Git e GitHub - Controle de Versão e Colaboração

Asimov Academy

ASIMOV

Conteúdo

01. Git e GitHub - Controle de Versão e Colaboração

Olá, seja muito bem-vindo ao curso **"Git e GitHub - Controle de Versão e Colaboração"**!

Neste curso, vamos explorar o universo do versionamento de código e da colaboração em projetos. Se você já desenvolve projetos ou está começando agora, sabe como é importante manter tudo bem organizado e seguro. Nessa jornada vamos entender como o Git e o GitHub podem transformar a forma como você gerencia e compartilha seu trabalho.

O que é Git?

Git é uma ferramenta de controle de versão que permite salvar, gerenciar e recuperar o progresso do seu código de maneira eficiente. Ele é fundamental para manter seus projetos organizados e seguros, possibilitando que você acompanhe as alterações realizadas e retorne a versões anteriores quando necessário.

Por que Git e GitHub?

A adoção do Git é universal entre os desenvolvedores, devido à sua robustez e flexibilidade. Complementando essa ferramenta, o GitHub oferece uma plataforma em nuvem colaborativa que potencializa o uso do Git, permitindo a integração de equipes, a hospedagem de projetos na nuvem e facilitando a comunicação e o gerenciamento de tarefas. Juntos, Git e GitHub formam uma dupla poderosa que simplifica o desenvolvimento e a manutenção de qualquer projeto.

Estrutura do Curso

Este curso foi planejado para levá-lo dos fundamentos até as práticas mais avançadas de versionamento e colaboração, seguindo a ordem abaixo:

1. **O que é Git**
    - Entenda os conceitos básicos, a história e a importância de um sistema de controle de versão.
2. **Utilizando Git no Terminal e no VSCode**
    - Aprenda os comandos essenciais para criar, salvar e gerenciar repositórios tanto no terminal quanto em um editor de código popular.
3. **O que é GitHub**

- Descubra como o GitHub funciona, como criar uma conta, criar repositórios remotos e entender as principais funcionalidades da plataforma.

4. **Trabalhando em Branchs e resolvendo conflitos**
    - Saiba como criar e gerenciar ramificações (branches), fazer merges e lidar com conflitos de forma eficiente.
5. **Trabalhando em projetos colaborativos**
    - Veja como organizar tarefas, criar Pull Requests, revisar códigos e usar as ferramentas colaborativas que o GitHub oferece.
6. **Criando um repositório para seu perfil no GitHub**
    - Entenda como hospedar projetos públicos e privados, além de personalizar o seu perfil com repositórios que mostrem seu trabalho.

O que esperar deste curso?

Ao final deste curso, você terá uma compreensão sólida de como utilizar o Git e o GitHub para gerenciar seus projetos de forma eficiente. Você estará preparado para aplicar essas ferramentas em diversos contextos, seja em projetos pessoais, acadêmicos ou profissionais, garantindo organização, segurança e colaboração aprimorada.

Espero que você aproveite cada aula e que este curso transforme sua forma de trabalhar com código. Este é apenas o começo da sua jornada no mundo do versionamento e da colaboração com Git e GitHub. Vamos lá para a primeira aula! # Módulo 1: Introdução ao Git e Controle de Versão

Se você já perdeu arquivos ou precisou voltar atrás em uma mudança de texto ou código sem sucesso, saiba que o Git pode resolver esse problema de forma eficiente. Ao longo deste módulo, você vai entender como o Git funciona e por que ele é fundamental para qualquer projeto e como ele pode tornar seu fluxo de trabalho mais seguro e organizada!

1.1 O que é Git e por que usá-lo?

O Git é um software de controle de versão que permite gerenciar o histórico de alterações no código. Em algumas distribuições Linux e versões do macOS, ele já vem pré-instalado, mas em muitos casos, é necessário instalar manualmente. No Windows, o Git não vem instalado por padrão, sendo preciso baixar e configurar no sistema.

Ao longo das aulas, veremos como instalar e utilizar o Git na prática. Mas, antes disso, é importante entender o que ele faz e como funciona para conseguirmos utilizar ele da melhor forma.

O que ele faz, exatamente?

Imagine o Git como um cartão de memória para o seu código. Se você já trabalhou em um projeto grande – seja em Python, JavaScript, outra linguagem ou com outros tipos de arquivos –, sabe como pode ser arriscado fazer mudanças sem um backup.

O Git permite que você crie versões de salvamento do seu progresso, como aqueles muitos arquivos que muita gente já gerou ao escrever um TCC ou um relatório importante:

```
trabalho_versao1.docx
trabalho_versao2.docx
trabalho_final.docx
trabalho_final_corrigido.docx
trabalho_final_corrigido_definitivo.docx
```

Com o tempo, essa bagunça de versões se torna difícil de gerenciar. O Git resolve esse problema de forma organizada e eficiente, criando "savepoints" ou checkpoints no seu código. Se algo der errado ou você precisar voltar para um momento anterior do desenvolvimento, o Git permite recuperar qualquer versão sem precisar manter dezenas de arquivos diferentes.

Em resumo, o Git mantém tudo sob controle, permitindo que você programe com segurança e sem medo de perder suas mudanças.

Mas por que não basta salvar os arquivos em uma pasta no computador?

Imagine que você está desenvolvendo um site com diversos arquivos e, de repente, uma modificação causa um erro inesperado. Se você estiver salvando tudo apenas em uma pasta, será difícil identificar exatamente onde está o problema ou reverter para um momento em que estava funcionando.

Já com o Git você não só armazena os arquivos, mas também mantém um histórico completo de cada alteração feita. Isso é chamado de **controle de versão**, e ele traz vários benefícios:

- **Segurança**: Você pode voltar para uma versão anterior do código caso algo dê errado.
- **Histórico**: Todas as mudanças são registradas, permitindo que você analise o que foi alterado ao longo do tempo.
- **Organização**: Cada etapa do desenvolvimento pode ser documentada, com mensagens explicando as alterações.
- **Colaboração**: Várias pessoas podem trabalhar simultaneamente no mesmo projeto sem perder o controle das mudanças.

- **Autonomia**: Para trabalhar ou fazer commits você não precisa estar conectado na internet ou buscar essa informação em um servidor! Se você quiser viajar e trabalhar offline você consegue ir salvando e versionando seu código no git sem problemas!

Controle de versão: segurança, histórico e reversão de erros

O controle de versão funciona como uma linha do tempo do seu projeto. Sempre que você "salva" com o Git (faz um commit), ele cria um "checkpoint" que guarda o estado atual do seu projeto. Por exemplo:

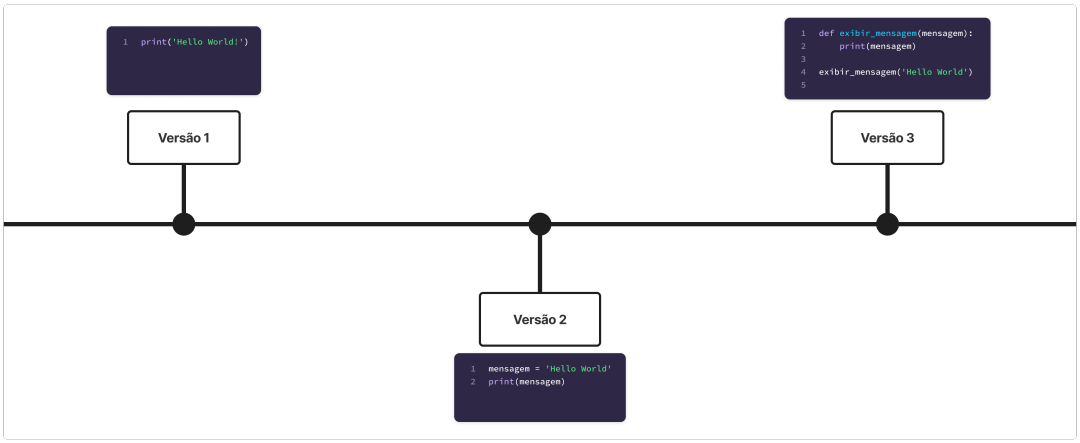


Figure 1: Commits na linha do tempo do código

Com o Git, cada **checkpoint** (commit) contém o estado exato do projeto em um ponto específico no tempo. Isso permite que você acompanhe todas as mudanças, identifique o que foi modificado e até corrija erros sem comprometer o restante do código.

Além disso, o Git garante a **integridade dos arquivos**. Ou seja, nenhuma mudança passa despercebida, tudo é verificado para que nada seja perdido ou corrompido no processo. Para isso, o Git usa um identificador único chamado **hash**, que é uma sequência de 40 caracteres hexadecimais calculada com base no conteúdo do commit e em suas alterações.

Mesmo que você mude **apenas** uma vírgula em um arquivo, o hash do commit muda completamente. Isso garante que cada versão seja única e rastreável.

Veja abaixo o exemplo de um hash (em amarelo) no histórico de commits do git:

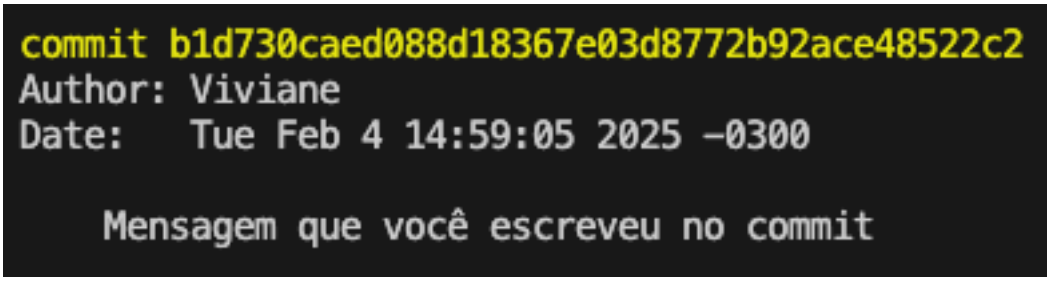


Figure 2: Hash de um commit

Mas por que isso é importante? Você não precisa entender exatamente como esse cálculo é feito, mas é essencial saber que ele existe, pois você verá **hashes** com frequência ao utilizar o git e futuramente o github. Eles são importantes para buscar commits no histórico, reverter alterações e acessar versões antigas do código.

E quanto a projetos em equipe?

Bom galera, vocês viram que até aqui eu tô falando só de "git", não falei nenhuma vez de "github". E essa é uma distinção importante: o programa que faz esse controle de versão é o **git**, e qualquer pasta que estiver sendo gerenciada por ele é chamado de um **repositório git**.

Mas e se eu quiser que outras pessoas colaborem comigo no mesmo projeto? Vamos supor que eu esteja trabalhando em uma parte do código, e meu amigo está desenvolvendo outra parte no PC dele. Como sincronizamos isso sem precisar trocar arquivos por e-mail ou WhatsApp?

É aqui que entra o **GitHub**: uma plataforma online que funciona como um centralizador de repositórios Git. Com ele, podemos armazenar projetos na nuvem, compartilhar o progresso e trabalhar em equipe, mesmo que os colaboradores estejam em diferentes lugares.

Se o Git controla versões localmente, no seu computador, o GitHub expande esse uso para o ambiente online, permitindo que várias pessoas colaborem em um mesmo projeto de forma organizada e segura!

Com o GitHub, é possível:

- Enviar alterações feitas no seu computador para um repositório remoto;
- Gerenciar contribuições de outros colaboradores;
- Revisar alterações antes de integrá-las ao projeto principal;
- Armazenar projetos de maneira segura e acessível a qualquer momento.

No decorrer do curso, vamos explorar o GitHub em mais detalhes e entender como ele complementa o Git para facilitar o trabalho colaborativo.

1.2 Instalação do Git

Como já vimos, o **Git** é um sistema de controle de versão que permite gerenciar mudanças no código, colaborar com outras pessoas e manter um histórico detalhado do desenvolvimento. Porém, antes de começar a utilizar o git, é necessário instalar corretamente no seu sistema operacional.

A instalação do Git vai variar conforme o sistema operacional. Abaixo, explicamos como instalar no **Windows**, **macOS** e **Linux**.

Instalando o Git no Windows

Diferente do macOS e Linux, o Git não vem instalado por padrão no Windows. Para instalar, siga os passos abaixo:

1. Acesse o site oficial: git-scm.com.
2. Baixe a versão mais recente do instalador para Windows.
3. Durante a instalação:
    - Escolha o **Git Bash** como terminal padrão.
    - Mantenha as opções recomendadas.
4. Após a instalação, abra o **Git Bash** e teste se tudo está funcionando com:

```
git --version
```

Se o comando exibir a versão do Git instalada, significa que o processo foi bem-sucedido. Caso contrário, tente reinstalar ou reiniciar o computador.

Instalando o Git no macOS e Linux

A maioria das distribuições **Linux** e versões do **macOS** já possuem o **Git** pré-instalado. Para verificar se ele já está disponível no seu sistema, abra o terminal e execute:

```
git --version
```

Se o Git estiver instalado, o terminal mostrará a versão atual. Caso o comando não retorne um número de versão, será necessário instalar manualmente:

Acesse o tutorial oficial de instalação do Git

No site, selecione seu sistema operacional e siga o passo a passo indicado.

Configuração Inicial do Git

Após a instalação, é necessário configurar o Git com suas informações de usuário. Isso garante que todas as alterações feitas por você sejam corretamente identificadas.

- Definindo nome e email

  O Git utiliza um nome e um email para registrar as contribuições de cada usuário. Para configurá-los, execute os comandos abaixo, substituindo com suas informações:

  ```
  git config --global user.name "Seu Nome"
  git config --global user.email "seuemail@exemplo.com"
  ```

- Verificando a configuração

  Para conferir se a configuração foi aplicada corretamente, rode no terminal:

  ```
  git config --global --list
  ```

A saída será algo como:

```
user.name=Seu Nome
user.email=seuemail@exemplo.com
```

Caso precise alterar alguma informação, basta rodar os comandos novamente e definir um novo nome ou email.

No próximo tópico vamos dar os primeiros passos no Git, aprendendo a criar um repositório local e executando os primeiros comandos no terminal e no VSCode!

Módulo 2 - Utilizando Git no Terminal e no VSCode

O Git pode ser utilizado tanto pelo **terminal** quanto por interfaces gráficas, como o **VSCode**. Neste módulo, aprenderemos a criar um **repositório local**, utilizar os **comandos básicos** do Git e realizar operações diretamente pelo VSCode.

2.1 Criando o Primeiro Repositório Local

Para começar a versionar um projeto, precisamos criar um repositório Git. Vamos seguir os passos abaixo para criar e configurar um repositório local.

Criando um diretório e inicializando o repositório

Abra o terminal e execute os seguintes comandos para criar um diretório e inicializar o Git:

```
mkdir meu-projeto
cd meu-projeto
```

Para controlar a versão de um projeto e registrar suas mudanças, é necessário transformá-lo em um repositório Git. Para isso, usamos o comando `git init`, que deve ser executado apenas uma vez para inicializar um repositório vazio e começar o rastreamento pelo Git.

Para inicializar o git na sua pasta e tranformá-la em um repositório git execute no terminal o comando:

```
git init
```

Agora o diretório está pronto para receber arquivos sob controle de versão!

Se o diretório já for um repositório Git, rodar esse comando novamente não faz sentido e resultará na mensagem *"Reinitialized existing Git repository"*, indicando que o repositório foi reinicializado.

Caso tenha feito isso por engano, não há problema, pois o comando não altera nada. Para evitar esse erro, antes de usar `git init`, execute `git status`. Se a mensagem *"fatal: not a git repository"* aparecer, significa que o diretório ainda não é um repositório Git e o comando pode ser usado.

Criando um arquivo Python simples

Para utilizarmos o controle de versão precisamos de arquivos de código no nosso repositeorio, para isso vamos criar um arquivo Python simples com um comando básico de saída:

```
echo "print('Hello, Git!')" > hello.py
```

Adicionando o arquivo ao controle de versão

Pronto, podemos agora adicionar nosso arquivo na área de preparação do Git para depois criamos nosso primeiro ponto de salvamento:

```
git add hello.py
```

Esse comando, por debaixo dos panos, move o arquivo na estrutura do repositório Git para a staging area (área de preparação), onde ele começa a ser monitorado pelo Git e fica pronto para ser registrado no histórico do projeto.

Para criar um ponto de salvamento e registrar essa versão do código, usamos o comando **commit**(confirmação):

```
git commit -m "Adicionando arquivo inicial"
```

O commit funciona como um checkpoint no histórico do projeto, registrando exatamente como o código estava naquele momento. Isso permite que possamos recuperar essa versão futuramente, sem precisar criar cópias manuais do arquivo.

Sempre que rodamos o comando `git commit`, precisamos incluir uma mensagem descritiva. Essa mensagem, que deve ser colocada entre aspas `"..."`, deve ser curta, mas clara, fornecendo contexto sobre a alteração feita.

Isso é importante para que, caso seja necessário voltar para um commit anterior ou entender o histórico do projeto, possamos identificar rapidamente qual mudança foi feita em cada commit.

Ou seja, o processo de versionamento no Git segue um fluxo organizado, desde a criação ou modificação de um arquivo até sua inclusão no histórico do repositório. Esse caminho pode ser visualizado da seguinte forma:

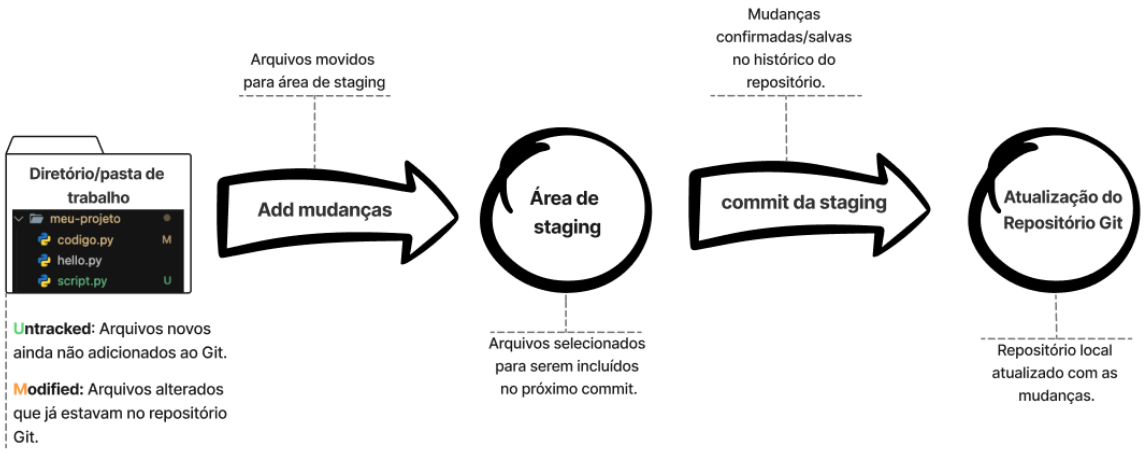


Figure 3: Principais etapas para o versionamento

Estados de Arquivo no Git

Após realizar seu primeiro commit, podemos verificar se ele foi registrado corretamente acessando o histórico do repositório. Para isso, usamos o comando git log, que exibe uma lista detalhada de todos os commits feitos, incluindo informações como hash, autor, data e mensagem do commit.

- Se quisermos uma versão mais compacta e visual do histórico, podemos usar:

```
git log --oneline --graph --decorate
```

- Esse comando exibe um resumo mais organizado, mostrando os commits de forma simplificada e destacando a estrutura do projeto.
- **Importante**: O histórico de commits é exibido em ordem cronológica inversa, ou seja, os commits mais recentes aparecem primeiro.

Outro comando essencial para acompanhar o **estado** do repositório é o `git status`. Ele mostra quais arquivos foram modificados, quais estão na área de staging (área de preparação) prontos para commit e quais ainda não estão sob controle do Git.

Esse comando é útil antes e depois de rodar um commit, pois permite conferir se todas as alterações estão sendo rastreadas corretamente.

No Git, os arquivos dentro de um repositório podem estar em diferentes **estados**. Aqui estão alguns dos principais status que um arquivo pode ter:

- **U** (*Untracked* - Não rastreado) → O arquivo ainda não está sendo monitorado pelo Git. Ele foi criado dentro do repositório, mas ainda não foi adicionado à área de staging/preparação.
- **A** (*Added* - Adicionado) → O arquivo foi adicionado a área de staging com o comando `git add`, ou seja, agora ele está pronto para ser incluído no próximo commit.
- **M** (*Modified* - Modificado) → O arquivo já estava no repositório, mas foi alterado depois do último commit. Essas mudanças ainda não foram adicionadas à área de staging.
- **S** (*Staged* - Preparado) → O arquivo foi modificado e adicionado à área de staging, aguardando ser salvo no próximo commit.

Guia Rápido de Comandos Git

Revisando os comandos principais que mais usamos ao trabalhar com Git:

- `git init` → Inicializa um repositório Git no diretório atual.

- `git add <arquivo>` → Adiciona arquivos a área de **staging**, preparando-os para o commit.

- `git commit -m "<mensagem>"` → Registra as alterações no histórico do repositório.
- `git log` → Exibe o histórico de commits.
- `git status` → Mostra o estado atual do repositório e quais arquivos estão modificados, adicionados ou prontos para commit.

Agora que nós já criamos nosso primeiro repositório, adicionamos arquivos e entendemos como o Git organiza as mudanças, vamos facilitar ainda mais o nosso fluxo de trabalho! Vamos ver **como usar o Git no VSCode**, aproveitando a interface gráfica para deixar tudo mais prático e intuitivo. Bora lá!

2.2 Git no VSCode

O **VSCode** possui uma **integração nativa com o Git**, permitindo que você execute operações de versionamento sem precisar do terminal. Essa funcionalidade é útil para quem prefere uma abordagem mais visual e interativa no gerenciamento de código.

Por mais que o terminal seja uma ferramenta poderosa para executar comandos e versionar nosso código, o **VSCode** oferece uma interface gráfica que torna esse processo mais intuitivo. Ele facilita a visualização das mudanças, a organização dos commits e a administração do versionamento, permitindo um controle mais prático do repositório sem depender apenas da linha de comando.

Como usar o Git no VSCode:

1º - Abrir a pasta do projeto no VSCode

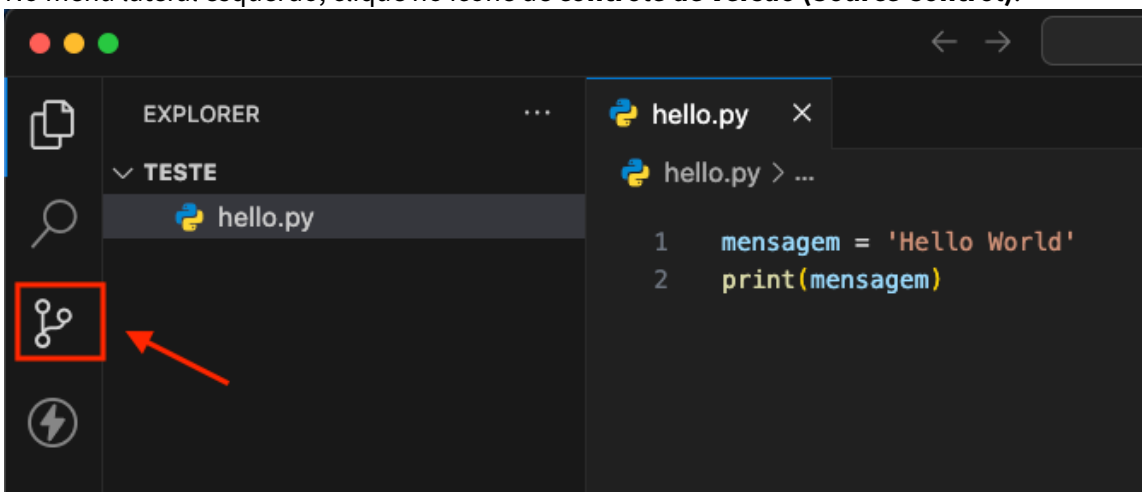
- No terminal, dentro da pasta do projeto, digite:

```
code .
```

- Caso você esteja usando **macOS** ou **Linux** e o comando `code .` não funcione, será necessário configurá-lo manualmente.
- Você pode ler mais sobre isso na documentação oficial:
  - Configuração no macOS
  - Configuração no Linux
- Alternativamente, você pode abrir a pasta do projeto diretamente no **VSCode**, indo em: **File → Open Folder...** e selecionando o diretório do seu projeto.

2º - Acessar a aba de controle de versão**

- No menu lateral esquerdo, clique no ícone de **controle de versão (Source Control)**.



- O VSCode detectará automaticamente se o diretório é um repositório Git.
- Se o repositório ainda **não estiver inicializado**, o VSCode oferecerá a opção de inicializá-lo diretamente pela interface.

3º - Adicionar e confirmar alterações

- O VSCode exibirá os arquivos modificados na aba de controle de versão.
- Para adicioná-los ao **staging area**, clique no botão **+** ao lado do nome do arquivo.
- Digite uma **mensagem descritiva** para o commit.
- Clique no botão **Commit** para registrar as alterações no histórico do Git.

Comparação: Terminal vs. VSCode

Ação	Comando Terminal	VSCode
Inicializar repositório	`git init`	Criar novo repositório pela interface
Adicionar arquivo ao Git	`git add <arquivo>`	Clicar no ícone + na aba de controle de versão

Ação	Comando Terminal	VSCode
Fazer commit	`git commit -m "<mensagem>"`	Escrever mensagem e clicar em **Commit**
Verificar histórico	`git log`	Interface não exibe diretamente, mas há extensões que ajudam

Como podemos ver no quadro acima, as duas formas de utilizar o Git possuem vantagens e permitem realizar as mesmas operações de maneira eficiente. No entanto, a escolha entre **terminal** e **interface gráfica** dependerá do fluxo de trabalho de cada desenvolvedor e da sua preferência pessoal.

Desafio Prático: Criando Seu Primeiro Repositório Git

Agora chegou o momento de praticar! Vamos criar nosso primeiro repositório local e explorar o Git na prática. O objetivo é aprender a versionar seu código utilizando tanto o **terminal** quanto a **interface gráfica do VSCode**.

Passo a passo

1 - Criar um repositório e inicializar o Git

1. Abra o **terminal** ou o **Git Bash**.
2. Escolha um local no seu computador onde deseja criar o repositório.
3. Execute o seguinte comando para criar uma pasta para o projeto:

```
mkdir projeto-git
```

4. Entre na pasta criada:

```
cd projeto-git
```

5. Agora, inicialize o repositório Git dentro dessa pasta:

```
git init
```

O Git foi inicializado! Isso significa que essa pasta agora é um repositório Git e podemos começar a versionar os arquivos dentro dela.

2 - Criar um arquivo `mensagem.py`

Agora vamos criar um arquivo simples dentro do repositório para testar o controle de versão.

1. Ainda no terminal, execute:

```
echo "print('Olá, Git!')" > mensagem.py
```

2. Esse comando cria um arquivo chamado `mensagem.py` com um código simples em Python.

3. Para verificar se o arquivo foi criado, digite:

```
dir
```

4. Se estiver no **Linux ou macOs**, use:

```
ls
```

O arquivo `mensagem.py` deve aparecer na lista de arquivos dentro do diretório.

3 - Adicionar e commitar o arquivo pelo terminal

Agora que temos um arquivo no repositório, precisamos adicionar e salvar essa alteração no histórico do Git.

1. Primeiro, verifique o status do repositório para ver quais arquivos foram modificados ou adicionados:

```
git status
```

2. Adicione o arquivo `mensagem.py` ao **staging area** (área de preparação para o commit):

```
git add mensagem.py
```

3. Agora, registre essa mudança no histórico do Git com um **commit**:

```
git commit -m "Adicionando mensagem"
```

4. Para conferir se o commit foi realizado, execute:

```
git log
```

Agora, seu primeiro commit está salvo e faz parte do histórico do repositório!

4 - Abrir o projeto no VSCode e verificar a integração com o Git

Agora, vamos abrir o projetopelo VSCode, alterar o arquivo e subir um commit pela interface gráfica.

1. Abra a pasta do projeto no VSCode.
2. No **menu lateral esquerdo**, clique no ícone de **controle de versão (Source Control)**.
3. Você verá que o repositório já está sendo monitorado pelo Git.

5 - Modificar o arquivo e realizar um novo commit pelo VSCode

Agora, vamos fazer uma alteração no arquivo `mensagem.py` e registrar essa mudança pelo **VSCode**, sem usar o terminal.

1. No **VSCode**, abra o arquivo `mensagem.py`.

2. Modifique a linha dentro do arquivo para:

```
mensagem = 'Olá, Git e VSCode!'
print(mensagem)
```

3. Salve a alteração (`Ctrl + S` ou `Cmd + S` no Mac).

4. No menu lateral esquerdo, clique no ícone de **controle de versão**.
5. Você verá que o `mensagem.py` foi modificado.

6. Passe o mouse em cima do arquivo e clique no **+** ao lado do arquivo para adicioná-lo ao **staging**.

7. No campo de mensagem, digite algo como:

```
Atualizando mensagem com variável
```

8. Clique no botão **Commit** para salvar a alteração.

Pronto! Agora você realizou um commit usando o VSCode!

As duas formas de versionar seu código com o git são válidas! O terminal oferece **mais controle e flexibilidade**, enquanto a interface gráfica do VSCode pode ser **mais intuitiva e visual**.

Agora que você aprendeu a criar um repositório, adicionar arquivos e registrar alterações de duas formas diferentes, está pronto para começar a gerenciar seus projetos com mais organização e controle!

Bora aprender **como salvar o projeto com o controle de versão no GitHub**, permitindo que ele fique armazenado na nuvem, acessível de qualquer lugar e pronto para colaboração com outras pessoas. Isso tornará seu fluxo de trabalho mais seguro e eficiente!

Módulo 3 - Introdução ao GitHub

Agora que já entendemos o Git e como ele gerencia o versionamento do código localmente, surge uma pergunta: vamos entender um pouco mais do GitHub, o carinha de quem falamos no início do curso. O GitHub permite que você armazene seus repositórios online, colabore com outras pessoas e também acompanhe a evolução dos seus projetos de forma organizada e profissional.

Se você deseja compartilhar o seu código, trabalhar em equipe ou até mesmo construir um portfólio para o mercado de trabalho, o GitHub vai ser uma ferramenta fundametal no seu dia a dia.

3.1 O que é o GitHub e sua importância?

Bom, mas de fato, o que é o GitHub? O GitHub é uma plataforma baseada em nuvem que permite armazenar e gerenciar repositórios Git de forma remota. Ele é amplamente utilizado por empresas, desenvolvedores e no mercado de trabalho para colaboração em projetos de código aberto e privados.

O que você pode fazer no GitHub?

- Armazenar seus códigos na nuvem, aqueles que criamos diretamente no nosso computador ou até mesmo já versionamos no git localmente, garantindo acesso a eles de qualquer lugar.
- Compartilhar e demonstrar seu trabalho, deixando seus projetos de código e automações visíveis para recrutadores ou colaboradores.
- Trabalhar em equipe sem conflito no código, com o controle de versão permitindo que várias pessoas contribuam no mesmo projeto.

Diferença entre Git e GitHub

Muitas pessoas confundem Git com GitHub, mas são ferramentas diferentes:

- **Git** → Software de controle de versão local, instalado no seu computador.
- **GitHub** → Plataforma online para armazenar e compartilhar repositórios Git, facilitando a colaboração e integração de equipes.

Criando uma conta e configurando o GitHub

Agora que entendemos o que é o GitHub, precisamos criar nossa conta para começar a utilizá-lo:

1. Acesse o site oficial: https://github.com
2. Clique em **Sign up** (Cadastrar-se).
3. Preencha os campos com:
    - **Username (Nome de usuário)** → Escolha um nome único.
    - **Email** → Recomendamos utilizar o mesmo email que configuramos no Git para facilitar a integração.
    - **Password (Senha)** → Escolha uma senha segura.
4. Clique em **Create account** e siga as instruções na tela.

A dica aqui é utilizar o mesmo email que utilizamos lá nas aulas iniciais para configurar o Git no nosso computador. Se quiser usar um email diferente, depois será necessário atualizar a configuração do Git local para evitar problemas de autenticação.

Boa, pessoal! Nesta aula aprendemos os conceitos básicos do GitHub. Agora vamos aprofundar um pouco mais e partir para a prática, vamos aprender como conectar um repositório local ao repositório remoto.

Conectando um repositório no GitHub

Vamos fazer a conexão que comentamos na última aula? Do nosso repositório local com um repositório no GitHub? Para iniciar esse processo, vamos criar um repositório remoto no GitHub e a partir dele fazer a conexão com o repositório que já temos no nosso computador. Vamos ver também como configurar um Token de Acesso Pessoal (PAT) para autenticação, já que o GitHub não permite mais o uso direto de senhas no terminal.

Primeiro, precisamos criar um repositório remoto no GitHub:

Criando um repositório no GitHub

1. Acesse o GitHub e clique no botão **New repository**.
2. Escolha um nome para o repositório e defina se ele será **público** ou **privado**.
3. Não marque a opção “Initialize this repository with a README” (pois já temos um repositório local).
4. Clique em **Create repository**.

Prontinho, temos um repositório remoto criado para receber arquivos!

Outra maneira de enviar arquivos para o repositório é fazendo o **upload manualmente** direto pelo GitHub.

Se você já tem um projeto totalmente pronto no seu computador e quer subir para a nuvem sem usar o terminal, siga esses passos:

1. Acesse seu repositório no GitHub e clique na opção **“uploading an existing file”**.

2. Arraste os arquivos do seu projeto para a página ou clique em **“choose your files”** para selecionar manualmente.

Dica: Em vez de arrastar a pasta inteira, arraste apenas os arquivos que estão dentro dela. Se você arrastar a pasta, o GitHub criará um diretório dentro do repositório, e ao acessar o repositório, verá apenas a pasta e não os arquivos diretamente.

Esse método é útil quando você já tem um projeto pronto e quer adicionar ele ao GitHub. Mas, se for um projeto em desenvolvimento, o ideal é usar o Git desde o início para versionar o código e garantir que nada se perca.

3.2 Conectando o Repositório Local ao GitHub

Agora que o repositório foi criado no GitHub, precisamos conectá-lo ao nosso projeto local. Isso pode ser feito de **duas formas**: pelo **terminal** ou diretamente pelo **VSCode**.

Conectando o repositório pelo Terminal

Se você prefere fazer todo o processo pelo terminal, siga os passos abaixo:

No terminal, dentro da pasta do projeto, execute o comando:

```
git remote add origin https://github.com/seu-usuario/seu-projeto.git
```

Esse comando cria um vínculo entre o seu repositório local e o repositório remoto no GitHub. Como o GitHub não permite mais autenticação via senha diretamente pelo terminal, precisamos gerar um **Token de Acesso Pessoal (PAT)**. Para isso:

1. Acesse o GitHub e vá até **Settings → Developer settings → Personal access tokens → Tokens (classic)**
2. Clique em **Generate new token (classic)**
3. Escolha um nome para o token e selecione as permissões necessárias (marque a opção **repo** para acesso aos repositórios)
4. Clique em **Generate token** e **copie** o token gerado. **Atenção!** Esse token só será exibido uma vez

Agora que temos o token, podemos usá-lo para enviar o código para o GitHub. No terminal, rode:

```
git push -u origin main
```

Se for a primeira vez que está enviando o código para o GitHub nesse repositório, o terminal pode pedir o usuário do GitHub antes de solicitar a senha.

1. Digitar seu nome de usuário do GitHub.
2. Quando ele pedir a senha, colar o token gerado no lugar da senha.

Se você já configurou previamente as credenciais do Git no seu computador, pode ser que o Git não peça o nome de usuário e vá direto para a solicitação do token. Caso queira evitar essa solicitação toda vez que fizer um push, pode configurar o Git para lembrar suas credenciais com:

```
git config --global credential.helper store
```

Assim, ao colar o token uma vez, ele será salvo e não será necessário digitá-lo novamente nas próximas operações.

O comando `git push` será um dos mais utilizados ao trabalharmos com o GitHub. Ele é responsável por enviar ("empurrar") os commits feitos no repositório local para o repositório remoto, garantindo que as mudanças fiquem salvas na nuvem e possam ser acessadas de qualquer lugar. Sempre que finalizarmos uma etapa do projeto e quisermos atualizar o código no GitHub, utilizaremos o `git push` para sincronizar as alterações, seja pelo terminal ou diretamente pelo VSCode.

Outro comando importante que utilizamos quando trabalhamos com repositórios remotos é o `git pull`. Esse comando é essencial para manter seu repositório local atualizado com as mudanças realizadas no repositório remoto. Sempre que alguém atualizar o projeto na nuvem, utilizar o `git pull` garante que você esteja trabalhando com a versão mais recente, pois ele "puxa" do repositório remoto as atualizações, deixando seu repositório local em sincronia com o remoto para que você possa continuar trabalhando sem conflitos.

Conectando o repositório pelo VSCode

Se preferir fazer o processo diretamente pelo **VSCode**, vai ser preciso seguir os seguintes passos:

1. Certifique-se de que a pasta do seu projeto está aberta no VSCode.

Adicionando o repositório remoto

1. Acesse a aba de **Controle de Versão** no menu lateral esquerdo
2. Clique nos **três pontinhos (...),** selecione **Remote > Add Remote**
3. Cole a **URL do repositório** do GitHub e pressione Enter

4. O VSCode solicitará que você **defina um nome para o repositório remoto**. O nome mais usado é **origin**, pois é o nome padrão para referenciar repositórios remotos no Git
5. Confirme pressionando **Enter**

Após adicionar o repositório remoto, o VSCode pode exibir um pop-up solicitando a autenticação no GitHub:

1. Clique em **Allow** para permitir a conexão entre o VSCode e o GitHub
2. O VSCode abrirá um navegador pedindo para confirmar o login com sua conta do GitHub

Enviando os arquivos para o GitHub (Push pelo VSCode)

- Após confirmar o login, no **Source Control**, digite uma mensagem para o commit e clique no ícone para confirmar

Depois de fazer o commit, precisamos enviar as alterações para o repositório remoto no GitHub. No VSCode, temos duas opções para isso: **Push** e **Sync Changes**.

- **Push** → Envia apenas os commits locais para o repositório remoto. Essa opção é útil quando temos certeza de que ninguém mais alterou o repositório no GitHub.
- **Sync Changes** → Além de enviar os commits locais com o **Push**, também verifica se há mudanças no repositório remoto e baixa essas alterações para o seu computador através do comando **Pull**. Isso é útil quando estamos trabalhando em equipe, garantindo que nosso código esteja sempre atualizado.

Quando o botão **Sync Changes** fica disponível, indica que há alterações locais prontas para serem enviadas ao GitHub. Para finalizar o processo, basta clicar nesse botão e aguardar a sincronização.

Confirmando no GitHub

1. Acesse o seu repositório no **GitHub** e atualize a página
2. Agora, todos os arquivos e commits devem aparecer lá!

Agora que aprendemos a conectar nosso repositório ao GitHub, podemos seguir para os próximos passos do curso!

3.3 Clonando e colaborando com repositórios existentes

Além de criar repositórios do zero, muitas vezes a gente precisa clonar um repositório existente para trabalhar nele no nosso computador ou até contribuir com um projeto que já está em andamento.

Isso é bem útil quando queremos testar um código pronto, fazer alguma modificação ou até ajudar a melhorar um projeto de um colega ou de código aberto.

Por exemplo, vamos clonar o repositório do Pandas, que é uma das bibliotecas mais populares para análise de dados em Python. O código dela está disponível no GitHub para qualquer pessoa acessar e até contribuir.

Para fazer isso, basta entrar no repositório, clicar no botão **<>Code** e copiar o link HTTPS. Esse link permite que a gente clone o repositório para o nosso computador, criando uma cópia exata do código que está no GitHub. Assim, podemos explorar os arquivos, testar mudanças e até sugerir melhorias.

Vamos ver esse processo com mais detalhes logo abaixo.

Clonando um repositório do GitHub

Lembrando que clonar um repositório pode ser útil quando você deseja:

- Trabalhar em um repositório já existente.
- Baixar um projeto open-source para estudar ou contribuir.
- Criar uma cópia do seu próprio repositório para trabalhar em outra máquina.

Depois de copiar o link HTTPS vai ser preciso utilizar o comando `git clone` para copiar o repositório do GitHub para o seu computador.

Se for realizar o clone pelo terminal basta entrar na pasta e que você deseja colocar o projeto e rodar pelo terminal:

```
git clone <URL_COPIADA>
```

Passo a passo para clonar um repositório pelo VSCode Se preferir, também é possível clonar um repositório diretamente pelo VSCode:

1. **Acesse o repositório no GitHub**
    - No GitHub, vá até o repositório que deseja clonar.
    - Clique no botão **Code** e copie a **URL HTTPS**.
2. **Abra o VSCode**
    - No **VSCode**, acesse o menu **Source Control** (Controle de Versão).
    - Clique em **Clone Repository**.
    - Cole a **URL do repositório** copiada no GitHub.
    - Escolha uma pasta no seu computador para armazenar o projeto.

3. **Abrir o repositório clonado**
    - Após a clonagem, o VSCode perguntará se deseja abrir a pasta clonada.
    - Clique em **Open** para começar a trabalhar no projeto.

Agora, o repositório clonado já está pronto para ser modificado e versionado no seu ambiente local.

Um ponto importante a ser destacado é que, ao realizar commits, você só vai conseguir enviar as alterações para o GitHub caso tenha autorização para aquele repositório. Por isso, realizamos a conexão do GitHub e a configuração do Git com nosso email, garantindo que tenhamos permissão para versionar e sincronizar as mudanças nos nossos próprios repositórios.

Caso o projeto seja open-source e público como o Pandas, você pode criar uma nova branch com suas alterações e enviar para o repositório original. O dono do projeto poderá revisar seu código e, se aprovado, incorporar suas contribuições no repositório oficial. Vamos ver ao longo do curso como esse processo funciona com mais detalhes.

Desafio Prático: Clonando e modificando um repositório

Agora que aprendemos como clonar repositórios, vamos praticar baixando o repositório que criamos anteriormente no GitHub, editando um arquivo e enviando a alteração de volta para a nuvem.

Passo a passo

1. Clone o repositório “Hello World” no VSCode
    - Acesso o repositório do Octocat: https://github.com/octocat/Hello-World(
    - No **Source Control**, clique em Clone Repository e cole a **URL** do repositório que você copiou.
    - Escolha uma pasta no seu computador para armazená-lo.
    - Abra o repositório no VSCode.
2. Modifique um arquivo
    - No VSCode, abra o arquivo do projeto.
    - Adicione uma nova linha de texto ou faça qualquer modificação simples.
    - Salve o arquivo (`Ctrl + S` ou `Cmd + S` no Mac).
3. Adicione e salve alterações de forma local no seu computador

- No Source Control, clique em **+** para adicionar o arquivo ao staging.
- Escreva uma mensagem no commit e clique em **Commit**.

Agora você conseguiu clonar um repositório, modificar um arquivo e salvar as alterações localmente usando apenas o VSCode!

No próximo módulo, vamos entender como trabalhar com múltiplas branches e como resolver conflitos ao juntar essas branchs!

3.4 O que é o .gitignore e como utilizá-lo

O arquivo **.gitignore** é uma ferramenta essencial no Git para indicar quais arquivos ou pastas devem ser ignorados pelo controle de versão. Quando você cria um repositório, seja no GitHub ou localmente, esse arquivo permite que você defina quais arquivos não serão rastreados, evitando que dados sensíveis, arquivos temporários ou dependências desnecessárias sejam enviados para o repositório remoto.

Qual a função do .gitignore?

A função principal do **.gitignore** é manter o repositório limpo e seguro. Ao ignorar arquivos desnecessários, você reduz o risco de expor informações confidenciais e torna o histórico de commits mais organizado. Por exemplo, você pode querer ignorar:

- Arquivos de configuração local;
- Arquivos de cache ou logs;
- Chaves de API, tokens ou outras credenciais sensíveis.

Exemplo prático

Imagine que você possui um arquivo contendo uma chave de API que não deve ser compartilhada. O conteúdo deste arquivo pode ser algo como:

```
API_KEY=123456789abcdef
```

Para garantir que este arquivo não seja enviado para o repositório, você deve listá-lo no **.gitignore**:

```
# Ignorar arquivo contendo chave de API
.env
```

Nesse exemplo, o arquivo `.env` contém variáveis de ambiente, como chaves de API, e ao adicioná-lo ao **.gitignore**, garantimos que essas informações fiquem apenas no seu ambiente local.

Criação e utilização do .gitignore

Você não precisa esperar que o GitHub crie o arquivo **.gitignore** automaticamente. Você pode criá-lo localmente, seguindo estas etapas:

1. Crie um novo arquivo em seu projeto e nomeie-o como **.gitignore**.
2. Dentro dele, siga o padrão de listagem para ignorar arquivos ou pastas. Por exemplo:

```
# Ignorar arquivos de ambiente
.env

# Ignorar diretórios de dependências
node_modules/

# Ignorar arquivos de log
*.log
```

3. Salve o arquivo e, ao fazer o commit, o Git já passará a ignorar os itens listados.

Com o **.gitignore**, você garante que somente os arquivos relevantes e seguros façam parte do repositório, facilitando o trabalho colaborativo e mantendo seu projeto mais organizado.

Essa aula em texto traz uma visão sobre o **.gitignore** e sua importância para o controle de versão. Agora, você já sabe como criar e configurar esse arquivo tanto localmente quanto utilizando as opções do GitHub, protegendo informações sensíveis e mantendo seu repositório limpo.

3.5 Autenticação com SSH: Uma Alternativa ao Token PAT

Como vimos anteriormente, o **token PAT** serve para autenticar operações com o GitHub, permitindo que você execute comandos e acesse seus repositórios de forma segura e prática. Entretanto, existe outra abordagem que também garante segurança e facilidade de uso: a autenticação via **chave SSH**.

Por que utilizar a chave SSH?

A chave SSH é uma ferramenta que cria uma conexão criptografada entre o seu computador e o GitHub. Ao gerar um par de chaves (pública e privada), você adiciona a chave pública à sua conta no GitHub,

enquanto a chave privada permanece protegida em seu dispositivo. Assim, sempre que você interagir com um repositório, o GitHub verifica essa correspondência e permite o acesso sem a necessidade de digitar senhas ou tokens.

Exemplo prático

Ao clonar um repositório com SSH, o comando seria:

```
git clone git@github.com:usuario/nome-do-repositorio.git
```

Esse método é especialmente útil para quem prefere uma abordagem mais direta e evita a necessidade de atualizar tokens periodicamente.

Criando sua chave SSH localmente

Você pode criar a chave SSH localmente seguindo estes passos:

1. **Gerar o par de chaves SSH:**
   Abra o terminal e execute o comando abaixo, substituindo `"seu_email@exemplo.com"` pelo seu e-mail:

   ```
   ssh-keygen -t ed25519 -C "seu_email@exemplo.com"
   ```

   Caso seu sistema não suporte o algoritmo `ed25519`, utilize:

   ```
   ssh-keygen -t rsa -b 4096 -C "seu_email@exemplo.com"
   ```

2. **Salvar a chave:**
   Durante o processo, aceite o local padrão (geralmente, `~/.ssh/id_ed25519` ou `~/.ssh/id_rsa`) e defina uma senha para maior segurança, se desejar.

3. **Adicionar a chave ao agente SSH:**
   Inicie o agente SSH e adicione sua chave:

   ```
   eval "$(ssh-agent -s)"
   ssh-add ~/.ssh/id_ed25519
   ```

4. **Copiar a chave pública:**
   Para copiar a chave pública e adicioná-la à sua conta do GitHub, utilize:

   ```
   cat ~/.ssh/id_ed25519.pub
   ```

5. **Adicionar a chave ao GitHub:**
   Vá até as configurações da sua conta no GitHub, na seção “SSH and GPG keys”, clique em “New SSH key”, cole a chave pública e salve.

Com a chave SSH configurada, suas operações com o GitHub se tornam ainda mais seguras e práticas, oferecendo uma alternativa robusta ao uso do token PAT. Ambas as abordagens visam facilitar a autenticação, mas a escolha entre elas depende das suas preferências e necessidades no gerenciamento dos repositórios.

Módulo 4 - Trabalhando com Branches e Resolvendo Conflitos

Agora que já entendemos como versionar nosso código e como conectar nossos repositórios ao GitHub, chegou o momento de dar um passo além no controle de versão. Até agora, trabalhamos apenas com uma única linha do tempo, onde cada commit era feito diretamente na branch principal. Mas e se quisermos testar algo novo sem bagunçar nosso código principal? Ou se estivermos trabalhando em equipe e cada pessoa precisar desenvolver uma funcionalidade separadamente?

É aí que entram as **branches**. Elas permitem que possamos criar versões paralelas do código, testar mudanças, corrigir bugs e depois integrar tudo de volta sem comprometer o projeto principal.

4.1 Branches no Git: A árvore do seu projeto

Lembra que, quando estávamos trabalhando com o Git nas aulas anteriores, falamos que **nossas versões do código ficam salvas em uma linha do tempo**? Pois é, agora vamos aprofundar um pouco mais nisso. Até agora, trabalhamos em uma única linha do tempo, onde cada commit era um novo ponto de progresso no projeto.

Mas e se quisermos desenvolver algo novo sem alterar essa linha do tempo principal? E se precisarmos testar um recurso sem comprometer o código que já está funcionando? É aí que entram as **branches**!

Se pensarmos em um projeto de software como uma **árvore**, podemos imaginar que o tronco principal representa a **branch principal** (`main` ou `master`). Assim como em uma árvore real, esse tronco é a base que sustenta todo o crescimento do projeto.

Quando queremos desenvolver uma nova funcionalidade, corrigir um bug ou testar uma ideia sem interferir diretamente no tronco principal, criamos **galhos**, ou seja, **branches**. Esses galhos crescem paralelamente, mas ainda fazem parte da mesma árvore:

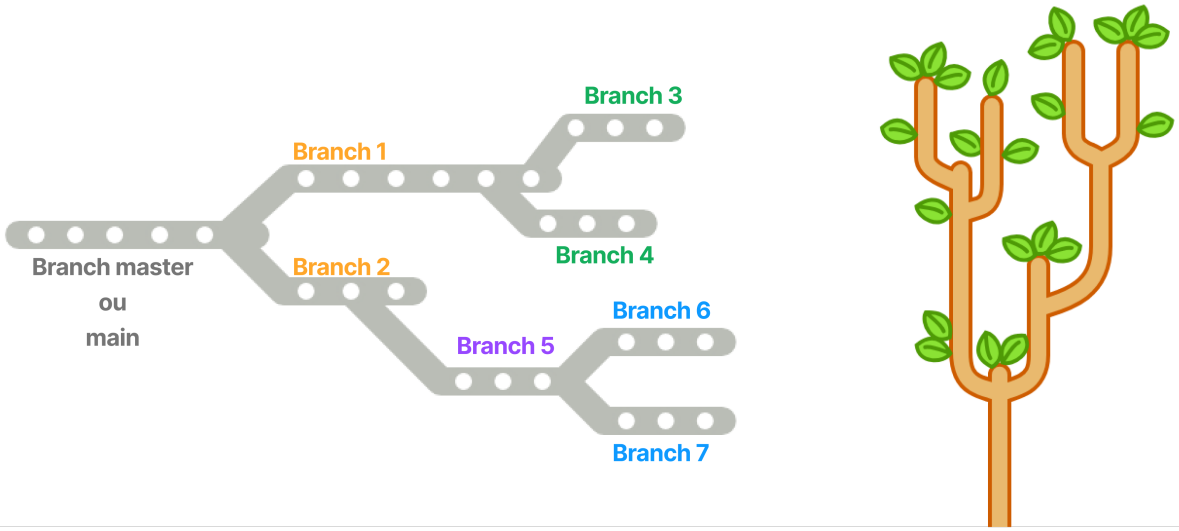


Figure 4: Árvore de código

Na imagem, podemos ver como as branches no Git funcionam exatamente como os galhos de uma árvore. Cada nova branch se desenvolve de maneira independente, permitindo testar e implementar novas funcionalidades sem alterar o código original.

Como as branches funcionam no fluxo de trabalho

Quando um projeto começa, geralmente temos apenas a branch principal (`main`). Mas, conforme o desenvolvimento avança, precisamos adicionar novos recursos, testar ideias ou corrigir erros. Para isso, criamos **branches secundárias**, que podem ser integradas de volta à branch principal quando estiverem prontas.

Agora que já entendemos que cada branch é um galho independente do projeto, vamos dar um passo além e entender como os commits funcionam dentro de cada branch.

Cada commit é um ponto de salvamento do código dentro de uma branch. Isso significa que podemos trabalhar em diferentes partes do projeto ao mesmo tempo, fazendo commits separadamente em cada branch.

Na imagem abaixo, cada ponto colorido representa um commit em uma branch específica:

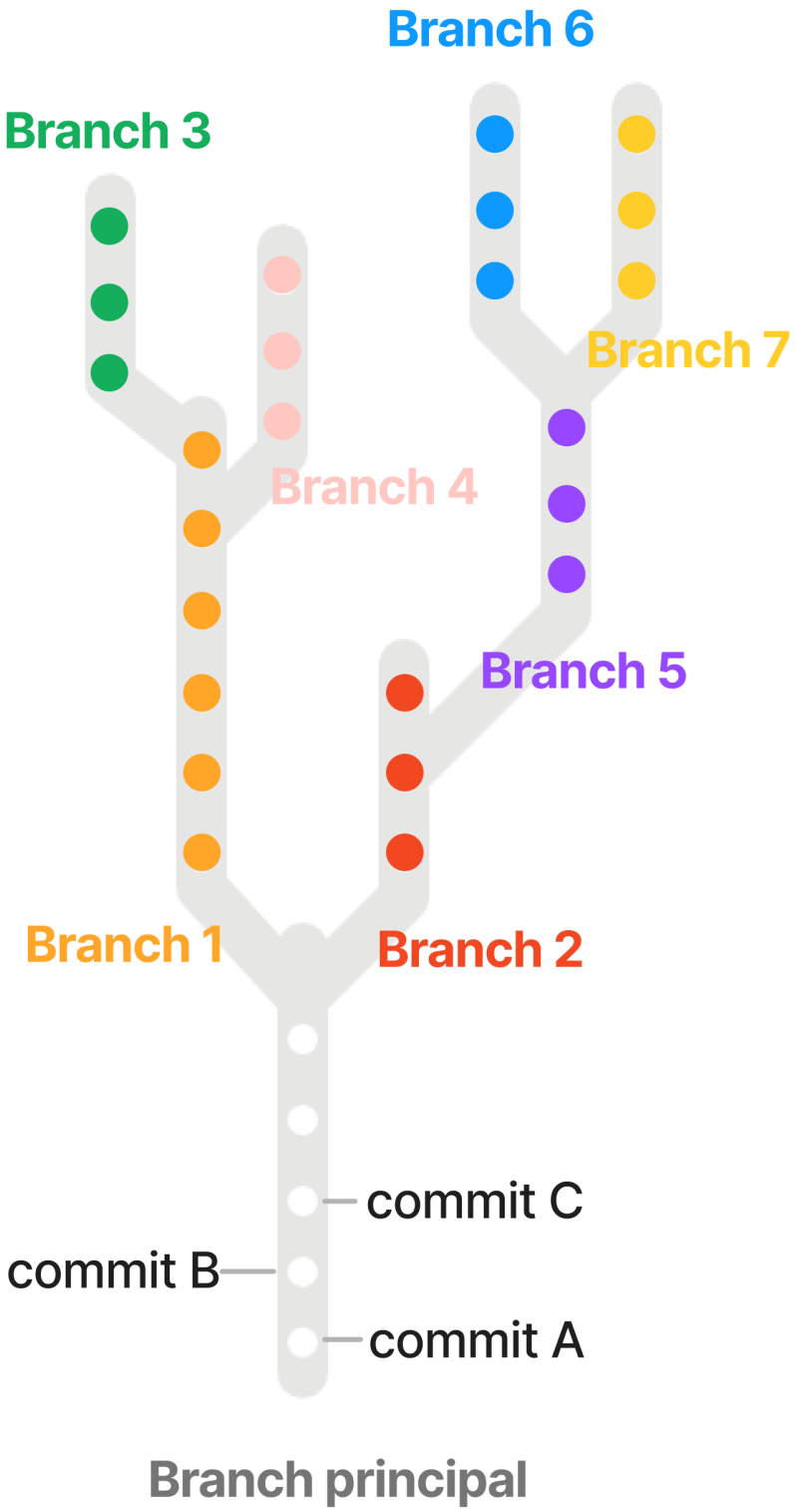


Figure 5: Detalhe das branchs/galhos

Cada ponto colorido representa um commit feito em uma branch específica. Podemos observar que:

- Cada branch possui seu próprio conjunto de commits, permitindo o desenvolvimento independente de funcionalidades.
- Os commits são organizados dentro de cada branch, criando um histórico separado para cada linha de desenvolvimento.
- Quando necessário, as branches podem ser unidas novamente à branch principal, combinando todas as mudanças realizadas.

Isso mostra que o desenvolvimento de um projeto não é linear, mas sim ramificado, como o crescimento de uma árvore! Assim, podemos testar, corrigir e melhorar o código de maneira organizada, sem impactar diretamente a versão principal do projeto.

Por que usar branches?

Imagine que você está criando um **bot de automação** para enviar notificações por e-mail. No meio do desenvolvimento, você decide adicionar uma nova funcionalidade: integrar o bot com um sistema de mensagens no Telegram.

Em vez de modificar diretamente o código principal na branch `main`, o ideal é criar uma nova branch chamada `bot-telegram`. Assim, você pode desenvolver essa nova funcionalidade sem interferir no que já está funcionando.

Enquanto isso, um colega pode estar trabalhando em uma outra melhoria, como um sistema de **agendamento automático de envios**, dentro da branch `agendamento-envios`. Dessa forma, cada um pode desenvolver suas funcionalidades separadamente, garantindo que nenhuma mudança afete o código do outro antes de estarem prontas para serem integradas.

Quando as funcionalidades estiverem prontas e testadas, podemos juntar elas de volta à branch principal sem causar problemas ou misturar códigos inacabados.

Criando e gerenciando branches

Agora que entendemos o conceito de **branches**, vamos criar e gerenciar uma nova branch (usando o terminal Git Bash ou o próprio terminal do VSCode).

Criando uma nova branch

1. Abra o terminal na pasta do seu projeto.

2. Antes de criar uma nova branch, vamos conferir as que já existem no repositório com o comando:

```
git branch
```

Aqui vai aparecer a branch atual e as demais, caso existam. Como estamos usando o projeto que criamos na aula de Git deve aparecer apenas a branch `main`.

3. Agora que verificamos as branchs que existem, vamos criar uma nova branch. Para isso, digite o comando para criar uma nova branch chamada `nova-feature`:

```
git branch nova-feature
```

4. Boa! Vamos entrar na nossa nova branch. Para alternar para qualquer branch no Git, basta rodar o comando `git checkout` seguido do nome da branch desejada. No nosso caso, para entrar na `nova-feature`, usamos:

```
git checkout nova-feature
```

Estando tudo certo, qualquer alteração que fizermos a partir de agora ficará registrada apenas na branch `nova-feature`, sem afetar a `main`.

Fazendo uma alteração e commitando na nova branch

Então pessoal, já que criamos uma nova branch, vamos modificar um arquivo dentro dela e salvar essa alteração para ver como isso funciona na prática.

1. Abra o arquivo `mensagem.py` no seu editor de código.

2. Edite o código, adicionando uma nova linha:

```
mensagem = "Olá, Git e VSCode!"
print(mensagem)
print("Nova funcionalidade adicionada!")
```

3. Salve o arquivo (`Ctrl + S` no Windows).

4. Agora, no terminal, verifique o status do repositório:

```
git status
```

O Git vai mostrar que o arquivo `mensagem.py` foi modificado.

5. Adicione essa alteração ao **staging**:

```
git add mensagem.py
```

6. Agora, crie um commit registrando essa alteração na `nova-feature`:

```
git commit -m "Adicionando nova funcionalidade"
```

7. Para garantir que estamos na **nova-feature**, podemos conferir a branch ativa com:

```
git branch
```

O Git vai mostrar todas as branches disponíveis e marcar com `*` a branch ativa.

Aqui o comportamento esperado é que as alterações commitadas estejam apenas na branch `nova-feature`, então que tal vamos alternarmos para a branch `main` e ver o que acontece?!

```
git checkout main
```

Estando na `main`, se você abrir o arquivo `mensagem.py`, verá que a alteração que fizemos desapareceu!

Isso acontece porque cada branch tem seu próprio histórico de commits e alterações. A `main` continua exatamente na sua própria linha do tempo, como estava antes da criação da nova branch.

Se voltarmos para `nova-feature`, a alteração reaparece. Isso demonstra como as branches funcionam como versões paralelas do código.

Enviando a nova branch para o GitHub

Avançamos bastante explorando o uso das branchs nessa aula, alternando entre branchs localmente. Mas e o GitHub? Bom, vamos enviar essa **nova branch** para o **GitHub** e visualizar como ela aparece lá!

1. Primeiro, verifique se a branch `nova-feature` está ativa:

```
git branch
```

Se necessário, mude para ela:

```
git checkout nova-feature
```

2. Agora, envie essa branch para o GitHub:

```
git push -u origin nova-feature
```

Quando queremos enviar nossas mudanças do computador para o GitHub, usamos `git push`, que literalmente “empurra” os arquivos para lá. Esse comando avisa ao Git que queremos salvar e sincronizar nossas alterações na nuvem, deixando o código acessível de qualquer lugar.

O `origin` nesse comando já indica que estamos enviando as mudanças para o repositório remoto. Mas, para garantir que estamos mandando uma branch específica, podemos rodar `git push -u origin nome-da-branch`. Isso faz com que o Git lembre dessa branch para os próximos envios, permitindo que no futuro a gente só precise digitar `git push`.

Visualizando as branches no GitHub

Depois de enviarmos pro GitHub a `nova-feature`, podemos conferir como ela aparece lá no repositório. Para isso:

1. No GitHub, acesse o repositório.
2. No canto superior esquerdo, clique no menu de branches.
3. Selecione `nova-feature` e veja se a alteração que fizemos está lá.
4. Agora, alterne para `main` e verifique se a modificação não aparece nessa branch.

Viu como é simples esse processo? Dá para visualizar e entender o código e as branchs tanto no nosso computador, onde temos o código, quanto diretamente no GitHub, que podemos acessar de qualquer PC ou celular.

Depois de aprender como criar, editar e alternar entre as branches, fica mais claro como cada uma pode ter mudanças isoladas sem interferir uma na outra, não é mesmo?

Bom, vamos ver como podemos juntar essa nova branch na nossa branch principal e também como resolver possíveis conflitos que podem surgir nesse processo. Afinal, quando tentamos unir diferentes versões de um código, pode ser que nem sempre tudo se encaixe perfeitamente de primeira.

4.2 Combinando branches e lidando com conflitos

É hora de unir tudo no código principal! No Git, esse processo de juntar duas branches é chamado de **merge**, e é ele que permite combinar as alterações feitas em uma branch dentro de outra.

Para entender bem como funciona, vamos testar duas situações:

1. **Fazer um merge sem conflitos**, quando as mudanças das branches não interferem uma na outra.

2. **Criar um conflito propositalmente**, para entender como resolver quando o Git não consegue decidir automaticamente quais alterações manter.

Merge sem conflitos no VSCode

Primeiro, vamos pegar o projeto que estávamos trabalhando na última aula e mesclar a branch que criamos (`nova-feature`) com a branch principal (`main`). Se você já estiver com o projeto aberto no **VSCode**, seguimos direto os passos abaixo. Caso contrário, abra o seu projeto para fazermos o merge juntos!

Passo a passo

1. Volte para a branch principal (`main`)
    - No **VSCode**, clique no nome da branch atual (canto inferior esquerdo).
    - Selecione **main** para alternar.
2. Inicie o **merge**
    - Vá até **Source Control > ... > Branch > Merge Branch**.
    - Escolha `nova-feature` como a branch que queremos mesclar.

Se as mudanças feitas na `nova-feature` não entrarem em conflito com o código da `main`, o Git adiciona tudo automaticamente. Caso isso aconteça, parabéns! Você fez seu primeiro merge sem conflitos.

Vamos ver o que acontece quando duas branches tentam modificar a mesma parte do código?!

Criando um conflito para testar o merge

O Git não consegue mesclar automaticamente alterações que afetam a mesma linha de código em duas branches diferentes. Para entender isso melhor, vamos criar esse cenário.

Passo a passo para criar um conflito

1. Crie uma nova branch para simular um conflito
    - No **VSCode**, vá até **Source Control > ... > Branch > Create Branch**.
    - Nomeie essa nova branch como `ajuste-conflito`.
2. Modifique um arquivo
    - Abra um arquivo do projeto (por exemplo, `mensagem.py`).
    - Substitua a linha com a mensagem original por outra diferente, como:
      ```
      mensagem = "Alteração feita na branch ajuste-conflito!"
      print(mensagem)
      ```
    - Salve o arquivo e faça um **commit** dessa alteração.
3. Volte para a branch `main` e faça uma outra modificação no mesmo arquivo

- Vá até **Source Control > ... > Branch > Switch to Branch** e selecione `main`.

- Agora, abra **o mesmo arquivo (`mensagem.py`)** e altere a **mesma** linha, mas com um texto diferente:

```
info = "Alteração feita diretamente na branch main!"
print(info)
```

- Salve o arquivo e faça um **commit**.

Agora temos um cenário onde **a mesma linha do código foi alterada em duas branches diferentes**. Isso gera um **conflito** na hora do merge porque o Git não sabe automaticamente qual das alterações deve ser mantida. Nesse momento, ele **pede para você decidir** qual versão do código faz mais sentido: manter uma das alterações, combinar as duas ou fazer um novo ajuste manualmente antes de concluir o merge.

Fazendo um merge com conflito

Com esse cenário criado, vamos tentar juntar as duas branches e ver como o Git lida com isso.

1. Volte para a branch `main`
    - No **VSCode**, clique na branch atual e selecione `main`.
2. Tente fazer o merge da `ajuste-conflito`
    - Vá até **Source Control > ... > Branch > Merge Branch**.
    - Escolha `ajuste-conflito`.

Agora o Git vai detectar o problema e exibir um **conflito**.

Resolvendo Conflitos no VSCode

Quando um conflito acontece, o **VSCode** marca os arquivos problemáticos e exibe diferentes versões do código.

- Passo a passo para resolver o conflito

1. Abra o arquivo com conflito.
    - O VSCode mostrará as duas versões:
        - HEAD (a versão da `main`)
        - A versão da `ajuste-conflito`

2. Escolha como resolver:
    - Você pode optar por **manter a versão da `main`**, **manter a versão da branch `ajuste-conflito`**, ou **combinar as duas manualmente**.
    - O VSCode facilita esse processo mostrando opções como **"Aceitar versão atual"**, **"Aceitar versão vinda da branch"** ou **"Aceitar ambas"**.
3. Edite o código para ficar como deseja.

    Se quisermos combinar as duas versões, o código final pode ficar assim:

    ```
    mensagem = "Alteração feita na branch ajuste-conflito e ajustada na main!"
    print(mensagem)
    ```

4. Marque o conflito como resolvido e faça um commit.

Depois de ajustar o código, vá até a aba **Source Control**, clique em **Mark as Resolved** e faça um novo commit.

Pronto! Você resolveu seu primeiro conflito de merge.

Visualizando o Histórico de Branches

Agora que trabalhamos com **branches** e resolvemos conflitos, podemos visualizar melhor como essas mudanças foram registradas no histórico do Git.

Para isso, vamos abrir o **terminal integrado do VSCode**:

1. No **VSCode**, clique em **View** (ou **Exibir** se estiver em português).
2. Selecione **Terminal** (ou use o atalho `Ctrl + J` no Windows/Linux ou `Cmd + J` no macOS).

Agora, execute o seguinte comando:

```
git log --graph
```

Isso gera um **gráfico visual** mostrando todas as **branches** e seus **commits**, facilitando a organização e o acompanhamento do fluxo de trabalho no projeto.

No próximo módulo, vamos entender o que são PRs, os Pull Requests no GitHub e como eles ajudam a organizar e revisar mudanças em projetos colaborativos.

Por mais que a gente possa fazer merges rapidamente como vimos aqui, em projetos com várias pessoas isso pode gerar conflitos ou perda de informações. Os PRs evitam esses problemas ao permitir que as alterações sejam revisadas antes de serem integradas.

Módulo 5 - Trabalhando em equipe com GitHub e Pull Requests

Fala pessoal! Até aqui, aprendemos a usar o Git para salvar versões do nosso código, criar branches para organizar mudanças e até resolver conflitos quando unimos diferentes versões. Mas como isso funciona quando estamos trabalhando em equipe?

Seja dentro de uma empresa, em um projeto pessoal com um amigo ou até mesmo em contribuições para repositórios públicos, precisamos de uma forma segura e organizada para revisar e integrar as mudanças de cada pessoa. E é aí que entram os **Pull Requests** no GitHub.

5.1 Trabalhando em equipe com GitHub

No GitHub, podemos adicionar colaboradores ao nosso repositório para que mais pessoas tenham acesso ao projeto. Assim, cada um pode trabalhar em diferentes funcionalidades ao mesmo tempo, sem interferir no código do outro.

Mas o que é um Pull Request (PR)?

Quando trabalhamos em equipe, nem sempre é uma boa ideia sair unindo mudanças no código principal sem revisar antes. Para isso, usamos um Pull Request (PR).

O Pull Request permite que uma pessoa peça para que suas mudanças sejam revisadas antes de serem integradas ao projeto principal. Isso garante que todos possam conferir as alterações, sugerir melhorias ou até identificar possíveis erros antes que o código seja incorporado ao repositório.

Como funciona esse fluxo de trabalho?

Vamos imaginar que estamos desenvolvendo um bot em Python para automatizar o envio de e-mails na nossa empresa. Você e seu colega estão programando juntos, mas cada um cuida de uma parte:

- Você está criando o código que gera os relatórios.
- Seu colega está implementando a função de envio dos e-mails.

Ambos precisam fazer alterações no código ao mesmo tempo, mas sem bagunçar nada. Para isso, o fluxo básico seria:

1. Cada pessoa trabalha em uma branch separada → Isso evita que mudanças interfiram diretamente no código principal.

2. As alterações são revisadas antes de entrarem na branch principal → A gente usa um Pull Request para verificar o que foi alterado e garantir que está tudo certo.

3. Se tudo estiver correto, o código é mesclado na branch principal → Assim, garantimos que o projeto final está sempre organizado e funcionando.

Esse processo é essencial para equipes que colaboram no mesmo código, garantindo um fluxo estruturado e evitando conflitos desnecessários. O diagrama abaixo ilustra esse fluxo em detalhes:

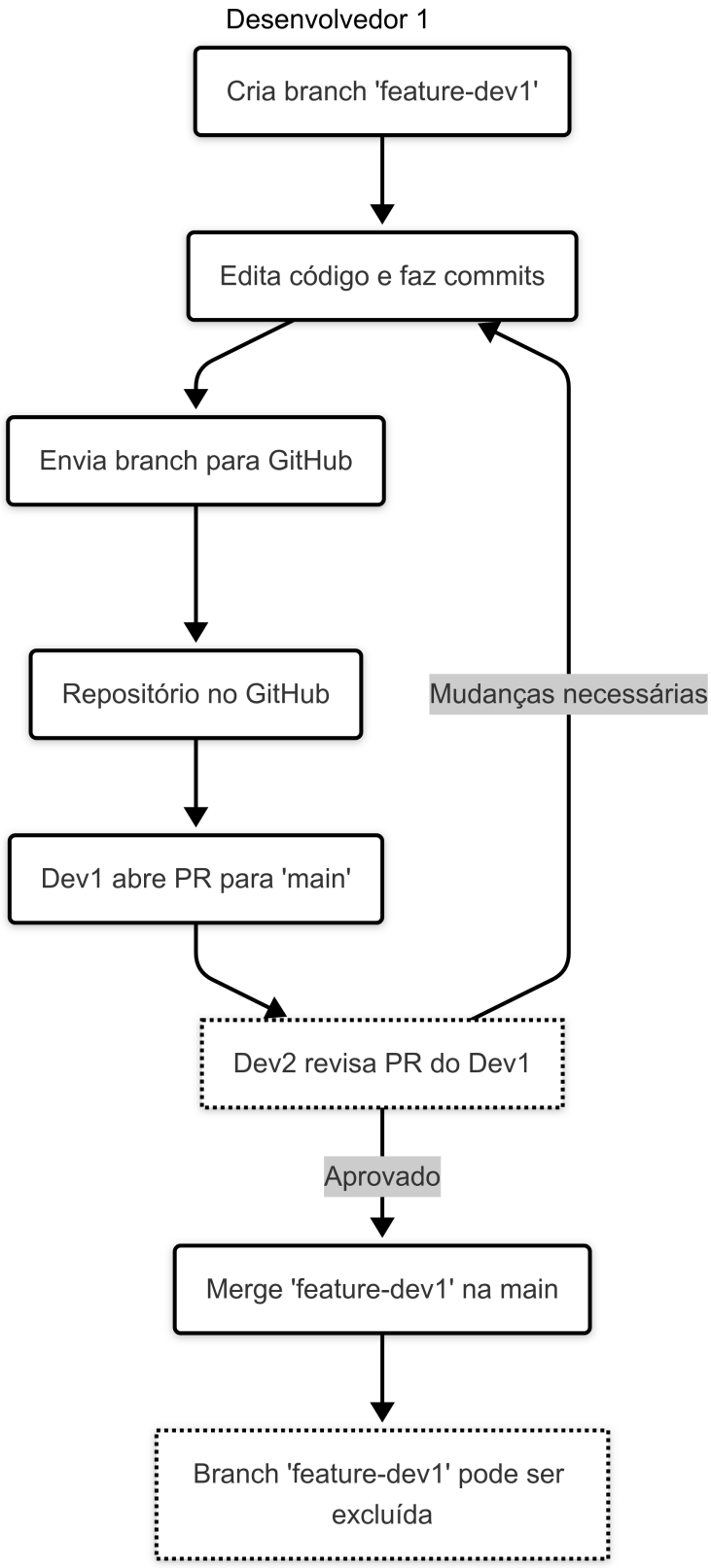


Figure 6: Fluxo de uma PR

5.2 Criando e Gerenciando Pull Requests no GitHub

Agora que já entendemos o fluxo de trabalho em equipe, vamos ver como colocar isso em prática!

Criando um Pull Request no GitHub

Para criarmos um **Pull Request**, vamos simular um trabalho em equipe no nosso projeto. Bora abrir o projeto da última aula e imaginar que temos um colega também colaborando no código. Para organizar esse fluxo, cada um deve trabalhar em uma **branch separada**, evitando os conflitos que discutimos antes. Vamos editar a branch **nova-feature**, subir nossa alteração e, em seguida, abrir um **PR** para que outra pessoa da equipe revise e aprove antes de mesclarmos as mudanças na branch principal.

Passo a passo:

- 1. Certifique-se de estar na branch correta

1. No **VSCode**, vá até o **Source Control**.
2. No canto inferior esquerdo, clique no nome da branch atual `main`.

3. No menu que aparece, selecione a branch em que estávamos trabalhando `nova-feature`.

Agora, qualquer alteração feita será registrada nessa branch e não afetará a `main`.

- 2. Faça alterações no código e commit no Git

1. No **VSCode**, abra o arquivo `mensagem.py`.

2. Modifique o código adicionando uma nova funcionalidade:

```
mensagem = "Olá, Git e VSCode!"
print(mensagem)

def saudacao(nome):
    print(f"Seja bem-vindo, {nome}!")

saudacao("Asimover")
```

3. Salve o arquivo (`Ctrl + S` ou `Cmd + S` no Mac).

4. Vá até a aba **Source Control** e clique no botão **+** ao lado do arquivo modificado para adicioná-lo ao staging.

5. No campo de mensagem, escreva algo como `"Adicionando função de saudação"` e clique em **Commit**.

- 3. Enviar a branch para o GitHub

Agora vamos enviar essa branch para o GitHub.

1. No **Source Control**, clique nos três pontinhos **(...)** no topo.

2. Selecione **Push to...** e escolha `origin`.

3. Isso enviará a branch para o GitHub e deixará ela disponível para criarmos um Pull Request.

4. Criando um Pull Request no GitHub

1. Vá até o seu repositório no GitHub.

2. O GitHub já vai sugerir a criação de um **Pull Request** para a branch recém-enviada.

3. Clique em **Compare & pull request**.

4. No campo de descrição, explique brevemente as mudanças feitas no código.

5. Clique em **Create Pull Request**.

Agora o Pull Request está aberto e pode ser revisado pelos seus colegas antes de ser mesclado na branch principal.

5. Revisando e Aceitando um Pull Request

Se você fosse o colega trabalhando no projeto, receberia uma notificação de que um **Pull Request (PR)** foi criado no repositório. Quando você tem permissão para gerenciar o repositório, pode visualizar e aprovar os PRs antes que as alterações sejam mescladas na branch principal.

Bora simular essa etapa de aprovação do PR? Para isso:

1. Vá até a aba **Pull Requests** no GitHub.
2. Clique no **PR** que deseja revisar.
3. Verifique as mudanças no código e, se necessário, comente sugerindo ajustes.
4. Se estiver tudo certo, clique em **Merge Pull Request** para integrar a mudança ao projeto.

Pronto! Agora a branch de funcionalidade foi adicionada ao projeto principal e está pronta para ser usada por toda a equipe!

Agora você já sabe como trabalhar em equipe usando o GitHub! Criamos branches para separar funcionalidades, usamos **Pull Requests** para garantir que tudo seja revisado antes de entrar no código principal e organizamos melhor o fluxo de trabalho.

Para finalizar o conteúdo do nosso curso, vamos ver como você pode usar o GitHub como portfólio para destacar seus projetos pessoais ou profissionais.

5.3 Colaborando com Projetos Abertos: Entendendo o Fork

Você já aprendeu como utilizar o **git clone** para pegar projetos abertos e trazê-los para o seu computador. Mas, quando o objetivo é colaborar com projetos de outras pessoas, usamos o **fork**. O fork cria uma cópia do repositório original na sua própria conta do GitHub, permitindo que você faça alterações e contribuições sem afetar diretamente o projeto original.

Qual a função do Fork?

O fork tem duas funções principais:

- **Criação de uma cópia independente:** Ao fazer um fork, você gera uma cópia do repositório original na sua conta. Essa cópia é totalmente independente, permitindo que você modifique, experimente e adicione novas funcionalidades sem interferir no projeto principal.
- **Facilitar a colaboração:** Após fazer as alterações desejadas em seu fork, você pode abrir um *Pull Request* (PR) para sugerir que suas modificações sejam incorporadas ao repositório original. Dessa forma, o mantenedor do projeto pode revisar suas mudanças e decidir se elas serão integradas.

Fluxo de Trabalho com Fork

1. **Fazer o Fork:**
   No GitHub, acesse o repositório do projeto que deseja contribuir e clique no botão **Fork**. Isso criará uma bifurcação do projeto, ou seja, uma cópia do projeto na sua conta.

2. **Clonar o Repositório Forked:**
   Depois de ter seu fork, você pode cloná-lo para o seu computador com o comando:

```
git clone git@github.com:seu_usuario/nome-do-repositorio.git
```

3. **Realizar as Alterações:**
   Com o repositório clonado, faça as modificações necessárias no seu ambiente local. Você pode criar uma nova branch para organizar melhor suas mudanças:

```
git checkout -b minha-contribuicao
```

4. **Enviar as Alterações:**
   Após testar e confirmar suas mudanças, envie os commits para o seu repositório remoto:

```
git push origin minha-contribuicao
```

5. **Abrir um Pull Request:**
   Com as alterações publicadas no seu fork, vá até o GitHub e abra um *Pull Request* no repositório original, solicitando que suas contribuições sejam avaliadas e possivelmente incorporadas ao projeto principal.

Utilizando o fork, você contribui de forma organizada e segura para projetos abertos, permitindo que suas alterações sejam avaliadas e integradas ao repositório original sem causar conflitos ou interferir diretamente no desenvolvimento do projeto. Essa abordagem é fundamental para a colaboração em larga escala e para a evolução contínua dos projetos de código aberto.

Módulo 6: Criando e Personalizando o Perfil no GitHub

Chegamos à etapa final do nosso curso! Com tudo o que vimos até aqui, você já tem as ferramentas necessárias para versionar seus códigos, subir seus projetos para o GitHub e até colaborar com outras pessoas em repositórios compartilhados.

Agora, vamos entender como configurar seu GitHub para que seja um espaço organizado para armazenar e compartilhar suas soluções. Ele pode ser um repositório pessoal de automações úteis para seu trabalho, um local para documentar seus projetos ou até mesmo um portfólio profissional, caso você queira exibir suas habilidades para possíveis oportunidades na área de tecnologia.

Aula 6.1 Mas como o GitHub pode ser um portfólio para seus projetos e automações?

O GitHub, como vimos, não é apenas para desenvolvedores. Ele é uma ferramenta excelente para organizar e compartilhar códigos.

Se você criou um código para automatizar tarefas na empresa, otimizar um processo repetitivo ou gerar relatórios mais eficientes, pode armazená-lo no GitHub com segurança. Assim, além de acessar seus projetos de qualquer lugar, você pode compartilhá-los com colegas e até receber sugestões de melhorias.

Se, além de organizar seus projetos, você também quer mostrar suas habilidades para oportunidades profissionais, o GitHub pode funcionar como um portfólio público, permitindo que outras pessoas vejam seu código e acompanhem seus projetos.

Como visualizar perfis e projetos no GitHub?

O GitHub não serve apenas para armazenar os seus próprios códigos. Ele também permite que você encontre projetos de outras pessoas, visualize perfis e colabore em soluções interessantes.

Para pesquisar usuários e repositórios, basta usar a barra de pesquisa do GitHub e buscar por nomes, palavras-chave ou tecnologias específicas.

Alguns perfis interessantes para explorar no GitHub:

- Linus Torvalds → Criador do Linux, onde você pode conferir o código-fonte do kernel.
- Guido van Rossum → Criador do Python, com contribuições relevantes para a linguagem.
- Mario Souto → Desenvolvedor destaque e GitHub Star, compartilhando conteúdos sobre desenvolvimento web e boas práticas.

Quando você acessar um perfil no GitHub, verá algumas informações importantes, como:

- **Repositórios públicos** → Projetos disponíveis para qualquer pessoa visualizar.
- **Atividade e histórico de commits** → Registros de alterações feitas ao longo do tempo.
- **Destaque para projetos importantes** → Repositórios fixados no topo do perfil.
- **Informações sobre o usuário** → Breve descrição sobre a pessoa, incluindo onde trabalha, áreas de interesse e formas de contato, como redes sociais e LinkedIn.

Na prática, cada perfil no GitHub funciona como um espaço organizado para armazenar códigos e documentar projetos. Você pode manter projetos privados, compartilhar apenas com quem precisar ou deixar públicos para que outras pessoas também visualizem.

Como manter um perfil organizado no GitHub?

Se você quer utilizar o GitHub para organizar e compartilhar suas soluções, aqui vão algumas boas práticas:

1. Nome claro para os projetos → Exemplo: `automacao-relatorios` é mais intuitivo que `script1.py`.
2. Descrição bem explicada no README → Informe o que sua automação faz e como utilizar ela.
3. Código bem estruturado → Separe arquivos, utilize pastas e mantenha tudo organizado.
4. Commits com mensagens claras → Isso ajuda a entender o que foi alterado em cada atualização.
5. Definir se o repositório será privado ou público → Caso seja um projeto interno, pode mantê-lo privado.

Agora que entendemos como o GitHub pode ajudar a organizar e compartilhar seus projetos, vamos aprender a personalizar seu perfil e criar um repositório especial para destacar seus principais trabalhos!

Aula 6.2 Criando um Repositório para o Perfil

O GitHub permite que você tenha um README especial no seu perfil, onde pode adicionar informações sobre seus projetos e habilidades. Esse README funciona como uma “capa” personalizada para o seu GitHub.

Passo 1: Criando um Repositório Público para o Perfil

1. Acesse o GitHub e vá até a aba Repositórios.
2. Clique em **New** (Novo Repositório).

3. No campo de nome, digite exatamente seu nome de usuário no GitHub.
    - Exemplo: Se seu usuário for `joaodev`, o repositório deve se chamar `joaodev`.
4. Deixe o repositório como Público.
5. Marque a opção **Add a README file**.
6. Clique em **Create repository**.

Agora, esse repositório será usado para personalizar seu perfil!

Passo 2: Personalizando o README do Perfil

Com o repositório criado, vamos editar o arquivo `README.md` para deixar seu perfil mais profissional.

1. No repositório, clique no arquivo `README.md` e depois em Edit (Editar).
2. No editor, adicione informações básicas sobre você, como:

Dicas de revisão:

-> https://markdownlivepreview.com/ -> https://readme.so/pt -> https://www.makeareadme.com/

```
# Olá! Eu sou "Fulano"

Bem-vindo ao meu perfil! Aqui você encontra meus projetos, estudos e automações.

Sou um desenvolvedor apaixonado por automação, inteligência artificial
e análise de dados.
Gosto de resolver problemas do dia a dia com soluções inovadoras e tecnologia.

- Explorando novas tecnologias e aprimorando minhas habilidades
- Interesse especial em **IA, Automação, Web Scraping e Data Science**
- Trabalhando com **Python, Selenium, LangChain e Bancos de Dados**
- Sempre aprendendo e compartilhando conhecimento

## Tecnologias & Ferramentas

    ![Git](https://img.shields.io/badge/
    -Git-05122A?style=flat&logo=git)
    ![GitHub](https://img.shields.io/badge/
```

```
    -GitHub-05122A?style=flat&logo=github)
    ![Python](https://img.shields.io/badge/
    -Python-05122A?style=flat&logo=python)
    ![Selenium](https://img.shields.io/badge/
    -Selenium-05122A?style=flat&logo=selenium)
    ![LangChain](https://img.shields.io/badge/
    -LangChain-05122A?style=flat&logo=chainlink)
    ![AI](https://img.shields.io/badge/
    -IA-05122A?style=flat&logo=openai)
    ![Pandas](https://img.shields.io/badge/
    -Pandas-05122A?style=flat&logo=pandas)

## Entre em Contato

    [![LinkedIn](https://img.shields.io/badge/
    LinkedIn-blue?style=flat-square&logo=linkedin&logoColor=white)]
    (https://www.linkedin.com/in/seu-linkedin/)
   [![Email](https://img.shields.io/badge/Email-D14836?style=flat-square
    &logo=gmail&logoColor=white)](mailto:seuemail@example.com)
```

3. Depois de editar, clique em Commit changes para salvar.

Passo 3: Destacando Projetos no Perfil

Além do `README.md`, você pode fixar repositórios importantes no seu perfil.

1. Vá até seu perfil no GitHub.
2. No topo da página, clique em Customize profile.
3. Role até a seção Pinned Repositories.
4. Escolha até seis repositórios que deseja destacar.
5. Clique em Save changes.

Isso faz com que seus projetos mais importantes fiquem visíveis logo de cara para quem acessar seu perfil.

Então é isso, pessoal! Espero que tenham curtido essa jornada pelo versionamento de código e que agora o Git e o GitHub façam mais sentido para vocês.

O Git e o GitHub podem parecer complexos no início, mas, com o tempo e a prática, vocês vão perceber como eles facilitam o dia a dia. Mesmo que usem apenas para projetos privados, manter um histórico das suas automações e soluções já é um grande diferencial.

Agora é seguir praticando, versionando seus códigos e integrando isso à sua rotina. Valeu por acompanharem o curso e até a próxima!