

# Resumo Git e Github

---

## # Sumário

### 1. Introdução

- O que é controle de versão e por que é importante.
- Diferença entre controle de versão centralizado e distribuído.
- Apresentação do Git e GitHub.

### 2. Entendendo Git

- Instalação do Git.
- Configuração básica do Git.
- Os três estados de um arquivo no Git: Modificado, preparado e confirmado (committed).

### 3. Principais Comandos Git

- `git init` : Inicializando um repositório local.
- `git clone` : Clonando um repositório existente.
- `git add` : Adicionando arquivos ao stage.
- `git commit` : Confirmando alterações.
- `git status` e `git diff` : Verificando o estado dos arquivos.
- `git log` : Visualizando histórico de commits.

### 4. Ramificação (Branching) e Mesclagem (Merging)

- `git branch` : Criando e listando branches.
- `git checkout` : Trocando entre branches.
- `git merge` : Juntando alterações de diferentes branches.

### 5. Trabalhando com GitHub

- Criando uma conta no GitHub.
- Como criar um novo repositório no GitHub.
- `git push` : Enviando alterações para o GitHub.
- `git pull` : Obtendo alterações do GitHub.
- `git fetch` : Sincronizando seu repositório local com o repositório remoto.

### 6. Contribuindo com Projetos no GitHub

- Forking e Cloning: Contribuindo para projetos open source.
- Pull requests: Como sugerir mudanças em outros repositórios.

### 7. Boas práticas com Git e GitHub

- Mensagens de commit significativas.
- Uso adequado do `.gitignore`.

- Mantendo os branches atualizados com as mudanças.

## 8. Ferramentas de Suporte no GitHub

- GitHub Actions: Automatizando tarefas.
  - GitHub Pages: Hospedando seu site ou blog.
- 

# # Introdução

## I - O que é controle de versão e por que é importante?

O controle de versão, também conhecido como controle de revisão, é um sistema que registra alterações feitas em um arquivo ou conjunto de arquivos ao longo do tempo, de modo que você possa recuperar versões específicas mais tarde.

O controle de versão é extremamente útil em vários aspectos, particularmente em projetos de desenvolvimento de software, para várias pessoas:

1. Permite que várias pessoas trabalhem simultaneamente em um projeto sem se preocuparem com conflitos de versão.
2. Os desenvolvedores podem reverter seu projeto para um estado anterior, caso algo dê errado.
3. Proporciona um log de quem fez o quê e quando, o que é útil para entender por que as alterações foram feitas e por quem.
4. Ajuda na experimentação, pois os desenvolvedores podem criar branches separados para tentar novas ideias e mesclá-las de volta, se bem-sucedidas.

## II - Diferença entre controle de versão centralizado e distribuído

Existem dois tipos principais de sistemas de controle de versão: Centralizado (CVCS) e Distribuído (DVCS).

- **Controle de Versão Centralizado (CVCS)**: Nesse tipo de sistema, há um servidor central que contém todas as versões do projeto, e os colaboradores obtêm a última versão do projeto desse servidor central. Os exemplos mais comuns deste tipo de sistema são o Subversion (SVN) e o CVS. A desvantagem deste sistema é que se o servidor central falhar, ninguém poderá colaborar ou salvar versões de seu trabalho.
- **Controle de Versão Distribuído (DVCS)**: Nesse tipo de sistema, cada colaborador possui uma cópia local completa do histórico do projeto. Isso significa que, mesmo se um servidor falhar, o repositório completo pode ser restaurado a partir de qualquer um dos colaboradores. Além disso, esses sistemas lidam melhor com a colaboração entre múltiplos repositórios. O Git é um exemplo de DVCS.

## III - Apresentação do Git e GitHub

- **Git**: É um sistema de controle de versão distribuído de código aberto. Ele foi projetado para lidar com tudo, desde pequenos a muito grandes projetos com

velocidade e eficiência. Foi criado por Linus Torvalds em 2005 para o desenvolvimento do kernel do Linux.

- **GitHub:** É uma plataforma de hospedagem de código-fonte com controle de versão usando o Git. Ele permite que você colabore com outros em seu projeto, tornando-o fácil de compartilhar seu projeto e trabalhar com outras pessoas. Além disso, possui várias funcionalidades que facilitam a gestão de projetos de software, como issues tracking, pull requests, actions para CI/CD e muito mais.
- 

## # Entendendo Git

### 1. Instalação do Git:

- **Windows:** Você pode baixar o Git do site oficial <https://git-scm.com/downloads>. O instalador é um executável que te guiará pelas etapas de instalação.
- **Mac:** Também pode ser baixado do site oficial, ou instalado através do gerenciador de pacotes Homebrew, com o comando `brew install git` no terminal.
- **Linux:** A maioria das distribuições Linux inclui o Git nos repositórios padrão, então você pode instalá-lo com o gerenciador de pacotes da sua distribuição. Por exemplo, no Ubuntu, você usaria `sudo apt-get install git`.

### 2. Configuração básica do Git:

Após a instalação, é importante configurar algumas informações básicas. Os principais comandos de configuração são:

- `git config --global user.name "Seu Nome"` : Configura seu nome de usuário, que será associado a todos os seus commits.
- `git config --global user.email "seu-email@exemplo.com"` : Configura o email que será associado a todos os seus commits.

Para verificar suas configurações, você pode usar `git config --list`.

### 3. Os três estados de um arquivo no Git: Modificado, preparado e confirmado (committed):

Os arquivos em um repositório Git podem estar em um de três estados: modificado, preparado (staged) e confirmado (committed).

- **Modificado:** Significa que você alterou o arquivo mas ainda não o preparou para a próxima versão. O Git sabe que o arquivo mudou, mas ainda não tomou nota dessas mudanças para o próximo commit.
- **Preparado:** Significa que você marcou um arquivo modificado em sua versão atual para ser incluído em seu próximo commit. Você faz isso com o comando `git add`.

- **Confirmado (Committed):** Significa que os dados estão armazenados em seu banco de dados local. Quando você faz um commit, o Git grava um snapshot de todos os arquivos que estão no estado preparado, e esse conjunto de arquivos torna-se um commit.
- 

## # Principais Comandos Git

**git init:** Cria um repositório vazio ou converte uma pasta existente sem controle de versão em um repositório.

**git clone https://url-do-link:** Baixa o código-fonte de um repositório remoto. Para desvincular sua cópia do original, use **git remote rm origin**.

**git add:** Adiciona as alterações de um arquivo para o próximo commit.

- Para adicionar apenas um arquivo: **git add arquivo**
- Para adicionar todos os arquivos modificados: **git add -A** ou **git add .**

**git commit:** Cria um ponto de verificação no desenvolvimento.

- Para adicionar uma mensagem ao commit: **git commit -m "mensagem explicando a mudança no código"**
- Para adicionar e inserir a mensagem do commit: **git commit -a -m 'commit message'**

**git status:** Mostra informações sobre a branch atual, incluindo se está atualizada em relação à master e quais arquivos foram alterados.

**git diff:** Compara as fontes de dados Git e mostra as linhas adicionadas e removidas.

**Git log:** Permite visualizar o histórico de commits ou um específico.

- Parâmetro **-pretty=format:** para customizar apresentação de logs
- Opções específicas:
  - **%h:** hash reduzido do commit
  - **%d:** branch e tag do commit
  - **%s:** mensagem do commit
  - **%cn:** nome do autor do commit
  - **%cr:** data relativa do commit
- Utilizar cores diferentes para facilitar visualização (**%C**)

Exemplo: `git log --pretty=format:'%C(blue)%h%C(red)%d %C(white)%s - %C(cyan)%cn, %C(green)%cr'`

**git log --oneline:** É uma maneira mais compacta e resumida de visualizar o histórico de commits.

**git revert:**

- Desfaz commits: **git revert 'número do hash'**
- Para obter o número do hash utiliza-se: **git log -- oneline**

**Git stash:** salva alterações sem commit.

- Salvar alterações: **git stash**
- Listar stashes: **git stash list**
- Aplicar stash específico: **git stash apply stash@{2}.**

## # Ramificação (Branching) e Mesclagem (Merging)

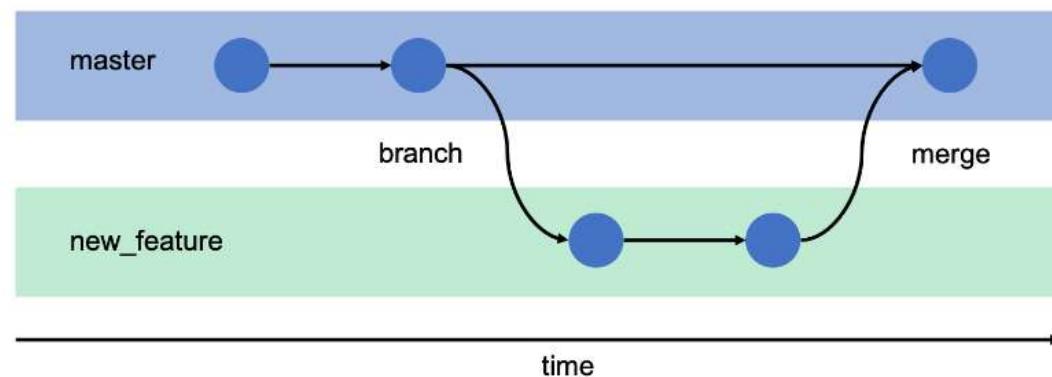
**git branch:** Cria, lista, exclui e gerencia branches.

- Para criar uma nova branch local: **git branch nome-da-branch**
- Para enviar a nova branch ao repositório remoto: **git push -u remote nome-da-branch**
- Para listar ramificações: **git branch** ou **git branch --list**
- Para excluir uma branch: **git branch -d nome-da-branch**

**git checkout:** Muda de uma branch para outra ou verifica arquivos e commits.

- Para mudar de ramificação: **git checkout nome-da-ramificação**
- Atalho para criar e mudar para uma nova branch: **git checkout -b nome-da-branch**

**git merge:** Juntando as alterações de diferentes branches. **git merge nome-da-branch**



## # Trabalhando com GitHub

**Criando uma conta no GitHub:**

1. Visite <https://github.com/>.
2. Clique em "Sign Up".
3. Escolha um nome de usuário único, insira seu endereço de e-mail e escolha uma senha.
4. Você terá a opção de escolher um plano. Para uso geral, o plano gratuito é suficiente.
5. Leia e aceite os termos de serviço e clique em "Complete setup".
6. Em seguida, você será solicitado a verificar seu endereço de e-mail. Vá para seu e-mail e clique no link de verificação enviado para você pelo GitHub.

### **Como criar um novo repositório no GitHub:**

1. Após entrar na sua conta do GitHub, clique em "+" no canto superior direito e selecione "New repository".
2. Escolha um nome para seu repositório.
3. Escolha se deseja que este repositório seja público (visível para todos) ou privado (visível apenas para você e colaboradores convidados).
4. Você pode optar por inicializar o repositório com um README, um .gitignore e/ou uma licença.
5. Clique em "Create repository".

### **Comandos para criar um novo repositório local e conectar ao remoto.**

```
git init
git add .
git commit -m "first commit"
git branch -M main
git remote add origin <link do repositório>
git push -u origin main
ou envie um repositório existente a partir da linha de comando.
```

```
git remote add origin <link do repositório>
git branch -M main
git push -u origin main
```

### **Comandos de linha git push, pull e fetch**

#### **git push : Enviando alterações para o GitHub:**

Depois de fazer commit das suas alterações localmente, você pode enviá-las para o GitHub com `git push`. Aqui está um exemplo básico:

```
git push origin master
```

Este comando envia os commits na branch "master" do seu repositório local para o repositório "origin" no GitHub. Você pode substituir "master" pelo nome da branch que deseja enviar.

Para enviar alterações a um branch específico:

```
git push remote nome-do-branch
```

Para fazer upload de um novo branch:

```
git push -u origin nome-do-branch
```

ou

```
git push --set-upstream origin nome-do-branch
```

#### **git pull : Obtendo alterações do GitHub:**

`git pull` é usado para buscar as últimas alterações de um repositório remoto e mesclá-las com o seu trabalho atual. Por exemplo:

```
git pull origin master
```

Este comando puxa as alterações da branch "master" do repositório "origin" e as mescla com a branch atual no seu repositório local.

#### **git fetch : Sincronizando seu repositório local com o repositório remoto:**

`git fetch` é usado para buscar todas as alterações do repositório remoto que não existem no seu repositório local e armazená-las localmente. Diferentemente do `git pull`, o `git fetch` não mescla as alterações automaticamente. Você terá que usar `git merge` para mesclar as alterações, se quiser.

```
git fetch origin
```

Este comando busca todas as novas informações (branches, commits etc.) do repositório "origin", mas não altera seu trabalho atual. Isso é útil para verificar as alterações antes de realmente mesclá-las com o seu trabalho.

---

## # Contribuindo com Projetos no GitHub

### Forking e Cloning: Contribuindo para projetos open source

- Forking:** O Forking é o processo de fazer uma cópia de outro repositório para a sua conta no GitHub. Esta é uma maneira comum de contribuir para projetos open source. Para fazer um fork de um repositório, vá até a página do repositório e clique no botão "Fork" no canto superior direito.
- Cloning:** Depois de fazer um fork de um repositório, você pode fazer um clone dele para o seu computador local para trabalhar no código. Para fazer isso, clique no botão "Clone or download" no repositório que você forked, copie a URL, e então no terminal, digite `git clone [URL do repositório]`.

Agora, você tem uma cópia local do repositório e pode começar a fazer alterações no código.

### Ferramentas e Funcionalidades Essenciais para Colaboração e Gerenciamento de Projetos no GitHub

- Issues:** As issues do GitHub são uma ótima maneira de acompanhar tarefas, melhorias e bugs para seus projetos. Elas são como um fórum de discussão para ideias e podem

ser atribuídas a colaboradores, rotuladas para fácil categorização e referenciadas em pull requests e commits.

**2. Actions:** O GitHub Actions é uma ferramenta de CI/CD (Integração Contínua / Entrega Contínua) que permite automatizar tarefas como testes de software e implantação de projetos. Com ele, você pode criar fluxos de trabalho personalizados que são acionados com base em eventos específicos em seu repositório, como quando um commit é feito para uma branch específica ou quando uma issue é criada.

**3. Projects:** A ferramenta de Projects do GitHub é como um quadro de Kanban para o seu repositório. Ela permite criar quadros com colunas personalizadas (como "Para fazer", "Em progresso" e "Feito") e adicionar cartões a essas colunas para acompanhar o progresso das tarefas. Esses cartões podem ser ligados a issues e pull requests para uma melhor organização do projeto.

**4. Wiki:** Cada repositório do GitHub vem com uma seção de Wiki que pode ser usada para hospedar documentação do projeto, guias de contribuição, FAQs e qualquer outra informação relevante. As Wikis suportam a marcação Markdown, então você pode formatar sua documentação de maneira limpa e legível.

**5. Insights:** Insights é uma seção do GitHub que fornece informações analíticas sobre seu repositório. Ela fornece dados sobre atividade do repositório, como visualizações de página, clones, participação da comunidade, tempo para resolver issues e muito mais.

**6. Pull Requests:** Um Pull Request é uma sugestão de alteração no repositório original enviada por um colaborador. Ele permite que você mostre suas alterações a outros colaboradores, discuta sobre possíveis modificações e, uma vez aprovado, essas alterações podem ser integradas (ou "mergeadas") ao projeto original.

Os Pull Requests permitem que múltiplas pessoas trabalhem no mesmo projeto sem conflitos, pois cada colaborador trabalha em sua própria "branch" (uma versão do projeto) e sugere alterações via Pull Requests. Uma vez que um Pull Request é criado, outros colaboradores podem revisá-lo, discutir alterações e até mesmo sugerir alterações adicionais antes de ser mergeado.

Os Pull Requests também têm integração com outros recursos do GitHub, como Issues e Actions. Por exemplo, você pode mencionar uma Issue específica em um Pull Request e, quando o Pull Request for mergeado, a Issue será automaticamente fechada. Além disso, as Actions do GitHub podem ser configuradas para realizar testes automaticamente quando um Pull Request é criado ou alterado.

Assim, os Pull Requests são uma parte fundamental do fluxo de trabalho colaborativo no GitHub.

---

## # Boas práticas com Git e GitHub

1. **Faça commits pequenos e frequentes:** Cada commit deve ser uma unidade de trabalho autossuficiente que inclua apenas as alterações necessárias para realizar uma tarefa específica. Evite grandes commits que abranjam muitas tarefas diferentes.
2. **Escreva mensagens de commit claras e descritivas:** A mensagem de commit deve resumir as alterações que foram feitas e o motivo dessas alterações. Isso facilita a revisão do histórico do commit e a compreensão do que cada commit faz.
3. **Use Branches para tarefas/seções específicas:** Use branches para trabalhar em novas funcionalidades ou bugs. Isso mantém seu trabalho organizado e evita que alterações de diferentes tarefas se misturem.
4. **Mantenha a branch principal (geralmente "master" ou "main") limpa:** A branch principal deve ser mantida "limpa" e funcional. As alterações só devem ser mescladas nela após terem sido testadas em outra branch.
5. **Faça rebase frequentemente:** O rebase ajuda a manter um histórico de commit limpo, já que evita a criação de commits de merge desnecessários. Além disso, facilita a manutenção do código atualizado com as últimas alterações.
6. **Use Pull Requests para revisão de código:** Pull Requests permitem que outros colaboradores revisem seu código antes de ser mesclado. Isso ajuda a manter a qualidade do código e a evitar bugs.
7. **Responda e solucione Issues:** As Issues são uma maneira útil de rastrear bugs e solicitações de funcionalidades. Responda a elas prontamente e tente resolvê-las em um prazo razoável.
8. **Adicione um README e outros documentos úteis:** Um bom README fornece informações sobre o que é o projeto, como instalá-lo, como usá-lo e como contribuir para ele. Outros documentos, como CONTRIBUINDO, CÓDIGO DE CONDUTA, e LICENÇA, também podem ser úteis.
9. **Use .gitignore:** Use um arquivo .gitignore para evitar o rastreamento de arquivos que não precisam estar no repositório Git, como arquivos temporários, logs, dependências e arquivos de configuração específicos do seu ambiente local.
10. **Respeite a convenção de nomes:** Ao nomear seus arquivos, commits, branches, etc., siga as convenções padrão para manter tudo consistente e compreensível para todos os colaboradores.
11. **Mantenha um arquivo requirements.txt:** Se o seu projeto é baseado em Python, é uma boa prática ter um arquivo `requirements.txt` no repositório. Este arquivo deve listar todas as bibliotecas das quais o seu projeto depende e as suas respectivas versões. Isso permite que qualquer pessoa que deseje executar o seu projeto possa instalar facilmente as dependências corretas com o comando `pip install -r requirements.txt`.

**12. Inclua um Procfile se necessário:** Se você estiver usando o Heroku ou outra plataforma de PaaS para hospedar sua aplicação, você pode precisar de um `Procfile`. Este é um arquivo de texto simples que especifica os comandos que são executados pelo aplicativo em inicialização. Por exemplo, se você estiver executando uma aplicação web em Python usando o Gunicorn, seu `Procfile` pode parecer com isto: `web: gunicorn app:app`.

**13. Use outros arquivos de configuração conforme necessário:** Além dos mencionados acima, pode haver outros arquivos de configuração que são úteis ou necessários para o seu projeto. Por exemplo, se você estiver usando o Docker, você precisará de um `Dockerfile` e possivelmente um `docker-compose.yml`. Se você estiver usando o GitHub Actions para CI/CD, você precisará de um arquivo de fluxo de trabalho em `.github/workflows`. Sempre verifique quais arquivos de configuração são necessários para as ferramentas que você está usando e inclua-os no seu repositório.

#### 14. Commits Semânticos:

Commits Semânticos são uma padronização de commits em processos de versionamento e alterações no código. A estrutura de um commit semântico é:

```
*tipo*[escopo opcional]: *Descrição*
*corpo opcional*
*rodapé opcional*
```

#### Tipos:

- **build:** alterações no sistema de construção ou dependências externas
- **ci:** mudanças nos arquivos e scripts de configuração de métodos de integração contínua (CI)
- **docs:** inclusão ou alteração apenas de arquivos de documentação
- **feat:** adições de novas funcionalidades ou implantações ao código
- **fix:** correções de bugs
- **perf:** melhorias de desempenho no código
- **refactor:** mudanças no código sem alterar a funcionalidade final
- **style:** formatações que não afetam o significado do código
- **test:** adição ou correção de testes automatizados
- **chore:** atualizações de tarefas sem alterar o código de produção
- **env:** modificações ou adições em arquivos de configuração de métodos de integração contínua (CI)

#### Escopos:

Utilizados em commits específicos e pontuais para especificar o contexto imediato da mudança.

Exemplo: `feat(login/routes): change in route settings for the login`

### Descrições:

Parâmetro obrigatório, iniciando com letra minúscula e sendo suficientemente claro.

Exemplo: `test: ensure DbLoadSurveys throws if LoadSurveysRepository throws`

### Corpo:

Utilizado quando a descrição não é suficiente para explicar o conteúdo do commit. Deve iniciar com uma linha em branco após a descrição.

Exemplo: `feat: ensure LoadSurveysController returns 204 if there is no content - Returns code 204 if the search load method does not return content`

### Rodapé:

Uso não obrigatório, destinado a alterações de estado via smart commit, resoluções de problemas ou sprints de projetos. Iniciar com uma linha em branco após o corpo e seguir o formato de token de palavra, símbolo ":", espaço em branco e símbolo "#".

Exemplo: `fix: correct minor typos in code - see the issue for details on typos fixed. - Reviewed-by: Elisandro Mello - Refs #133`

---

## # Ferramentas de Suporte no GitHub

### 1. GitHub Actions: Automatizando tarefas

GitHub Actions é uma ferramenta poderosa que permite automatizar qualquer aspecto do seu fluxo de trabalho de desenvolvimento. Ela permite criar, testar, empacotar, liberar e implantar seu software diretamente do GitHub.

Aqui estão alguns exemplos de como você pode usar o GitHub Actions:

- **CI/CD (Integração contínua/Entrega contínua):** Você pode criar um fluxo de trabalho que testa, constrói e implanta seu código sempre que você faz um commit ou cria um pull request. Isso ajuda a garantir que todos os commits e pull requests sejam de alta qualidade.
- **Automatizar a gestão de issues e pull requests:** Você pode criar um fluxo de trabalho que responda automaticamente a issues e pull requests, rotule-os ou até mesmo os feche se eles não seguirem certas regras.

- **Programar tarefas:** Você pode criar fluxos de trabalho que são acionados em um horário específico para realizar tarefas como limpeza de dados, geração de relatórios, ou envio de notificações.

## 2. GitHub Pages: Hospedando seu site ou blog

GitHub Pages é um serviço de hospedagem gratuito do GitHub para hospedar sites estáticos diretamente dos repositórios do GitHub. Isso é ideal para projetos pessoais, documentação de projetos, blogs, e mais.

Com GitHub Pages, você pode:

- **Hospedar sites estáticos gratuitamente:** Você só precisa de um repositório GitHub e seus arquivos de site estático (HTML, CSS, JavaScript, e imagens).
- **Usar Jekyll para blogs e sites:** Jekyll é um gerador de sites estáticos que é integrado ao GitHub Pages. Ele permite criar sites completos a partir de texto simples, como Markdown.
- **Personalizar a URL do seu site:** Por padrão, o URL do seu site GitHub Pages será `seunome.github.io/repositorio`. Mas você pode usar seu próprio domínio personalizado se quiser.