria dos desenvolvedores não pode gravar diretamente no repositório abençoado. Eles não podem empurrá-lo, apenas podem puxá-lo. Somente uma pessoa, ou algumas pessoas no projeto, têm esse poder de levar as coisas ao repo abençoado. Eles geralmente são chamados de mantenedores. Neste exemplo aqui, apenas Nick pode enviar por push para o repo abençoado, e os outros desenvolvedores, aqueles que não têm acesso por push ao repo abençoado, às vezes são chamados de colaboradores. Fiz as setas cinza para significar que são conexões somente leitura ou somente puxar. Portanto, a equipe ainda usa o repositório abençoado como um hub de comunicação e, quando um commit está no repositório abençoado, é oficial, e todo mundo o fará eventualmente. Agora, digamos que Jane tem novos compromissos para contribuir, mas ela não é a mantenedora. Ela não tem o poder de avançar para o repositório central, então como ela pode contribuir com esses compromissos para o projeto? O truque aqui é que o mantenedor também pode ver os colaboradores para reposicionar, para que Jane possa dizer oi Nick, puxe minhas alterações, veja se você gosta delas e fique à vontade para empurrá-las para o repo abençoado. Talvez Jane possa simplesmente ir até a mesa de Nick e contar a ele sobre as mudanças, ou talvez Jane e Nick nem compartilhem o mesmo escritório, então ela deve avisá-lo de alguma outra maneira, por exemplo, enviando uma mensagem dizendo: olha, eu tenho novos commits no meu repositório que você pode querer usar. Essa mensagem pode ser um email, por exemplo, ou pode ser gerenciada por algum tipo de serviço, como o GitHub, e isso geralmente é chamado de solicitação de recebimento. Agora, Nick recebeu uma solicitação de recebimento, para que ele saiba das mudanças de Jane. Se ele estiver de acordo com essas mudanças, ele poderá colocá-las em seu próprio repositório, resolver qualquer conflito de mesclagem que ele possa ter e enviá-las para o repositório abençoado. E agora as mudanças de Jane foram feitas parte oficial do o projeto, e outros colaboradores como Ralph também podem retirá-los. A solicitação de recebimento é a mecânica definidora desse modelo, para que você possa chamá-lo de modelo de solicitação de recebimento. O Git possui alguns recursos que facilitam a preparação de uma solicitação pull. Na verdade, existe um comando pull de solicitação do git, mas na verdade o envio da solicitação de pull é um recurso do Git. Você precisa usar correio antiquado ou algum outro meio de fazer isso. E essa é uma das razões pelas quais serviços como o GitHub são tão populares, porque automatizam o envio de solicitações pull, tanto que a solicitação pull se tornou o recurso definidor do GitHub, e o modelo pull request se tornou o mais popular. modelo para desenvolvimento de código aberto. A vantagem mais significativa do modelo de solicitação de recebimento é que ele ajuda a gerenciar a confiança. Você pode obter contribuições de fontes confiáveis, como Nick, neste caso, e também de fontes menos confiáveis, como Ralph e Jane, e isso é necessário em projetos de código aberto, é claro, porque, em código aberto, qualquer pessoa na Internet pode ser um colaborador e você provavelmente não confia na Internet inteira para enviar diretamente para o repositório do seu projeto, para que possa ter alguns mantenedores confiáveis ​​e qualquer número de colaboradores, mas esse conceito de gerenciamento de confiança também é útil para projetos de código fechado. Sempre que você tiver um projeto em que não deseja conceder acesso push ao repositório principal para toda a equipe de desenvolvimento. Nesse caso, você pode nomear um mantenedor interno, talvez chamá-lo de gerente de integração, e os desenvolvedores devem solicitar que essa pessoa faça suas alterações. Uma última coisa sobre o modelo de solicitação de recebimento antes de prosseguirmos. Se você observar os projetos no GitHub, eles usam esse modelo em uma variação um pouco mais complexa. A complexidade adicional aqui é que, no GitHub e em outros serviços como o GitHub, todos esses repositórios estão realmente na nuvem; portanto, os desenvolvedores não estão trabalhando diretamente nesses repositórios. Em vez disso, cada desenvolvedor tem dois repositórios, o repositório particular em seu próprio computador e o repositório público na nuvem. Portanto, o conceito é praticamente o mesmo que o modelo básico de solicitação de puxar, mas a mecânica de empurrar e puxar e coisas do gênero estão um pouco mais envolvidas. Se você quiser uma explicação detalhada desse tipo específico de modelo de solicitação de recebimento, poderá encontrar um no último módulo do treinamento Como o Git Funciona, para que eu não me repita aqui. Em vez disso, vou falar sobre um último modelo de distribuição. Este é como o modelo de solicitação de recebimento, mas leva as coisas um passo adiante. No modelo de solicitação de recebimento, você tem um repositório abençoado para todo o projeto com um ou mais mantenedores que podem acessá-lo. Aqui eu mostro no repo abençoado e que Jane é a mantenedora. Nesse outro modelo, no entanto, o projeto é dividido em subprojetos, e cada subprojeto tem seu próprio repositório abençoado e seu próprio mantenedor, ou mantenedores, e existem outros colaboradores em geral. Devo admitir que estou reutilizando os gráficos para os colaboradores um pouco aqui, mas tudo bem. Você sabe o que eles dizem sobre descolados, todos parecendo iguais. Também mudei um pouco o estilo deste diagrama. Não desenhei nenhuma flecha para evitar torná-la muito confusa. Em vez disso, lembre-se de que, neste modelo, qualquer pessoa pode extrair dados de praticamente qualquer lugar, mas apenas os mantenedores podem enviar dados para os repositórios abençoados. Então, aqui temos três níveis. Você pode ter ainda mais se quiser ter vários níveis de subprojetos, e há solicitações de recebimento, é claro. Contribuintes regulares geralmente enviam PRs para subprojetos, e os mantenedores do subprojeto enviam PRs para o mantenedor do projeto principal, e todo mundo puxa dados para seus próprios repositórios, para que os dados se espalhem para cima em resposta às solicitações de recebimento, e para baixo à medida que as pessoas o puxam. dos níveis superiores. Às vezes, esse modelo é usado em projetos muito grandes que são grandes demais para uma única equipe de mantenedores. O exemplo clássico é o kernel do Linux. No Linux, os mantenedores do subprojeto são chamados de tenentes, e o mantenedor global é chamado de ditador benevolente; portanto, você pode chamar esse modelo de modelo de ditador e tenentes. Também é popular em grandes empresas que têm grandes projetos ou, às vezes, realmente gostam de hierarquias. Então, para recapitular, vimos quatro modelos de distribuição, o modelo ponto a ponto, sem repo abençoado, sem controle centralizado. É puramente distribuído. O modelo centralizado é como o gerenciamento de configuração tradicional e não distribuído. Você tem um repositório central abençoado, um servidor e todos enviam para esse servidor. O modelo de solicitação de recebimento ainda possui um repositório centralizado, mas a maioria das pessoas só pode obter dele. Alguém que tem o direito de enviar por push para o repositório central deve enviar os motores de outros repositórios em resposta a algum tipo de solicitação de recebimento. E o modelo mais complicado é o ditador e os tenentes, onde você tem vários subprojetos, cada um é como um projeto de modelo de solicitação pull e, em seguida, um nível mais alto, ou talvez até vários níveis mais altos de integração, porque o subprojeto também é baseado em solicitações de recebimento. Um ponto importante para fechar. Lembre-se de que esses são padrões. Eles não são receitas de como estruturar seu próprio projeto. Portanto, você pode querer ter uma abordagem mista em seu próprio projeto. Muitos projetos fazem isso. Talvez a maioria dos desenvolvedores possa precisar de um projeto para trabalhar em um modelo centralizado, exceto a equipe de desenvolvimento offshore que segue o modelo de solicitação de recebimento e envia solicitações de recebimento aos desenvolvedores internos. Ou talvez você trabalhe centralizado, exceto que às vezes os desenvolvedores estão trabalhando juntos em um recurso e, em seguida, optam por sincronizar seus repositórios como se estivessem trabalhando em um modelo ponto a ponto. Isso é bom. Você não precisa seguir um modelo religiosamente. Use o que funciona para você. Esses nomes que eu dei a você são apenas rótulos. Eles são úteis para conceituar e discutir suas opções; portanto, por exemplo, você pode facilmente dizer que estamos usando um modelo centralizado aqui, e seu colega de equipe pode responder, não, vamos usar um modelo de solicitação de recebimento. Esse é o objetivo dos padrões. Modelos de ramificação Logo após o modelo de distribuição, o segundo elemento importante para garantir o fluxo de trabalho do Git é algum tipo de política para gerenciar filiais. Todo projeto tem essa política, seja explícita ou não, então vamos ver alguns padrões comuns para ramificações. Primeiro, deixe-me fazer uma distinção entre ramos estáveis ​​e instáveis. Uma ramificação é estável quando a ponta da ramificação sempre contém uma versão funcional do projeto. Ou seja, os testes são verdes, não há bugs conhecidos e assim por diante. Aqui, você pode empacotar o que estiver em um ramo estável e reutilizá-lo. Usarei essa marca de seleção verde para significar a versão de trabalho do projeto. Portanto, mesmo que alguém adicione novas confirmações à ramificação, por exemplo, pressionando-as, a ponta da ramificação ainda contém um sistema em funcionamento. Em um ramo instável, você não necessariamente tem isso. Quando alguém envia mais uma confirmação ao ramo, a ponta do ramo pode estar funcionando ou pode estar quebrada. Não há garantia. Agora que temos uma definição de ramificação estável e ramificação instável, voltemos às ramificações típicas de um projeto. Quase todo projeto tem um ramo principal que você usa para juntar tudo. Geralmente, esse é o ramo que as pessoas consideram o ramo mais importante do sistema. As pessoas podem trabalhar em outros ramos, mas os outros ramos tendem a se ramificar nesse ramo principal. Eles tendem a permanecer razoavelmente alinhados com o ramo principal e acabam voltando ao ramo principal. Estou usando cores diferentes para os commits aqui para fazer com que os segmentos dos ramos se estendam um do outro. As cores não têm nenhum significado específico. Esse ramo tão importante, que eu desenhei em vermelho na foto, geralmente é chamado de mestre, mas seja como for. Você pode chamá-lo de ramo de integração porque é o lugar onde as coisas se reúnem e geralmente o lugar onde você resolve os conflitos que você pode ter quando as coisas se reúnem. As pessoas também o chamam de ramo principal, ou a linha principal, às vezes, ou ramo de desenvolvimento, ou apenas o ramo principal, mas temos que escolher um nome, então chamarei de ramo de integração. Agora, o ramo de integração é estável ou instável? Bem, isso depende do projeto. Na maioria dos casos, quanto mais estável for o ramo de integração, melhor. Afinal, este é o ramo central do projeto. Principalmente todo mundo está trabalhando nisso, e ninguém gosta de trabalhar em uma base de código instável. Por outro lado, é difícil manter uma ramificação estável quando você está constantemente integrando coisas novas sobre ela; portanto, na prática, a maioria dos projetos parece uma ramificação de integração praticamente estável. Na verdade, é para isso que servem as máquinas de construção. O principal trabalho de uma máquina de compilação na Jenkins, ou algum outro tipo de sistema de compilação automatizado, é verificar o que estiver no ramo de integração, provavelmente executar os testes e informar se a compilação atual está funcionando ou quebrada . Outra pergunta importante na maioria dos projetos é de qual ramo você libera seu software? Em algum momento, você precisa implantar o software em um servidor da Web, ou talvez empacotá-lo e distribuí-lo em uma loja de aplicativos ou qualquer que seja o seu método de distribuição. Alguns projetos fazem isso a partir do ramo de integração, colocando principalmente uma marca em confirmações específicas para marcar os pontos de onde são liberados. Outros projetos preferem ter uma ramificação separada para liberações, uma ramificação de liberação. Qual o sentido de ter uma ramificação separada para lançamentos? Bem, existem algumas vantagens, mas a mais óbvia é que você pode manter o código nesse ramo mais estável do que no ramo de integração. Por exemplo, você pode mesclar a ramificação de integração na ramificação de liberação somente depois de verificar se é estável. Essencialmente, um ramo de versão separado fornece um buffer para manter as alterações liberáveis ​​separadas das alterações ainda não totalmente liberáveis. Eu falei sobre um ramo de lançamento, mas em alguns projetos você precisa manter vários lançamentos ao mesmo tempo. e, nesse caso, você pode ter várias ramificações de liberação. Talvez eles se ramifiquem do ramo de integração no momento em que o lançamento acontece e depois prossigam. Se você precisar adicionar documentação específica ou corrigir um bug ou a versão 1.1, mas não a versão 1.2, bem, você terá um ramo de versão específico para trabalhar. Há outro tipo de filial que é um grampo de muitos e muitos projetos. Digamos que você tenha dois desenvolvedores trabalhando em dois recursos diferentes. Em alguns projetos, ambos iriam diretamente para o ramo de integração. Estou usando duas cores diferentes para os dois recursos aqui. Com essa maneira de trabalhar, você pode ter uma integração muito frequente, que eu gosto em geral, mas às vezes pode ser difícil fazer o certo e, além disso, sua história fica difícil de entender porque os commits pertencentes a diferentes recursos podem ser todos interligado, como neste caso, e com essa abordagem, você também precisa viver com recursos semi-desenvolvidos no ramo de integração na maior parte do tempo. Por exemplo, agora, talvez o recurso verde esteja pronto, mas o recurso azul está ainda em andamento, você terá metade do recurso azul no ramo de integração. Uma alternativa possível é criar uma nova ramificação para cada recurso. Assim, por exemplo, um ramo para o recurso A e um para o recurso B. E quando as pessoas começam a avançar para esses ramos, elas divergem e progridem em paralelo até eventualmente voltarem para o ramo de integração. Nesse caso, estamos integrando ao mesclar, mas você também pode refazer o processo, se desejar. E depois disso, você pode excluir esses ramos, se desejar, ou deixá-los lá para referência futura. Esses ramos são chamados ramos de recursos. Algumas pessoas preferem o nome tópico ramos, o que também é bom. Usarei ramificações de recursos aqui apenas porque parece ser um pouco mais comum. Muitos projetos usam ramificações de recursos. Já vimos alguns estilos de ramificações, e você pode pensar que, depois de ter uma ramificação de integração, talvez uma ou mais ramificações de lançamento e várias ramificações de recursos, você tem tudo o que precisa. Mas há um problema adicional que pode resultar em mais ramificações para alguns projetos. Imagine que você tenha uma situação como essa com alguns ramos. Deixe-me dar nomes a eles, para que seja mais fácil referenciá-los. Não importa quais ramificações elas são exatamente, o que importa é que essas ramificações estão divergindo há algum tempo, mas a ramificação 1 tem uma confirmação, essa confirmação vermelha aqui, que você também deseja ter na ramificação 2. Você não não quero nenhum dos outros commit das ramificações 1 e 2, apenas esta confirmada. Então, como você faz isso? Como você copia um único commit de um ramo para o outro? Bem, o Git tem um comando especial que é exatamente isso, e é chamado de escolha de cereja. Você pode escolher uma única confirmação, ou algumas confirmações específicas, de uma ramificação e copiá-las em cima de outra ramificação. Problema resolvido, exceto por um detalhe. Uma escolha de cereja é como uma pequena recuperação. É a reformulação de um commit específico, e os subprojetos não gostam de reformulações. Eles querem usar a mesclagem em todos os lugares. Portanto, as escolhas de cereja não são uma opção para projetos baseados em mesclagem. Como você ainda pode ter o mesmo commit em dois ramos separados sem escolher a cereja? Bem, há outra solução para o problema. Quando você deseja que o mesmo commit seja compartilhado por dois ramos separados, basta ter um terceiro ramo e colocar esse commit compartilhado no terceiro ramo. E então você mescla o terceiro ramo no primeiro e no segundo ramo, assim. E é isso. Agora você tem um lugar para colocar confirmações compartilhadas e acaba de mesclar, sem rebotes. Esta é realmente uma situação comum, especialmente quando você tem uma correção de bug. Deixe-me mudar os nomes desses ramos. Você tem um ramo de lançamento e um ramo de integração, e eles estão divergindo. Você acabou de encontrar um bug desagradável em sua versão mais recente e deseja corrigi-lo imediatamente e preparar outra versão. Mas você também deseja que a mesma correção de bug esteja no ramo de integração, para que você também tenha o bug corrigido na próxima versão. Bem, basta colocar a correção de bug em outro ramo chamado, digamos, hotfix, e mesclar o hotfix no ramo de lançamento e de integração. Agora, essa correção está no histórico dos dois ramos. Melhor dos dois mundos. Você manteve as ramificações separadas e ainda compartilhou dados entre elas. Também existem outros casos de uso para o ramo de acoplamento como este, mas as correções de bugs são comuns, portanto, não se surpreenda se você vir projetos que possuem os chamados ramos de hotfix para esse tipo de coisa. Então vamos ver. Falamos sobre a diferença entre ramificações estáveis ​​e instáveis ​​e, em seguida, sobre diferentes padrões de ramificação, integração, versão, recurso e ramificações de hotfix. Pode haver mais, você provavelmente criaria alguns ramos especializados. Mas esses são comuns o suficiente para merecerem ser chamados de padrões. Você os vê surgindo várias vezes em vários projetos diferentes. E é isso sobre modelos de ramificação. Vejamos o terceiro e o último componente de um fluxo de trabalho distribuído. Restrições O terceiro e o último elemento de um fluxo de trabalho distribuído, após o modelo de distribuição e o modelo de ramificação, é um conjunto de restrições. Estou usando restrições como um nome completo para todas as coisas que claramente não pertencem ao modelo de distribuição ou ao modelo de ramificação. Além disso, comportamento, se desejar. Todo projeto em que trabalhei tinha suas próprias regras, às vezes muito específicas, e às vezes surpreendentes, então não posso mostrar nenhum padrão específico aqui, porque existem muitas restrições possíveis, mas posso apresentar alguns exemplos de restrições que eu já vi em projetos da vida real. Eu já mencionei um dos exemplos mais comuns, a escolha entre mesclagens e rebotes. Em alguns projetos, as pessoas preferem sempre mesclar, portanto, se você acabou de trabalhar em uma ramificação de recursos, por exemplo, você as mescla na ramificação de integração. Por outro lado, alguns projetos refazerão toda a ramificação de recursos em cima da ramificação de integração, e as duas políticas têm trocas diferentes, que geralmente são sutis e, bem, estão fora de tópico aqui, na verdade. Eu falei sobre essas compensações no treinamento Como o Git Funciona. Mais uma vez, não quero me repetir. Meu argumento aqui é que depende de você usar mesclagem ou rebase, mas você provavelmente deve tentar ser consistente com isso, para que provavelmente não queira metade da equipe usando mesclagem e a outra metade usando rebase. Isso seria confuso, portanto, essa geralmente é uma decisão para todo o projeto. Você provavelmente quer que seja uma construção oficial. Outra questão importante para alguns projetos é quem pode fazer o que com quais filiais? Talvez seja esperado que alguns desenvolvedores se comprometam apenas com alguns ramos e não com outros, ou talvez seu projeto esteja usando tags no ramo de lançamento para marcar um novo lançamento, para que apenas o responsável pelas liberações possa marcar o ramo de lançamento. Outros desenvolvedores devem evitar fazer isso, mesmo que tecnicamente eles possam fazer isso. Aqui está outro exemplo. Este não é necessariamente comum, mas é interessante. Eu estava trabalhando em um projeto com muitos desenvolvedores fazendo integração contínua, e a build-machine foi executada nos testes de unidade no ramo de integração. De vez em quando, os testes podiam ser interrompidos, em vermelho; portanto, tínhamos uma regra em vigor que dizia que, se você perceber que a compilação é vermelha, pare de empurrar para o servidor. Aguarde até a pessoa que interrompeu a construção perceber que há um problema para que ela possa consertar a construção e torná-la verde novamente. Caso contrário, se você continuar pressionando, tornará o trabalho dessa pessoa realmente difícil. Você mudará o código sob seus pés e acionará mais compilações. Então, essa foi a restrição. Não empurre quando a construção estiver vermelha. Espere que ele fique verde novamente. A parte interessante é que às vezes as pessoas se esqueciam de verificar a máquina de construção antes de empurrar, então elas violavam essa restrição, mesmo que não quisessem. Então a equipe fez algo inteligente. Eles usaram o recurso Git chamado ganchos. É basicamente uma maneira de executar um script sempre que um evento específico acontece, então eles tinham esse gancho no servidor que iniciou esse script sempre que alguém iniciou um push no servidor, e o script verificou a máquina de compilação e, se a compilação era vermelha, o script lhe deu um aviso e a oportunidade de abortar o envio. Eu acho que é um bom exemplo de como ajudá-lo a respeitar a restrição com alguma automação simples. Aqui está outra decisão que algumas equipes optam por regular. Que tipo de histórico ou refatoração você deve fazer nas ramificações de recursos antes de mesclá-las na ramificação de integração? Alguns projetos gostam de compactar o recurso inteiro em um único commit, por exemplo, enquanto outros preferem manter um commit pequeno e granular no histórico da linha principal, e assim por diante. Não quero continuar reclamando aqui, porque isso pode sugerir que essas são todas as áreas em que você deve tomar decisões. Na verdade, em seu projeto, você pode não se importar com esses assuntos e, em vez disso, pode se preocupar com outras coisas que eu nem pensaria. Eu só queria criar um monte de exemplos. O que estou dizendo é que os fluxos de trabalho mais distribuídos têm algumas restrições e existem muitas restrições diversas; portanto, é bom que você torne explícita a restrição do projeto onde descreve seu fluxo de trabalho. Portanto, agora discutimos os elementos que constituem um fluxo de trabalho Git, um modelo de distribuição, um modelo de ramificação e um conjunto de restrições. Como combinamos esses elementos em um fluxo de trabalho? Para responder a isso, vamos primeiro examinar um exemplo popular da vida real de um fluxo de trabalho distribuído. Precisamos conversar sobre o GitFlow Se você procurar fluxos de trabalho do Git no Google, é provável que você encontre esta postagem no blog entre os primeiros resultados. É um post de Vincent Driessen, um desenvolvedor holandês. Ele o publicou em 2010. Naquela época, o Git começava a se espalhar por toda a comunidade de software e muitas pessoas ficaram impressionadas com todas as possibilidades que o Git oferece; portanto, procuravam conselhos sobre como trabalhar com o Git, e muitas delas acharam esse conselho nesta postagem do blog. Ele descreve o fluxo de trabalho que Vincent e sua equipe usaram. É uma maneira muito específica de trabalhar que ficou conhecida como GitFlow. Não precisamos descrevê-lo em detalhes aqui, mas vale a pena dar uma olhada rápida nele. Em poucas palavras, o GitFlow é baseado em um modelo de distribuição centralizado. Todo desenvolvedor pode enviar para um repositório abençoado central. Mas o GitFlow também incentiva os desenvolvedores a trocar dados diretamente quando apropriado, portanto esse é um modelo de distribuição misto, centralizado com elementos ponto a ponto. O núcleo do GitFlow é seu modelo de ramificação, que é muito detalhado. Ele define um número de ramificações. Para entender as ramificações do GitFlow, você pode particioná-las amplamente em dois grupos, as ramificações instáveis, usadas para o trabalho de desenvolvimento, e as ramificações estáveis, usadas principalmente para lançamentos. Na maioria das ramificações instáveis, você tem uma ramificação de integração, denominada develop, e as ramificações de recursos, uma por recurso. Na segunda categoria, os ramos estáveis, você tem outro ramo de integração chamado mestre. Isso é diferente de desenvolver porque o desenvolvimento não é estável em geral, enquanto o mestre deve ser estável. Portanto, você só mescla o desenvolvimento ao mestre quando sabe que possui um sistema em funcionamento. E então você tem as ramificações da versão, uma separada para cada versão. E você também tem uma ramificação de hotfix. E depois existem as restrições. Muitos deles. Eles definem principalmente quais ramificações podem ramificar de quais outras ramificações e quais ramificações podem ser mescladas em quais outras ramificações. A propósito, eu digo mesclar, porque está sempre mesclando, nunca rebatendo. O GitFlow acredita em manter uma história verdadeira e rastreável. Você não deve mudar essa história rebatizando. E há mais regras sobre o que marcar e quando, e assim por diante. O GitFlow também fornece as convenções de nomenclatura que você deve usar para algumas de suas ramificações. Todas essas restrições significam que o GitFlow está bem definido e essa precisão tem algumas vantagens definidas. Por exemplo, você pode encontrar as extensões Git na Internet que fornecem comandos específicos para o GitFlow. Operações como criar uma ramificação de recurso ou mesclar um hotfix estão bem definidas o suficiente para que possam ser automatizadas. No mínimo, a precisão contribui para uma documentação muito boa. Essas são outras razões pelas quais o GitFlow se tornou popular. De qualquer forma, é muito popular. Tornou-se o fluxo de trabalho do Git para muitas pessoas, e muitas equipes apenas adotaram como estão. Agora, tenho uma opinião bastante forte sobre isso. Em uma frase, sugiro que você não use o GitFlow e comece a usá-lo. Agora, por favor, não pense que estou desrespeitando o GitFlow aqui. Eu não acho que haja algo errado com o GitFlow. É um fluxo de trabalho tão bom quanto você pode ter. E também inclui muitas práticas sólidas de controle de versão. O que eu recomendo é apenas pegar e usar qualquer fluxo de trabalho pré-enlatado, e muitas equipes fazem exatamente isso com o GitFlow, especialmente em grandes empresas. Eles acessam a Internet, encontram o GitFlow e o tornam obrigatório para todos. Mas existem muitos projetos para os quais o GitFlow não é a solução ideal. Por exemplo, projetos que podem evitar todas essas regras complexas de ramificação. Imagine que seu produto é um aplicativo da Web e não um aplicativo empacotado; portanto, você mantém apenas uma versão de produção por vez. Talvez você não precise de várias ramificações de lançamento. Você pode se safar com um. Ou talvez algumas das regras do GitFlow possam ser contraproducentes para você. Talvez você esteja em um projeto de ponta que faz implantação contínua. Ele implanta a produção toda vez que alguém integra um recurso estável. E todas essas ramificações introduziram muito na direção da integração à implantação. Algumas das restrições no GitFlow podem até incentivar comportamentos prejudiciais em projetos que não se encaixam nelas. Por exemplo, o GitFlow exige ramificações de recursos, e você deve mesclar uma ramificação de recursos ao desenvolvimento depois que o recurso for concluído. Porém, alguns grandes projetos herdados podem ter dezenas de recursos em desenvolvimento a qualquer momento, e cada recurso pode levar meses para ser implementado e tende a conter muito código. Portanto, quando esse processo é concluído e você finalmente mescla um recurso no ramo de desenvolvimento, tudo de uma vez, em uma única grande mesclagem, que pode causar grandes conflitos para outras pessoas na equipe. Em tais projetos, eu encorajaria as pessoas a se integrarem com mais frequência. Eu não sou um grande fã de ramos de recursos para dizer a verdade. Geralmente, prefiro fazer uma integração mais frequente quando posso. Mas algumas equipes respeitam essa restrição do GitFlow à letra e acabam tendo ramificações de recursos de folhas muito longas. Bem, você entendeu a minha deriva. O GitFlow é ótimo. É um fluxo de trabalho muito bem projetado, mas isso não o torna o fluxo de trabalho perfeito para qualquer projeto. Quando falamos sobre fluxos de trabalho, falamos sobre pessoas, a maneira como elas funcionam e, quando se trata disso, um tamanho definitivamente não serve para todos. Contexto é tudo, então, mais uma vez, não use apenas o GitFlow. Então, o que você deve fazer? Crescendo um fluxo de trabalho Como você cria seu próprio fluxo de trabalho? Fazer isso é mais arte do que ciência, portanto não há regras rígidas. Sem receitas. Mas posso lhe dar uma orientação importante aqui, que é evitar a tentação de apenas sentar e projetar um fluxo de trabalho. Essa abordagem tende a gerar um fluxo de trabalho super-projetado e super-complicado que ainda não trata dos problemas específicos que seu projeto pode ter no futuro. Mesmo se você for muito esperto, ainda é difícil prever todas as situações possíveis em que seu projeto entrará, e mesmo se você for bastante experiente, isso ajuda, mas ainda não é suficiente porque cada projeto é diferente, geralmente de maneiras sutis, para que suas experiências passadas possam realmente enganá-lo. Portanto, em vez de criar um fluxo de trabalho, você deve se esforçar para aumentar seu fluxo de trabalho. Comece pequeno. Estou pensando muito pequeno aqui. Algo assim. Os detalhes realmente não importam aqui. Este é apenas um exemplo do que quero dizer com pequeno. Modelo de distribuição, o modelo centralizado neste caso, mas escolha o que for mais adequado ao tamanho do seu projeto. Um modelo de ramificação simples, sem complexidades, a menos que você saiba que essas complexidades são necessárias e um punhado de restrições. Três restrições, como neste exemplo, podem ser suficientes. Você sempre pode adicionar mais restrições posteriormente. Talvez você precise de mais do que isso se tiver um grande projeto ou se trabalha em uma organização tradicional e muito estruturada. Mas, de qualquer forma, tente ficar do lado pequeno. Você pode achar que seu fluxo de trabalho muito simples é quase tudo o que você precisa, na verdade, e quando se mostra insuficiente e você descobre que precisa de algo mais complexo, basta adicionar essa complexidade à medida que avança. Você pode adicionar restrições, adicionar novas ramificações, incorporar idéias de outros modelos de distribuição. É isso que quero dizer quando digo aumentar o seu fluxo de trabalho. Faça evoluir para se adequar ao seu projeto. Adicione apenas as regras em resposta a problemas reais que se tornam visíveis e sempre esteja pronto para remover regras que não estão tendo um impacto positivo no projeto. Um cavalheiro chamado Dee Hock disse algo que se relaciona com o que estou sugerindo aqui. Aqui está essa citação. Se você deseja um comportamento complexo e inteligente, crie princípios simples, porque princípios simples dão às pessoas o espaço necessário para serem flexíveis e inteligentes, para resolver os problemas inesperados que sempre surgem em um ambiente complexo. Por outro lado, regras e regulamentos complexos criam burocracia, eles criam um ambiente inadequado para a solução de problemas. Dee Hock foi o fundador e CEO da empresa de cartões de crédito Visa, então provou em primeira mão o quão poderosa essa ideia pode ser. Muitos de nós nunca resolverão o tipo de problemas complexos que ele resolveu, mas ainda tento manter essa citação em mente quando sou tentado a ser inteligente e adicionar regras e procedimentos às minhas equipes em minha própria maneira de trabalhar. Mantenha simples. Você raramente se arrependerá de fazer isso. Pensando em Git Chegamos ao final deste treinamento, então aqui está um resumo rápido de tudo o que falamos. Na visão de mundo do Git, o mundo é dividido em quatro áreas. A área mais importante é o repositório onde seus dados permanecem seguros e com versão, a área de trabalho é sua própria área privada, onde você interage com os dados, o índice separa a área de trabalho do repositório, oferecendo muita flexibilidade para decidir quando e como mover dados para o repositório e o stash é como uma área de transferência que você pode usar para armazenar dados temporariamente enquanto manipula as outras áreas. A maioria dos comandos no Git pode ser entendida em termos de dois elementos, como eles movem dados entre as áreas e o que eles fazem no repositório em particular. Se você entende esses dois elementos, provavelmente entende o comando. Examinamos os comandos básicos sob essa luz, aqueles que você usa para fluxos de trabalho locais básicos, como adicionar, confirmar e check-out, e prestamos atenção especial aos comandos menos comuns, como redefinição, que são mais especializados, mas podem ser útil em várias situações. Esses tópicos foram objeto dos quatro primeiros módulos deste treinamento. Uma coisa que um usuário mestre do Git faz muito é interagir com o histórico, o que significa navegar no histórico para compreendê-lo, com comandos como git log, git diff e git blame, e também significa realmente editar o histórico ou refatorá-lo , com técnicas como rebase interativa, confirmação de alterações e o comando git revert. E isso significa até usar ferramentas como o reflog para auditar o histórico e se recuperar do erro destrutivo estranho. Tivemos dois modelos inteiros dedicados à cirurgia de história. Finalmente, neste último módulo, falamos sobre fluxos de trabalho, quais são os elementos e padrões comuns de um fluxo de trabalho distribuído do Git. O modelo de distribuição, o modelo de ramificação e as restrições adicionais. E dissemos que você não deveria apenas projetar um fluxo de trabalho. Em vez disso, você deve aumentar um fluxo de trabalho para se ajustar à sua situação específica. E foi isso. Chegamos ao fim de dominar o Git. Claro, ainda há muito que você pode aprender. Existem tantos comandos, recursos e opções que eu não mencionei porque eles não se encaixavam no fluxo deste treinamento, mas agora você tem o que conta. Você tem o modelo. Você entende como o Git vê o mundo e o conjunto de ferramentas básicas que o Git fornece para manipular esse mundo. Então você está pensando no Git agora. Você sabe cozinhar a lasanha, usar a metáfora que usei no começo. E isso é conhecimento suficiente para seguir por conta própria. Parabéns. Eu me diverti muito preparando esse treinamento. Espero que você tenha se divertido tanto, então obrigado de Paulo Perrotta e até o próximo treinamento