SENAI Serviço Nacional de Aprendizagem Industrial

Git e GitHub para Desenvolvedores

Módulo 4: Branches e Colaboração

1. Criar e trocar de branch

Branches são fundamentais para o fluxo de trabalho colaborativo no Git. Elas permitem que você trabalhe em novas funcionalidades ou correções de bugs de forma isolada, sem afetar a linha principal de desenvolvimento do projeto.

`git branch`

Este comando lista todas as branches no seu repositório local. A branch atual será destacada (geralmente com um asterisco).

`git checkout -b <nome-da-branch>`

Este comando cria uma nova branch com o nome especificado e, em seguida, muda para essa nova branch. É um atalho para `git branch <nome-da-branch>` seguido de `git checkout <nome-da-branch>` .

Para trocar para uma branch existente, use `git checkout <nome-da-branch>` .

2. Mesclar branches (merge)

Depois de desenvolver uma funcionalidade ou corrigir um bug em uma branch separada, você precisará mesclar essas alterações de volta para a branch principal (ou outra branch).

`git merge <nome-da-branch>`

Para mesclar, primeiro certifique-se de estar na branch para a qual você deseja trazer as alterações (por exemplo, `main`). Em seguida, execute o comando `git merge` com o nome da branch que contém as alterações que você deseja mesclar.

3. Resolver conflitos de merge

Conflitos de merge ocorrem quando o Git não consegue mesclar automaticamente as alterações de duas branches porque as mesmas linhas de código foram modificadas de maneiras diferentes em ambas as branches. Quando um conflito ocorre, o Git marca os arquivos conflitantes e você precisa resolvê-los manualmente.
O Git indicará os conflitos no arquivo com marcadores especiais:

```
<<<<<<< HEAD
// Código da sua branch atual
=======
// Código da branch que você está mesclando
>>>>>>> <nome-da-branch>
```

Você precisará editar o arquivo, remover os marcadores e escolher qual versão do código deseja manter (ou combinar as duas). Após resolver os conflitos, adicione o arquivo novamente com `git add` e faça um novo commit.

4. Trabalhando com pull e push

`pull` e `push` são os comandos que você usará para sincronizar seu repositório local com o repositório remoto.

`git pull`

Este comando baixa as alterações do repositório remoto para o seu repositório local e as mescla automaticamente na sua branch atual. É uma combinação de `git fetch` (baixar) e `git merge` (mesclar).

`git push`

Este comando envia os commits do seu repositório local para o repositório remoto. Você só pode enviar commits que ainda não existem no repositório remoto.

5. Fork e Pull Request no GitHub

Fork: Um "fork" é uma cópia pessoal de um repositório de outra pessoa. Ele permite que você faça alterações em um projeto sem afetar o repositório original. É uma prática comum em projetos de código aberto.

Pull Request (PR): Depois de fazer alterações em seu fork e enviá-las para o seu repositório remoto, você pode propor que essas alterações sejam incorporadas ao repositório original criando um Pull Request no GitHub. O PR permite que outros desenvolvedores revisem suas alterações, discutam-nas e, eventualmente, as mesclem no projeto principal.

Módulo 5: Boas Práticas e Casos Reais

1. .gitignore

Ao trabalhar com Git, nem todos os arquivos do seu projeto precisam ser versionados. Arquivos gerados automaticamente (como `node_modules`), arquivos de configuração com informações sensíveis (`.env`), ou arquivos temporários não devem ser incluídos no repositório. Para isso, usamos o arquivo `.gitignore`.

O arquivo `.gitignore` é um arquivo de texto simples onde você lista os padrões de nomes de arquivos e diretórios que o Git deve ignorar. Cada linha no arquivo `.gitignore` representa um padrão.

Exemplos de padrões comuns:

- `node_modules/` : Ignora a pasta `node_modules` (comum em projetos JavaScript/Node.js).
- `.env` : Ignora o arquivo `.env` (comum para variáveis de ambiente).
- `*.log` : Ignora todos os arquivos com a extensão `.log`.
- `/build` : Ignora a pasta `build` na raiz do projeto.
- `temp/` : Ignora a pasta `temp` em qualquer subdiretório.

É uma boa prática criar o arquivo `.gitignore` logo no início do projeto para evitar que arquivos indesejados sejam acidentalmente adicionados ao repositório.

2. Commits semânticos

As mensagens de commit são cruciais para manter um histórico de projeto claro e compreensível. Commits semânticos são uma convenção que padroniza as mensagens de commit, tornando-as mais informativas e fáceis de ler.

Uma mensagem de commit semântica geralmente segue o formato:

`<tipo>(<escopo>): <descrição>`

[corpo opcional]

[rodapé opcional]

Tipos comuns:

`feat` : Uma nova funcionalidade.

`fix` : Uma correção de bug.

`docs` : Alterações na documentação.

`style` : Alterações que não afetam o significado do código (espaços em branco, formatação, ponto e vírgula ausente, etc.).

`refactor` : Uma mudança de código que não corrige um bug nem adiciona uma funcionalidade.

`test` : Adição ou correção de testes.

`chore` : Alterações na construção do processo ou ferramentas auxiliares e bibliotecas (geração de documentação, etc.).

Exemplos:

`feat: adiciona funcionalidade de login de usuário`

`fix(auth): corrige erro de autenticação no cadastro`

`docs: atualiza README com instruções de instalação`

3. GitHub README.md

O arquivo README.md é o cartão de visitas do seu projeto no GitHub. Ele é o primeiro arquivo que as pessoas veem ao visitar seu repositório e deve fornecer informações essenciais sobre o projeto, como o que ele faz, como instalá-lo, como usá-lo e como contribuir.

Estrutura básica de um bom README:

Título do Projeto: Um título claro e conciso para o seu projeto.

Descrição: Uma breve descrição do que o projeto faz e qual problema ele resolve.

Instalação: Instruções passo a passo sobre como instalar e configurar o projeto.

Uso: Exemplos de como usar o projeto, incluindo trechos de código, se aplicável.

Contribuição: Orientações sobre como outras pessoas podem contribuir para o projeto.

Licença: Informações sobre a licença do projeto.

Créditos/Agradecimentos: Opcional, para agradecer a colaboradores ou recursos utilizados.

Um bom README.md não apenas ajuda outros desenvolvedores a entenderem e usarem seu projeto, mas também demonstra profissionalismo e cuidado com a documentação.

Módulo 6: Ferramentas Avançadas

1. GitHub Pages

O GitHub Pages é um serviço de hospedagem de sites estáticos gratuito oferecido pelo GitHub. Ele permite que você hospede sites diretamente de um repositório GitHub, sendo ideal para projetos de código aberto, blogs pessoais, documentação ou até mesmo portfólios. É amplamente utilizado para projetos front-end desenvolvidos com JavaScript, React, Vue, Angular, ou apenas HTML/CSS puro.

Como funciona:

Você cria um repositório no GitHub, adiciona seus arquivos HTML, CSS e JavaScript (ou o build de seu projeto React/Vue) e configura o GitHub Pages para servir esses arquivos. O GitHub Pages pode ser configurado para usar a branch `main` (ou `master`), uma branch `gh-pages` específica, ou até mesmo uma pasta `docs/` dentro da sua branch principal.

Passos básicos para configurar o GitHub Pages:

1. Crie um repositório no GitHub (ou use um existente).
2. Certifique-se de que seus arquivos de site (ex: `index.html`, `style.css`, `script.js`) estejam na raiz da branch que você deseja usar para o GitHub Pages (geralmente `main` ou `gh-pages`).
3. Vá para as configurações do seu repositório no GitHub (aba "Settings").
4. No menu lateral, clique em "Pages".
5. Em "Source", selecione a branch e a pasta (ex: `main` e `/root` ou `gh-pages` e `/root`) que você deseja usar como fonte para o seu site.
6. Clique em "Save".

Após alguns minutos, seu site estará disponível em um URL como https://<seu usuario>.github.io/<nome-do-repositorio> .

2. Integração com VS Code (extensões de Git)

O Visual Studio Code (VS Code) possui uma integração nativa poderosa com o Git, o que facilita muito o fluxo de trabalho de controle de versão. Além da integração nativa, existem diversas extensões que podem aprimorar ainda mais sua experiência com Git e GitHub.

Recursos nativos do VS Code para Git:

- **Controle do Código-Fonte (Source Control):** Uma aba dedicada no VS Code (ícone de três círculos conectados) mostra todas as alterações no seu repositório, permitindo que você visualize arquivos modificados, adicione-os à área de staging, faça commits, desfaça alterações e muito mais.

Visualização de Diferenças (Diff View): O VS Code facilita a comparação de versões de arquivos, mostrando claramente as linhas adicionadas, removidas ou modificadas.

Histórico de Commits: Você pode visualizar o histórico de commits diretamente na interface, com detalhes sobre cada commit.

Branches: Trocar de branch, criar novas branches e mesclar branches são operações simples de realizar através da interface do VS Code.

Extensões úteis para Git e GitHub no VS Code:

GitLens — Git supercharged: Uma extensão extremamente popular que adiciona recursos poderosos, como a visualização de quem alterou cada linha de código (Git blame), histórico de arquivos, navegação entre commits e muito mais.

GitHub Pull Requests and Issues: Permite que você gerencie Pull Requests e Issues do GitHub diretamente no VS Code, facilitando a revisão de código e a colaboração.

Git Graph: Oferece uma visualização gráfica interativa do seu histórico de commits, branches e merges, tornando mais fácil entender o fluxo do seu repositório.

Essas ferramentas e extensões tornam o VS Code um ambiente de desenvolvimento muito produtivo para projetos que utilizam Git e GitHub, especialmente, GitHub.