FUNDAMENTOS DE DESIGN DE SISTEMAS

AULA 3

Prof. Vinicius Pozzobon Borin

CONVERSA INICIAL

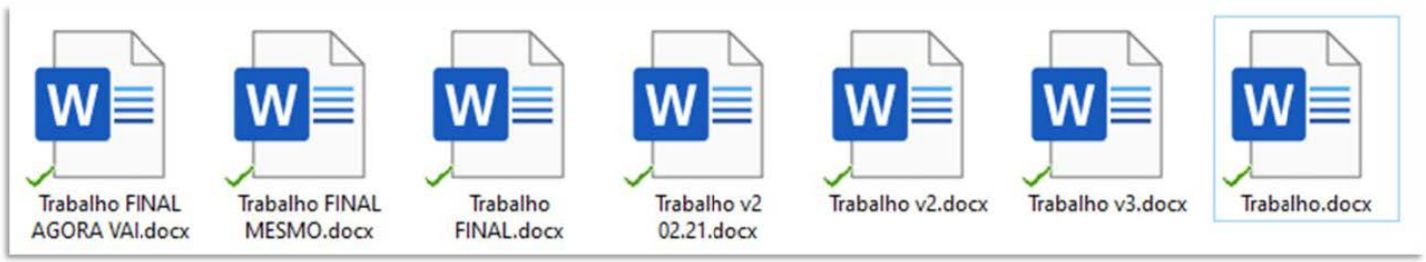
O objetivo desta etapa é conhecer sobre o Git e como ele auxilia no controle de versionamento de arquivos. Veremos:

- O que é Git e versionamento de arquivos;
- Repositórios Git;
- Terminologia do Git;
- Comandos Git.

TEMA 1 – GIT

O controle de versão é uma prática essencial para documentos importantes e, também, para qualquer projeto de software, e o Git é uma das ferramentas mais populares e amplamente utilizadas para esse propósito. Aliás, quem nunca precisou criar diversos arquivos diferentes para um mesmo trabalho e depois ficou perdido com tantas versões? Veja:

Figura 1 – Diferentes versões de um mesmo arquivo



Fonte: Borin, 2023.

Aqui estão algumas das principais razões pelas quais o Git é importante no desenvolvimento de software:

- **Rastreabilidade**: o Git permite que você mantenha uma linha cronológica completa de todas as alterações feitas em seu código. Isso significa que você pode facilmente reverter qualquer alteração que possa ter causado problemas ou simplesmente verificar como o código evoluiu ao longo do tempo;
- **Colaboração**: o Git é projetado para trabalhar com múltiplos desenvolvedores em um projeto. Ele permite que várias pessoas trabalhem no mesmo código e mesclem as suas contribuições sem causar conflitos;
- ***Backup***: o Git mantém uma cópia de todas as versões do seu código, o que significa que você sempre pode recuperar uma versão anterior se algo der errado. Isso também significa que você não precisa se preocupar em perder o seu trabalho, mesmo se o seu computador falhar;
- **Integração contínua**: o Git é compatível com muitas ferramentas de integração contínua, o que significa que você pode automatizar vários processos, como compilação, testes e implantação;
- **Comunidade**: o Git é *open source* e tem uma comunidade muito ativa e vibrante. Isso significa que você pode obter ajuda com problemas e melhorias frequentes e novas funcionalidades.

A ferramenta mais comum de Git é o Git SCM. Hoje pode ser baixada no link a seguir. Deixamos também outros links úteis:

- Download – <https://git-scm.com/>
- Windows, somente – <https://gitforwindows.org/>
- Guia prático – <https://rogerdudler.github.io/git-guide/index.pt_BR.html>

Figura 2 – Ferramenta do Git

Créditos: Postmodern Studio/Shutterstock.

1.1 REPOSITÓRIO GIT

Um repositório Git é uma coleção centralizada de arquivos e diretórios que são controlados pelo Git. É o lugar onde todas as versões do seu código são armazenadas e gerenciadas. Cada repositório Git é único e representa um projeto individual.

Um repositório Git mantém a história completa de todas as alterações feitas nos arquivos e diretórios em seu projeto. Isso permite que você reverta facilmente as alterações, compare versões diferentes e veja como o seu código evoluiu ao longo do tempo.

TEMA 2 – CONFIGURANDO O GIT

O Git é manipulado por meio de diferentes comandos. Lembra que no Linux nós aprendemos alguns comandos de terminal? Aqui aprenderemos comandos com o objetivo de trabalhar com o Git. Os comandos do Git são os mesmo para qualquer sistema operacional.

2.1 COMANDO *GIT CONFIG*

O comando *git config* é usado para configurar as opções do Git e normalmente é o primeiro comando que damos após instalar o Git. Ele permite que você especifique informações sobre você

mesmo, como seu nome e endereço de e-mail, bem como opções de comportamento do Git, como a cor das saídas. Aqui estão alguns exemplos de como usar o comando *git config*:

1. Configurar o seu nome de usuário:

```
$ git config --global user.name "Seu Nome"
```

2. Configurar o seu endereço de e-mail:

```
$ git config --global user.email "seu.email@exemplo.com"
```

3. Exibir todas as configurações do Git:

```
$ git config -list
```

2.2 COMANDO *GIT INIT*

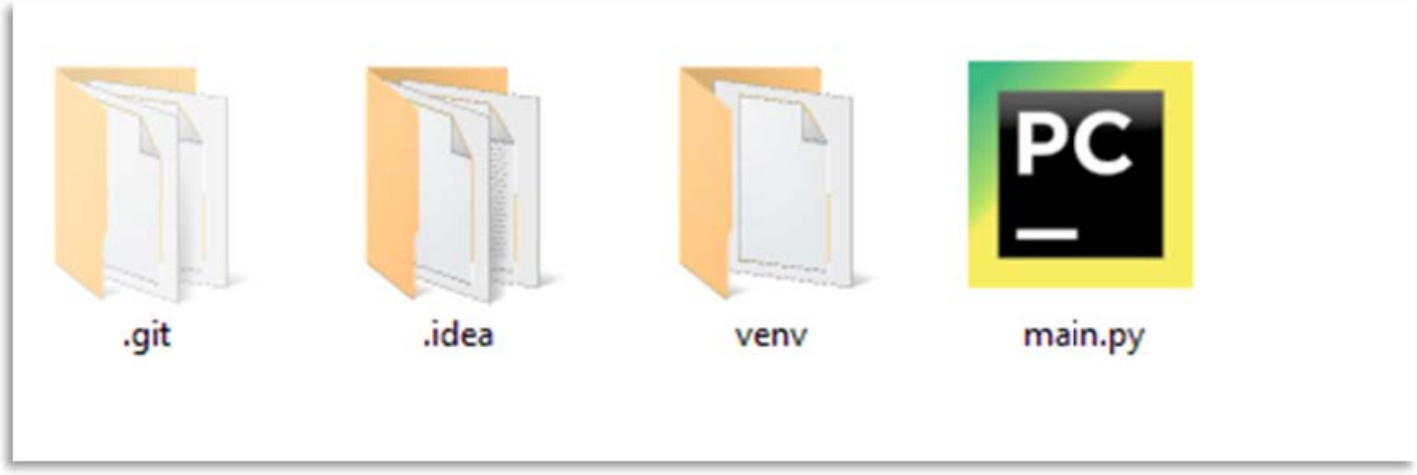
O comando git init é usado para inicializar um novo repositório Git em um diretório existente. Ao ser inicializado, ele cria uma pasta oculta chamada *.git* no diretório atual, que armazena todos os arquivos e informações necessárias para o Git gerenciar as versões do seu código. Aqui está um exemplo de como usar o comando *git init*:

```
$ mkdir novo-projeto
 $ cd novo-projeto
 $ git init
     Inicializando um repositório Git vazio em /caminho/para/novo-
projeto/.git/
```

Note que usamos, no exemplo acima, comandos do Linux também. Primeiro criamos uma pasta (*mkdir*), depois andamos até esta pasta (*cd*). Por fim, inicializamos o repositório. Após executar o comando *git init*, o diretório atual se torna um repositório Git e você pode começar a adicionar, fazer *commit* (conceituaremos isso em breve) e controlar as versões dos seus arquivos.

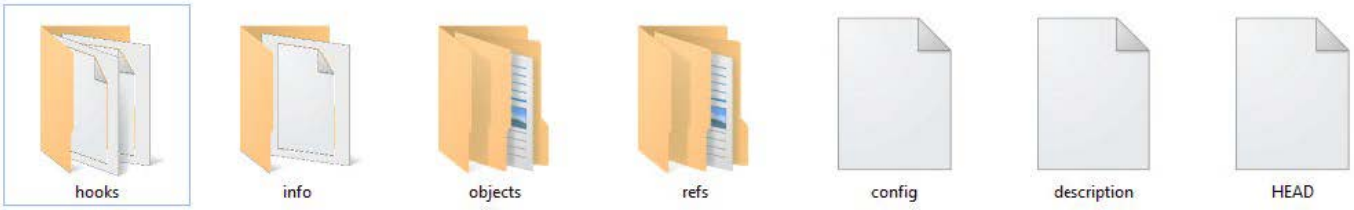
Na Figura 3 e Figura 4, vemos um exemplo de pastas de um projeto de software contendo controle de versionamento Git. Na Figura 3, temos o projeto de um código em linguagem Python. Vemos, também, a pasta .git. Na Figura 4, entramos na pasta do Git para ver os arquivos dentro dela.

Figura 3 – Estrutura de pastas projeto com o Git



Fonte: Borin, 2023.

Figura 4 – Estrutura de pastas projeto com o Git



Fonte: Borin, 2023.

Na estrutura de arquivos da Figura 4 temos um arquivo relevante de ser mencionado, que é o HEAD. Ele serve para manter o histórico das alterações realizadas no projeto.

TEMA 3 – TERMINOLOGIA GIT

Vamos, agora, conhecer alguns termos bastante utilizados no meio Git.

3.1 *BRANCHES* (RAMIFICAÇÕES)

Em Git, um *branch* é uma linha separada (ramificação) de desenvolvimento que permite a você trabalhar em vários recursos ou correções de bugs ao mesmo tempo, sem afetar o código principal da aplicação. Isso significa que você pode criar uma cópia separada da linha principal do código (conhecido como *branch principal* ou *branch master*) e fazer todas as alterações e experimentos

nesse novo *branch* sem afetar o código estável no *branch* principal. Quando você estiver satisfeito com as alterações, pode mesclar (*merge*) o *branch* de volta ao *branch* principal.

Criar *branches* em Git é fácil e eficiente, o que permite a você experimentar novas ideias sem prejudicar o código principal, e ainda pode ser usado para dividir o trabalho em equipe em diferentes tarefas ou para manter diferentes versões do software. É uma das principais características do Git e um dos motivos pelos quais ele é tão popular entre desenvolvedores de software.

Figura 5 – Esquema de ramificações e mesclagem

3.2 FLUXO DE TRABALHO

O fluxo de trabalho básico no Git envolve três etapas principais: adicionar (*add*), cometer (*commit*) e enviar (*push*). Aqui está uma breve explicação de cada etapa:

- *Add*: esta etapa envolve preparar as alterações que você fez no código para serem incluídas em um *commit*. Você usa o comando *git add* para selecionar os arquivos ou partes dos arquivos que deseja incluir no próximo *commit*. É possível adicionar alterações aos arquivos aos poucos, antes de fazer o *commit* com todas as alterações juntas.
- *Commit*: esta etapa envolve registrar as alterações adicionadas em um repositório Git local. Todas as alterações são registradas no arquivo de HEAD. Você usa o comando *git commit* para criar um *commit*, que é uma entrada no histórico do repositório que descreve as alterações realizadas. Cada *commit* tem uma mensagem associada que explica o que foi alterado e por quê.
- *Push*: esta etapa envolve enviar as alterações do repositório local para um repositório remoto, como o GitHub. Você usa o comando *git push* para enviar seus *commits* para o repositório remoto, onde podem ser compartilhados com outras pessoas ou usados como *backup*.

Figura 6 – Fluxo de trabalho do Git

Fonte: Borin, 2023.

3.3 COMANDO *GIT ADD*

Aqui estão alguns exemplos de uso do comando *git add*:

1. Adicionar todos os arquivos:

```
$ git add *
```

2. Adicionar um arquivo específico:

```
$ git add file.txt
```

3. Adicionar uma pasta específica:

```
$ git add folder/
```

3.4 COMANDO *GIT COMMIT*

Aqui estão alguns exemplos de uso do comando *git commit*:

1. Fazer um *commit* com uma mensagem padrão:

```
$ git commit
```

2. Fazer um *commit* com uma mensagem personalizada:

```
$ git commit -m "Adicionado nova funcionalidade X"
```

Em todos os exemplos, o comando *git commit* registra as alterações adicionadas no repositório Git local.

3.5 COMANDO *GIT PUSH*

Aqui estão alguns exemplos de uso do comando *git push*:

1. Enviar todos os *branches* locais para o repositório remoto:

```
$ git push origin –all
```

2. Enviar um branch específico para o repositório remoto:

```
$ git push origin nome-do-branch
```

TEMA 4 – OUTROS COMANDOS GIT

Existem outros comandos Git que precisam ser conhecidos para que tenhamos o melhor uso de todos os seus recursos e funcionalidades. Vejamos a seguir.

4.1 COMANDO *GIT LOG*

O *git log* é um comando que exibe o histórico de *commits* de um repositório Git. Ele mostra uma lista de todos os *commits* realizados no repositório, incluindo informações sobre o autor, a data de *commit*, a mensagem de *commit* e a identificação única do *commit* (a *hash*). Vejamos alguns exemplos de como o *git log* pode ser usado:

1. Exibir o histórico de *commit* completo:

```
$ git log
```

2. Exibir o histórico de *commit* para um determinado arquivo:

```
$ git log <file>
```

3. Exibir o histórico de *commit* para um determinado período:

```
$ git log --since=<YYYY-MM-DD> --until=<YYYY-MM-DD>
```

4.2 COMANDO *GIT STATUS*

O *git status* é um comando que exibe o estado atual de um repositório Git. Ele mostra informações sobre arquivos que foram modificados, adicionados ou excluídos, bem como informações sobre quais arquivos estão sendo seguidos e quais ainda não foram realizados *commit*.

Aqui estão alguns exemplos de como o *git status* pode ser usado:

1. Exibir o estado atual do repositório:

```
$ git status
```

2. Exibir o estado atual dos arquivos de uma determinada *branch*:

```
$ git status <branch>
```

4.3 COMANDO *GIT RESET*

O *git reset* é um comando do Git que permite desfazer uma ou mais ações no seu repositório. Ele permite que você reverta uma operação específica, como um *commit* ou uma operação de *merge*, ou remove um arquivo do índice antes de realizar um *commit*. O *git reset* é útil quando você precisa desfazer alguma coisa que foi feita por engano ou quando precisa reajustar o histórico de seu repositório antes de prosseguir com o desenvolvimento.

Existem três modos de operação do *git reset*: *--soft, --mixed* e *--hard*. Aqui estão alguns exemplos de uso do *git reset*:

1. Desfazer o último *commit*:

```
$ git reset --soft HEAD^
```

2. Remover um arquivo do índice: Para remover um arquivo do índice antes de realizar um *commit*, use o comando:

```
$ git reset <nome-do-arquivo>
```

É importante ter cuidado ao usar o comando *git reset*, pois ele pode remover permanentemente informações do seu repositório (se usar o modo *hard*). Por isso, é recomendável fazer *backup* dos seus dados antes de usar o *git reset* em modo --hard.

4.4 COMANDO *GIT DIFF*

O comando *git diff* é um comando do Git que permite comparar duas versões de um ou mais arquivos para mostrar as diferenças entre eles. É útil para ver as alterações que você fez em seus arquivos antes de realizar um *commit* ou para comparar diferentes versões de um projeto. Aqui estão alguns exemplos de uso do *git diff*:

1. Comparar duas versões de um arquivo:

```
$ git diff <hash-do-commit> <nome-do-arquivo>
```

2. Comparar dois *commits*:

```
$ git diff <hash-do-commit1> <hash-do-commit2>
```

3. Ver as diferenças entre o repositório atual e a última versão enviada (*push*) para o repositório remoto:

```
$ git diff origin/master
```

O comando *git diff* é uma ferramenta poderosa para comparar versões de arquivos, e ajuda a garantir que você esteja fazendo as alterações corretas em seu projeto antes de realizar um *commit*.

TEMA 5 – TRABALHANDO COM *BRANCHES*

Vamos conhecer comandos para criação de ramificações em nossos projetos.

5.1 COMANDO *GIT BRANCH*

O comando g*it branch* é uma característica do Git que permite criar ramificações (*branches*) do seu projeto. Cada *branch* representa uma linha independente de desenvolvimento e permite que você trabalhe em diferentes versões do seu projeto ao mesmo tempo sem afetar a outra. Isso é útil quando você precisa corrigir um *bug*, implementar uma nova funcionalidade ou fazer uma alteração significativa em seu projeto sem interromper o trabalho principal.

Aqui estão alguns exemplos de uso do *git branch*:

1. Para listarmos todas as *branches* existentes no projeto e checarmos em qual *branch* estamos, fazemos somente:

```
$ git branch
```

2. Criar uma nova *branch*:

```
$ git branch <nome-da-branch>
```

3. Excluir uma *branch*:

```
$ git branch -d <nome-da-branch>
```

5.2 COMANDO *GIT CHECKOUT*

O comando *git checkout* é um comando do Git que permite alternar entre *branches*, repositórios remotos ou versões de arquivos. Ele é usado para mudar o contexto do seu repositório local, permitindo que você trabalhe em diferentes versões do seu projeto sem interferir nas outras.

Aqui estão alguns exemplos de uso do *git checkout*:

1. Mudar de *branch*:

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

2. Criar uma nova *branch* e mudar para ela:

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

3. Recuperar uma versão antiga de um arquivo:

```
$ git checkout <hash-do-commit> <nome-do-arquivo>
```

4. Desfazer alterações em um arquivo:

```
$ git checkout -- <nome-do-arquivo>
```

FINALIZANDO

Em conclusão, Git é uma ferramenta incrivelmente poderosa e útil para o controle de versões de código. Ao dominar seus conceitos básicos, como o uso de repositórios, *commits*, *branches* e *merges*, você será capaz de trabalhar de maneira mais eficiente e colaborativa em projetos de software. Além disso, na grande comunidade de desenvolvedores que usam Git significa que você tem acesso a uma ampla variedade de recursos e ferramentas adicionais para aprimorar sua experiência de trabalho. Se você ainda não está familiarizado com Git, esperamos que esta etapa tenha fornecido uma base sólida para continuar seu aprendizado e uso desta poderosa ferramenta.

REFERÊNCIAS

CHACON, S.; STRAUB, B. **Pro Git**. 2. ed. [S.l.]: Editora Apress, [s.d.].

DOCUMENTAÇÃO oficial do Git. **Git-scm**, s.d. Disponível em: <https://git-scm.com/docs>. Acesso em: 23 fev. 2023.

GIT. Guia prático. **Roger Dudler**, s.d. Disponível em: <https://rogerdudler.github.io/git-guide/index.pt_BR.html>. Acesso em: 23 fev. 2023.