GIT e GITHUB - NA PRÁTICA

> 🕮 Treinamento : # Pedro Henrique S. Silva

Objetivo do Treinamento

O objetivo desse treinamento é ensinar a quem não sabe e também para quem sabe boas práticas com git e como utilizar essas ferramentas que são úteis e eficientes para diversos projetos.

Teoria: O que você precisa saber

O que é Git

É um sistema de controle de versão distribuído, utilizado principalmente para rastrear alterações no código-fonte durante o desenvolvimento de software. Ele permite que os desenvolvedores trabalhem em projetos simultaneamente, mantendo um histórico detalhado de todas as alterações feitas em cada arquivo.

Palavras chaves:

Existem muitas palavras-chave que são utilizadas no dia a dia com Git. Algumas das principais são:
principais são:

- **Repositório:** local onde o código-fonte e seu histórico de alterações são armazenados. Pode ser local ou remoto (ex.: GitHub).
- **Commit**: registro de uma alteração no projeto.
- **Rebase:** reorganiza o histórico de commits para manter uma linha linear.
- **Branch:** ramificação do código que permite criar uma linha de desenvolvimento isolada.
- **Merge**: Ação de mesclar duas ramificações (branches) diferentes
- **HEAD:** referência que aponta para o commit atual no qual você está trabalhando.
- **Remote:** repositório hospedado remotamente, como no GitHub.

O que é GitHub?

GitHub é uma plataforma de hospedagem de código que utiliza o Git para controle de versão. Ele oferece uma interface amigável para colaboração em projetos, compartilhamento de código e gerenciamento eficiente de repositórios. Resumindo, é um repositório central compartilhado com todos os colaboradores de um projeto de software.

Palavras chaves:

Assim como GIT o GitHub também possuí palavras chaves muito importantes no dia a dia, que são:

- **README**: Arquivo principal de apresentação do projeto, arquivo .MD (markdown)
- **Releases:** área para disponibilizar versões do software com changelog e arquivos binários
- **Actions:** funcionalidade de CI/CD do GitHub para automatizar processos como testes, builds e deploys.
- **Contributors:** lista de pessoas que contribuíram com o repositório.
- **Pull Request (PR):** solicitação para revisar e integrar mudanças de uma branch para outra no GitHub.

Principais recursos do GitHub:

- **Repositórios públicos e privados:** Os desenvolvedores podem hospedar seus projetos de forma gratuita em repositórios públicos ou optar por repositórios privados para projetos comerciais ou sensíveis.
- **Colaboração eficiente:** O GitHub facilita a revisão e comentários de código, o rastreamento de bugs e a solicitação de alterações específicas no código.
- **Integração com ferramentas de automação:** O GitHub oferece integração com várias ferramentas de automação, *como integração contínua (CI) e entrega contínua (CD)*, para automatizar processos de compilação, teste e implantação.

Prática: Rodando no terminal

Passo a Passo Prático no Terminal para Usar Git e GitHub

---

0 - Instale o Git

Git - Downloads

Após instalar o Git na sua máquina abra o terminal. Vamos configurar sua conta do Github:

```
git config --global user.name "Fulano de Tal"
git config --global user.email fulanodetal@exemplo.br
```

> Troque o e-mail e nome por suas credencias do GitHub

---

1 - Iniciar um repositório local

```
git init
```

> Cria um repositório Git dentro da sua pasta atual.
> O Git começa a controlar alterações dos arquivos.

---

2 - Ver o status do repositório

```
git status
```

> Mostra o que foi alterado, o que ainda não foi salvo, e quais arquivos estão prontos para commit.

---

3 - Adicionar arquivos ao controle de versão

```
git add .
```

> Coloca todos os arquivos modificados na **área de preparação (staging)** para o próximo commit.

---

4 - Criar um commit

```
git commit -m "Mensagem que explica o que foi alterado"
```

> Grava as alterações no histórico do projeto com uma mensagem explicativa.

---

5 - Conectar com um repositório remoto (exemplo: GitHub)

```
git remote add origin https://github.com/usuario/repo.git
```

> Conecta seu projeto local com um repositório remoto.

---

6 - Enviar o código para o repositório remoto (primeira vez)

```
git push corigin main
```

> Envia suas alterações para o GitHub e vincula a branch local à remota.

---

7 - Trazer novidades do repositório remoto

Buscar atualizações sem aplicar mudanças ainda

```
git fetch
```

> Baixa as atualizações do repositório remoto, sem alterar seu código local.

Trazer e aplicar as mudanças direto no projeto

```
git pull
```

> Baixa e aplica as atualizações do remoto no seu código local.

---

8 - Criar uma nova branch

```
git checkout -b minha-nova-branch
```

> Cria e muda para uma nova branch para desenvolver recursos isoladamente.

---

9 - Ver em qual branch você está

```
git branch
```

> Lista todas as branches, indicando em qual você está trabalhando.

---

10 - Enviar uma nova branch para o GitHub

```
git push --set-upstream origin minha-nova-branch
```

> Envia a branch para o repositório remoto e vincula localmente.

---

11 - Mudar de branch

```
git checkout nome-da-branch
```

> Troca para a branch informada.

---

12 - Fazer merge de uma branch em outra

1. Vá para a branch que vai receber as mudanças:

```
git checkout main
```

2. Faça o merge:

```
git merge minha-nova-branch
```

> Junta as alterações da sua branch na branch atual (`main`).
> Se houver conflitos, o Git avisará para que sejam resolvidos.

Resumo Visual dos Comandos

Ação	Comando
Iniciar repositório	`git init`
Ver status	`git status`
Adicionar arquivos	`git add .`
Criar commit	`git commit -m "mensagem"`
Conectar repositório	`git remote add origin URL`
Enviar código (push)	`git push -u origin main`
Buscar atualizações	`git fetch`
Atualizar e aplicar	`git pull`
Criar branch	`git checkout -b nome-branch`
Ver branches	`git branch`
Enviar nova branch	`git push -u origin nome-branch`
Trocar de branch	`git checkout nome-branch`
Fazer merge	`git checkout main` + `git merge nome-branch`

Boas Práticas

Existem muitás práticas a se considerar quando se usa Git e Github, a mais utilizada são os padrões de commit. Utilizando padrões de commit é possível manter os históricos de commits organizados possibilitando um entendimento melhor pelos gestores e outros técnicos sobre o que foi feito naquela alteração da codebase.

1 - Commits pequenos, tenha em mente um objetivo para cada commit.

2 - O commit semântico possui os elementos estruturais abaixo (tipos), que informam a intenção do seu commit ao utilizador(a) de seu código.

- `feat` - Commits do tipo feat indicam que seu trecho de código está incluindo um **novo recurso** (se relaciona com o MINOR do versionamento semântico).
- `fix` - Commits do tipo fix indicam que seu trecho de código commitado está **solucionando um problema** (bug fix), (se relaciona com o PATCH do versionamento semântico).
- `docs` - Commits do tipo docs indicam que houveram **mudanças na documentação**, como por exemplo no Readme do seu repositório. (Não inclui alterações em código).
- `test` - Commits do tipo test são utilizados quando são realizadas **alterações em testes**, seja criando, alterando ou excluindo testes unitários. (Não inclui alterações em código)
- `build` - Commits do tipo build são utilizados quando são realizadas modificações em **arquivos de build e dependências**.
- `perf` - Commits do tipo perf servem para identificar quaisquer alterações de código que estejam relacionadas a **performance**.
- `style` - Commits do tipo style indicam que houveram alterações referentes a **formatações de código**, semicolons, trailing spaces, lint... (Não inclui alterações em código).
- `refactor` - Commits do tipo refactor referem-se a mudanças devido a **refatorações que não alterem sua funcionalidade**, como por exemplo, uma alteração no formato como é processada determinada parte da tela, mas que manteve a mesma funcionalidade, ou melhorias de performance devido a um code review.
- `chore` - Commits do tipo chore indicam **atualizações de tarefas** de build, configurações de administrador, pacotes... como por exemplo adicionar um pacote no gitignore. (Não inclui alterações em código)
- `ci` - Commits do tipo ci indicam mudanças relacionadas a **integração contínua** (*continuous integration*).
- `raw` - Commits do tipo raw indicam mudanças relacionadas a arquivos de configurações, dados, features, parâmetros.
- `cleanup` - Commits do tipo cleanup são utilizados para remover código comentado, trechos desnecessários ou qualquer outra forma de limpeza do código-fonte, visando aprimorar sua legibilidade e manutenibilidade.
- `remove` - Commits do tipo remove indicam a exclusão de arquivos, diretórios ou funcionalidades obsoletas ou não utilizadas, reduzindo o tamanho e a complexidade do projeto e mantendo-o mais organizado.

Veja mais sobre padrões de commit

Bônus: Resolução de Conflitos e Rollback

Resolvendo Conflitos Durante o Merge

1. Ao fazer um merge, se houver conflito, o Git avisa assim:

```
Auto-merging arquivo.txt
CONFLICT (content): Merge conflict in arquivo.txt
Automatic merge failed; fix conflicts and then commit the result.
```

2. Para listar os arquivos com conflitos:

```
git status
```

> Os arquivos com conflito estarão listados como "both modified".

3. Abra o arquivo com conflito no editor de texto. O Git marca as partes conflitantes assim:

```
<<<<<<< HEAD
Conteúdo da branch atual
=======
Conteúdo da branch que está sendo mergeada
>>>>>>> minha-nova-branch
```

> Edite o arquivo para manter o conteúdo correto e remova essas marcações

4. Após resolver todos os conflitos, adicione os arquivos para marcar como resolvidos:

```
git add arquivo.txt
```

5. Finalize o merge com um commit:

```
git commit
```

> O Git pode abrir o editor para mensagem de commit. Salve e feche para concluir.

Rollback: Como desfazer alterações

> Rollback é o ato de reverter para uma versão anterior para corrigir algum problema

Desfazer mudanças locais antes de adicionar ao staging

```
git checkout -- arquivo.txt
```

> Restaura o arquivo para o último commit, descartando alterações locais.

Desfazer arquivos que já foram adicionados ao staging

```
git reset HEAD arquivo.txt
```

> Remove o arquivo da área de staging, mantendo as alterações locais.

Desfazer o último commit, mantendo as alterações no código (soft reset)

```
git reset --soft HEAD~1
```

> Desfaz o commit mas deixa os arquivos preparados para um novo commit.

Desfazer o último commit e as alterações feitas (hard reset)

```
git reset --hard HEAD~1
```

> Apaga o commit e descarta as alterações feitas nos arquivos.

Reverter um commit que já foi enviado ao remoto (cria um novo commit desfazendo as alterações)

```
git revert <hash-do-commit>
```

> Essa é a forma segura de desfazer um commit já compartilhado, pois mantém o histórico.

Dicas para evitar problemas

- Sempre faça `git pull` antes de iniciar seu trabalho para evitar conflitos.
- Revise conflitos com calma e teste o código antes de finalizar o merge.
- Use branches para organizar o trabalho e manter o `main` estável.
- Faça commits pequenos e com mensagens claras para facilitar rollbacks.
