



Versionamento de Código com Git e GitHub

☰ Resumo	Curso para apresentar Git e GitHub
☰ Links	GitHub: https://github.com/marlonprado04/DIO/tree/main/BOOTCAMP_desenvolvimento_java_com_cloud_aws/02_git_e_github
⊕ Categoria	Curso
☼ Status	Done
↗ Conteúdos DB	Git, GitHub
↗ Plano de estudos DB	Coding The Future - GFT e AWS Desenvolvimento Java com Cloud AWS
📅 Data	@05/08/2023 → 07/08/2023
↗ Instituições e plataformas DB	</> DIO

Versionamento de Código com Git e GitHub

▼ Sumário

[Versionamento de Código com Git e GitHub](#)

[Sumário](#)

[O que é Versionamento de Código](#)

[Tipos de Sistemas de Controle de Versão](#)

[O que é Git](#)

[O que é GitHub](#)

[Configurando o Git](#)

[Configurando o nome e email do usuário atual](#)

[Configurando branch padrão](#)

[Autenticação via Token](#)

[Autenticação via Chave SSH](#)

[Criando e Clonando Repositórios](#)

[Lista de comandos](#)

[Salvando alterações no repositório local](#)

[Comandos git add e git commit](#)

[Arquivo .gitignore](#)

[Arquivo .gitkeep](#)

[Desfazendo Alterações no Repositório Local](#)

[Desativar repositório .git](#)

[Comando `git restore`](#)

[Comando `git commit --amend`](#)

[Comando `git reset`](#)

[Trabalhando com branches - Criando, mesclando, deletando e tratando conflitos](#)

[Comando `git checkout`](#)

[Comando `git branch -v`](#)

[Comando `git branch -d`](#)

[Comando `git merge`](#)

[Tratamento de conflitos](#)

[Trabalhando com Branches - Comando úteis no dia a dia](#)

[Comando `git fetch`](#)
[Clonando apenas branch específica do repositório remoto](#)
[Comando `git stash`](#)
[Links úteis e materiais de apoio](#)

O que é Versionamento de Código

O versionamento de código nasceu para revolver vários problemas que surgem no trabalho em equipe ao criar um código.

Com sistemas de versionamento de código **podemos controlar quem realiza cada as alterações**, além de poupar trabalho na hora de recuperar códigos antigos e realizar alterações.

Tipos de Sistemas de Controle de Versão

Existem 2 tipos principais de sistemas de controle de versão, que são:

- **VCS Centralizado (CVCS)**
 - São sistemas que permitem edição de código apenas no servidor, o que pode acarretar problemas no caso de quedas de energia ou falha na sincronização caso não exista um sistema de backup.
 - Exemplos de software: **CVS**, **Subversion**
- **VCS Distribuído (DVCS)**
 - Para contornar os problemas dos CVCS surgiram os programas distribuídos, que criam uma cópia local na máquina de cada usuário, o que permite edição mesmo quando o servidor original estiver fora do ar
 - Cada clone é um backup, que permite um fluxo de trabalho mais flexível
 - Exemplos de software: **Git**, **Mercurial**

O que é Git

- Git está ligado ao projeto do kernel do Linux, onde começaram a utilizar o **BitKeeper**, um **DVCS proprietário**.
- Por cota do Linux ser de código aberto, acabou que o **Linus Torvalds**, criador do Linux, **desenvolveu uma ferramenta própria para o projeto**.

O que é GitHub

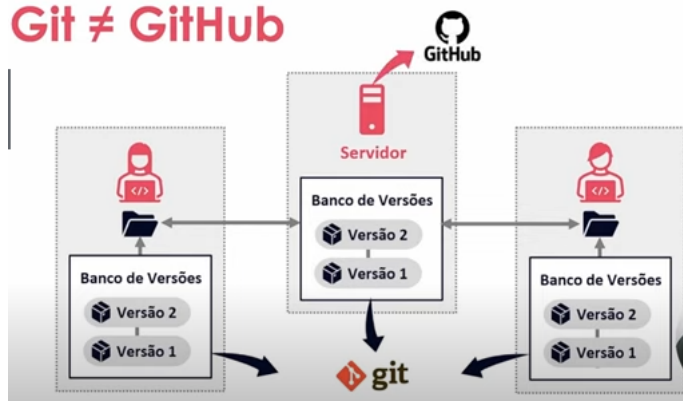
Apesar do nome parecido, o GitHub é uma **plataforma de hospedagem de código para controle de versão com Git, e colaboração**.

Algumas características:

- Possui uma comunidade muito ativa.
- Utilizado mundialmente;
- Possui o Octocat como mascote;

A diferença do Git e GitHub fica clara com o organograma abaixo:

Git ≠ GitHub



O Git funciona como gerenciador do repositório localmente, já o GitHub hospeda esse repositório funcionando como o servidor.

Configurando o Git

Para visualizar algumas opções do Git podemos utilizar o comando `git config` no terminal. Desta forma vamos visualizar a seguinte tela:

```
marlon@ubuntu-marlon:~$ git config
uso: git config [<options>]

Config file location
--global      use global config file
--system      use system config file
--local       use repository config file
--worktree    use per-worktree config file
-f, --file <arquivo> use given config file
--blob <blob-id> read config from given blob object

Action
--get          get value: name [value-pattern]
--get-all     get all values: key [value-pattern]
--get-regexp   get values for regexp: name-regex [value-pattern]
--get-urlmatch get value specific for the URL: section[.var] URL
--replace-all replace all matching variables: name value [value-pattern]
--add          add a new variable: name value
--unset        remove a variable: name [value-pattern]
--unset-all   remove all matches: name [value-pattern]
--rename-section rename section: old-name new-name
--remove-section remove a section: name
-l, --list     list all
--fixed-value  use string equality when comparing values to 'value-pattern'
-e, --edit     open an editor
--get-color    find the color configured: slot [default]
--get-colorbool find the color setting: slot [stdout-is-tty]
```

Por enquanto é importante saber que:

- `--global` → Se refere às configurações globais **do usuário atual** que está utilizando o sistema
- `--system` → Se refere às configurações gerais do sistema, abrindo **todos os usuários do sistema**.

Configurando o nome e email do usuário atual

Inicialmente precisamos configurar o **nome** e **email** do Git dentro do `--global`.

Por exemplo, para configurar o nome como **Marlon Prado** e email como **marlonprado04@gmail.com**:

```
marlon@ubuntu-marlon:~$ git config --global user.name "Marlon Prado"
marlon@ubuntu-marlon:~$ git config --global user.email "marlonprado04@gmail.com"
marlon@ubuntu-marlon:~$ git config user.name
Marlon Prado
marlon@ubuntu-marlon:~$ git config user.email
marlonprado04@gmail.com
marlon@ubuntu-marlon:~$
```

Essas informações servem para identificar o usuário no momento em que ele salvar os commits de código.

Obs: Caso elas sejam alteradas, estas informações só vão aparecer nos commits criados a partir do momento da alteração

Configurando branch padrão


Para consultar qual o branch principal podemos usar o comando `git config init.defaultBranch`, conforme abaixo:

```
Elidiana@NCC-1701-A MINGW64 /d/dio-git-e-github
$ git config init.defaultBranch
master
```

Para atribuir um novo nome à branch padrão podemos setar o atributo `--global` e adicionar ao fim do comando o nome da branch, conforme abaixo:

```
marlon@ubuntu-marlon:~$ git config --global init.defaultBranch main
marlon@ubuntu-marlon:~$ git config init.defaultBranch
main
```

Para mais informações de configuração do Git acessar:

Git - git-config Documentation
Check your version of git by running
 <https://git-scm.com/docs/git-config>

Autenticação via Token

Por questões de segurança o GitHub não permite que usuários clonem e acessem repositórios apenas informando o usuário e senha através do Git.

Dessa forma é necessário configurar um Token único de acesso sempre que formos configurar uma nova máquina ou projeto localmente.

Para testarmos isso, podemos seguir os passos abaixo:

1. Criar um repositório privado no GitHub
2. Tentar clonar o repositório via comando `git clone <url_do_repositorio>` dentro do terminal Git
3. Informar usuário e senha do Git (conforme será solicitado)

Os passos acima resultarão em erro, sendo necessário **ir nas configurações do GitHub para criar um Token**. Após criado o Token com as devidas permissões, podemos tentar repetir os passos 2 e 3, mas ao invés de digitar a senha no passo 3, digitamos o Token criado.



Observação

Esse Token precisa ser **configurado** para que o Git **não solicite toda vez** que tentarmos acessar ou modificar os repositórios do GitHub

Para configurar o Token podemos utilizar o seguinte comando:

- `git config --global credential.helper cache` → Caso desejemos que o Token seja armazenado **no cache do usuário atual**
- `git config --global credential.helper store` → Caso desejemos que o Token seja armazenado de forma **permanente nas configurações do usuário**

Dessa forma, na próxima vez que o Token for utilizado para acessar ou modificar um repositório da conta, o Git irá armazená-lo localmente.

Para mais informações acessar o site abaixo:

Git - Credential Storage

This book is available in English.

 <https://git-scm.com/book/en/v2/Git-Tools-Credential-Storage>

Autenticação via Chave SSH

Uma outra opção para configurar o acesso ao GitHub é por meio de chaves SSH.

Para configurar desta forma basta seguir os passos fornecidos pelo GitHub no link abaixo:

Conectar-se ao GitHub com o SSH - GitHub Docs

Você pode conectar-se a GitHub usando o protocolo Secure Shell (SSH), que fornece um canal seguro por meio de uma rede insegura.

 <https://docs.github.com/pt/authentication/connecting-to-github-with-ssh>

GitHub

Criando e Clonando Repositórios

Para criar um repositório local novo podemos usar o comando `git init` dentro de uma pasta.

Ao utilizar este comando será criada uma pasta oculta chamada `.git` que contém todas as informações e configurações do repositório.

Dentro dessa pasta há o arquivo chamado `config` que armazena algumas configurações como a URL de origem do repositório no GitHub.

Abaixo um exemplo de um repositório novo criado localmente:

```

marlon@ubuntu-marlon:~$ cd git_teste/
marlon@ubuntu-marlon:~/git_teste$ git init
Repositório vazio Git inicializado em /home/marlon/git_teste/.git/
marlon@ubuntu-marlon:~/git_teste$ ls -la
total 12
drwxrwxr-x  3 marlon marlon 4096 ago  5 22:42 .
drwx----- 40 marlon marlon 4096 ago  5 22:42 ..
drwxrwxr-x  7 marlon marlon 4096 ago  5 22:42 .git
marlon@ubuntu-marlon:~/git_teste$ cd .git/
marlon@ubuntu-marlon:~/git_teste/.git$ ls
branches  config  description  HEAD  hooks  info  objects  refs
marlon@ubuntu-marlon:~/git_teste/.git$ cat config
[core]
    repositoryformatversion = 0
    filemode = true
    bare = false
    logallrefupdates = true
marlon@ubuntu-marlon:~/git_teste/.git$

```

Abaixo um repositório clonado:

```

marlon@ubuntu-marlon:~$ git clone https://github.com/marlonprado04/testando_git.git
Cloning into 'testando_git'...
remote: Enumerating objects: 52, done.
remote: Counting objects: 100% (52/52), done.
remote: Compressing objects: 100% (39/39), done.
remote: Total 52 (delta 17), reused 40 (delta 12), pack-reused 0
Receiving objects: 100% (52/52), 8.04 KiB | 8.04 MiB/s, done.
Resolving deltas: 100% (17/17), done.
marlon@ubuntu-marlon:~$ cd testando_git/
marlon@ubuntu-marlon:~/testando_git(main)$ ls
arquivo_botao.md  LICENSE  README.md
marlon@ubuntu-marlon:~/testando_git(main)$ cd .git
marlon@ubuntu-marlon:~/testando_git/.git(main)$ cat config
[core]
    repositoryformatversion = 0
    filemode = true
    bare = false
    logallrefupdates = true
[remote "origin"]
    url = https://github.com/marlonprado04/testando_git.git
    fetch = +refs/heads/*:refs/remotes/origin/*
[branch "main"]
    remote = origin
    merge = refs/heads/main

```

Lista de comandos

Abaixo a lista com alguns comandos possíveis de se usar para controlar o `git clone`:

- `git clone <url_do_repositorio> --branch <nome_da_branch> --single-branch` → Comando para clonar apenas uma branch de um repositório
- `git clone <url_do_repositorio> <nome_da_pasta>` → Comando para clonar um repositório com um nome de pasta específica

Salvando alterações no repositório local

Agora que sabemos criar e clonar repositórios, podemos aprender a salvar as modificações no repositório local.

Para iniciar podemos utilizar o comando `git status` que serve para listar o status da árvore de trabalho ou da área de preparação.

Como podemos ver, o comando retorna que é necessário usar `git add` para adicionar arquivos da pasta:

```
marlon@ubuntu-marlon:~/git_teste$ git status
No ramo main

No commits yet

nada para enviar (crie/copie arquivos e use "git add" para registrar)
marlon@ubuntu-marlon:~/git_teste$
```

Ao usar o comando `touch` que é próprio do terminal para criar um arquivo `README.md`, o comando `git status` retorna que este **arquivo criado não está sendo monitorado**, ou seja, é necessário adicioná-lo ao commit com o comando `git add <nome_do_arquivo>`:

```
marlon@ubuntu-marlon:~/git_teste$ touch README.md
marlon@ubuntu-marlon:~/git_teste$ git status
No ramo main

No commits yet

Arquivos não monitorados:
  (utilize "git add <arquivo>..." para incluir o que será submetido)
    README.md

nada adicionado ao envio mas arquivos não registrados estão presentes (use "git
add" to registrar)
marlon@ubuntu-marlon:~/git_teste$
```

Antes de continuar é necessário entender um pouco sobre markdown. Para isso, podemos acessar os sites abaixo:

Sintaxe básica de gravação e formatação no GitHub - GitHub Docs

Crie formatação sofisticada para narração e código no GitHub com sintaxe simples.

 <https://docs.github.com/pt/get-started/writing-on-github/getting-started-with-writing-and-formatting-on-github/basic-writing-and-formatting-syntax>

GitHub

Abaixo um site para editar markdown online:

readme.so

Use readme.so's markdown editor and templates to easily create a ReadMe for your projects

 <https://readme.so/pt>



Comandos git add e git commit

Após criar alguns arquivos dentro da pasta, podemos utilizar os comandos `git add .` para adicionar todos os arquivos e pastas existentes e depois o comando `git commit -m "commit inicial"` para adicionar um commit com a descrição "commit inicial".

Após, se utilizarmos o comando `git status` o retorno será de que não há arquivos a serem commitados.

Note que ao adicionar pastas sem arquivos dentro, o git status não reconhece. É necessário que haja arquivos dentro das pastas se desejarmos que eles sejam adicionados.

Abaixo uma exemplo com os comandos acima:

```
marlon@ubuntu-marlon:~/git_teste$ ls
README.md
marlon@ubuntu-marlon:~/git_teste$ nano README.md
marlon@ubuntu-marlon:~/git_teste$ git add .
marlon@ubuntu-marlon:~/git_teste$ git commit -m "commit inicial"
[main (root-commit) 36e2bed] commit inicial
1 file changed, 6 insertions(+)
create mode 100644 README.md
marlon@ubuntu-marlon:~/git_teste(main)$ git status
No ramo main
nothing to commit, working tree clean
marlon@ubuntu-marlon:~/git_teste(main)$
```

Arquivo .gitignore

Outro ponto é que se desejarmos que determinado arquivo ou pasta não seja trackeado pelo Git, podemos adicionar o caminho no arquivo `.gitignore`. Dessa forma o Git irá ignorar os arquivos anotados.

Abaixo um exemplo:

```
marlon@ubuntu-marlon:~/git_teste(main)$ ls
README.md
marlon@ubuntu-marlon:~/git_teste(main)$ mkdir exemplo
marlon@ubuntu-marlon:~/git_teste(main)$ l
exemplo/ README.md
marlon@ubuntu-marlon:~/git_teste(main)$ git status
No ramo main
nothing to commit, working tree clean
marlon@ubuntu-marlon:~/git_teste(main)$ touch exemplo/exeplo_arquivo.md
marlon@ubuntu-marlon:~/git_teste(main)$ ls
exemplo README.md
marlon@ubuntu-marlon:~/git_teste(main)$ git status
No ramo main
Arquivos não monitorados:
  (utilize "git add <arquivo>..." para incluir o que será submetido)
    exemplo/

nada adicionado ao envio mas arquivos não registrados estão presentes (use "git add" to registrar)
marlon@ubuntu-marlon:~/git_teste(main)$ echo exemplo/ > .gitignore
marlon@ubuntu-marlon:~/git_teste(main)$ git status
No ramo main
Arquivos não monitorados:
  (utilize "git add <arquivo>..." para incluir o que será submetido)
    .gitignore

nada adicionado ao envio mas arquivos não registrados estão presentes (use "git add" to registrar)
marlon@ubuntu-marlon:~/git_teste(main)$ echo > .gitignore
marlon@ubuntu-marlon:~/git_teste(main)$ git status
No ramo main
Arquivos não monitorados:
  (utilize "git add <arquivo>..." para incluir o que será submetido)
    .gitignore
    exemplo/

nada adicionado ao envio mas arquivos não registrados estão presentes (use "git add" to registrar)
marlon@ubuntu-marlon:~/git_teste(main)$
```

Arquivo .gitkeep

Para contornar o problema de pastas vazias que o Git não reconhece, há uma convenção de se criar um arquivo chamado `.gitkeep` dentro de diretórios vazios para que o Git possa incluir o diretório no commit.

Abaixo um exemplo:

```
marlon@ubuntu-marlon:~/git_teste(main)$ ls
exemplo  README.md
marlon@ubuntu-marlon:~/git_teste(main)$ git status
No ramo main
nothing to commit, working tree clean
marlon@ubuntu-marlon:~/git_teste(main)$ mkdir diretorio
marlon@ubuntu-marlon:~/git_teste(main)$ git status
No ramo main
nothing to commit, working tree clean
marlon@ubuntu-marlon:~/git_teste(main)$ touch diretorio/.gitkeep
marlon@ubuntu-marlon:~/git_teste(main)$ git status
No ramo main
Arquivos não monitorados:
  (utilize "git add <arquivo>..." para incluir o que será submetido)
    diretorio/

nada adicionado ao envio mas arquivos não registrados estão presentes (use "git add" to registrar)
marlon@ubuntu-marlon:~/git_teste(main)$
```

Desfazendo Alterações no Repositório Local

Ao trabalhar com versionamento de código podemos cometer alguns erros que desejamos corrigir.

Abaixo algumas situações em que possamos querer desfazer possíveis erros:

Desativar repositório .git

Para **desativar o git de uma pasta que não deveria ser repositório** basta deletar a pasta `.git` que é criada:

Comando `git restore`

Para **restaurar alterações indesejadas em alguns arquivos ou pastas**, podemos usar o comando `git restore <caminho_do_arquivo_ou_pasta>`. Esse comando restaura todas as informações dos arquivos passados para a versão mais recente commitada. **Tomar cuidado para não apagar tudo.**

Comando `git commit --amend`

Para **alterar o a mensagem do último commit realizado**, podemos usar o comando `git commit --amend -m <nova_mensagem_do_commit>` ou também podemos apenas usar o comando `git commit --amend`, desta **segunda forma o Git irá abrir o documento no editor de texto do terminal para que possa ser editado.**

Abaixo um exemplo

```

marlon@ubuntu-marlon:~/git_teste(main)$ git log
commit fe615528dcb3a022567310684d2f92e6c4741104 (HEAD -> main)
Author: Marlon Prado <marlonprado04@gmail.com>
Date: Mon Aug 7 20:36:49 2023 -0300

    nova mensagem: commit inicial
marlon@ubuntu-marlon:~/git_teste(main)$ git commit --amend -m "mensagem aleatoria"
[main a713cc8] mensagem aleatoria
Date: Mon Aug 7 20:36:49 2023 -0300
4 files changed, 7 insertions(+)
create mode 100644 .gitignore
create mode 100644 README.md
create mode 100644 diretorio/.gitkeep
create mode 100644 exemplo/exemplo_arquivo.md
marlon@ubuntu-marlon:~/git_teste(main)$ git log
commit a713cc879b8fec5d23abf8b3659e02d4eaf006ca (HEAD -> main)
Author: Marlon Prado <marlonprado04@gmail.com>
Date: Mon Aug 7 20:36:49 2023 -0300

    mensagem aleatoria
marlon@ubuntu-marlon:~/git_teste(main)$

```

Comando `git reset`

Para **restaurar os arquivos / resetar os arquivos submetidos anteriormente** usamos o comando `git reset` que tem como opções de parâmetros `--soft`, `--mixed` e `--hard` que servem para:

- `--soft`: Restaura um commit passado **adicionando as diferenças no staging area** do commit;
- `--mixed`: Restaura um commit passado **sem adicionar as diferenças no staging area**. *Esse é o comportamento padrão do git reset quando não passamos um parâmetro;*
- `--hard`: Restaura um commit anterior **removendo todas as diferenças** submetidas em commits posteriores.

Outro uso do comando é remover um arquivo ou pasta da submissão de commit usando `git reset <caminho_do_arquivo>`

Trabalhando com branches - Criando, mesclando, deletando e tratando conflitos

Branch's são ramos que permitem divergir o código em commits separados.

Comando `git checkout`

Para criar uma branch nova e já entrar nela usamos o comando `git checkout -b <nome_da_branch>`

Dessa forma, será criada uma ramificação a partir do commit atual.

Se desejarmos trocar de branch podemos usar o comando `git checkout <nome_da_branch>`, sem o parâmetro `-b`.

Comando `git branch -v`

Podemos listar o último commit realizado em cada branch usando o comando `git branch -v`

Comando `git branch -d`

Para deletar as informações de uma branch podemos usar o comando `git branch -d <nome_da_branch>`.

Comando `git merge`

Se desejarmos unificar as informações de uma branch com outra, podemos usar o comando `git merge <nome_da_branch>`, mas para isso precisamos dentro da branch que desejamos mesclar.

Ex: estar dentro da branch `main` e usar o comando `git merge exemplo` para trazer as mudanças do `exemplo` para `main`.

Tratamento de conflitos

Supondo que existam alterações no repositório local e remoto no mesmo arquivo, simultaneamente.

Ao tentar baixar as informações do repositório remoto o git irá informar que existe um conflito.

Dessa forma é necessário decidir se mantemos as informações do local ou do individual.

Trabalhando com Branches - Comando úteis no dia a dia

Existem comandos que facilitam nosso trabalho com branches no dia a dia. Abaixo alguns deles.

Comando `git fetch`

Caso a gente deseje **apenas baixar as alterações do repositório remoto sem mesclá-las** com o repositório local podemos usar o comando `git fetch <apelido_repositorio_remoto> <branch_local>`.

Por exemplo: `git fetch origin main`

Após, podemos visualizar as diferenças com o comando `git diff main origin/main`. Dessa forma aparecem as mudanças commitadas no repositório remoto que não estão no local.

Para finalizar, podemos usar o `git merge origin/main` para incluir as mudanças da origem na branch main local.

Clonando apenas branch específica do repositório remoto

Em casos específicos podemos querer clonar apenas uma branch específica de um repositório remoto.

Para fazer esse clone usamos o comando `git clone <url_do_repositorio> --branch <nome_da_branch> --single-branch`

Por exemplo → `git clone <url> --branch teste --single-branch`

Comando `git stash`

Podemos usar o comando `git stash` para **arquivar modificações e evitar que elas sejam passadas para uma nova branch** criada.


Por exemplo, se estamos em uma branch e deletamos determinado arquivo, mas ao criar uma branch desejamos que esse arquivo não seja deletado.

Para isso, podemos usar o comando `git stash` antes de criar a nova branch e depois usar o `git stash list` para listar as modificações arquivadas.

Depois disso podemos criar uma nova branch que as alterações realizadas atualmente não serão passadas para ela. Ao voltar para a branch com as modificações arquivadas devemos usar um dos dois comandos abaixo para restaurar as mudanças:

- `git stash pop` → Recupera as a alterações arquivadas e descarta elas;
- `git stash apply` → Recupera as alterações arquivadas e mantém elas arquivadas.

Para saber mais:

Git - git-branch Documentation
Check your version of git by running
 <https://git-scm.com/docs/git-branch>

Links úteis e materiais de apoio

Repositório no GitHub com anotações do curso:

<https://github.com/elidianaandrade/dio-curso-git-github>