Exercícios de Fixação sobre Git e GitHub

Esses exercícios têm como objetivo ajudar a fixar os conceitos sobre Git e GitHub, essenciais para o controle de versões e colaboração em projetos de desenvolvimento de software. Responda os seguintes exercícios para treinar e testar seus conhecimentos.

Exercício 1: Configuração Inicial do Git

1. Como configurar seu nome e e-mail no Git? Escreva os comandos e explique a importância de cada um.

R:

Usuário: git config --global user.name (Este comando define o nome de usuário que será associado aos seus commits no Git. O --global faz com que essa configuração seja aplicada a todos os repositórios no seu computador, a menos que você a sobrescreva para um repositório específico. Se você não usar o --global, o nome será configurado apenas para o repositório atual.)

Email: git config --global user.email (Define o e-mail associado aos seus commits. Isso permite que outras pessoas saibam quem fez as alterações e, muitas vezes, também facilita a comunicação e a colaboração. O --global configura esse e-mail para todos os repositórios, mas você pode mudar isso para um repositório específico se necessário.)

Exercício 2: Comandos Básicos do Git

2. O que faz o comando 'git init'? Explique a diferença entre 'git init' e 'git clone'.

R:

- **o "git init" é utilizado para iniciar um novo repositório Git em um diretório existente(.git). Quando usar?** Quando você começa um novo projeto ou quer transformar um diretório existente em um repositório Git.
- **o gti clone, clona um repositório. Quando usar?**: Quando você deseja obter uma cópia de um repositório remoto existente e começar a trabalhar nele localmente

Exercício 3: Criando e Mudando de Branch

3. Como você criaria uma nova branch chamada 'desenvolvimento' e mudaria para ela? Escreva os comandos e explique o que cada um faz.

1- git branch desenvolvimento

Explicação: Este comando cria uma nova branch chamada desenvolvimento, mas **não muda para ela automaticamente**. A branch será criada a partir da branch atual em que você está no momento.

2-git checkout desenvolvimento

Explicação: O comando git checkout é usado para **mudar para a branch especificada**. Quando você executa git checkout desenvolvimento, o Git muda o seu diretório de trabalho para a branch desenvolvimento, fazendo com que você comece a trabalhar nela.

(OBS: O comando git checkout -b cria a branch desenvolvimento **e** muda para ela imediatamente. Essa é uma forma mais eficiente de fazer ambas as ações em uma única linha.)

Exercício 4: Adicionando e Commitando Arquivos

4. Após criar um arquivo de texto, como você o adicionaria ao Git e faria um commit? Escreva os comandos e explique o que acontece após cada comando.

- **echo "Conteúdo do arquivo" > arquivo.txt**

Explicação: Este comando cria um arquivo arquivo.txt (se ainda não existir) e escreve o conteúdo "Conteúdo do arquivo" nele.

- **git add arquivo.txt**

Explicação: O comando git add move o arquivo (ou as mudanças feitas em um arquivo) para a "staging area" (área de preparação). A partir daqui, o Git saberá que você deseja incluir esse arquivo nas próximas alterações (commit). O arquivo **não é** ainda parte do repositório até o commit ser feito.

- **(fazer o commit)**

git commit -m "Adicionar arquivo.txt com conteúdo inicial"

Explicação: O comando git commit registra as alterações na história do repositório. O parâmetro -m permite que você forneça uma **mensagem** de commit explicando o que

foi feito. Neste caso, a mensagem "Adicionar arquivo.txt com conteúdo inicial" descreve a ação de adicionar o arquivo.

- **git push origin nome-da-branch**

Explicação: O comando git push envia suas alterações (os commits feitos localmente) para o repositório remoto. origin é o nome padrão do repositório remoto e nome-da-branch é a branch em que você está trabalhando (por exemplo, main, master ou desenvolvimento).

Exercício 5: Trabalhando com Repositórios Remotos

5. Explique os comandos 'git remote add', 'git push', 'git pull' e 'git fetch'. Quando e por que devemos usar cada um deles?

- **git remote add**

Função: Adiciona um repositório remoto ao repositório local.

exemplo: git remote add <nome> <url_do_repositório>

- **git push**

Função: Envia as mudanças locais (commits) para um repositório remoto.

exemplo: git push <nome_do_remoto> <nome_da_branch>

- **git pull**

Função: Obtém alterações de um repositório remoto e as integra ao seu repositório local (equivalente a git fetch seguido de git merge).

exemplo: git pull <nome_do_remoto> <nome_da_branch>

git pull origin main (Isso busca as alterações da branch main no repositório remoto origin e tenta integrá-las à sua branch local main.)

- **git fetch**

Função: Obtém as alterações de um repositório remoto, mas não as integra automaticamente à sua branch local.

exemplo: git fetch <nome_do_remoto>

Exercício 6: Resolvendo Conflitos

6. O que é um conflito de merge? Como você resolveria um conflito de merge entre duas branches no Git?

Um conflito de merge ocorre quando o Git não consegue automaticamente mesclar as alterações de duas branches diferentes porque as modificações feitas nas mesmas linhas de código ou em arquivos diferentes são incompatíveis.

Isso normalmente acontece quando:

- Modificações simultâneas no mesmo trecho de código: Duas pessoas (ou duas branches) alteraram o mesmo arquivo na mesma parte (linha ou bloco de código), mas de maneiras diferentes.
- Remoção e modificação do mesmo arquivo: Um arquivo foi deletado em uma branch e modificado em outra.

Quando isso acontece, o Git não consegue decidir qual versão do código deve ser mantida, e o desenvolvedor precisa intervir manualmente para resolver o conflito.

Identificar os arquivos conflitantes

Quando você tenta mesclar duas branches e ocorre um conflito, o Git vai informar quais arquivos têm conflito. Você pode verificar isso com o comando git status.

Abrir os arquivos conflitantes

Abra os arquivos que estão em conflito em um editor de código. O Git marca as áreas conflitantes no arquivo com delimitadores especiais.

Resolver o conflito

Você precisa decidir qual código deve ser mantido. Há várias maneiras de resolver o conflito:

- **Manualmente editar**: Você pode editar o arquivo para escolher qual versão do código deve ser mantida (a versão da sua branch ou a versão da branch que você está mesclando).
- **Manter ambos os trechos**: Em alguns casos, você pode querer combinar os dois trechos de código e mesclar as alterações manualmente.
- **Apagar o código conflitante**: Se uma das alterações não for mais necessária, você pode apagar uma das versões.

Exercício 7: Usando GitHub

7. Como você cria um repositório no GitHub e o conecta ao seu repositório local? Escreva os comandos necessários para isso.

No GitHub:

- **Passo 1: Crie um repositório novo no github.**
- **Passo 2: Não inicialize com README ou outros arquivos (se seu repositório local já tiver).**

No seu repositório local:

- **Inicie um repositório Git local (se não tiver um) : .git .**
- **código:** git init

Adicionar o repositório remoto do GitHub com o código:

git remote add origin <URL do repositório no GitHub>

Adicionar e fazer o commit dos arquivos:

código:

git add .

git commit -m "Primeiro commit"

Enviar as alterações para o GitHub.

código: git push -u origin main

Exercício 8: Histórico de Commits

8. Como você visualiza o histórico de commits no Git? Quais comandos você usaria para ver o histórico de commits e detalhes de um commit específico?

(OBS: No Git, um hash é uma sequência única de caracteres gerada a partir do conteúdo de um commit. Ele é usado para identificar de maneira única cada commit, garantindo integridade e permitindo navegação eficiente pelo histórico do repositório.)

git log: Exibe o histórico completo de commits, mostrando hash, autor, data e mensagem de cada commit.

git log --oneline: Exibe o histórico de commits de forma resumida, com o hash curto e a mensagem do commit.

git log <hash_do_commit>: Exibe detalhes de um commit específico, identificado pelo seu hash.

git show <hash_do_commit>: Exibe informações detalhadas sobre um commit, incluindo as alterações feitas no código.

git log --graph --oneline: Exibe o histórico com um gráfico visual das branches.

Exercício 9: Trabalhando com Forks

9. Explique o que é um fork no GitHub e qual é o fluxo de trabalho típico ao trabalhar com forks em projetos colaborativos.

Um **fork** no GitHub é uma cópia de um repositório que permite que você faça alterações sem afetar o repositório original. O fluxo de trabalho típico ao trabalhar com forks é:

1. Fork o repositório no GitHub.
2. Clone o repositório forkado para sua máquina local.
3. Adicione o repositório original como upstream para manter seu fork atualizado.
4. Crie uma nova branch para suas mudanças.
5. Faça as alterações e commits.
6. Puxe atualizações do repositório original para evitar conflitos.
7. Push suas alterações para o seu repositório forkado no GitHub.
8. Crie um Pull Request para submeter suas alterações ao repositório original.
9. Revisão e feedback para ajustes, se necessário.
10. Merge ou fechamento do Pull Request.

Exercício 10: Revertendo Commits

10. O que o comando 'git revert' faz? Qual a diferença entre 'git revert' e 'git reset'?

O comando git revert no Git é usado para criar um novo commit que desfaz as alterações feitas por um commit anterior. Ou seja, ele inverte as mudanças introduzidas em um commit específico, mas sem alterar o histórico do repositório.

O comando git reset: Move a referência da branch para um commit anterior, podendo **desfazer commits** de forma local (em sua cópia do repositório). Pode alterar o histórico de

commits, removendo commits e deixando o histórico diferente.

Ambos os comandos são usados para desfazer mudanças, mas de maneiras diferentes e com implicações distintas no histórico de commits.