

# ***Versionamento de Código***

## ***Módulo 3***



**TIC em Trilhas**

# **MÓDULO 3**

**Este módulo tem como objetivo compreender técnicas avançadas e processos automatizados de análise de código, bem como citar algumas ferramentas importantes e úteis no mercado que auxiliam e facilitam o desenvolvimento de software.**

## CAPÍTULO 1

# FORK NO GIT

O Git possui um meio de colaborar em repositórios de código aberto chamado de *forking*.

**N**o Git, *forking* é importante para promover desenvolvimento independente, facilitar a contribuição em projetos de código aberto (*open-source*), permitir aprendizado e experimentação e promover a construção de uma comunidade no entorno de um projeto. Abaixo estão listados alguns motivos em detalhes da importância do processo:

- **Desenvolvimento Independente:** O *forking* permite que um desenvolvedor crie uma cópia pessoal do projeto de outra pessoa. Isso possibilita trabalhar no projeto de forma independente sem afetar o repositório original. Essa prática é particularmente útil quando se quer experimentar mudanças ou adicionar novos recursos sem o risco de impactar a base de código original do projeto clonado.

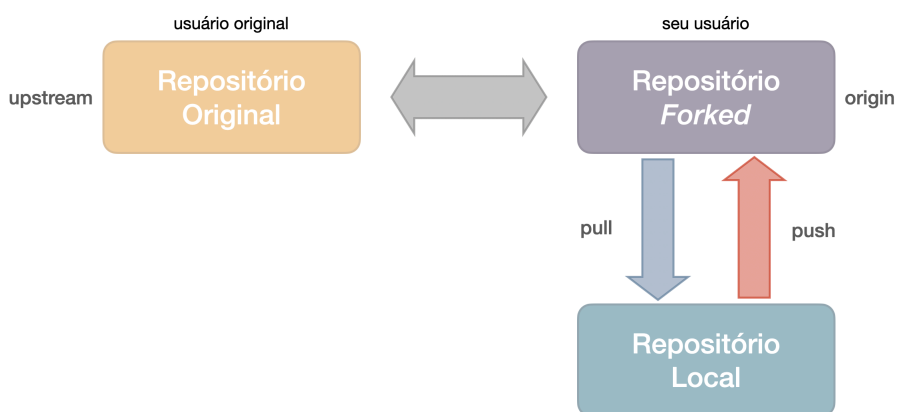
- **Contribuição para Projetos de Código Aberto:** Projetos de código aberto prosperam com contribuições da comunidade. O *forking* é o primeiro passo para um desenvolvedor externo fazer alterações em um projeto. Após fazer o *fork* de um repositório, os contribuidores podem realizar suas mudanças em seu *fork* e, em seguida, submeter um *pull request* para o repositório original a fim de integrar suas contribuições nele.
- **Aprendizado e Experimentação:** Para aprendizes e desenvolvedores que desejam entender como um determinado software funciona, o *forking* oferece um ambiente de teste (também chamado de *sandbox*). Eles podem dissecar, modificar e experimentar com o código sem quaisquer implicações para a base de código original.
- **Diversificação de Projetos:** Às vezes, um *fork* pode levar a uma nova direção para um projeto, criando um caminho completamente separado de desenvolvimento. Isso pode acontecer quando existem visões ou objetivos diferentes para o projeto, ou quando o repositório original não é mais mantido.
- **Backup da Base de Código:** Fazer um *fork* de um projeto também serve como uma forma de *backup*. Caso o repositório original seja deletado ou corrompido, o código ainda existe nos *forks* feitos por outros usuários.
- **Construção de Comunidade:** O *forking* incentiva um senso de propriedade comunitária e desenvolvimento

colaborativo. Quando vários usuários fazem *fork* de um projeto, muitas vezes leva a uma rede de desenvolvedores trabalhando em direção a objetivos comuns ou complementares.

Afinal, o que é *fork*? *Forking* no Git refere-se ao processo de criar uma cópia completa de um repositório existente. Essa ação é muito comum em plataformas de hospedagem de código como, por exemplo, o GitHub. Quando um desenvolvedor faz um *fork* de um repositório, ele cria uma cópia independente desse repositório na sua própria conta. Esse *fork* é totalmente separado do repositório original, o que significa que é possível fazer alterações nele sem afetar o repositório original. O *forking* permite também que o desenvolvedor trabalhe em um projeto sem ter que pedir permissão aos proprietários originais do repositório. Isso é especialmente útil em projetos de código aberto, onde pode-se querer contribuir com melhorias ou correções. Existe a possibilidade em que, depois de fazer alterações em um repositório que foi feito *fork*, poder propor que essas alterações sejam integradas no repositório original através de um *pull request*. Isso permite que os mantenedores do repositório original revisem suas alterações e, se aprovadas, as integrem.

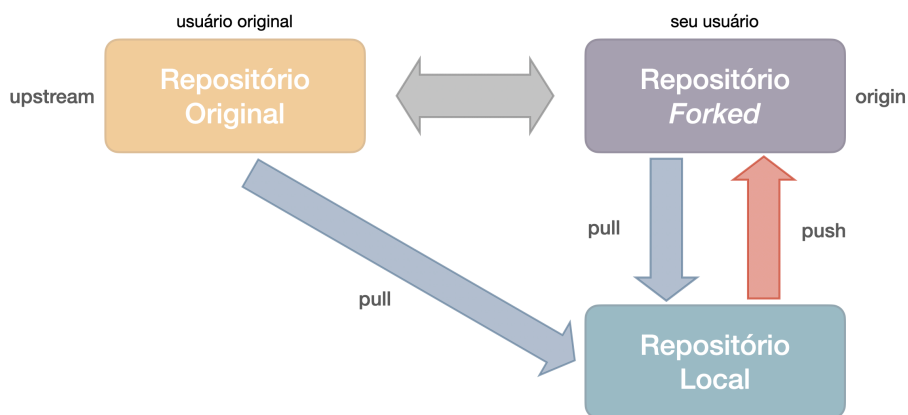
Para criar um *fork* no GitHub primeiro é preciso encontrar um repositório e navegar até ele. No site, o botão 'Fork' estará disponível para ser clicado. Ao clicar, uma tela de criação do *fork* aparecerá solicitando ao usuário escolher um nome para o repositório que será copiado (por padrão, *forks* recebem o mesmo nome do repositório original) e a

opção de fazer a cópia do repositório só com a branch *main*. Após ter o *fork* criado, basta clonar o repositório que agora está no repositório do usuário com o comando `git clone <link do repositório>`. Neste momento, o usuário pode enviar mudanças e fazer operações de *fetch* e *pull* de e para seu repositório remoto. A figura abaixo mostra o estado atual das operações possíveis para o repositório *origin*.



Entretanto, para manter o *fork* do usuário sincronizado com o repositório original, é preciso adicionar um remoto comumente chamado de *upstream* apontando para o repositório original. Isso pode ser feito com o seguinte comando: `git remote add upstream <link do repositório original>`. Para verificar se o repositório foi adicionado basta listar os repositórios com o comando `git remote` (ou use a versão com o parâmetro `-v` para ver os links atrelados aos repositórios). A partir daí é possível puxar versões atualizadas com *fetch* e *pull* do repositório original para qualquer branch que o usuário quiser. A

operação de push não será permitida em nenhum caso porque o repositório original não pertence ao usuário. Existem apenas duas maneiras de tentar enviar alterações para este repositório: a primeira é o dono do repositório original adicionar o usuário como colaborador do projeto (improvável que isso aconteça) e a segunda é fazendo um *pull request* para que o proprietário do repositório possa avaliar as alterações e decidir se quer adicioná-las ao seu repositório. A figura abaixo mostra o estado atual das operações possíveis, depois de o link com o repositório *upstream* ser feito.



A partir de agora, sempre que o usuário precisar atualizar seu *fork* com as mudanças do repositório original, basta buscar (operação de *fetch*) as mudanças do *upstream* e fazer merge delas na branch principal. Isso pode ser feito com a seguinte sequência de comandos:

1. `git fetch upstream`



2. `git checkout main`

3. `git merge upstream/main`

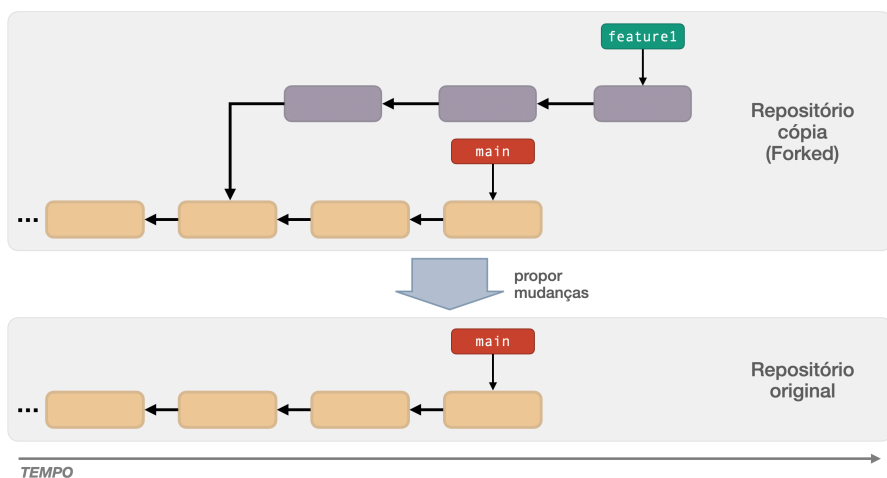
O primeiro comando busca as mudanças do repositório original, o segundo comando muda a branch atual para a `main` e o terceiro comando faz a operação de merge das mudanças que estão no repositório local (foram buscadas) e que estão apontadas pela branch `upstream/main` na branch `main`.

Tudo certo, agora é possível fazer mudanças no *fork*, commitar e fazer *push* dessas mudanças para o repositório no GitHub. Se for de desejo do usuário, ele pode contribuir com as mudanças do seu *fork* para o repositório original criando um *pull request* através do GitHub.

Note que, ao fazer *fetch* das informações, o usuário precisa especificar o nome do repositório remoto no comando `git fetch <repositório>`. Entretanto, ao enviar as mudanças com o comando `git push` isso não é necessário. Por que? Como o repositório cópia (o que foi feito *fork*) foi clonado para o dispositivo do usuário, o Git já configurou as branches locais e as respectivas *tracking* branches com o remoto, ou seja, ele já sabe qual é a branch equivalente no remoto a qual ele tem que sincronizar as mudanças quando solicitado. É possível verificar quais branches locais estão atreladas a branches remotas com o seguinte comando: `git branch -vv`.

## PULL REQUEST PARA REPOSITÓRIO ORIGINAL

Supondo o cenário hipotético em que o desenvolvedor fez um *fork* de um repositório de outra pessoa, clonou o repositório copiado (que está em seu GitHub) e fez alterações em uma nova branch criada, se ele quiser propor essas mudanças no repositório original ele precisa abrir um *pull request* para que o dono do repositório original revise as mudanças e decida se quer unir essas mudanças em seu repositório, se não quer ou se quer que o desenvolvedor faça alterações antes de permitir. A imagem abaixo ilustra de forma visual o objetivo do cenário hipotético.



Para solicitar o envio das mudanças da branch *feature1* no repositório cópia para o repositório original, basta entrar no site do GitHub e criar um *pull request*. A principal diferença na hora da criação do *pull request* é dada ao fato de que existem dois campos a mais a se preencher

informações no momento de se decidir qual branch é a fonte e qual é a destino. Em um *pull request* normal existe a branch base ou destino (as mudanças vão para essa branch) e a branch de comparação ou fonte (as mudanças que devem ser enviadas estão nessa branch). Ao solicitar alterações em um repositório diferente dois campos novos surgem: o repositório base ou destino (para qual repositório vão as mudanças?) e o repositório *head* ou fonte (de qual repositório serão pegas as mudanças para envio?). Basicamente, o esquema de decisão dos repositórios é análogo ao esquema de decisão das branches, respectivamente, ou seja, branch destino é equivalente ao repositório destino e branch fonte é equivalente ao repositório fonte. Se for aceito, as alterações do *pull request* são unidas (operação de merge) no repositório original e é possível, a partir daí, continuar o desenvolvimento normalmente. Note que, para uma revisão de um *pull request* ser aprovada (no caso de *fork*) é preciso de pelo menos uma aprovação de algum colaborador com permissão de escrita naquele repositório e, mesmo que tenha sido aprovado, o usuário não poderá fazer a operação de merge, apenas os colaboradores com permissão de escrita nele.

## CAPÍTULO 2

# AUTOMAÇÕES E CI/CD

Este capítulo explica quais automações se podem usar e porque Continuous Integration, Continuous Delivery e Continuous Development são práticas de desenvolvimento de software importantes.

**A** integração contínua (Continuous Integration), a entrega contínua (Continuous Delivery) e o desenvolvimento contínuo (Continuous Development) são práticas que automatizam os testes, versões e o lançamento (deployment) dessas versões para que o projeto e/ou organizações enviem mudanças de código mais rápido e de maneira mais confiável para o mercado.

O Git e o GitHub oferecem uma variedade de recursos e ferramentas destinadas à automação, e CI/CD. É possível explorar algumas delas como, por exemplo:

- **Ações (Actions):** As GitHub Actions são um dos recursos mais poderosos para automação, CI/CD no GitHub. Elas permitem que o desenvolvedor crie fluxos de trabalho customizados para automatizar o processo de desenvolvimento de software. Isso pode incluir testes, construção, implantação, e muito mais, tudo acionado por eventos específicos no repositório.
- **Pacotes (Packages):** O GitHub Packages é um serviço de hospedagem de pacotes de software que permite aos desenvolvedores publicar, instalar e gerenciar versões de pacotes de software. Ele suporta várias linguagens e ferramentas de empacotamento, como npm, Maven, RubyGems, entre outros.
- **APIs:** O GitHub fornece APIs robustas (REST e GraphQL) que permitem a integração com outros sistemas e ferramentas. Essas APIs podem ser usadas para automatizar tarefas, acessar dados de repositórios, manipular *issues* e *pull requests*, entre outras funcionalidades.
- **GitHub Pages:** O GitHub Pages permite hospedar sites estáticos diretamente dos repositórios de um desenvolvedor no GitHub. É comumente usado para hospedar portfólios pessoais, projetos de documentação, blogs e outros tipos de sites que não requerem processamento no lado do servidor.
- **Webhooks:** Os Webhooks permitem que aplicativos externos sejam notificados quando ocorrem eventos

específicos em um repositório ou organização. Isso é útil para desencadear ações personalizadas em outros sistemas quando algo acontece no GitHub, como um novo commit ou a criação de um *pull request*.

- **Hosted Runners:** Para executar *jobs* de GitHub Actions, você pode usar executores hospedados pelo próprio GitHub. Estes são ambientes virtuais que são automaticamente provisionados para executar os *jobs*, e depois descartados.
- **Self-Hosted Runners:** Em contraste com os executores hospedados, o desenvolvedor também pode configurar os próprios executores em um ambiente de sua escolha. Isso permite mais controle sobre o ambiente de execução e pode ser necessário para casos de uso específicos ou restrições de segurança.
- **Gerenciamento de Segredos (Secrets Management):** O GitHub permite armazenar segredos, como chaves de API e credenciais, que são necessários para os processos de CI/CD. Esses segredos são criptografados e podem ser usados em GitHub Actions de maneira segura, sem expor informações sensíveis.

Essas ferramentas e funcionalidades proporcionam uma poderosa infraestrutura para automação e CI/CD, permitindo que os desenvolvedores criem, testem, e implantem seus softwares de maneira eficiente e segura.

Como visto anteriormente, CI/CD é um conjunto de práticas usadas no desenvolvimento de software para melhorar e agilizar a forma como o software é desenvolvido e entregue. Mas, como isso funciona na prática? No contexto do GitHub, essas práticas são frequentemente implementadas usando Actions do próprio GitHub, que são *scripts* automatizados que podem construir, testar e implantar um código sempre que uma mudança é feita no repositório. Para implementar, por exemplo, uma Integração Contínua é preciso ter um repositório no GitHub e familiaridade com o Git e GitHub. De forma geral, para configurar CI com GitHub Actions basta seguir os seguintes passos:

1. Criar um Workflow de GitHub Actions;
2. Configurar o arquivo de Workflow;
3. Commit e push do arquivo de Workflow;
4. Verificar a execução do Workflow;
5. Ajustar e melhorar (caso necessário).

Como este capítulo tem como objetivo explicar e demonstrar conceitos de Git, os conceitos de CI/CD não serão aprofundados, visto que este tópico sozinho já pode ser considerado um caso de estudo muito mais aprofundado.

## CAPÍTULO 3

# SUMÁRIO DE FUNCIONALIDADES

O Git e GitHub são repletos de funcionalidades que podem ser aproveitadas dependendo do projeto em que ela melhor se encaixar. Este capítulo elenca a maioria das funcionalidades disponíveis para o desenvolvedor.

**O** Git e GitHub oferecem várias funcionalidades e práticas para aumentar a colaboração no desenvolvimento, a automação dos processos para deixar o fluxo de trabalho mais rápido, a segurança no desenvolvimento e manutenção do software, a gerencia de projetos de forma eficiente, a administração do time de colaboradores e a facilidade de comunicação e contato com a comunidade.

Na parte de segurança, existem vários tópicos pontuais que podem aumentar a proteção do projeto no desenvolvimento. Dentre elas, é possível citar:



- **Repositórios Privados:** No GitHub são visíveis apenas para pessoas especificamente autorizadas pelo proprietário do repositório. Isso ajuda a proteger o código-fonte e outros arquivos de serem acessados ou modificados por pessoas não autorizadas.
- **Autenticação de Dois Fatores (2FA):** A 2FA adiciona uma camada extra de segurança ao processo de login. Além do nome de usuário e senha, os usuários precisam fornecer uma segunda forma de verificação, como um código de um aplicativo de autenticação ou uma mensagem SMS.
- **Revisões Obrigatórias (*Required Reviews*):** Esta configuração exige que um ou mais revisores aprovem as mudanças em um *pull request* antes que ele possa ser mesclado na branch principal. Isso garante que o código seja examinado e aprovado por outras pessoas qualificadas antes de se tornar parte do projeto.
- **Verificações de Status Obrigatórias (*Required Status Checks*):** Essas verificações automatizadas (como testes de integração contínua) devem passar antes que um *pull request* possa ser mesclado. Isso ajuda a garantir que o novo código não quebre a funcionalidade existente.
- **Varredura de Código (*Code Scanning*):** Esta ferramenta analisa automaticamente o código para identificar vulnerabilidades e erros de segurança. Pode ser configurada para rodar em cada *push* ou *pull request*,

ajudando a identificar problemas antes que eles entrem na base de código principal.

- **Varredura de Segredos (*Secret Scanning*):** Identifica segredos, como *tokens* de API e chaves privadas, que podem ter sido comprometidos ao serem carregados acidentalmente no repositório. Isso ajuda a prevenir vazamentos de informações sensíveis.
- **Grafo de Dependências (*Dependency Graph*):** Mostra todas as bibliotecas e dependências que o projeto usa e como elas estão conectadas. Isso é útil para entender como as atualizações ou vulnerabilidades em uma dependência podem afetar o projeto como um todo.
- **Alertas do *Dependabot* (*Dependabot Alerts*):** O *Dependabot* monitora as dependências do projeto e alerta quando vulnerabilidades são encontradas nelas. Ele também pode criar automaticamente *pull requests* para atualizar as dependências para versões mais seguras.

Essas ferramentas e práticas são vitais para manter a integridade e a segurança dos projetos de software, especialmente em um ambiente colaborativo e aberto como o GitHub.

Já na parte de gerência do projeto, o Git e GitHub oferecem uma série de recursos e ferramentas que facilitam a gestão de projetos de software. Dentre esses recursos e ferramentas, é possível citar:

- **Projetos:** No GitHub, os Projetos são usados para organizar e priorizar o trabalho. Eles podem ser configurados como quadros Kanban ou de listas para gerenciar tarefas, bugs, novas funcionalidades e mais. Isso ajuda as equipes a acompanhar o progresso do trabalho em um nível alto.
- **Etiquetas (Labels):** As Labels são usadas para categorizar e filtrar *issues* e *pull requests*. Elas podem indicar o status de uma *issue* (como "em progresso" ou "bloqueado"), a prioridade dela, o tipo (*bug*, melhoria, etc.), ou qualquer outra classificação que a equipe ache útil.
- **Marcos (Milestones):** Os Milestones são usados para rastrear o progresso de um conjunto de *issues* ou *pull requests* em direção a um objetivo ou prazo específico. Eles são úteis para planejar e acompanhar o progresso de lançamentos, *sprints*, ou grandes recursos.
- **Issues:** As *Issues* são usadas para rastrear ideias, melhorias, tarefas ou *bugs* para o trabalho no GitHub. Elas são fundamentais para a colaboração e discussão sobre o desenvolvimento do projeto.
- **Gráfico Unificado de Contribuições:** Este gráfico mostra uma visão geral das contribuições de um usuário em todos os repositórios no GitHub, fornecendo uma visão rápida do seu envolvimento e atividades.
- **Gráfico de Atividade da Organização:** Este recurso fornece uma visão geral da atividade dentro de uma

organização no GitHub. Ele pode mostrar tendências de commits, *pull requests*, *issues* e mais, ajudando os líderes de equipe a entender o fluxo de trabalho e a colaboração dentro da organização.

- **Visões Gerais de Dependências da Organização:** Este recurso dá uma visão geral das dependências usadas em todos os repositórios de uma organização. Ele pode ajudar a identificar dependências desatualizadas ou vulneráveis e a entender o impacto de mudanças nas dependências.
- **Visões Gerais do Repositório:** *Repository Insights* oferece análises detalhadas sobre a atividade de um repositório específico, incluindo frequência de commits, *pull requests*, *issues* e muito mais. É uma ferramenta valiosa para entender como o repositório está sendo usado e colaborado.
- **Wikis:** As Wikis no GitHub permitem às equipes criar e compartilhar documentações de forma colaborativa. Elas são úteis para manter documentações de projetos, guias, notas de reuniões e outras informações importantes em um local central e acessível.
- **GitHub Insights:** Este recurso fornece análises aprofundadas e dados sobre equipes e projetos no GitHub. Ele pode incluir métricas de produtividade, qualidade do código, velocidade de colaboração, entre outros, ajudando a tomar decisões informadas sobre a gestão de projetos e equipes.

Estas ferramentas e recursos são essenciais para uma gestão eficaz de projetos e equipes, aumentando a produtividade de colaboradores, projetos e organizações como um todo.

O Git e o GitHub também oferecem diversas funcionalidades específicas para a administração de equipes e organizações, facilitando o gerenciamento de colaboradores, a segurança e a eficiência operacional. Abaixo estão detalhados alguns tópicos:

- **Organizações (*Organizations*):** Permitem agrupar pessoas e repositórios sob uma única entidade. Isso facilita a administração de várias equipes e projetos, a configuração de permissões, a criação de times, e a gestão centralizada de políticas e segurança.
- **Convites (*Invitations*):** Os administradores de uma organização podem enviar convites para novos membros se juntarem à organização ou a times específicos dentro dela. Isso permite um controle centralizado sobre quem tem acesso aos recursos da organização.
- **Sincronização de Equipes (*Team Sync*):** Esta funcionalidade permite sincronizar grupos de usuários de um provedor de identidade externo (como o *Active Directory*) com equipes no GitHub. Isso automatiza o processo de adicionar ou remover membros de equipes com base em suas associações de grupo na organização.

- **Funções Personalizadas (*Custom Roles*):** No GitHub, os administradores podem criar funções personalizadas com conjuntos específicos de permissões. Isso permite uma granularidade maior no controle de acesso, dando aos usuários apenas as permissões que eles realmente precisam para suas funções.
- **Verificação de Domínio:** As organizações podem verificar seu domínio de e-mail corporativo no GitHub. Isso ajuda a garantir que apenas usuários com um e-mail desse domínio possam se juntar à organização, aumentando a segurança e a integridade dos membros.
- **API do Registro de Auditoria (*Audit Log API*):** Permite que os administradores acessem e interajam com o registro de auditoria da organização, o qual registra eventos como criações de repositório, mudanças de permissões e outras ações importantes. Isso é crucial para monitorar a segurança e a conformidade.
- **Restrições de Criação de Repositório:** Os administradores podem restringir quem pode criar repositórios dentro de uma organização. Isso ajuda a evitar a proliferação descontrolada de repositórios e mantém a organização e a gestão centralizadas.
- **Restrição de Notificações:** Esta funcionalidade permite aos administradores controlar como as notificações são enviadas dentro da organização, podendo limitar quem pode enviar ou receber certos tipos de notificações. Isso

é útil para evitar spam e garantir que as informações importantes sejam comunicadas eficientemente.

Essas ferramentas e funcionalidades são projetadas para ajudar os administradores de equipes e organizações a gerenciar eficientemente seus membros, projetos e segurança no ambiente colaborativo do GitHub. Ele também oferece várias funcionalidades focadas na construção e no suporte de sua comunidade de desenvolvedores como, por exemplo:

- **GitHub Marketplace:** É uma plataforma onde os desenvolvedores podem descobrir, comprar e vender ferramentas e soluções que se integram ao GitHub. Isso inclui aplicativos para melhorar o fluxo de trabalho de desenvolvimento de software, como ferramentas de CI/CD, revisão de código, gerenciamento de projetos, entre outros. É um espaço para a comunidade encontrar e compartilhar ferramentas que complementam e melhoram suas experiências no GitHub.
- **GitHub Sponsors:** É uma maneira de apoiar financeiramente os desenvolvedores e organizações que criam software de código aberto. Através dele, indivíduos e empresas podem patrocinar o trabalho de desenvolvedores de código aberto, fornecendo-lhes recursos para continuar a desenvolver e manter seus projetos. É uma iniciativa importante para sustentar o ecossistema de código aberto.

- **GitHub Skills:** É uma série de cursos de aprendizado e materiais educativos projetados para ajudar os desenvolvedores a aprimorar suas habilidades no uso do GitHub e em práticas de desenvolvimento de software em geral. Estes recursos educacionais cobrem uma ampla gama de tópicos, desde o básico do Git até práticas mais avançadas de desenvolvimento e colaboração.

Esses recursos e iniciativas mostram o compromisso do GitHub em criar uma comunidade forte e sustentável, oferecendo ferramentas que apoiam o trabalho dos desenvolvedores, incentivando a inovação e o compartilhamento de conhecimento e recursos dentro do ecossistema de desenvolvimento de software.



## CAPÍTULO 4

# GIT HOOKS

O Git pode executar *scripts* antes ou depois de eventos como *commit* ou *push* para automatizar tarefas e operações personalizadas que são importantes para o fluxo de trabalho.

**G**it Hooks são úteis para diversas tarefas no versionamento de código como, por exemplo:

- **Validação de código:** Antes de um *commit* ou *push*, é possível usar um *hook* para executar uma verificação de estilo de código ou uma análise estática para garantir que o código siga as diretrizes de codificação.
- **Execução de testes automatizados:** Para garantir que o código que está sendo *commitado* não quebre nada, é possível configurar um *hook* para rodar testes automatizados.
- **Notificações:** Após um evento como um *push* ou *merge*, um *hook* pode ser usado para enviar notificações para uma equipe ou ferramentas de integração contínua.

- **Deploy automático:** Em alguns fluxos de trabalho, um *hook* no servidor pode ser usado para disparar um processo de *deploy* automático após o recebimento de novos commits.

Git Hooks são scripts que o Git executa antes ou depois de eventos como *commit*, *push* e *receive*. Eles são usados para automatizar tarefas e operações personalizadas que são importantes para o fluxo de trabalho de desenvolvimento de software. Eles são divididos em dois tipos:

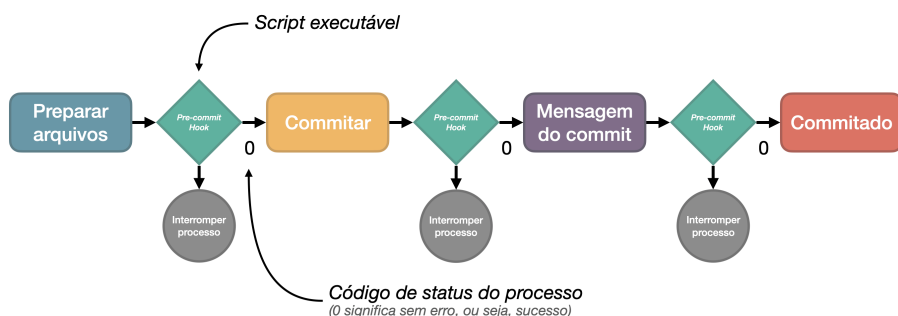
- **Client-side Hooks:** Executados no dispositivo local do desenvolvedor. Exemplos incluem "*pre-commit*", "*post-commit*", "*pre-push*", entre outros.
- **Server-side Hooks:** Executados no servidor onde os repositórios Git estão hospedados. Exemplos incluem "*pre-receive*", "*update*", "*post-receive*", entre outros.

Os *hooks* são armazenados no diretório `.git/hooks` de um repositório Git e podem ser escritos em qualquer linguagem de *script* que seu ambiente suporte (como *shell*, Python, Ruby, etc.). Por padrão, os *hooks* são desativados e precisam ser tornados executáveis para serem ativados.

O fluxo de commit padrão (sem Git Hooks) é o seguinte:



Por padrão, o Git permite que seja possível commitar qualquer arquivo, escrever qualquer mensagem no commit e não faz nenhuma verificação. Entretanto, é possível ativar os *hooks* para que *scripts* sejam executados no meio de ações que o Git faz. O fluxo de commit com Git Hooks ativo no repositório local do usuário (*client-side*) adiciona a possibilidade de executar *hooks* entre as ações.



Com esse novo esquema, Git Hooks podem ser usados para, por exemplo, verificar ou editar mensagens de commits, rodar testes da aplicação antes de cada commit ou antes de cada *push* para o repositório remoto, verificar e potencialmente arrumar erros de sintaxe no código usando *linters* (são ferramentas de análise de código estática, ou seja, eles sinalizam erros de programação, erros de estilo pré-definidos, bugs e construções suspeitas), gerar notificações após realizar um commit com sucesso, etc.

Quando o usuário cria um repositório, o Git cria junto uma pasta chamada `.git/hooks` e scripts de amostra dentro dela. Para habilitar um *hook* é preciso remover a extensão `.sample` do nome de algum arquivo que foi gerado. Por exemplo, ao gerar a pasta de *hooks*, o Git criou

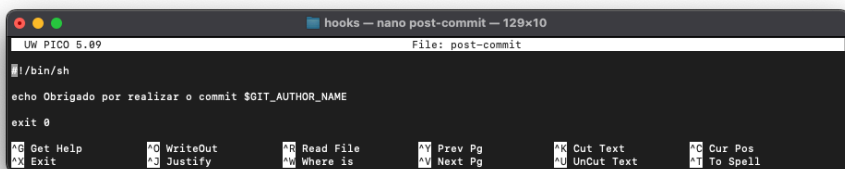
junto o `script commit-msg.sample`. Basta remover a extensão e o usuário terá o arquivo executável chamado de `commit-msg`. Caso ele queira adicionar um *script* customizado ou algum *script* que não foi gerado pelo Git, basta colocá-lo na pasta de *hooks*. Por exemplo, o *hook* de verificação pós-commit não é gerado por padrão e deve ser adicionado com o nome `post-commit` caso seja necessário efetuar uma verificação após o commit ser feito. Entretanto, ao tentar realizar um commit o Git irá gerar uma mensagem de aviso dizendo que o *hook* criado foi ignorado porque não está configurado para ser um arquivo executável. Para tornar um *hook* criado pelo usuário como executável é preciso executar o comando `chmod +x <arquivo>` no terminal. O comando basicamente torna o arquivo criado como executável para que, ao disparar o *script* de pós-commit, esse arquivo seja executado e possa realizar seu trabalho.

Todo *script* de *hook* precisa terminar com código 0 que indica sucesso (o processo continua) ou outro número de código que indica falha (o processo para naquele momento e não continua). Código 1 é comumente utilizado para indicar falha.

Para fins de demonstração simplificada do processo, o exercício será de criar um *script* de pós-commit, ou seja, um *script* que será rodado após um commit ser feito. O objetivo desse *script* será agradecer ao usuário por ter feito o commit.

Para isso, deve-se seguir os seguintes passos:

1. Criar um repositório novo ou abrir a pasta de um repositório existente.
2. Entrar na pasta `.git/hooks`.
3. Criar um novo arquivo dentro dela chamado `post-commit`.
4. Para que o script agradeça ao usuário é preciso escrever 2 comandos dentro do arquivo. O primeiro refere-se a mensagem que será dita no terminal e o segundo é o código de retorno. O comando `echo <mensagem>` faz esse trabalho. No caso desta demonstração será sempre sucesso, ou seja, código 0. O comando `exit <numero>` faz esse trabalho. A imagem abaixo mostra o resultado do conteúdo que o arquivo deve conter para atingir este objetivo:



The image shows a terminal window titled "hooks — nano post-commit — 128x10". The window displays the content of the `post-commit` file being edited in nano. The text inside the file is:

```
#!/bin/sh  
echo Obrigado por realizar o commit $GIT_AUTHOR_NAME  
exit 0
```

At the bottom of the window, the nano editor's status bar is visible, showing various keyboard shortcuts for editing and navigation.

5. Salvar o arquivo.
6. Tornar o arquivo criado em um arquivo executável através do comando `chmod +x <nome do arquivo>`. Neste caso, o comando é `chmod +x post-commit`.

7. Tudo pronto, agora é só realizar um commit e ver o *script* ser rodado.

Vale ressaltar que cada *hook* tem um nome específico que corresponde a um determinado evento no ciclo de vida do Git. Abaixo está a lista de alguns nomes comuns de *hooks* do Git e seus comportamentos:

- `pre-commit`: Executado antes de um commit ser finalizado. É comumente usado para realizar verificações de qualidade do código, como testes de *linter* ou testes unitários.
- `commit-msg`: Executado após a criação da mensagem de commit e antes do commit ser finalizado. Pode ser usado para validar ou formatar a mensagem de commit.
- `post-commit`: Executado imediatamente após um commit ser concluído. É útil para notificações ou outras tarefas que devem ocorrer após cada commit.
- `pre-push`: Executado antes que as mudanças sejam enviadas para um repositório remoto com `git push`. É comum usar este *hook* para rodar testes ou outras verificações antes de enviar o código.
- `pre-receive`: Executado no lado do servidor antes de aceitar commits enviados. Geralmente usado para controle de qualidade ou padrões de codificação no código que está sendo enviado.

- `post-receive`: Executado no lado do servidor após aceitar commits. Frequentemente utilizado para implementações automáticas ou notificações.

Cada um desses *hooks* é um *script* que pode ser escrito em qualquer linguagem de *script* que o seu ambiente suporta (como shell, Python, Perl, etc.), e o Git executará esses *scripts* nos momentos apropriados durante o seu fluxo de trabalho. Portanto, se o usuário nomear um arquivo de *hook* de forma incorreta, ele simplesmente não será acionado pelo Git, pois ele procura por arquivos com nomes específicos para cada tipo de evento.

Por padrão, *hooks* locais do Git não são enviados para o repositório remoto. Isso pode ser um problema, pois membros do time não conseguem sincronizar automaticamente os *hooks* da pasta `.git/hooks` entre dispositivos. Existem soluções para isso. A primeira seria criar um arquivo com o mesmo nome e conteúdo de um arquivo da pasta *hooks*, mas no Working Directory para que ele possa ser commitado. O problema disso é que todos os membros do time devem fazer um link simbólico manual para que esse arquivo seja interpretado pela pasta onde contém os *hooks*. A segunda solução seria usar pacotes ou aplicações que controlam os *hooks* do Git localizados na pasta `.git/hooks`. A configuração destes pacotes ficariam localizadas no Working Directory.

# GERENCIANDO HOOKS USANDO PACOTES

No mercado, os *hooks* são usados em ambas as formas: tanto através de arquivos nativos na pasta `.git/hooks` quanto por meio de pacotes. A escolha depende das necessidades específicas do projeto e da equipe. Cada abordagem possui suas vantagens.

Arquivos nativos na pasta `.git/hooks`:

- **Personalização e Controle Direto:** Usar arquivos nativos permite um alto grau de personalização. Os desenvolvedores podem escrever *scripts* específicos para suas necessidades.
- **Facilidade para Projetos Simples:** Para projetos menores ou menos complexos, essa abordagem pode ser mais direta e menos onerosa.
- **Dependência de Conhecimento da Equipe:** Requer que os membros da equipe tenham conhecimento sobre a criação e manutenção de *scripts* de *hook*.

Uso de pacotes (*packages*):

- **Frameworks e Ferramentas de Gerenciamento de Hooks:** Existem várias ferramentas e *frameworks*, como Husky, Lefthook, e pre-commit, que facilitam a configuração e o gerenciamento dos *hooks*.



- **Padronização e Facilidade de Uso:** Essas ferramentas oferecem uma maneira padronizada e mais fácil de configurar *hooks*, o que é especialmente útil em equipes grandes ou em projetos com muitos contribuidores.
- **Integração com Outras Ferramentas:** Muitas vezes, esses pacotes se integram bem com outras ferramentas de desenvolvimento, automatizando processos e garantindo a adesão a padrões de código.

Como visto acima, as considerações para a escolha de como usar *hooks* depende do tamanho e complexidade do projeto (projetos maiores podem se beneficiar do uso de pacotes devido à facilidade de manutenção e padronização), das necessidades específicas (se os requisitos do projeto exigem *hooks* altamente personalizados, a abordagem direta pode ser mais adequada) e experiência da equipe.

Existem vários pacotes e várias linguagens para cada pacote a fim de gerenciar *hooks* no Git. Um dos mais conhecidos é o Node.js. Node.js é uma plataforma de código aberto para execução de código JavaScript fora de um navegador da web. Para instalar o Node.js e o Node.js Package Manager (NPM) basta acessar o site <https://nodejs.org/en> e clicar no botão de download do Node.js recomendado para a maioria dos usuários. A instalação é simples e direta. Ao finalizar a instalação é possível verificar se tudo está funcionando corretamente digitando no terminal `node -v` (verifica se o Node.js foi instalado com sucesso) e `npm -v` (verifica se o Node.js Package Manager

foi instalado com sucesso). Se um número de versão do software aparecer significa que a ferramenta foi instalada.

Com a ferramenta instalada, é preciso entrar em um repositório local e inicializar o NPM digitando `npm init -y`. Este comando irá gerar um arquivo chamado `package.json` que irá conter informações dos *hooks* que os desenvolvedores quiserem adicionar a fim de serem instalados nos repositórios dos colaboradores. A partir daí, basta enviar as alterações criadas para o repositório remoto. Com isso, ao puxar o repositório para o dispositivo do usuário, ao digitar o comando `npm install` no terminal, todas as dependências registradas no arquivo `package.json` serão instaladas, incluindo os *hooks* que foram criados. Tudo pronto, agora é só criar *scripts* e adicioná-los ao pacote, enviar para o remoto e quando os outros colaboradores puxarem as mudanças e as instalarem, os *hooks* serão executados quando necessário.

Devido à complexidade do tema e ao fato de que a implementação de pacotes que gerenciam *hooks* excede o escopo deste capítulo, a instalação de um pacote gerenciador de *hooks* foi abordada apenas para fins demonstrativos (qualquer outro pacote gerenciador poderia ter sido utilizado). Portanto, uma explanação prática detalhada sobre a criação de *hooks* usando NPM se tornaria extensa e desviaria do foco principal deste capítulo, considerando que o estudo de Git Hooks em si já constitui um tópico que merece uma análise mais profunda.

## CAPÍTULO 5

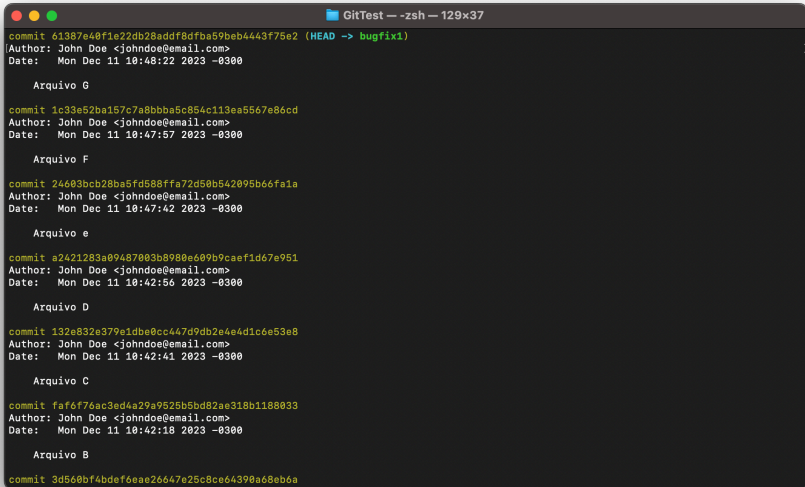
# GIT LOG FORMATADO

Exibir o log do Git é muito útil e informativo, ainda mais se for exibido de forma fácil e legível. Este capítulo demonstra maneiras elegantes de se listar o histórico do Git.

**V**erificar o histórico de commits é muito importante, pois ele contém várias informações valiosas sobre autor dos commits, data de criação, mensagem, hash, etc. Entretanto, o fluxo do histórico tende a ficar cada vez mais complicado de acordo com a quantidade de commits e branches criadas no projeto. Para isso, é essencial dominar técnicas e práticas de visualizar o histórico de maneira fácil, intuitiva e de rápida compreensão das informações essenciais ao desenvolvedor em um dado momento.

Este capítulo irá demonstrar algumas maneiras de se exibir o histórico ou *log* do Git com alguns comandos e parâmetros. Segue abaixo alguns comandos, suas formatações e o significado delas:

- `git log`: A operação mais básica para visualização de *logs* do Git. Este comando mostra o histórico de commits para a branch atual, iniciando pelo commit mais recente. O resultado do comando para cada commit mostra o hash do commit, autor, data e mensagem do commit.



```
GitTest -- zsh -- 129x37
commit 61387e4df1e22db78addf8dfba59be6443f75e2 (HEAD -> bugfix1)
Author: John Doe <johndoe@email.com>
Date: Mon Dec 11 10:48:22 2023 -0300

Arquivo G

commit 1c3e52ba157c7a8bbba5c854c113ea5567e86cd
Author: John Doe <johndoe@email.com>
Date: Mon Dec 11 10:47:57 2023 -0300

Arquivo F

commit 24683bcb28ba5fd588ffa72d50b542095b66fa1a
Author: John Doe <johndoe@email.com>
Date: Mon Dec 11 10:47:42 2023 -0300

Arquivo e

commit a2421282a09407003b0900e409b9caef1d67e951
Author: John Doe <johndoe@email.com>
Date: Mon Dec 11 10:42:56 2023 -0300

Arquivo D

commit 132e832e379e1dbe0cc447d9db2e4e4d1c6e53e8
Author: John Doe <johndoe@email.com>
Date: Mon Dec 11 10:42:41 2023 -0300

Arquivo C

commit fa6f76ac3ed4a27a97525b5d82ae318b1180033
Author: John Doe <johndoe@email.com>
Date: Mon Dec 11 10:42:18 2023 -0300

Arquivo B

commit 3d560bf4bdef6aae26647e25c8ce64390a68eb6a
```

- `git log [--after | --before | --since | --author]`: Quando se é trabalhado com repositórios grandes, o resultado do histórico pode ficar extenso. Este comando, junto de algum atributo, filtra o histórico de commits por data, autor, etc. Um dos mais utilizados é com o atributo `--since`.
- `git log --graph`: O histórico pode ser exibido uma forma gráfica com o parâmetro `--graph`. Neste modo, cada commit é representado por um todo. Os nodos são

conectados por linhas que indicam a branch e histórico de merge. Este comando é particularmente útil para visualizar o histórico do desenvolvimento de um repositório.

```
GitTest -- -zsh -- 129x30
* commit b2e8b64b7e2798ef1ea23329d01f9cdbea8a8d4 (HEAD -> main)
  Merge: 516829a 941fe38
  Author: AnndersonOreto <anndersonp@gmail.com>
  Date: Mon Dec 11 10:46:31 2023 -0300

    Merge branch 'feature1'

* commit 941fe38d1c8885843ba3e04b4f8ad870189f11f3 (feature1)
  Author: AnndersonOreto <anndersonp@gmail.com>
  Date: Mon Dec 11 10:42:56 2023 -0300

    Arquivo D

* commit 13bb59732847422bffd26a781d83c1ecbd219dc7
  Author: AnndersonOreto <anndersonp@gmail.com>
  Date: Mon Dec 11 10:42:41 2023 -0300

    Arquivo C

* commit 516829a7ecf111df4b6f995c5519ff7b6c1ad8
  Author: AnndersonOreto <anndersonp@gmail.com>
  Date: Mon Dec 11 10:42:18 2023 -0300

    Arquivo B

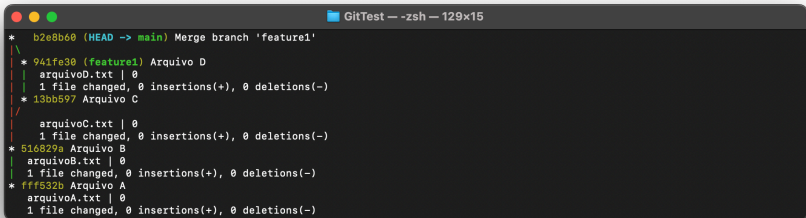
* commit fff532b016131059de3b12dd22b57544a4c3f287
  Author: AnndersonOreto <anndersonp@gmail.com>
  Date: Mon Dec 11 10:42:04 2023 -0300

    Arquivo A
```

- `git log --graph --oneline`: Produz o mesmo resultado do comando acima com a pequena diferença de que todos os commits são representados em apenas uma linha. Isso deixa o histórico enxuto para uma melhor visualização em casos de vários commits no histórico.

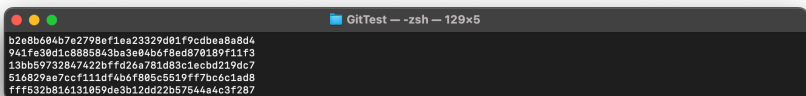
```
GitTest -- -zsh -- 129x7
* b2e8b60 (HEAD -> main) Merge branch 'feature1'
  /
  * 941fe38 (feature1) Arquivo D
  * 13bb597 Arquivo C
  /
* 516829a Arquivo B
* fff532b Arquivo A
```

- `git log --graph --oneline --stat`: Produz o mesmo resultado do comando acima com diferença de que este comando mostra também a quantidade de mudanças, inserções e exclusões de arquivos. O parâmetro `--stat` serve para mostrar estatísticas sobre as mudanças feitas em cada commit.



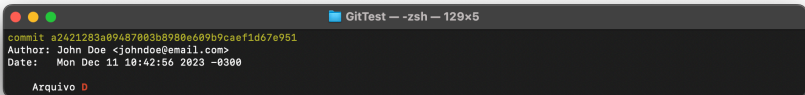
```
GitTest -- zsh -- 129x15
* b2e8b50 (HEAD -> main) Merge branch 'feature1'
* 941fe30 (feature1) Arquivo D
  arquivoD.txt | 0
  1 file changed, 0 insertions(+), 0 deletions(-)
* 13bb597 Arquivo C
  arquivoC.txt | 0
  1 file changed, 0 insertions(+), 0 deletions(-)
* 51a829a Arquivo B
  arquivoB.txt | 0
  1 file changed, 0 insertions(+), 0 deletions(-)
* fff532b Arquivo A
  arquivoA.txt | 0
  1 file changed, 0 insertions(+), 0 deletions(-)
```

- `git log --pretty="format:%H"`: O histórico do Git pode ser customizado a ponto de mostrar quaisquer informações específicas em qualquer formato. O parâmetro responsável pela customização é `--pretty` sucedido de uma formatação. Por exemplo, o comando acima mostra apenas o hash do commit. Existe um site que possui todas as combinações e possibilidades de formatação para todos os gostos. Para mais informações visite <https://devhints.io/git-log-format> e <https://git-scm.com/docs/pretty-formats>.



```
GitTest -- zsh -- 129x5
b2e8b504b7e2798ef1ea23329d01f9cdbea8a8d4
941fe30d1c8885843ba3e04b6f8ed870189f11f3
13bb59732847422bffd26a781d83c1ecbd219dc7
51a829ae7ccf111d74b6f885c5b519ff7bc0c1ad8
fff532b01e131889de3b12cd22b544a4c37207
```

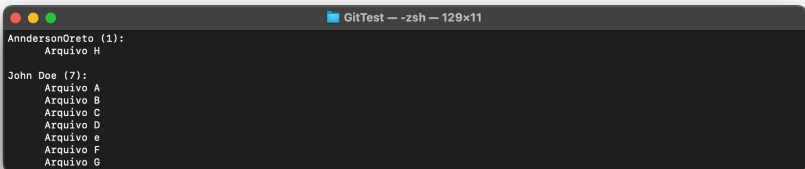
- `git log --grep="D"`: Este comando filtra commits pela mensagem dos commits. Se o desenvolvedor quiser procurar apenas commits que estejam relacionados a *bugs*, é possível que na mensagem esteja contida essa palavra. Por exemplo, o comando acima procura todos os commits com a letra "D" inclusa.



```
GitTest -- zsh -- 129x5
commit a2421283a094a87083b8980e409b9caef1d67e951
Author: John Doe <johndoe@email.com>
Date: Mon Dec 11 10:42:56 2023 -0300

Arquivo D
```

- `git shortlog`: Este comando é uma versão especial do `git log` feita para criar anúncios de lançamentos. Ele agrupa cada commit por autor e mostra a primeira linha de cada mensagem de commit. Esta é uma maneira fácil de ver quem está trabalhando em que.



```
GitTest -- zsh -- 129x11
AndersonDreto (1):
Arquivo H

John Doe (7):
Arquivo A
Arquivo B
Arquivo C
Arquivo D
Arquivo e
Arquivo F
Arquivo G
```

É possível visualizar apenas commits de merge com o comando `git log --merges`. O Git é cheio de comandos e parâmetros para melhorar a visualização e compreensão do histórico de commits. Para mais informações e detalhes sobre os possíveis parâmetros, visite o site <https://git-scm.com/docs/git-log>.

## CAPÍTULO 6

# GARBAGE COLLECTION

Esta ferramenta é muito útil na questão de performance e melhora de memória do Git. Com ela, é possível remover nós inalcançáveis e arquivos que não são mais usados.

**E**m um repositório no Git existem operações que podem ser feitas sobre commits que os deletam, os renomeiam gerando outro hash e inutilizando o commit anterior, que os reverterem e que os deslocam. O que essas operações têm em comum? A maioria delas podem fazer com que um commit seja esquecido na árvore criada no repositório pelo Git ao longo do tempo, deixando-o inacessível. Quando isso acontece, esse commit que não será mais usado fica ocupando memória no repositório. É aí que entra em ação o coletor de lixo (comumente chamado de *Garbage Collector*). O *Garbage Collector* (GC) no Git é uma ferramenta importante que ajuda a otimizar o repositório e melhorar o desempenho. Através dele, os commits que



estão ocupando espaço são removidos permanentemente do repositório, liberando espaço na memória.

Como isso funciona? Como visto anteriormente, no Git, os objetos que são salvos incluem commits, árvores, *blobs* (arquivos) e *tags*. Cada objeto é identificado por um hash SHA-1 e armazenado no repositório. Com o tempo, um repositório Git pode acumular objetos que não são mais necessários. Isso pode acontecer por várias razões, como a exclusão de branches ou a reescrita do histórico de commits. Esses objetos desnecessários podem ocupar espaço e reduzir a eficiência do repositório. O GC é um mecanismo do Git que identifica objetos inatingíveis, ou seja, objetos que não são acessíveis a partir de nenhuma das branches ou tags atuais. Isso inclui commits que foram substituídos por um rebase, por exemplo. Esses objetos inatingíveis não são imediatamente excluídos. Eles geralmente têm um prazo de expiração que, por padrão, é de 30 dias, após o qual o GC os considerará para exclusão. Esse período é um padrão para dar ao usuário uma janela de segurança, caso precisem recuperar algo que foi acidentalmente excluído ou alterado. Além de remover objetos inatingíveis, o GC também otimiza o repositório comprimindo arquivos e consolidando pacotes de objetos para economizar espaço e melhorar o desempenho.

A execução do *Garbage Collector* pode acontecer automaticamente ou manualmente. Em muitos casos, o Git executa o GC automaticamente em segundo plano após certas operações para manter a eficiência do repositório. Os usuários também podem invocar o GC manualmente

usando o comando `git gc`. Isso pode ser útil para manutenção periódica ou para limpar depois de grandes reestruturações no repositório. Tenha em mente que, a partir do momento que o usuário executa o comando, se ele salvou o hash de um commit para acessá-lo depois, ele não terá êxito, pois o commit será removido (caso ele seja inacessível). Em relação ao GC, note que:

- O prazo de 30 dias é uma configuração padrão que determina quanto tempo um objeto inatingível fica no repositório antes de ser elegível para exclusão pelo GC. O usuário pode alterar esse prazo com configurações do Git.
- O Git não roda o comando `git gc` a cada 30 dias. Ele decide quando executar o GC automaticamente com base em vários critérios, como o número de objetos soltos no repositório. Não há um intervalo de tempo fixo para essa execução automática. Entretanto, é possível dizer que isso geralmente acontece após certas operações que podem deixar muitos objetos inatingíveis, como fusões e *rebase* de branches.

Em resumo, o *Garbage Collector* é uma ferramenta de manutenção que ajuda a manter o repositório limpo e eficiente, removendo objetos desnecessários e otimizando os dados armazenados. Essa otimização já acontece nos repositórios porque o Git possui seu próprio sistema de armazenamento inteligente. Ao usar o GC, o que o Git faz é reestruturar os dados salvos no repositório de maneira que,

se ele encontrar uma melhor maneira de comprimir os dados, eles ocupem menos espaço e de forma otimizada.

## CAPÍTULO 7

# MASTERIZANDO HEADS

Este capítulo explica os conceitos de HEAD, MERGE\_HEAD, FETCH\_HEAD, ORIGIN\_HEAD e detached HEAD.

**N**o Git, existem vários termos técnicos que se referem a diferentes tipos de referências ou estados em que um repositório pode se encontrar. Basicamente uma HEAD é um ponteiro para um commit ou branch. Existem vários tipos de referências para diversos cenários gerais ou específicos. Para compreender o que cada uma significa, abaixo segue a lista dos tipos de referências encontradas no Git e GitHub e suas respectivas funções:

- **HEAD:** É uma referência ao último commit na branch atual em que o usuário está trabalhando. Em outras palavras, é um ponteiro para o commit mais recente na branch. Quando um commit é feito, o HEAD avança para esse novo commit.

- **MERGE\_HEAD:** Quando o usuário inicia um processo de merge (fusão) de branches, o Git cria uma referência chamada *MERGE\_HEAD*. Esta referência aponta para o commit com o qual o usuário está tentando fazer o merge. Essencialmente, é um marcador temporário usado durante o processo de merge. Por exemplo, ao criar uma branch chamada *feature3*, criar um commit nela com mudanças, fazer *checkout* para a branch *main*, criar um commit nela com outras mudanças e unir as mudanças com `git merge feature3`, a *MERGE\_HEAD* terá o hash do último commit da branch *feature3* (que é a branch fonte na qual se quer unir as mudanças). Isso acontece porque, caso ocorram conflitos de merge, o Git precisa saber qual era a branch na qual ele estava fazendo o processo de merge, para que o commit em questão não seja perdido ao concluir a resolução de conflitos.
- **FETCH\_HEAD:** Quando o usuário faz um *fetch* de um repositório remoto, o Git armazena o último commit buscado de cada branch remota em uma referência chamada *FETCH\_HEAD*. Isso é útil para ver o que foi trazido do repositório remoto antes de integrar essas mudanças ao seu repositório local.
- **ORIGIN\_HEAD:** Este é um ponteiro para o commit mais recente buscado de um repositório remoto chamado *origin*. *origin* é normalmente o nome padrão dado ao repositório remoto a partir do qual o usuário clonou. *ORIGIN\_HEAD* ajuda a manter o controle do estado mais recente do repositório remoto.

- **REBASE\_HEAD:** Este é um ponteiro usado durante o processo de *rebase*. Quando um *rebasing* é iniciado, o Git usa este ponteiro para manter o controle do estado atual da operação. Durante um *rebase*, **REBASE\_HEAD** aponta para o commit que está sendo trabalhado no momento. Ele ajuda a entender até onde o processo de *rebase* chegou e gerenciar o estado interno do *rebase*.
- **Detached HEAD:** Este é um estado especial no Git onde o HEAD não aponta para o final de uma branch, mas sim para um commit específico. Isso geralmente acontece quando o usuário faz *checkout* para um commit específico em vez de uma branch. Em um estado de *detached HEAD*, é possível fazer commits, mas esses commits não pertencem a nenhuma branch e podem ser difíceis de encontrar se o usuário voltar para uma branch normal depois.

## CAPÍTULO 8

# GITHUB PAGES

Serviço de hospedagem gratuito de sites estáticos diretamente em repositórios do GitHub ideal para página de projetos , portfólio e documentações.

**O** GitHub Pages é um serviço de hospedagem gratuito que permite aos usuários hospedar sites estáticos diretamente a partir de seus repositórios no GitHub. Ele é amplamente utilizado para hospedar páginas de projetos, portfólios pessoais, blogs e até mesmo documentações de projetos. Ele é especialmente popular entre desenvolvedores, pesquisadores, estudantes e profissionais de TI que já estão familiarizados com o GitHub e querem uma maneira simples e eficiente de criar e hospedar seus sites. Abaixo estão alguns motivos para se usar o Pages do Github:

- **Facilidade de Uso:** Se o usuário já usa o GitHub para controle de versão, o Pages se integra perfeitamente aos fluxos de trabalho existentes. É possível facilmente transformar um repositório em um site.

- **Gratuito:** Não há custos para hospedar o site, o que é ideal para projetos pessoais, portfólios ou pequenos projetos.
- **Suporte ao Jekyll:** O Pages tem suporte integrado ao Jekyll, um gerador de sites estáticos, o que permite criar sites dinâmicos sem a necessidade de um *back-end*.
- **Customização de Domínio:** Permite ao usuário usar seu próprio nome de domínio, oferecendo uma aparência mais profissional ao site.
- **Integração com Git:** Como parte do ecossistema do GitHub, ele se beneficia da robustez e dos recursos do Git, como controle de versão, *pull requests* e branches, para gerenciar o conteúdo do site.
- **Segurança:** Ele também fornece HTTPS automático para o site, garantindo que o tráfego seja criptografado e seguro.
- **Comunidade e Suporte:** Sendo parte do GitHub, o usuário se beneficia de uma grande comunidade de desenvolvedores e de uma vasta quantidade de documentação e fóruns de suporte.
- **Rápida Configuração e Implementação:** É possível configurar um site em minutos, especialmente útil para apresentar rapidamente um projeto ou ideia.



Criar um site com o Pages é um processo relativamente simples e direto. Para criar um site, primeiro é preciso criar um repositório no GitHub. O nome do repositório para sites pessoais ou de organização deve ser "<nome-do-usuário>.github.io". Para projetos, pode ser qualquer nome. Com o repositório criado, é preciso clona-lo para a máquina local e adicionar os arquivos do site (HTML, CSS, JavaScript, etc.) no repositório. No caso de usar geradores estáticos como Jekyll, é preciso configurar conforme necessário. Após as alterações serem feitas, é preciso fazer commit e *push* das modificações para o repositório remoto. Na página do repositório no GitHub, é preciso clicar em 'Settings' ou 'Configurações', acessar a seção 'Pages' e na parte de Branch escolher a branch em que os arquivos do site se encontram para hospedá-lo. Por fim, basta clicar em 'Salvar' e, uma vez que o Pages esteja habilitado e o conteúdo publicado, acessar o site <nome-do-usuário>.github.io para um site pessoal ou de organização e <nome-do-usuário>.github.io/<nome-do-repositório> para repositório de projeto.

**Neste módulo, foram explorados aspectos avançados do Git e GitHub, incluindo a eficácia do *fork* e a personalização por meio de *hooks*, além da automação e integração contínua com CI/CD, ampliando as habilidades de colaboração e eficiência. Foram abordadas funcionalidades sofisticadas do GitHub que potencializam a colaboração, imergindo nas nuances do histórico de commits formatado para um controle de versão mais perspicaz, e revelando o funcionamento interno do Git, incluindo o *Garbage Collector* e os diversos tipos de "HEADs". Com o domínio desses conceitos, torna-se possível enfrentar desafios complexos de desenvolvimento e colaboração, marcando um passo significativo na jornada como profissionais de software.**