



SUMMER ACADEMY

4 DE FEV

OFICINA GIT, GITHUB E VERSIONAMENTO

Agenda

1. Sistemas de Controle de Versão
2. Tipos de sistemas de Controle de Versões
3. GIT
4. Breve História
5. Instalando
6. Configurando
7. Como o GIT funciona
8. Estados do GIT e o ciclo de vida dos arquivo
9. Primeiros comandos
10. Branchs
11. Repositórios Remotos
12. Issues e Pull Requests
13. gitignore

Sistema de controle de versões

Um sistema de controle de versão (como o próprio nome já diz) tem a finalidade de **gerenciar diferentes versões de um documento**.

Com isso ele te oferece uma maneira muito mais inteligente e eficaz de organizar seu projeto, pois é possível acompanhar um **histórico de desenvolvimento, desenvolver paralelamente** e ainda te oferecer outras vantagens, como exemplo, **customizar uma versão**, incluir outros requisitos, finalidades específicas, layout e afins ou **resgatar o sistema em um ponto que estava estável**, isso tudo sem mexer na versão principal.

Como funciona um Controle de Versão?

Basicamente, os arquivos do projeto ficam armazenados em um **repositório** (um **servidor** em outras palavras) e o **histórico de suas versões** é salvo nele.

Os desenvolvedores podem acessar e resgatar a ultima versão disponível e fazer uma **cópia local**, na qual poderão trabalhar em cima dela e continuar o processo de desenvolvimento.

A **cada alteração** feita, é possível **enviar novamente ao servidor e atualizar a sua versão a partir outras feitas pelos demais desenvolvedores**.

E para **evitar problemas de conflitos**, o Sistema de Controle de Versão oferece **ferramentas uteis para mesclar o código e evitar conflitos**.

Como funciona um Controle de Versão?

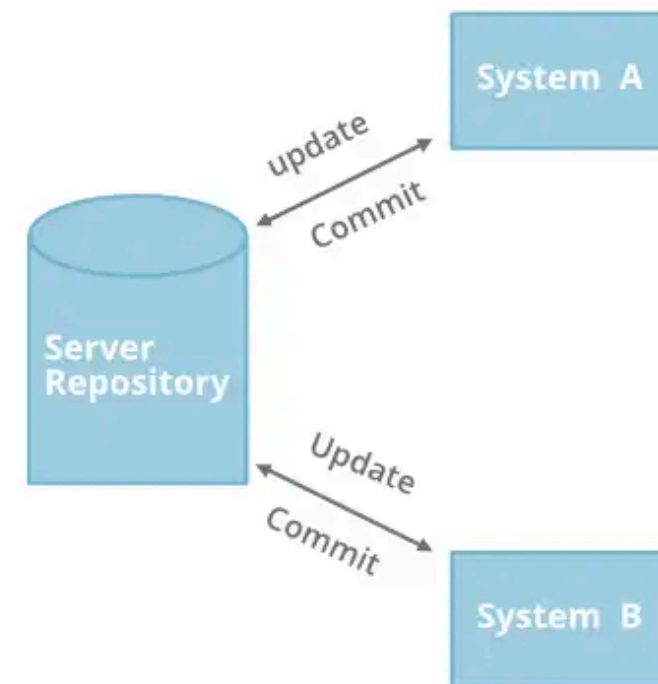
Por exemplo: Você atualizou seu projeto e começou a fazer suas alterações. Ao mesmo tempo, outro desenvolvedor fez alterações e atualizou a versão no servidor. Quando for enviar sua versão o Sistema de Controle de Versão irá alertar que o seu arquivo está desatualizado.

Ele enviará as novas informações adicionadas e permitirá mesclar as diferentes versões.

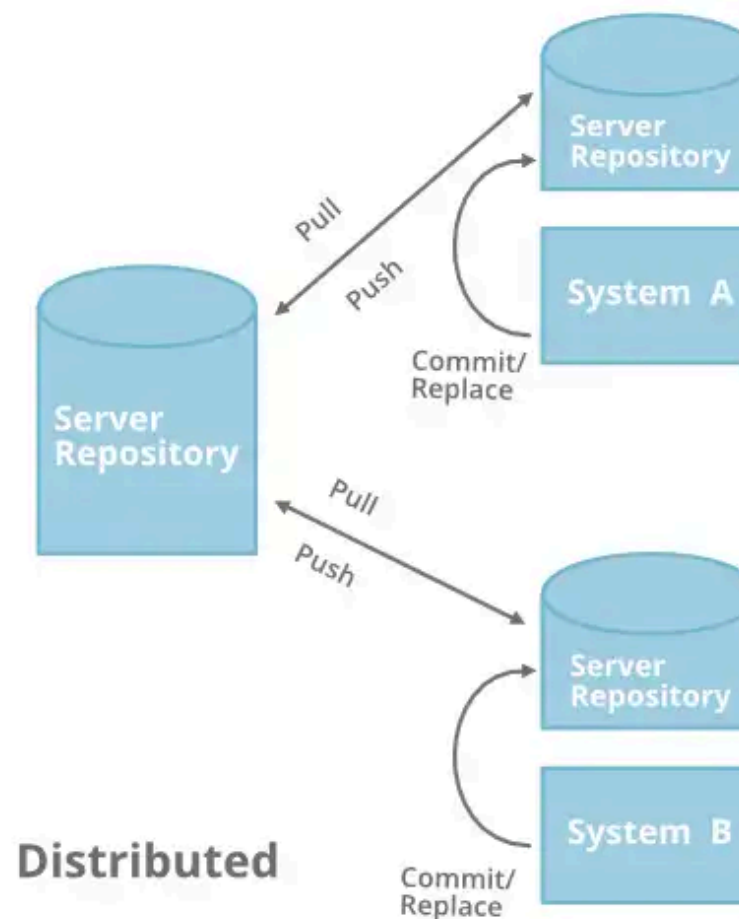
Não apenas isso, ele também mostrará onde foram feitas atualizações, trechos de código incluídos ou removidos e casos de conflito, onde linhas podem se sobrescrever e oferecerá opções para mesclar manualmente, escolhendo a melhor solução.

Tipos de sistemas de Controle de Versões

Atualmente, os sistemas de controle de versão são classificados em dois tipos: **Centralizados** e **Distribuídos**.



Centralized



Distributed

Tipos de sistemas de Controle de Versões

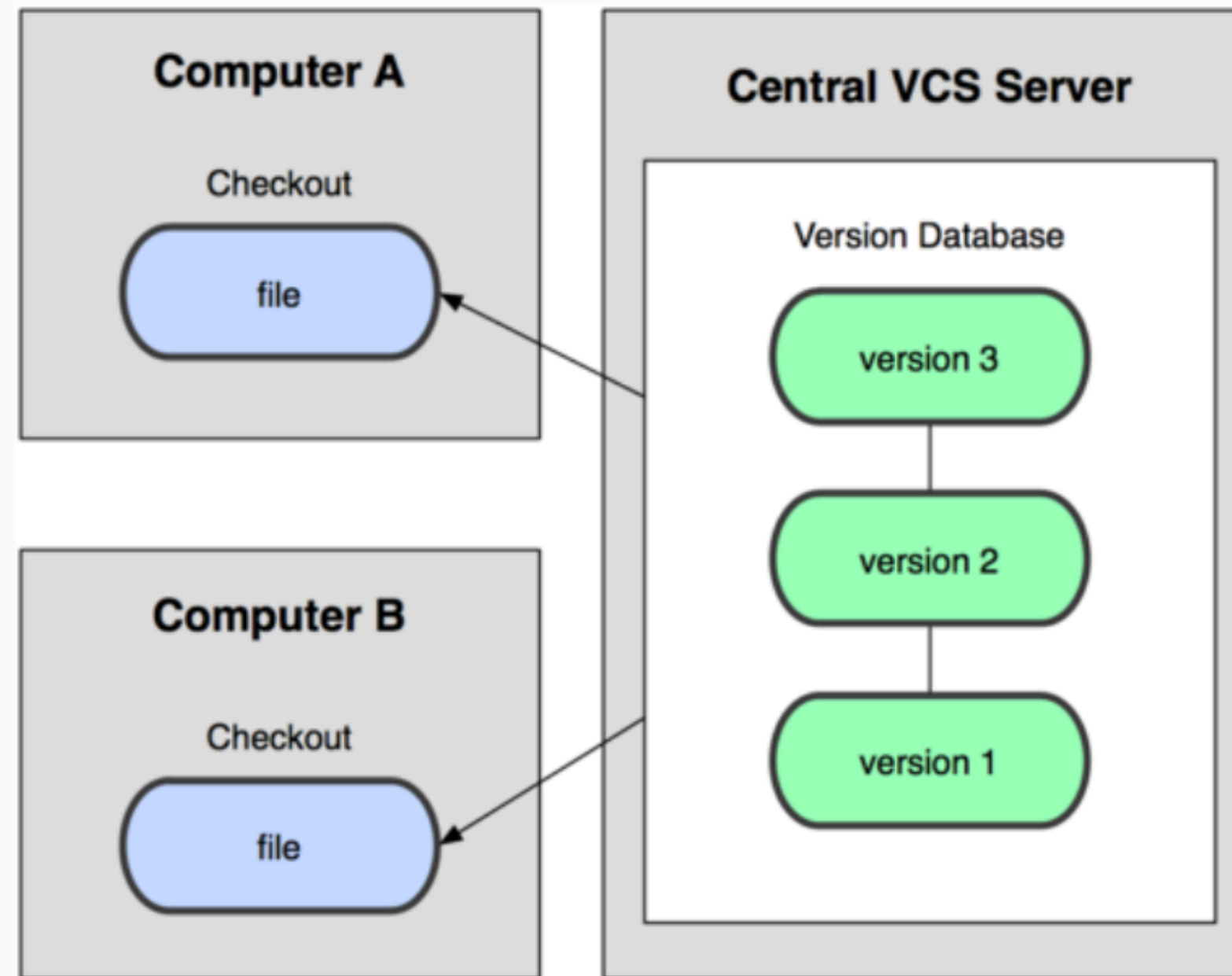
O **centralizado** trabalha apenas com um servidor central e diversas áreas de trabalho, baseados na arquitetura **cliente-servidor**.

Por ser centralizado, **as áreas de trabalho precisam primeiro passar pelo servidor para poderem comunicar-se**.

Essa versão atende muito bem a maioria das equipes de desenvolvimento que não sejam enormes e trabalhem em uma rede local, além de não ter problemas de velocidade para enviar e receber os dados e ter um bom tempo de resposta do servidor.

Um dos principais sistemas com o tipo de controle de versão centralizado é **Subversion**.

Tipos de sistemas de Controle de Versões



Controle de Versão Centralizado

Tipos de sistemas de Controle de Versões

O **distribuído** vai mais além. Ele é recomendado para equipes com muitos desenvolvedores e que se encontram em diferentes filiais.

Esta versão funciona da seguinte maneira: **cada estação de trabalho tem seu próprio “servidor”**, ou seja, as operações são feitas na própria máquina.

Apesar das áreas de trabalho poderem comunicar-se entre si, **recomenda-se usar um servidor como centro do envio dos arquivos para centralizar o fluxo e evitar ramificações do projeto e a perda do controle sobre o mesmo**, geralmente o sistema tem essa opção, oferecendo um **servidor remoto** para hospedar o projeto.

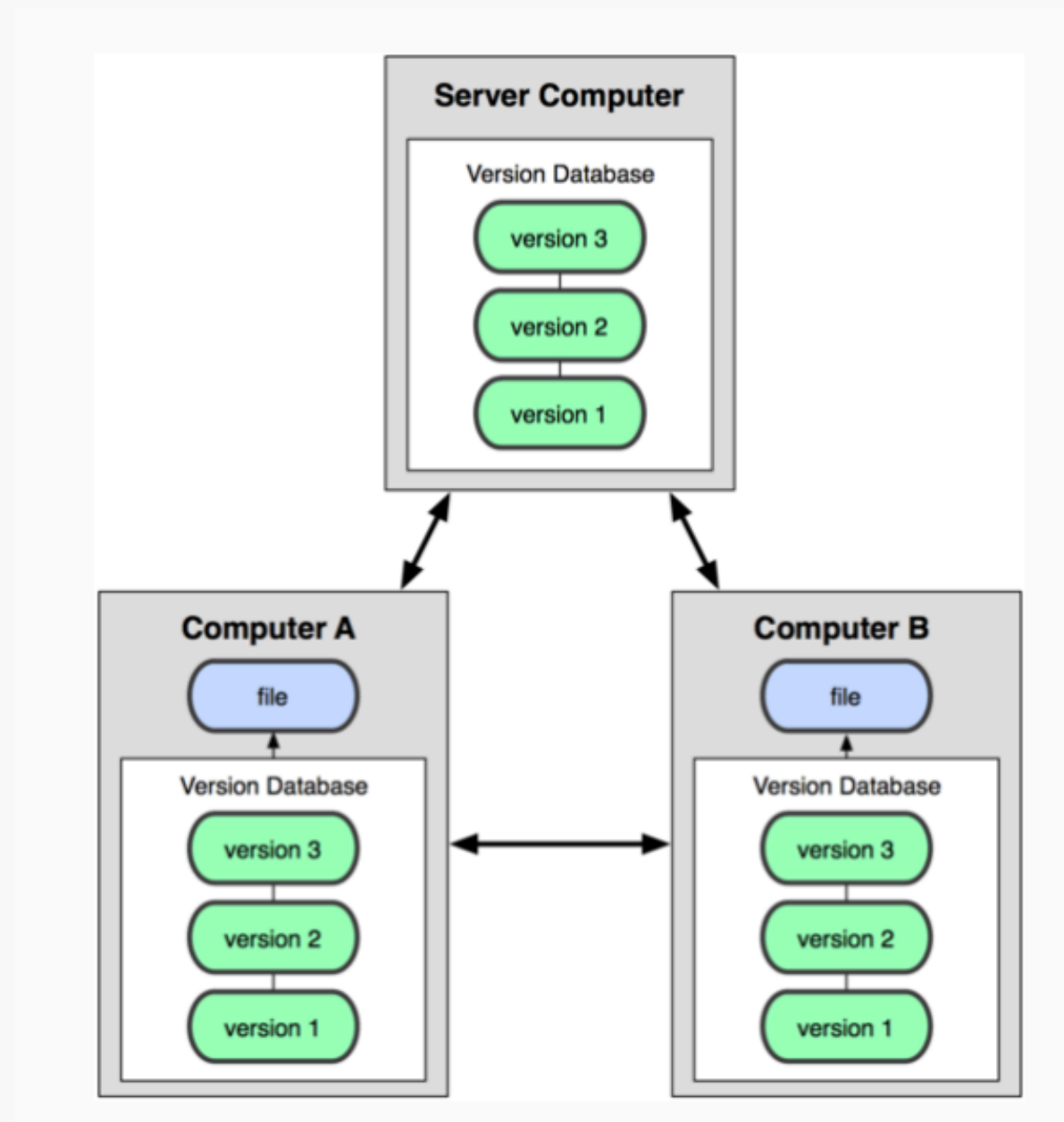
Tipos de sistemas de Controle de Versões

A comunicação entre o servidor principal e as áreas de trabalho funciona com duas operações para atualizar e mesclar o projeto, chamadas de pull e push (puxar e empurrar).

- pull: Com esta operação é possível pegar a versão de outra estação de trabalho e mesclar com a sua.
- push: Com esta operação temos o processo inverso do pull, ou seja, enviando para outra área a sua versão do projeto.

Os sistemas distribuídos mais conhecidos são o **Git** e o Mercurial.

Tipos de sistemas de Controle de Versões



Controle de Versão Distribuído

O que é o Git

Git é um sistema de controle de versão distribuído projetado para ser eficiente.



Breve história

- Criado por Linus Torvalds em 2005
- Para ser usado no desenvolvimento do Linux Kernel
- Metas do projeto:
 - Velocidade
 - Design simples
 - Permitir desenvolvimento não-linear (milhares de branches)
 - Capaz de manipular projetos grandes como o Linux Kernel de maneira eficiente

Git – Instalando

- <https://git-scm.com/download/win>

```
winget install --id Git.Git -e --source winget
```

Download for Windows

[Click here to download](#) the latest (2.40.0) 64-bit version of **Git for Windows**. This is the most recent [maintained build](#). It was released **22 days ago**, on 2023-03-14.

Other Git for Windows downloads

Standalone Installer

[32-bit Git for Windows Setup.](#)

[64-bit Git for Windows Setup.](#)

Portable ("thumbdrive edition")

[32-bit Git for Windows Portable.](#)

[64-bit Git for Windows Portable.](#)

Using winget tool

Install [winget tool](#) if you don't already have it, then type this command in command prompt or Powershell.

```
winget install --id Git.Git -e --source winget
```

The current source code release is version **2.40.0**. If you want the newer version, you can build it from [the source code](#).

Now What?

Git – Configurando o ambiente local

- Após a instalação do Git você precisará configurar seu usuário do GitHub em seu computador. Para isso, abra seu Terminal e execute os seguintes comandos:

```
git config --global user.name "SEU NOME" < enter >
```

```
git config --global user.email "EMAIL@exemplo.br" < enter >
```

Lembre-se de substituir “SEU NOME” pelo seu nome e

“EMAIL@exemplo.br” pelo seu e-mail

- Você pode verificar a lista de configurações usando o comando:

```
git config --list < enter >
```

Estados do GIT e o ciclo de vida dos arquivos

Um arquivo sempre estará em um dos estados fundamentais:

- **Não rastreado (untracked):**

- O arquivo foi **criado, modificado** ou **removido** porém não foi rastreado pelo GIT, ou seja, seria como se o GIT não soubesse do estado exato desse arquivo e **não tem controle do versionamento**.

- **Modificado (modified)**

- **Uma vez no repositório, qualquer arquivo que é adicionado, modificado ou removido é marcado como modificado.** Significa que o **arquivo sofreu alterações, mas ainda não foi dito que ele fará parte do próximo *commit***, ou seja, da próxima versão que será consolidada.
- Significa também que esses arquivos estão diferentes quando comparados com a última versão disponível no histórico.

Estados do GIT e o ciclo de vida dos arquivos

Um arquivo sempre estará em um dos estados fundamentais:

- **Preparado (staged):**
 - A partir do momento que o comando **git add** é executado em algum arquivo, ele assume o estado de preparado. Neste momento, **o Git sabe que o arquivo foi modificado e agora está na área de preparação para ser consolidado.**
- **Consolidado (committed)**
 - Após toda a preparação, os arquivos são finalmente salvos quando o comando **git commit é executado**. É importantíssimo destacar que se algum arquivo se encontra no estado modificado e é feito um git commit, estes arquivos continuarão como modificados, eles não farão parte do commit, pois nunca foram preparados.

Estados do GIT e o ciclo de vida dos arquivos

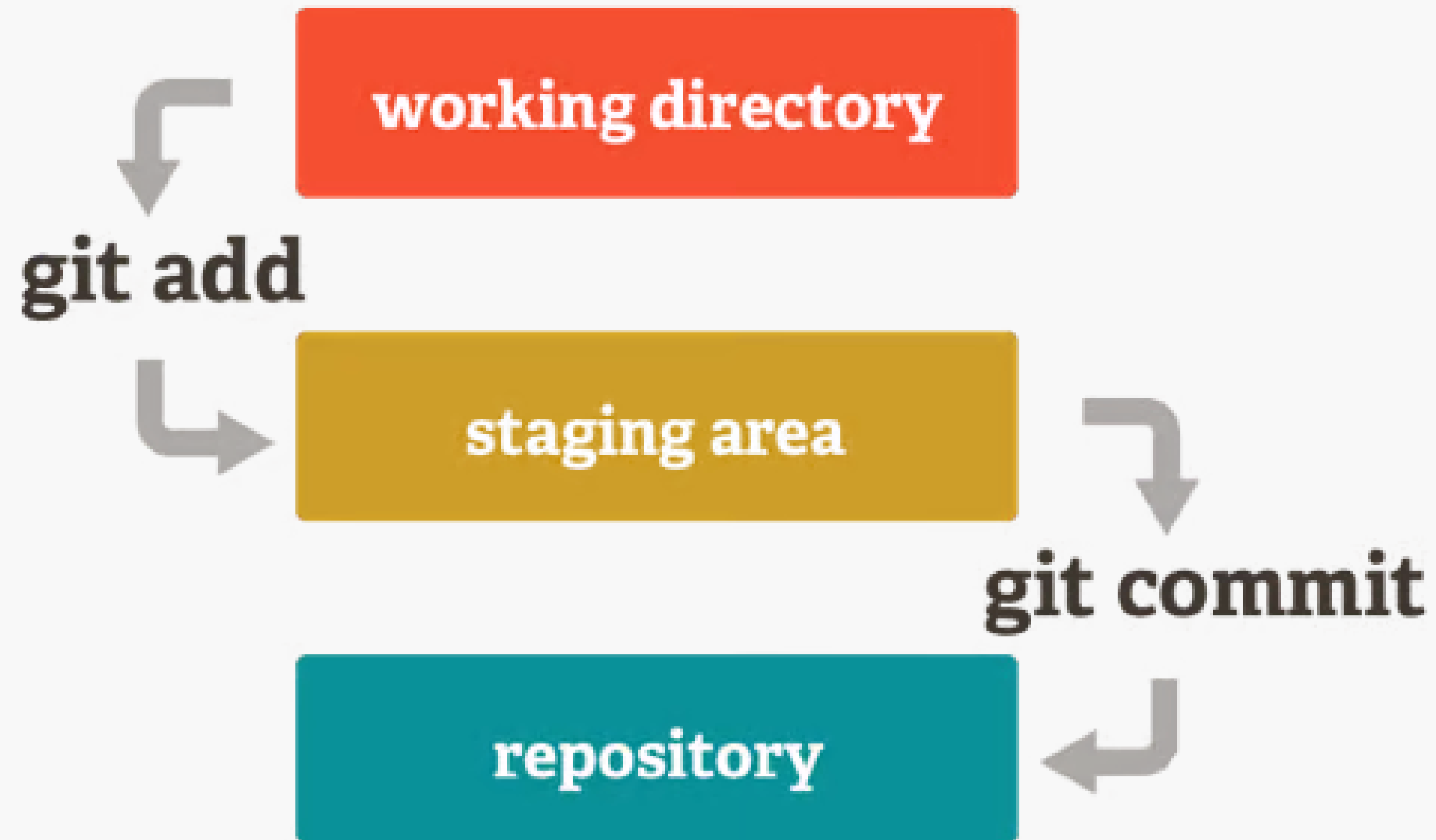
- **Staging Area**

“Staging Area” ou “Área de Preparação” é uma **área intermediária entre o diretório de trabalho e o repositório Git**. É onde os arquivos modificados são preparados para serem incluídos no próximo commit.

A Staging Area permite que os desenvolvedores selecionem quais alterações desejam incluir em um commit específico, separando as alterações em etapas distintas.

A Staging Area é como uma caixa de seleção, onde os desenvolvedores podem escolher quais alterações desejam incluir no próximo commit. Isso permite que os desenvolvedores revisem e organizem suas alterações antes de confirmá-las definitivamente.

Estados do GIT e o ciclo de vida dos arquivos



Projeto Base



Primeiros Comandos

- **Iniciando um repositório local:**
 - `git init` //rodar comando dentro do diretório no qual deseja iniciar o repositório
- **Mostrando os arquivos alterados desde o último commit:**
 - `git status`
- **Adicionando arquivos a staging area:**
 - `git add <nome do arquivo>` // Adicionando apenas um arquivo
 - `git add ".extensao_do_arquivo"` // Adicionando todos os arquivos com uma extensão
 - `git add .` // Adicionando todos os arquivos no diretório
- **Removendo arquivos da staging area:**
 - `git reset <nome do arquivo>`

Primeiros Comandos

- **Comitando as alterações:**
 - `git commit -m "Mensagem do Commit"`
- **Listando commits realizados:**
 - `git log`
- **Mostra as alterações realizadas no último commit:**
 - `git show`

Mensagens de commits

Não existe uma regra universal para a escrita das mensagens de commit, porém, algumas boas práticas podem ser seguidas para garantir que outras pessoas e até mesmo você no futuro entenda que alterações foram feitas e por quê.

As mensagens dos commits devem ser simples e objetivas. A seguir, temos algumas orientações que podem ser seguidas com esse objetivo:

- **Mantenha a mensagem curta e concisa:** a primeira linha da mensagem deve conter, no máximo, 72 caracteres. Caso seja necessária uma descrição adicional, você deve pular uma linha e adicionar os detalhes, como o contexto da mudança realizada.
- **Uso de verbo no infinitivo:** é comum que a mensagem do commit inicie com um verbo no infinitivo que descreva a alteração feita, como “adicionar”, “corrigir” ou “atualizar”. Em sequência, são adicionados detalhes concisos da mudança. Por exemplo: “Atualizar texto do título da página”.
- **Evite detalhes técnicos:** não inclua detalhes técnicos complexos na mensagem de commit. Esses detalhes podem ser adicionados nos comentários de código ou na documentação.

Commits Semânticos

Commit semântico ou conventional commit é **uma das formas que se pode fazer padronização de commits** dentro de um projeto de desenvolvimento de software.

A estrutura de um commit semântico é claro e de fácil identificação, pois utiliza um formato definido na sua sintaxe, possuindo **partes obrigatórias** e **outras opcionais**.

Abaixo, descreve-se uma estrutura básica de um commit semântico, contendo as partes obrigatórias: tipo e descrição; e as partes opcionais: escopo, corpo e rodapé.

<tipo>[escopo opcional]: <descrição>

[corpo opcional]

[rodapé(s) opcional(is)]

OBS: QUANTO MAIS COMPLETO O FORMATO DO COMMIT MAIOR SERÁ O TEMPO EM SEU PROCESSAMENTO.

PORTANTO, RECOMENDA-SE SEU USO SIMPLIFICADO, NÃO UTILIZANDO SUAS PARTES OPCIONAIS. HÁ CASOS ESPECÍFICOS QUE NECESSITAM UM MAIOR DESCRITIVO DO PROCESSO EXECUTADO.

Commits Semânticos – Tipos

A primeira e principal descrição de um commit semântico, refere-se a seu tipo, que informam a intenção do seu commit ao utilizador(a) de seu código.

Abaixo será enumerado os principais types descritos na documentação:

- **feat** - Tratam **adições de novas funcionalidades** ou de quaisquer outras novas implantações ao código;
- **fix** - Essencialmente definem o tratamento de **correções de bugs**;
- **docs** - Referem-se a inclusão ou alteração somente de arquivos de documentação;
- **test** - Commits do tipo test são utilizados quando são realizadas **alterações em testes**, seja criando, alterando ou excluindo testes unitários;
- **style** - Alterações referentes a formatações na apresentação do código que não afetam o significado do código, como por exemplo: espaço em branco, formatação, ponto e vírgula ausente etc.)

...

Commits Semânticos – Tipos

- **build** - Commits do tipo build são utilizados quando são realizadas **modificações em arquivos de build e dependências**;
- **refactor** - Commits do tipo refactor referem-se a mudanças devido a **refatorações que não alterem sua funcionalidade**, como por exemplo, uma alteração no formato como é processada determinada parte da tela, mas que manteve a mesma funcionalidade, ou melhorias de performance devido a um code review;
- **remove** - Commits do tipo remove **indicam a exclusão de arquivos, diretórios ou funcionalidades obsoletas ou não utilizadas**, reduzindo o tamanho e a complexidade do projeto e mantendo-o mais organizado.

Exemplo:

- `git commit -m "fix: Loop infinito na linha 50"`
- `git commit -m "feat : envia email para o cliente quando o produto é enviado"`

Branchs

Uma branch é uma **linha de desenvolvimento** do projeto.

Você pode ter vários branchs em seu repositório, cada branch representando uma versão específica de seu projeto, por exemplo.

Veja um branch como um fork('cópia') do projeto que pode seguir sua própria linha de desenvolvimento.

Todo novo repositório Git (após o primeiro commit) possui um branch chamado por padrão 'master'.

Branchs

- **Listando as branches:**
 - `git branch`
- **Criando um novo branch:**
 - `git branch <nome_da_branch>`
- **Criando um novo branch 'homologacao' baseado em um outro branch 'develop' que não é o corrente, ainda mantendo-se no branch atual:**
 - `git branch homologacao develop`
- **Outra forma de criar um branch e ao mesmo tempo tornar a nova branch como corrente é usando:**
 - `git checkout -b homologacao develop`

Branchs

- **Mudando para uma outra branch:**
 - `git checkout <nome_da_branch>`
- **Apagando uma branch:**
 - `git branch -D <nome_da_branch>`
 - OBS: você não pode apagar o branch corrente, alterne para um outro então execute novamente o comando.
- **Realizando o merge:**
 - `git merge <nome_da_branch>`

Repositórios Remotos

A ideia é a mesma de um repositório local, mas os remotos são hospedados em servidores na internet ou outra máquina que não a sua (mas pode ser a sua também).

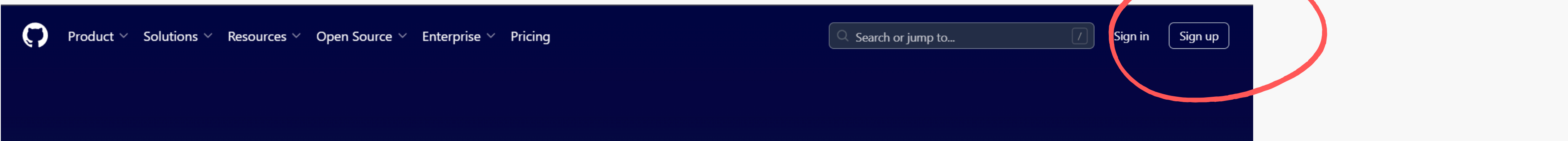
Existem sites que provêm hospedagem gratuita de código fonte para repositório Git, um deles é o GitHub.



Servidores de repositório de código

Repositórios Remotos

Vamos criar uma conta no Git Hub (github.com)



The screenshot shows the GitHub homepage. The top navigation bar is dark blue with the GitHub logo on the left and links for Product, Solutions, Resources, Open Source, Enterprise, and Pricing. A search bar is in the center. On the right, the 'Sign in' and 'Sign up' buttons are circled in red. Below the navigation bar, the main content area has a dark blue background with the text 'Build and ship single, collaborative projects' and 'Join the world's most widely adopted DevOps platform'. A form to 'Enter your email' is visible. To the right, there is a 'Create your free account' section with a dark background and colorful GitHub mascots. Further right, the 'Sign up to GitHub' form is shown with fields for Email, Password, and Username, and a 'Continue >' button.

Product ▾ Solutions ▾ Resources ▾ Open Source ▾ Enterprise ▾ Pricing

Search or jump to... /

Sign in Sign up

Build and ship single, collaborative projects

Join the world's most widely adopted DevOps platform

Enter your email Sign up

Create your free account

Explore GitHub's core features for individuals and organizations.

See what's included ▾

Sign up to GitHub

Email*

Email

Password*

Password

Password should be at least 15 characters OR at least 8 characters including a number and a lowercase letter.

Username*

Username

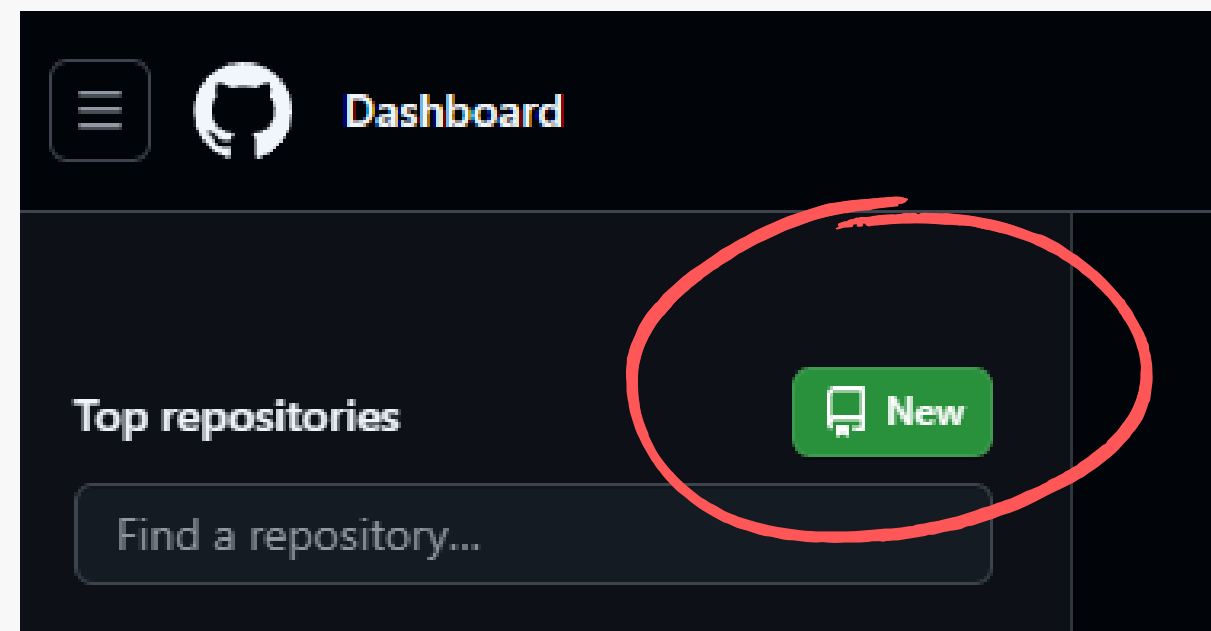
Username may only contain alphanumeric characters or single hyphens, and cannot begin or end with a hyphen.

Continue >

By creating an account, you agree to the [Terms of Service](#). For more information about GitHub's privacy practices, see the [GitHub Privacy Statement](#). We'll occasionally send you account-related emails.

Repositórios Remotos

No GitHub vamos criar um repositório




Create a new repository

A repository contains all project files, including the revision history. Already have a project repository elsewhere? [Import a repository.](#)

Required fields are marked with an asterisk (*).

Owner *

 **rodrifontes** ▾

Repository name *

/

Great repository names are short and memorable. Need inspiration? How about **bookish-eureka** ?

Description (optional)

☒  **Public**

Anyone on the internet can see this repository. You choose who can commit.

☐  **Private**

You choose who can see and commit to this repository.

Initialize this repository with:

☐ **Add a README file**

This is where you can write a long description for your project. [Learn more about READMEs.](#)

Add .gitignore


.gitignore template: **None** ▾

Choose which files not to track from a list of templates. [Learn more about ignoring files.](#)

Choose a license

License: **None** ▾

A license tells others what they can and can't do with your code. [Learn more about licenses.](#)

 You are creating a public repository in your personal account.

Create repository

Repositórios Remotos

- **Conectando ao repositório remoto:**
 - `git remote add origin <link do repositório>`
- **Enviando arquivos do repositório local para o remoto:**
 - `git push origin master` //origin é a referencia ao repositório e master a branch ou ainda
 - `git push -u <remote> <branch>`
 - Quando você usa a flag -u (abreviação de --set-upstream), você está dizendo ao Git para vincular a branch local atual à branch remota especificada. Isso faz com que, em futuros `git pull` ou `git push`, você não precise especificar o repositório e a branch

Repositórios Remotos

- **Atualizando repositório local com o remoto:**
 - `git pull <remote> <branch>`
- **Criando um clone (cópia de todo o projeto, incluindo todos commits) de um repositório remoto:**
 - `git clone <link do repositório>`

Lidando com Conflitos

- Escolha um projeto de sua preferência, altere algum arquivo diretamente pelo GitHub e crie um commit.
- Em seu VSCode, edite o mesmo arquivo nas mesmas linhas que você editou anteriormente e crie um commit.
- Faça um pull do repositório no GitHub.

```
Ⓢ PS C:\Users\rodri\Desktop\portfolio-git> git pull
Auto-merging index.html
CONFLICT (content): Merge conflict in index.html
Automatic merge failed; fix conflicts and then commit the result.
```

Issues

- Uma Issue é um mecanismo usado e para controlar tarefas, melhorias, bugs e outras discussões relacionadas a um projeto de software.
- Como funciona?
 - Um usuário cria uma Issue descrevendo um problema, sugestão ou tarefa.
 - A Issue pode ser categorizada com labels para melhor organização.
 - Desenvolvedores e colaboradores discutem a Issue nos comentários.
 - A Issue pode ser atribuída a membros do time para resolução.
 - Quando resolvida, a Issue é fechada.

Pull Request

- Um **Pull Request (PR)** é um mecanismo usado para propor alterações em um repositório. Ele permite que desenvolvedores solicitem a incorporação de novas funcionalidades, correções de bugs ou melhorias no código.
- Como funciona?
 - Um desenvolvedor cria uma branch e faz alterações no código.
 - Ele envia essas modificações para um repositório remoto.
 - Um Pull Request é aberto para revisão do código por outros desenvolvedores.
 - Comentários e sugestões podem ser feitos antes da aprovação.
 - Se aprovado, o código é mesclado (“merged”) na branch principal.

.gitignore

- O arquivo .gitignore é uma ferramenta essencial ao trabalhar com sistemas de controle de versão como o git. Ele é usado para **especificar quais arquivos e diretórios não devem ser rastreados pelo Git**, ou seja, quais arquivos **não devem ser incluídos no repositório**.
- Isso é útil para **evitar que arquivos temporários, arquivos de configuração específicos do ambiente, arquivos de compilação, ou qualquer outro tipo de arquivo desnecessário sejam adicionados acidentalmente ao repositório**.
- A estrutura do arquivo .gitignore é bastante simples. Cada linha do arquivo representa um padrão a ser ignorado. Padrões podem ser nomes de arquivos específicos, tipos de arquivos (por exemplo, todos os arquivos .log), ou até mesmo diretórios inteiros.

.gitignore

- Exemplo básico de arquivo .gitignore

```
# Comentários começam com cerquilha (#)
```

```
# Ignorar arquivos temporários
```

```
*.tmp
```

```
# Ignorar arquivos de log
```

```
*.log
```

```
# Ignorar arquivos de configuração específicos do ambiente
```

```
.env
```


.gitignore

- Existe um site chamado gitignore.io no qual você pode especificar qual é a tecnologia usada em seu projeto e ele gera automaticamente um arquivo .gitignore contendo os principais tipos de arquivos e diretórios que devem ser ignorados nesse contexto.

Atividade

- Crie um diretório chamado "NovoProjeto" e inicie um repositório Git dentro dele
- Dentro do repositório criado, adicione alguns arquivos
- Realize o commit e inclua a mensagem “Primeiro commit”
- Crie um repositório de GitHub
- Conecte o seu repositório local ao repositório remoto criado
- Crie uma issue realizar alguma alteração no seu projeto
- Crie uma branch para realizar a alteração
- Envie a alteração para o repositório remoto
- Abra um Pull Request para integrar o código da nova branch a branch principal e conecte o PR a Issue aberta. Por fim, aceite o Pull Request.



SUMMER ACADEMY

4 DE FEV

OFICINA GIT, GITHUB E VERSIONAMENTO