

Ronivaldo Domingues de Andrade

Capacitação em GitHub

Rio de Janeiro - RJ

2025

Ronivaldo Domingues de Andrade

Capacitação em GitHub

Guia Prático para Capacitação em GitHub

Rio de Janeiro - RJ
2025

Lista de ilustrações

Figura 1 – Página para a criação de conta no GitHub	12
Figura 2 – Primeira visão do GitHub depois de criar a conta	12
Figura 3 – Adicionar e verificar e-mail acadêmico no GitHub	15
Figura 4 – Página do GitHub Student Developer Pack	16
Figura 5 – Página do GitHub Student Developer Pack	16
Figura 6 – Iniciando a aplicação no GitHub Student Developer Pack	17
Figura 7 – Verificando se o Winget está instalado.	19
Figura 8 – Buscando o Git no Winget.	22
Figura 9 – Instalação do Git.	22
Figura 10 – Verificando se o Git está instalado.	23
Figura 11 – Instalando o GitHub-CLI.	24
Figura 12 – Verificando se o GitHub-CLI está instalado.	25
Figura 13 – Efetuando a autenticação com o GitHub-CLI - Passo 1.	27
Figura 14 – Efetuando a autenticação com o GitHub-CLI - Passo 2.	27
Figura 15 – Efetuando a autenticação com o GitHub-CLI - Passo 3.	27
Figura 16 – Efetuando a autenticação com o GitHub-CLI - Passo 4.	28
Figura 17 – Efetuando a autenticação com o GitHub-CLI - Passo 5.	28
Figura 18 – Configurando o Git.	30

Lista de tabelas

Tabela 1	–	GitHub Free (Pessoal) vs. GitHub Student Developer Pack (Pro)	. .	14
Tabela 2	–	Comandos do Git	75
Tabela 3	–	Comandos do GitHub-CLI (gh)	80
Tabela 4	–	Padrões de Commits Profissionais	90

Lista de abreviaturas e siglas

Winget	Windows Package Manager
GitHub	Plataforma de Controle de Versão Distribuído
git	Sistema de Controle de Versão Distribuído
gh	GitHub CLI (Command Line Interface)
git-lfs	Sistema de Controle de Versão Distribuído para Arquivos Grandes
Merge	Fusão de Branches no GitHub
Branch	Rama (branch) em um repositório GitHub
Commit	Confirmação de Modificações em um Branch
GitHub Pages	Serviço de Hospedagem de Páginas Estáticas
GitHub Actions	Plataforma de Automação de Fluxos de Trabalho
Pull Request (PR)	Solicitação de Mesclagem de Código
Markdown	Linguagem de Marcação Leve
README	Arquivo de Documentação do Projeto
.gitignore	Arquivo de Configuração para Ignorar Arquivos no Git
LICENSE	Arquivo de Licença do Projeto
MIT	Licença MIT -> Massachusetts Institute of Technology License
YML	YAML Ain't Markup Language -> YAML Não é uma Linguagem de Marcação de Texto, mas sim uma sintaxe para arquivos YAML
GPG	GNU Privacy Guard -> Guarda de Privacidade GNU, ferramenta de criptografia de dados e comunicação segura.

Sumário

Lista de ilustrações		2
Lista de tabelas		3
Sumário		5
1	INTRODUÇÃO	10
2	GITHUB	11
2.1	O que é?	11
2.2	Criando seu perfil no GitHub	11
2.2.1	E-mail acadêmico e GitHub Student Developer Pack	13
2.2.1.1	Por que usar o GitHub Student Developer Pack?	13
2.2.1.2	GitHub Free vs GitHub Student Developer Pack (GSDP)	13
2.2.1.3	GitHub Free vs GSDP	14
2.3	Obtendo o GitHub Student Developer Pack	15
3	WINGET	18
3.1	O que é?	18
3.2	Porque usar nessa capacitação?	18
3.2.1	Instalação	18
3.2.1.1	Atualização	19
4	GIT E GITHUB-CLI	21
4.1	O que é o Git?	21
4.1.1	Instalação do Git	21
4.2	O que é o GitHub-CLI?	23
4.2.1	Instalação do GitHub-CLI	24
4.3	Configuração do Git e GitHub-CLI	25
4.3.1	Autenticação	25

4.3.1.1	Autenticação com o GitHub-CLI	26
4.3.2	Configuração de usuário Git	29
4.3.2.1	Autenticação usando PAT (Opcional)	30
4.3.3	Configuração de Editor Padrão (Opcional)	31
4.3.4	Configurar a branch padrão para 'main' (Opcional)	31
4.4	Comandos Básicos do Git e GitHub-CLI	32
4.4.1	Comandos Básicos do Git	32
4.4.2	Comandos Básicos do GitHub-CLI	33
5	GIT LFS	34
5.1	O que é?	34
5.1.1	Motivos e Problemas que Resolve	34
5.1.2	Como Funciona	35
5.1.3	Vantagens	35
5.1.4	Limitações	36
5.1.5	Exemplo de Uso	36
5.1.6	Boas Práticas	36
6	COMMITTS, MERGES E PULL REQUESTS	38
6.1	Introdução	38
6.2	Commits	38
6.2.1	O que é um commit	38
6.2.2	Boas práticas de commits	38
6.2.3	Fazendo commits passo a passo	39
6.2.4	Editar o último commit / corrigir mensagens	40
6.2.5	Desfazer / alterar staging	40
6.2.6	Padrões de Commits	41
6.3	Merges	42
6.3.1	Tipos de merge	42
6.3.2	Merge local com merge commit (passo a passo)	42
6.3.3	Rebase (passo a passo) para um histórico linear	43
6.3.4	Resolver conflitos passo a passo	44
6.3.5	Squash e reescrita de commits (passo a passo)	45

6.4	Pull Requests (PR)	45
6.4.1	O que é um Pull Request	45
6.4.2	Fluxo básico criando um PR (via web)	45
6.4.3	Criar e gerenciar PRs via GitHub CLI (passo a passo)	46
6.4.4	Checklist para revisão de Pull Request	47
6.4.5	Depois do merge limpeza e sincronização	47
6.5	Boas práticas e recomendações finais	48
6.6	Exemplos rápidos de comandos úteis	48
7	GITHUB PAGES E GITHUB ACTIONS	50
7.1	O que é GitHub Pages?	50
7.2	O que é GitHub Actions?	50
7.3	Integração entre GitHub Pages e GitHub Actions	51
8	ASSINATURAS DE COMMITS COM CHAVE GPG	54
8.1	O que é?	54
8.1.1	Importância das assinaturas GPG	54
8.2	Como usar?	54
8.2.1	Passo 1: Instalar o GPG	54
8.2.2	Passo 2: Gerar uma chave GPG	55
8.2.3	Passo 3: Listar chaves e copiar o ID da chave	55
8.2.4	Passo 4: Configurar o Git para usar a chave GPG	56
8.2.5	Passo 5: Adicionar a chave GPG ao GitHub	56
8.2.6	Passo 6: Fazer commits assinados	56
8.2.7	Passo 7: Verificar commits assinados	56
8.3	Dicas de segurança e boas práticas	57
9	EXERCÍCIOS PRÁTICOS	58
9.1	Git e GitHub-CLI	58
9.1.1	Objetivo	58
9.1.2	Passo a Passo Detalhado	58
9.1.3	Problemas Comuns e Soluções	58
9.2	Criar um repositório no GitHub via CLI	59
9.2.1	Objetivo	59

9.2.2	Pré-requisito	59
9.2.3	Passo a Passo	59
9.2.4	README.md	59
9.2.4.1	O que é README.md	59
9.2.4.2	Como Criar	60
9.2.5	LICENSE	61
9.2.5.1	Por que usar LICENSE	61
9.2.5.2	Como Adicionar Licença MIT	61
9.3	Clonando um repositório do GitHub	62
9.3.1	Objetivo	62
9.3.2	Passo a Passo	62
9.4	Github Pages	63
9.4.1	Objetivo	63
9.4.2	Passo a Passo	63
9.5	Github Actions	64
9.5.1	Objetivo	64
9.5.2	Passo a Passo	64
9.5.3	Uso de [skip ci] no GitHub Actions	66
9.6	Merge	67
9.6.1	Objetivo	67
9.6.2	Passo a Passo	67
9.7	Pull Request	68
9.7.1	Objetivo	68
9.7.2	Passo a Passo	69
9.8	Materiais de Apoio	70
9.8.1	Checklist para Cada Exercício	70
9.8.2	Comandos Úteis para Consulta	70
9.8.3	Dicas para Boas Práticas	71
10	CONCLUSÃO	72
	REFERÊNCIAS	73

	APÊNDICES	74
	APÊNDICE A – COMANDOS GIT	75
	APÊNDICE B – COMANDOS GITHUB CLI	80
	APÊNDICE C – PADRÕES DE COMMITS	90
	ANEXOS	94
	ANEXO A – LISTA DE PRESENÇA	95
A.1	Listab de dos mebros presentes na capacitação	95

1 Introdução

O GitHub consolidou-se como uma das principais plataformas de desenvolvimento colaborativo, sendo amplamente adotado por equipes e desenvolvedores individuais para o controle de versão, a gestão de projetos e a integração contínua. No entanto, o uso eficiente de suas ferramentas exige não apenas familiaridade com conceitos básicos, mas também o domínio de boas práticas e fluxos de trabalho modernos.

Esta capacitação foi elaborada com o objetivo de oferecer um guia prático e acessível para o uso do GitHub e de suas tecnologias associadas, como Git, GitHub CLI, Git LFS, GitHub Pages e GitHub Actions. O material abrange desde a configuração inicial do ambiente até a execução de operações avançadas, como a assinatura de commits com GPG e a automação de fluxos de trabalho.

Além disso, são apresentados exercícios práticos que simulam situações reais de desenvolvimento, permitindo que os participantes vivenciem todo o ciclo de colaboração em projetos versionados. Com isso, espera-se que, ao final do curso, os participantes estejam aptos a contribuir de forma segura, organizada e profissional em repositórios locais e remotos, seja em projetos pessoais ou corporativos.

2 GitHub

2.1 O que é?

O GitHub é uma plataforma de hospedagem de código-fonte que utiliza o sistema de controle de versão Git. Ele permite que desenvolvedores colaborem em projetos, compartilhem código e gerenciem alterações de forma eficiente. Com o GitHub, é possível criar repositórios, realizar pull requests, revisar código e acompanhar o histórico de alterações.

Além disso, o GitHub oferece recursos adicionais, como GitHub Pages para hospedagem de sites estáticos, GitHub Actions para automação de fluxos de trabalho e integração com diversas ferramentas de desenvolvimento.

O GitHub é amplamente utilizado na indústria de software, sendo uma ferramenta essencial para desenvolvedores, equipes de desenvolvimento e organizações que buscam melhorar a colaboração e a gestão de projetos de software.

2.2 Criando seu perfil no GitHub

Para criar uma conta no GitHub, siga os passos abaixo:

1. Acesse o site do GitHub: [<https://github.com/>](https://github.com/)
2. Clique em "Sign up" no canto superior direito.
3. Preencha os campos solicitados, como endereço de e-mail, nome de usuário e senha - Figura 1, p.12.

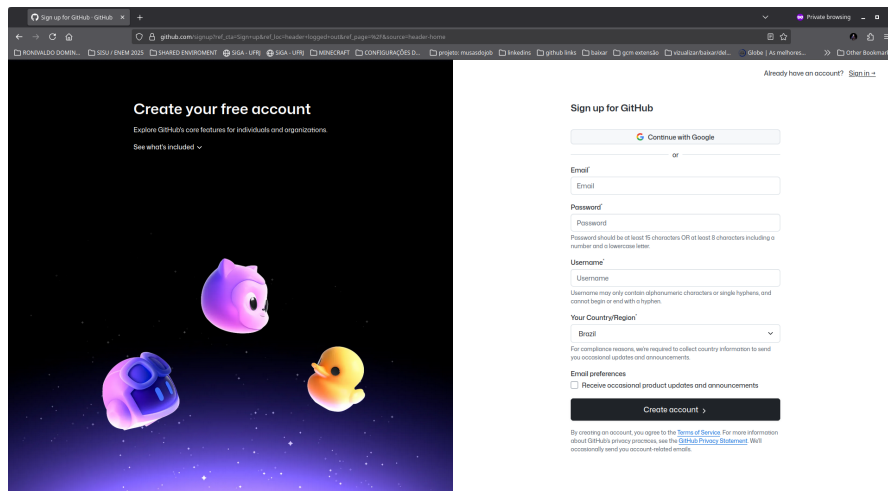


Figura 1 – Página para a criação de conta no GitHub

4. Siga as instruções na tela para concluir o processo de criação da conta.
5. Ao final você verá a tela inicial do GitHub - Figura 2, p.12.

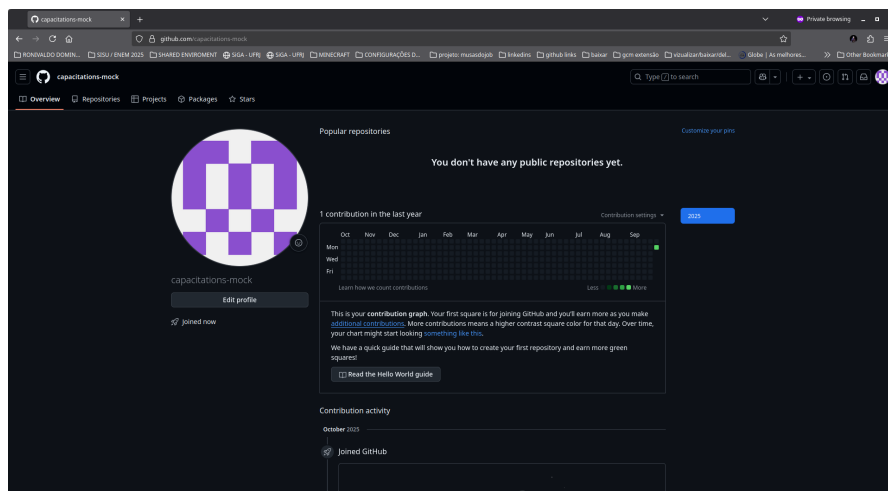


Figura 2 – Primeira visão do GitHub depois de criar a conta

Após criar a conta, você poderá acessar o GitHub e começar a explorar seus recursos.

2.2.1 E-mail acadêmico e GitHub Student Developer Pack

Para usar o GitHub Student Developer Pack e tornar sua conta uma GitHub Pro, você deve associar um e-mail acadêmico à sua conta. Isso pode ser feito nas configurações da conta, na seção "Emails". Adicionar um e-mail acadêmico pode ajudar a validar sua identidade como estudante ou profissional da área de tecnologia.

Com isso o GitHub Student Developer Pack fornece acesso gratuito a diversas ferramentas e serviços para estudantes. Para se inscrever, você precisará verificar seu status de estudante com um e-mail acadêmico válido.

2.2.1.1 Por que usar o GitHub Student Developer Pack?

O GitHub Student Developer Pack oferece uma série de benefícios, incluindo acesso gratuito a ferramentas de desenvolvimento, serviços de hospedagem e outros recursos que podem ser extremamente úteis para estudantes que estão aprendendo a programar e desenvolver software.

2.2.1.2 GitHub Free vs GitHub Student Developer Pack (GSDP)

A conta gratuita do GitHub oferece recursos básicos, como repositórios públicos e privados, colaboração em projetos e integração com outras ferramentas. Já a conta GitHub Student Developer Pro oferece benefícios adicionais, como acesso a ferramentas premium, maior capacidade de armazenamento e recursos avançados de colaboração.

2.2.1.3 GitHub Free vs GSDP

Recurso / Limite	GitHub Free (Conta Pessoal)	GitHub Student Developer Pack (GSDP)	Diferencial Estratégico
Acesso a Repositórios Privados	Ilimitado (Recursos Limitados)	Ilimitado (Recursos Pro/Avançados)	Governança de Código
Minutos do GitHub Actions (Mensal)	2,000 minutos	3,000 minutos	Maior Resiliência de CI/CD (+50%)
Armazenamento de Packages	500 MB	2 GB	Suporte a Artefatos e Contêineres (+400%)
Horas de Core do Codespaces (Mensal)	120 horas	180 horas	Desenvolvimento em Nuvem Estendido
Armazenamento Codespaces (Mensal)	15 GB	20 GB	Maior Capacidade de Workspace
Revisores Obrigatórios (Private Repos)	Não Disponível	Disponível (Recurso Pro)	Enforçamento de Qualidade e Compliance
Suporte	Suporte Comunitário	Suporte Comunitário	Base de Suporte
Acesso ao GitHub Copilot	Não Incluído (Subscrição Paga)	Incluído (Geralmente Copilot Pro)	Produtividade e Aceleração por IA

Tabela 1 – GitHub Free (Pessoal) vs. GitHub Student Developer Pack (Pro)

2.3 Obtendo o GitHub Student Developer Pack

1. Vá até as configurações da sua conta no GitHub.
2. Na seção "Emails", adicione seu e-mail acadêmico - Figura 3, p.15.

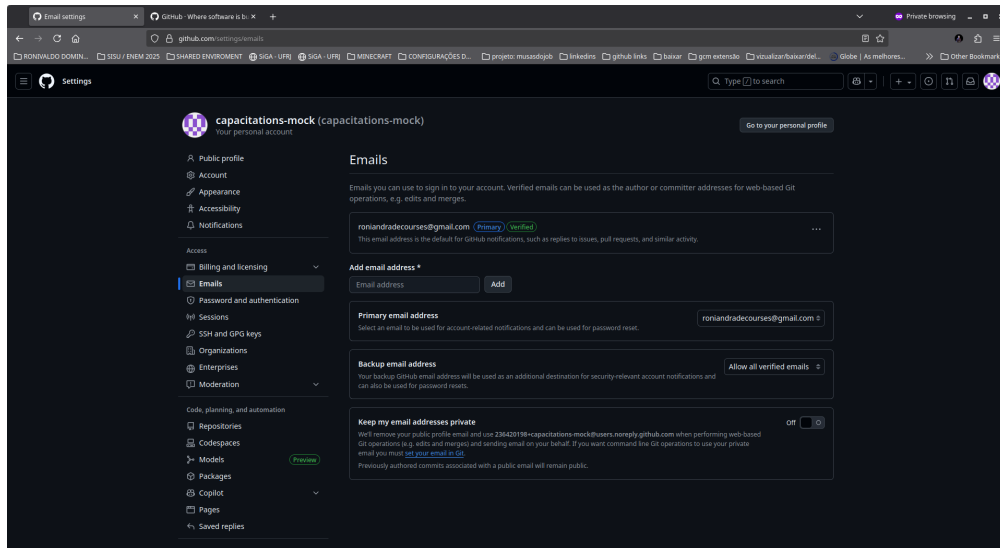


Figura 3 – Adicionar e verificar e-mail acadêmico no GitHub

3. Clique em "Add" para adicionar o e-mail.
4. Verifique o e-mail clicando no link enviado para sua caixa de entrada.
5. Após verificar o e-mail, você pode se inscrever no GitHub Student Developer Pack.
6. Acesse o site do GitHub Student Developer Pack: <https://education.github.com/pack> - Figura 4, p.16.

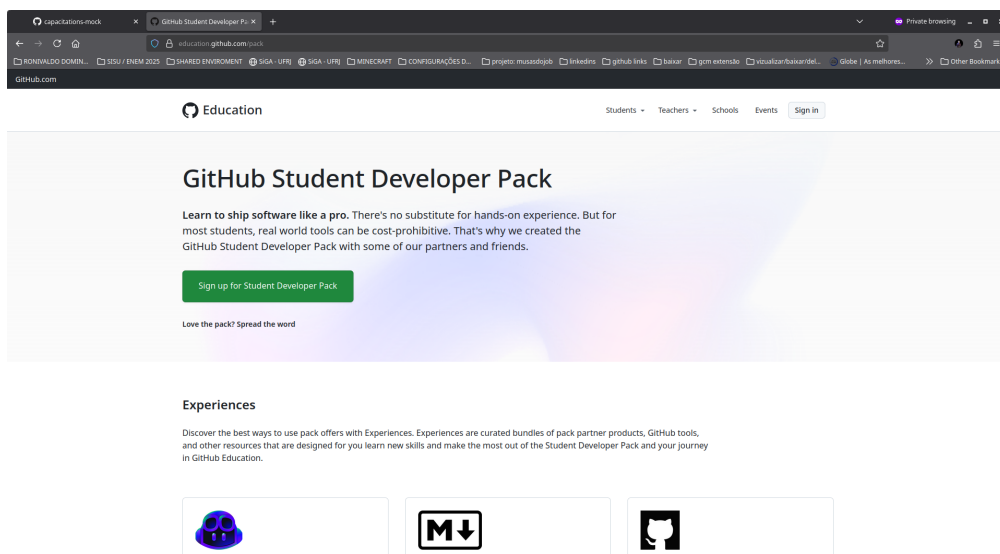


Figura 4 – Página do GitHub Student Developer Pack

7. Clique em "Sign up for Student Developer Pack".
8. Isso redirecionará para fazer login na sua conta do GitHub, caso não esteja logado - Figura 5, p.16.

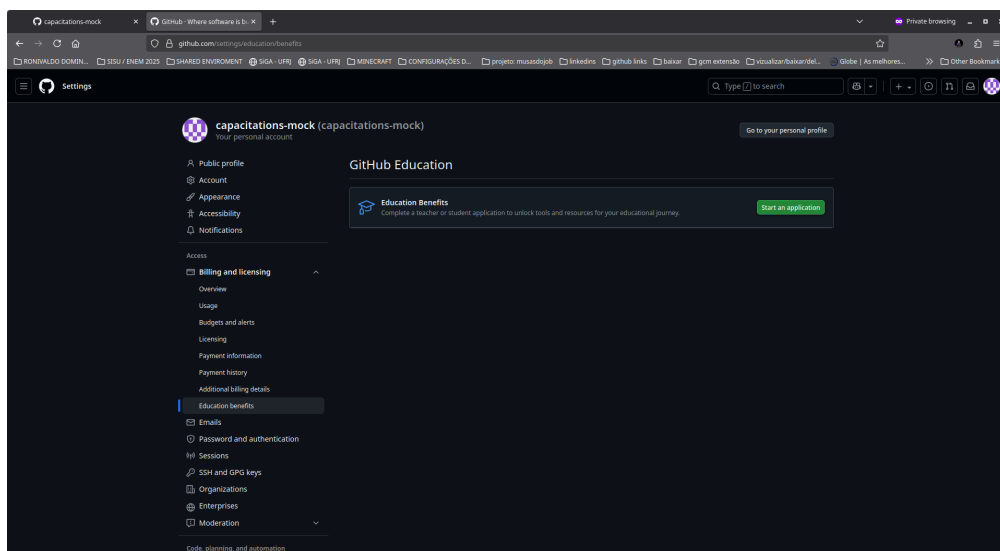


Figura 5 – Página do GitHub Student Developer Pack

9. Preencha os campos solicitados, incluindo seu e-mail acadêmico - Figura 6, p.17.

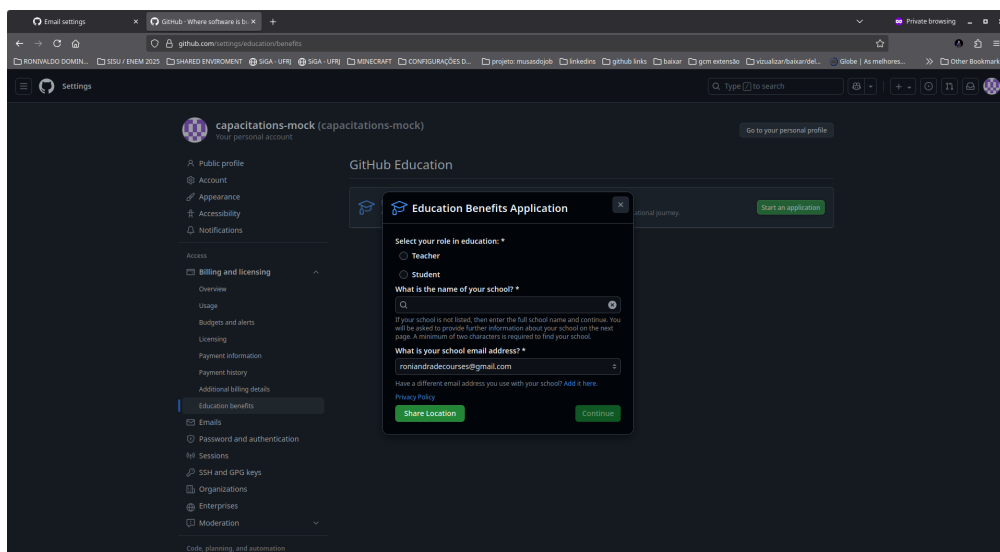


Figura 6 – Iniciando a aplicação no GitHub Student Developer Pack

10. Anexe um comprovante de matrícula ou uma carta da instituição de ensino, se solicitado. **(Importante: Use um documento oficial da instituição, por experiência própria, use a carteirinha de estudante que possui foto, para mim o processo foi mais rápido ao usar.)**
11. Envie a solicitação e aguarde a aprovação, que pode levar alguns dias.

3 Winget

3.1 O que é?

O Winget, ou Windows Package Manager, é uma ferramenta de linha de comando para Windows que permite instalar, atualizar e gerenciar aplicativos de forma simples e eficiente. Ele foi desenvolvido pela Microsoft e é uma solução nativa para gerenciamento de pacotes no Windows.

Para maiores informações e entendimento veja a documentação em ([Microsoft 2025](#)).

3.2 Porque usar nessa capacitação?

Para que todos possam acompanhar de maneira eficaz essa capacitação, preciso que todos tenham as ferramentas git e GitHub CLI instaladas em suas máquinas. Para facilitar esse processo, utilizaremos o Winget, que é o gerenciador de pacotes nativo do Windows. O Winget é um gerenciador de pacotes para Windows que facilita a instalação, atualização e remoção de aplicativos. Com ele, é possível automatizar a configuração do ambiente de desenvolvimento, economizando tempo e esforço.

3.2.1 Instalação

A ferramenta de linha de comando do WinGet só tem suporte no Windows 10 versão 1809 (build 17763) ou posterior. O WinGet não estará disponível até que você tenha feito login no Windows como usuário pela primeira vez, o que fará com que a Microsoft Store registre o Gerenciador de Pacotes do Windows como parte de um processo assíncrono. Se você tiver feito login recentemente como usuário pela primeira vez e o WinGet ainda não estiver disponível, abra o PowerShell e insira o seguinte comando para solicitar o registro dele:

```
Add-AppxPackage -RegisterByFamilyName -MainPackage  
Microsoft.DesktopAppInstaller_8wekyb3d8bbwe. Microsoft 2025.
```

1. Verifique se o Winget já está instalado no seu sistema. Abra o Prompt de Comando ou PowerShell e digite

```
winget --version
```

Se o comando retornar uma versão, o Winget já está instalado.

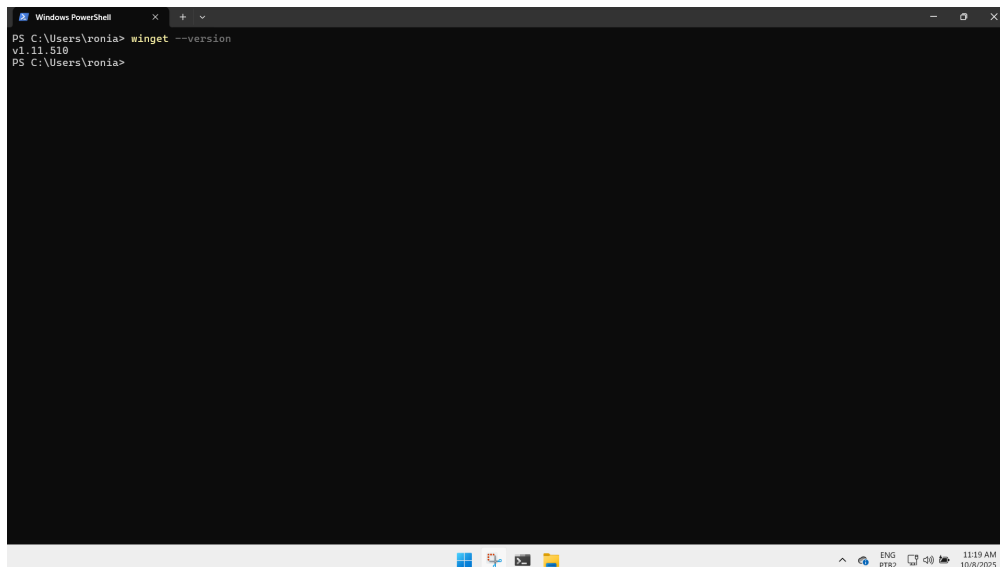


Figura 7 – Verificando se o Winget está instalado.

2. Caso o Winget não esteja instalado, você pode baixá-lo como parte do aplicativo "App Installer" da Microsoft Store. Acesse a Microsoft Store, procure por "App Installer" e clique em "Obter" para instalar. Caso enfrente alguma dificuldade, acesse a documentação em ([Microsoft 2025](#)).
3. Após a instalação, reinicie o Prompt de Comando ou PowerShell e verifique novamente a instalação com:

```
winget --version
```

O comando deve retornar a versão do Winget instalada.

3.2.1.1 Atualização

Para garantir que você está utilizando a versão mais recente do Winget, execute o comando:

```
winget upgrade --all
```

Este comando atualizará todos os pacotes instalados, incluindo o próprio Winget, se houver uma atualização disponível.

4 Git e GitHub-CLI

4.1 O que é o Git?

O Git é um software open source, gratuito e multiplataforma voltado para o versionamento de código. Ele oferece um sistema de controle de versão distribuído, amplamente utilizado no desenvolvimento de software, que permite que múltiplos desenvolvedores trabalhem simultaneamente em um mesmo projeto. O Git rastreia as alterações realizadas nos arquivos, possibilitando reverter modificações, comparar versões e gerenciar ramificações (branches) de forma eficiente e segura.

4.1.1 Instalação do Git

1. Abra o Prompt de Comando ou PowerShell.
2. Digite o comando:

```
winget search git
```

3. Na primeira vez que algum comando do Winget for executado, ele pedirá o aceite dos termos de uso. Ao aceitar, digitando Y ou y.
4. Nesse comando `winget search git`, será buscado na base do winget todas as ocorrências em que o termo `git` aparece e será retornado uma tabela com os resultados associando o nome do programa, seu [id](#) para a instalação e algumas outras informações.

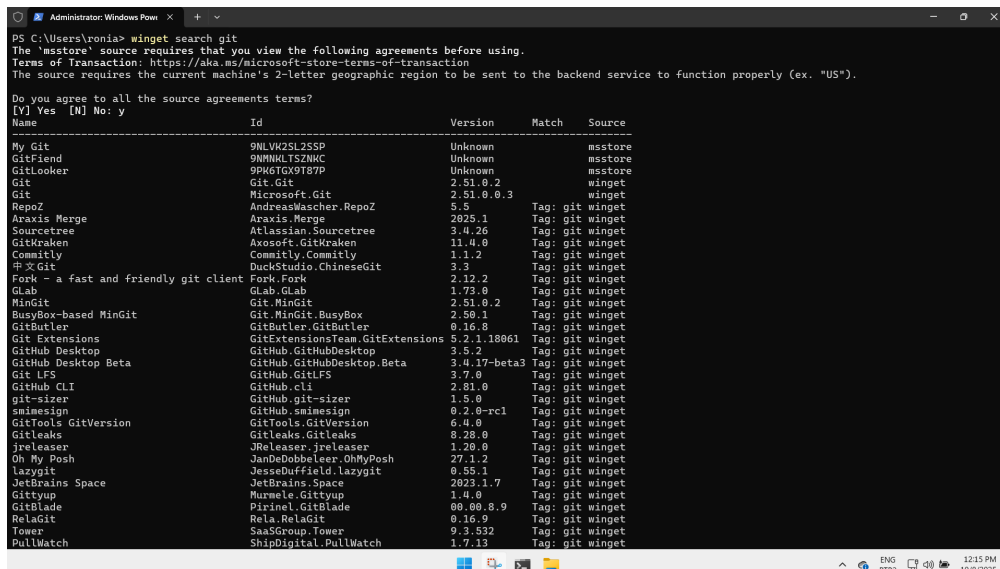


Figura 8 – Buscando o Git no Winget.

- Assim que localizar o software que deseja, copie ou memorize seu **id**, em nosso caso o **id = Git.Git**, e digite o comando:

```
winget install Git.Git
```

e pressione Enter.

- Siga as instruções na tela para concluir a instalação.
- Aguarde a conclusão da instalação.

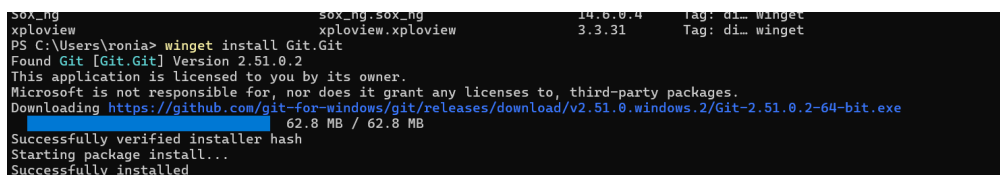
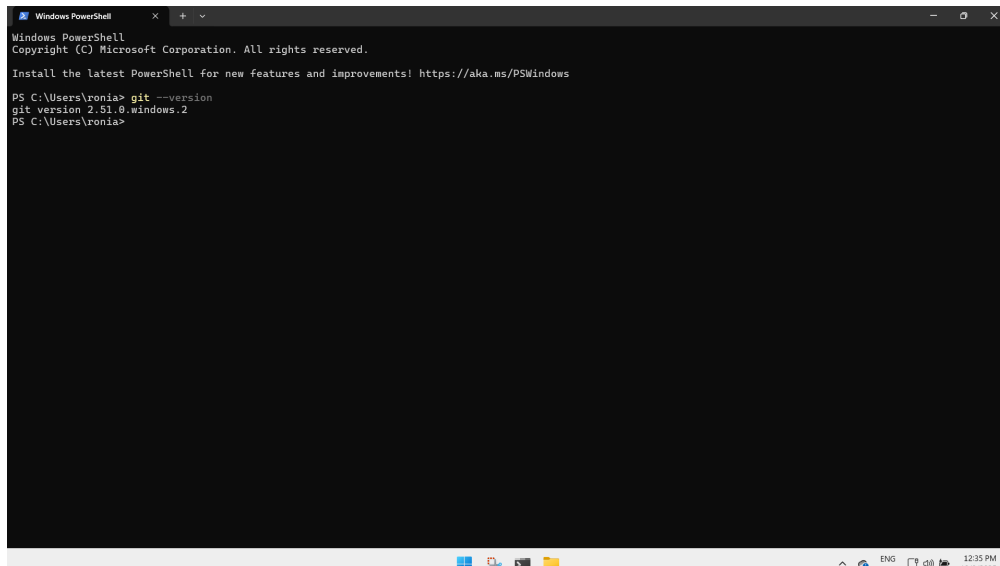


Figura 9 – Instalação do Git.

- Verifique a instalação digitando, em alguns casos o Windows exige que o terminal seja reiniciado para que as variáveis de ambiente adicionadas com a instalação sejam carregadas corretamente:

```
git --version
```

O comando deve retornar a versão do Git instalada.



```
Windows PowerShell
Copyright (C) Microsoft Corporation. All rights reserved.

Install the latest PowerShell for new features and improvements! https://aka.ms/PSWindows

PS C:\Users\ronia> git --version
git version 2.51.0.windows.2
PS C:\Users\ronia>
```

Figura 10 – Verificando se o Git está instalado.

4.2 O que é o GitHub-CLI?

O GitHub-CLI (Command Line Interface) é uma ferramenta de linha de comando desenvolvida pela GitHub, Inc., que permite interagir com os repositórios e recursos do GitHub diretamente pelo terminal, sem a necessidade de acessar a interface web.

Com o GitHub-CLI, é possível executar operações comuns como clonar repositórios, criar issues, abrir e revisar pull requests, gerenciar branches, autenticar usuários, visualizar status de workflows e automatizar fluxos de trabalho, integrando-se perfeitamente com o Git e com scripts de automação.

Essa ferramenta é especialmente útil para desenvolvedores que preferem trabalhar no terminal, proporcionando agilidade, automação e maior produtividade no gerenciamento de projetos hospedados no GitHub.

4.2.1 Instalação do GitHub-CLI

1. Abra o Prompt de Comando ou PowerShell.
2. Agora `id` = `GitHub.cli`, e digite o comando:

```
winget install GitHub.cli
```

e pressione Enter.

3. Aguarde a conclusão da instalação.

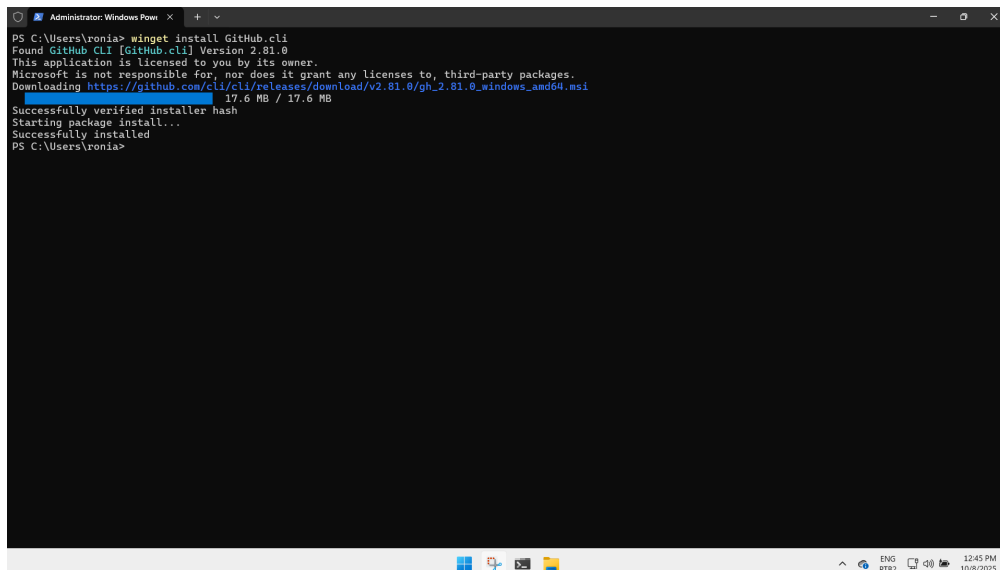
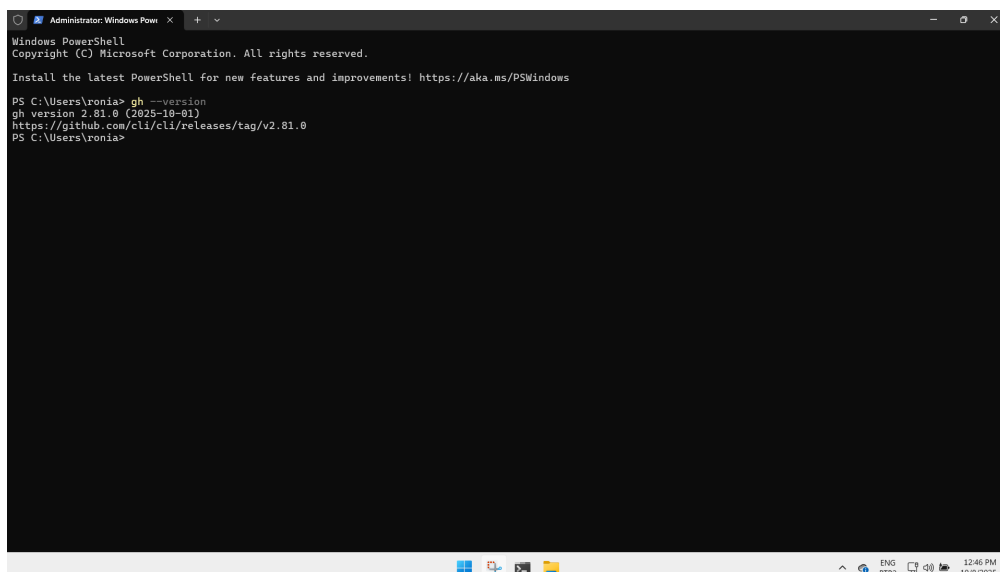


Figura 11 – Instalando o GitHub-CLI.

4. Verifique a instalação digitando:

```
gh --version
```

O comando deve retornar a versão do GitHub-CLI instalada.



```
Administrator: Windows PowerShell
Windows PowerShell
Copyright (C) Microsoft Corporation. All rights reserved.

Install the latest PowerShell for new features and improvements! https://aka.ms/PSWindows

PS C:\Users\ronia> gh --version
gh version 2.81.0 (2025-10-01)
https://github.com/cli/cli/releases/tag/v2.81.0
PS C:\Users\ronia>
```

Figura 12 – Verificando se o GitHub-CLI está instalado.

4.3 Configuração do Git e GitHub-CLI

Após a instalação do Git e do GitHub-CLI, é necessário realizar algumas configurações iniciais para garantir que suas informações estejam corretas ao fazer commits e interagir com o GitHub.

4.3.1 Autenticação

Existem duas formas seguras de autenticação para o Git, a primeira envolve a geração de um token de acesso pessoal (PAT - Personal Access Token) no GitHub, que é usado como senha ao fazer push ou pull de repositórios remotos. A segunda forma é a autenticação via SSH, que envolve a criação de um par de chaves SSH (pública e privada) e a adição da chave pública à sua conta do GitHub. A autenticação via SSH é geralmente mais segura e conveniente, pois elimina a necessidade de inserir o token ou senha repetidamente.

OBS.: Desde agosto de 2021, o GitHub não aceita mais autenticação via senha para operações Git que envolvem repositórios remotos. Portanto, é obrigatório o uso de tokens de acesso pessoal (PAT) ou autenticação via SSH para essas operações.

A opção dois é usando o GitHub-CLI, que facilita o processo de autenticação. A seguir, estão os passos para configurar a autenticação usando o GitHub-CLI.

4.3.1.1 Autenticação com o GitHub-CLI

Para autenticar-se no GitHub-CLI, execute o comando:

```
gh auth login
```

Siga as instruções na tela para concluir o processo de autenticação.

Figura 13 – Efetuando a autenticação com o GitHub-CLI - Passo 1.

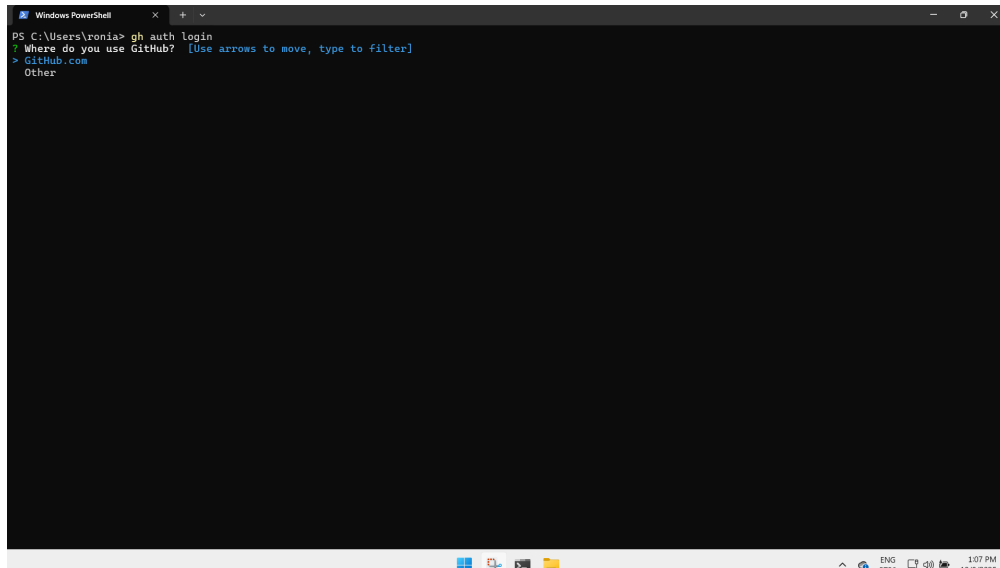


Figura 14 – Efetuando a autenticação com o GitHub-CLI - Passo 2.

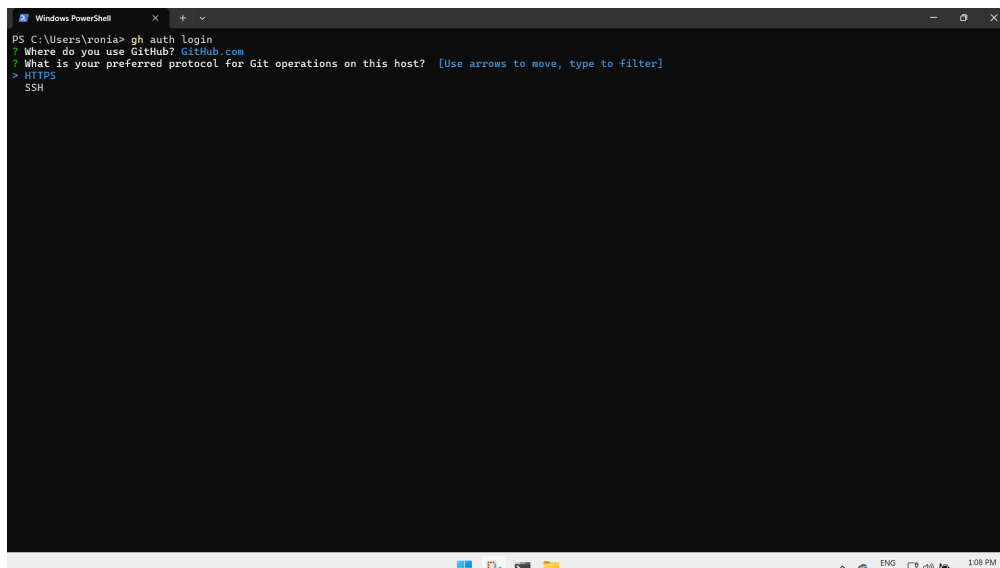


Figura 15 – Efetuando a autenticação com o GitHub-CLI - Passo 3.

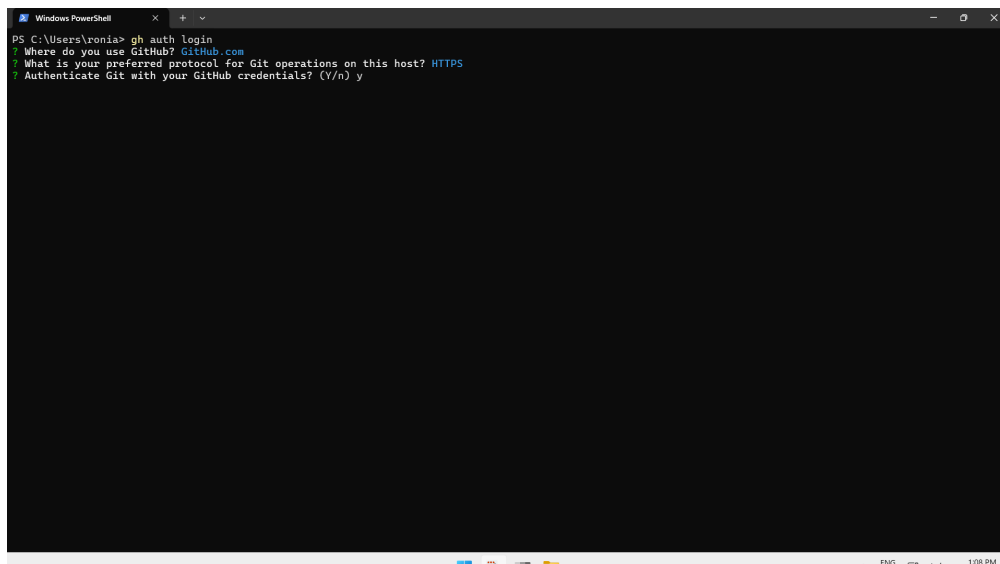
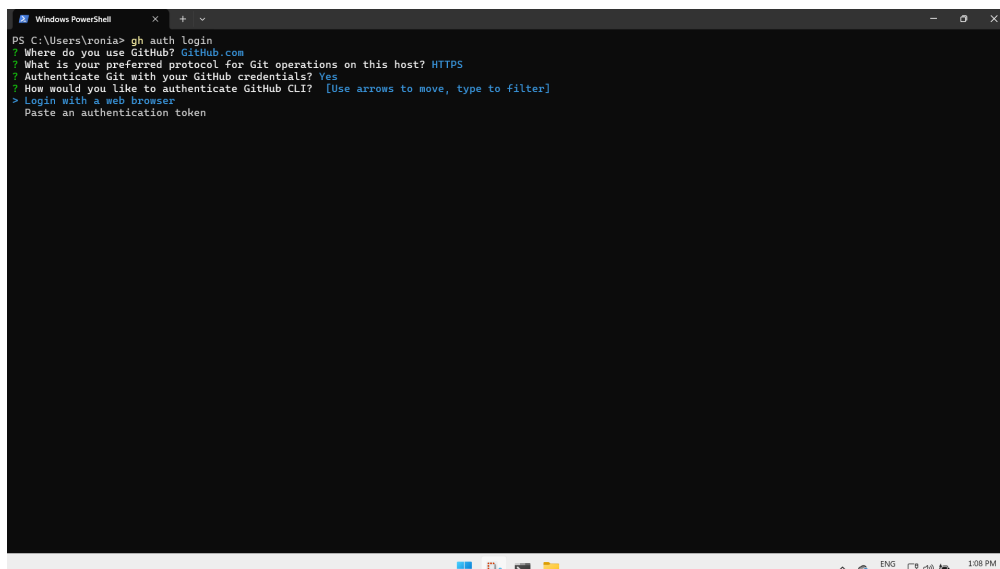
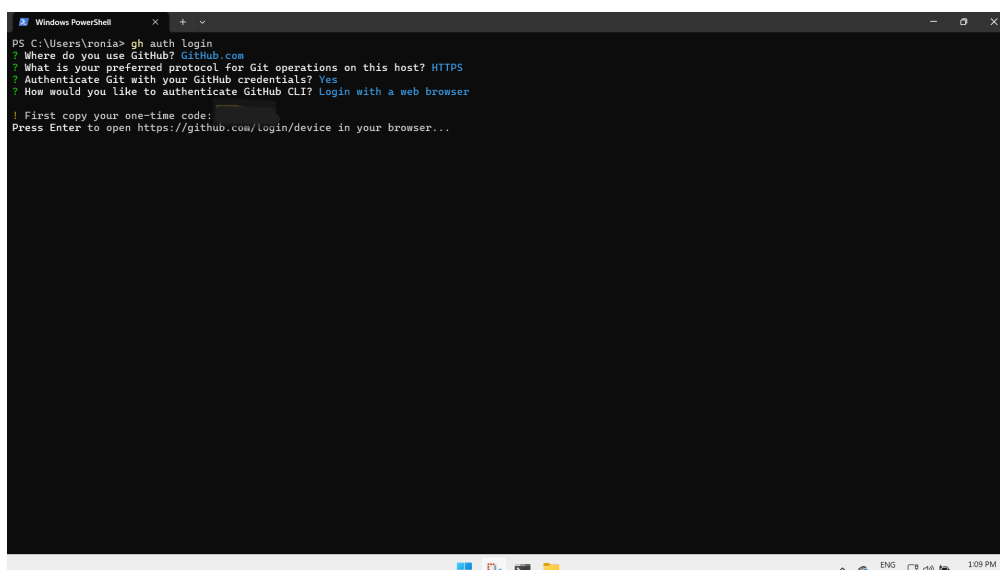


Figura 16 – Efetuando a autenticação com o GitHub-CLI - Passo 4.



```
PS C:\Users\ronia> gh auth login
? Where do you use GitHub? GitHub.com
? What is your preferred protocol for Git operations on this host? HTTPS
? Authenticate Git with your GitHub credentials? Yes
? How would you like to authenticate GitHub CLI? [Use arrows to move, type to filter]
> Login with a web browser
  Paste an authentication token
```

Figura 17 – Efetuando a autenticação com o GitHub-CLI - Passo 5.



```
PS C:\Users\ronia> gh auth login
? Where do you use GitHub? GitHub.com
? What is your preferred protocol for Git operations on this host? HTTPS
? Authenticate Git with your GitHub credentials? Yes
? How would you like to authenticate GitHub CLI? Login with a web browser
  First copy your one-time code:
  Press Enter to open https://github.com/login/device in your browser...
```

Após o passo 5 na Figura 17, p.28, será preciso copiar o código gerado abrir o navegador no link fornecido ou se precionar a tecla `Enter` abrirá o link automaticamente. Nessa tela será pedido que você se autentique e em seguida aparecerá o campo para digitar o código gerado e algumas permissões serão solicitadas.

4.3.2 Configuração de usuário Git

1. Abra o Prompt de Comando ou PowerShell.
2. Verifique a instalação do Git digitando:

```
git --version
```

O comando deve retornar a versão do Git instalada.

3. Configure seu nome de usuário com o comando:

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

Substitua "Seu Nome" pelo nome que você deseja associar aos seus commits e pressione Enter.

4. Verifique se o nome foi configurado corretamente com o comando:

```
git config --global user.name
```

Pressione Enter. O comando deve retornar o nome que você configurou.

5. Agora, configure seu e-mail com o comando:

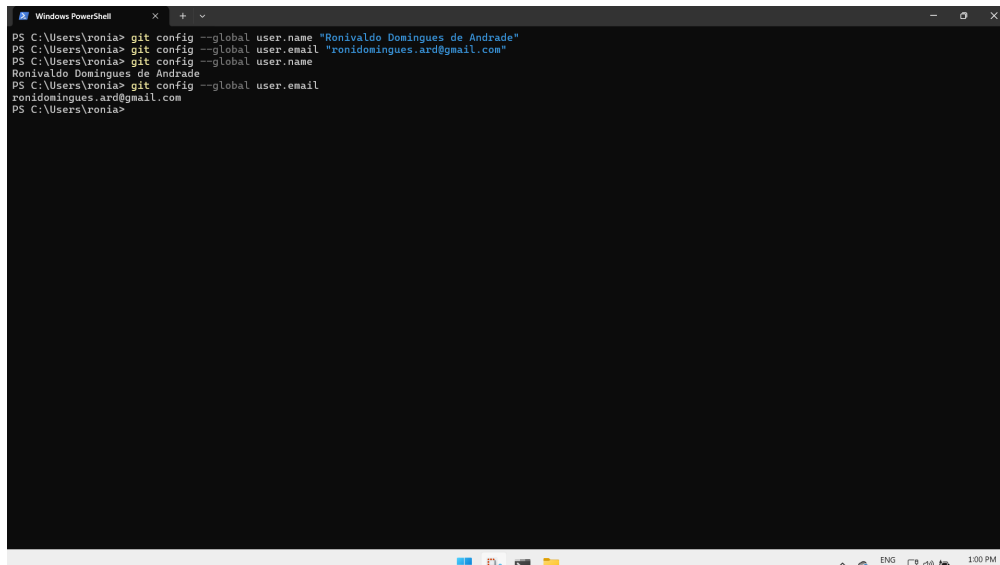
```
git config --global user.email "Seu E-mail"
```

Substitua "Seu E-mail" pelo e-mail que você deseja associar aos seus commits e pressione Enter.

6. Verifique se o e-mail foi configurado corretamente com o comando:

```
git config --global user.email
```

Pressione Enter. O comando deve retornar o e-mail que você configurou.



```
Windows PowerShell
PS C:\Users\ronia> git config --global user.name "Ronivaldo Domingues de Andrade"
PS C:\Users\ronia> git config --global user.email "ronidomingues.ard@gmail.com"
PS C:\Users\ronia> git config --global user.name
Ronivaldo Domingues de Andrade
PS C:\Users\ronia> git config --global user.email
ronidomingues.ard@gmail.com
PS C:\Users\ronia>
```

Figura 18 – Configurando o Git.

4.3.2.1 Autenticação usando PAT (Opcional)

O uso da autenticação com Personal Access Token (PAT) no GitHub oferece maior segurança em comparação à utilização de senhas tradicionais. Isso porque o token pode ser revogado a qualquer momento diretamente nas configurações da conta, além de possuir um prazo de validade configurável no momento da criação. Ademais, os PATs permitem definir escopos específicos de acesso, garantindo que o token tenha apenas as permissões necessárias para a operação desejada. Se você optar por usar um Token de Acesso Pessoal (PAT) para autenticação, siga os passos abaixo:

1. Acesse o link <https://github.com/settings/tokens> na sua conta do GitHub.
2. Clique em "Generate new token (classic)".
3. Marque escopos como:
 - repo (para acesso completo a repositórios privados e públicos)
 - workflows (para gerenciar e visualizar GitHub Actions)

4. Clique em "Generate token" na parte inferior da página e copie o token gerado (ele não será mostrado novamente!).
5. Agora, ao fazer um git push, quando o Git solicitar senha e usuário:
 - Use seu nome de usuário do GitHub como usuário.
 - Cole o token gerado como senha.
6. Você também pode salvar o token gerado no cache de credenciais do Git para evitar ter que digitá-lo toda vez que fizer push ou pull. Para isso, use o comando:

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

Com isso, na próxima vez que você digitar o token, ele será salvo no arquivo `/.git-credentials` e usado automaticamente nas próximas operações.

4.3.3 Configuração de Editor Padrão (Opcional)

Você pode configurar o editor de texto padrão que será usado para escrever mensagens de commit. Por exemplo, para configurar o Visual Studio Code como editor padrão, use o comando:

```
git config --global core.editor "code --wait"
```

Substitua `"code --wait"` pelo comando do editor de sua preferência.

4.3.4 Configurar a branch padrão para 'main' (Opcional)

Para configurar a branch padrão para 'main', use o comando:

```
git config --global init.defaultBranch main
```

Isso garantirá que novos repositórios criados localmente usem 'main' como a branch padrão. Caso não queira definir esse padrão é possível mudar a branch individualmente para cada repositório com o comando:

```
git branch -M main
```


Obs.: A partir de outubro de 2020, o GitHub alterou o nome da branch padrão de "master" para "main" em novos repositórios. Portanto, é recomendável usar "main" como a branch padrão para novos projetos.

4.4 Comandos Básicos do Git e GitHub-CLI

Aqui estão alguns comandos básicos do Git e do GitHub-CLI que você deve conhecer para começar a trabalhar com repositórios no GitHub.

4.4.1 Comandos Básicos do Git

- **git init**: Inicializa um novo repositório Git local.
- **git clone <url>**: Clona um repositório remoto para o seu computador.
- **git status**: Exibe o status atual do repositório, mostrando arquivos modificados, não rastreados e prontos para commit.
- **git add <arquivo>**: Adiciona um arquivo específico ao estágio para commit.
- **git add .**: Adiciona todos os arquivos modificados ao estágio para commit.
- **git commit -m "mensagem"**: Cria um commit com uma mensagem descritiva.
- **git push**: Envia os commits locais para o repositório remoto.
- **git pull**: Puxa as alterações do repositório remoto para o repositório local.
- **git branch**: Lista todas as branches no repositório.
- **git checkout <branch>**: Muda para a branch especificada.
- **git branch -M <branch>**: Renomeia a branch atual para o nome especificado forçando a sobrescrita se a branch especificada já existir.
- **git branch -m <branch>**: Renomeia a branch atual para o nome especificado e falha se a branch especificada já existir.
- **git checkout -b <branch>**: Cria uma nova branch e muda para ela.

- **git branch -d <branch>**: Deleta a branch especificada.
- **git remote rm origin**: Remove o repositório remoto chamado "origin".
- **git remote add origin <url>**: Adiciona um repositório remoto com o nome "origin".
- **git remote -v**: Exibe os repositórios remotos configurados.
- **git merge <branch>**: Mescla a branch especificada na branch atual.
- **Veja mais comandos do Git na tabela 2, p.75.**

4.4.2 Comandos Básicos do GitHub-CLI

- **gh auth login**: Autentica o usuário no GitHub-CLI.
- **gh repo create <nome-do-repositorio>**: Cria um novo repositório no GitHub.
- **gh repo clone <nome-do-repositorio>**: Clona um repositório do GitHub para o seu computador.
- **gh issue create**: Cria uma nova issue no repositório atual.
- **gh pr create**: Cria um novo pull request.
- **gh pr checkout <numero-do-pr>**: Faz checkout de um pull request específico.
- **gh pr merge <numero-do-pr>**: Mescla um pull request específico.
- **gh repo view**: Exibe informações sobre o repositório atual.
- **gh gist create <arquivo>**: Cria um novo gist com o arquivo especificado.
- **Veja mais comandos do GitHub-CLI na tabela 3, p.80.**

5 Git LFS

5.1 O que é?

O **Git LFS (Large File Storage)** é uma extensão oficial do Git projetada para o gerenciamento de **arquivos grandes ou binários** que não são tratados de forma eficiente pelo Git tradicional. Em vez de armazenar o conteúdo completo desses arquivos no histórico do repositório, o Git LFS substitui o arquivo original por um **ponteiro leve** — um pequeno arquivo de texto contendo informações sobre o objeto real, como seu identificador (*hash*) e tamanho. O conteúdo real é armazenado separadamente, em um **servidor LFS**, podendo estar hospedado no próprio provedor Git (como o GitHub, GitLab ou Bitbucket) ou em um servidor dedicado.

Essa estratégia mantém o repositório **mais leve e ágil**, reduzindo o tempo de *clone*, *checkout* e *fetch*, além de facilitar o versionamento de arquivos que mudam com frequência, como imagens, vídeos, áudios, modelos de aprendizado de máquina, arquivos de design (*.psd*), pacotes compactados (*.zip*), entre outros.

5.1.1 Motivos e Problemas que Resolve

Por padrão, o Git não é otimizado para lidar com arquivos grandes ou binários, pois ele foi projetado para versionar **texto**, como código-fonte. Existem várias limitações e problemas ao tentar versionar arquivos grandes diretamente:

- **Limite de tamanho:** serviços como o GitHub impõem limites de **100 MB por arquivo**, impedindo o envio de arquivos muito grandes via Git comum.
- **Histórico inflado:** cada nova versão de um arquivo binário é armazenada integralmente, sem compressão eficiente, o que faz o repositório crescer rapidamente.
- **Operações lentas:** com muitos arquivos grandes no histórico, operações como `git clone`, `git fetch` e `git checkout` tornam-se mais lentas.
- **Dificuldade de merge:** arquivos binários não podem ser mesclados (*merge*) facilmente, aumentando o risco de conflitos.

O **Git LFS** resolve esses problemas ao:

- Armazenar apenas **ponteiros leves** no repositório Git;
- Manter o conteúdo real em um **armazenamento separado**, acessível sob demanda;
- Permitir a **revogação ou substituição** de arquivos sem reescrever o histórico;
- Proporcionar uma experiência de versionamento **transparente**, pois os comandos Git (`add`, `commit`, `push`) continuam funcionando normalmente.

5.1.2 Como Funciona

O Git LFS utiliza um sistema de filtros configurados no Git:

- O filtro **clean** atua ao adicionar um arquivo rastreado pelo LFS, substituindo seu conteúdo real por um ponteiro antes de armazená-lo no repositório.
- O filtro **smudge** atua durante o `checkout` ou `clone`, baixando automaticamente o arquivo real do servidor LFS e substituindo o ponteiro pelo conteúdo original no diretório de trabalho.

O arquivo versionado no Git contém apenas algo como:

```
version https://git-lfs.github.com/spec/v1
oid sha256:3b6f1a8a...
size 1258291
```

Essas informações são suficientes para que o Git LFS localize e baixe o conteúdo correto quando necessário.

5.1.3 Vantagens

- Mantém o repositório leve e rápido;
- Suporta arquivos grandes (acima de 100 MB);
- Permite versionamento de arquivos binários;

- Integra-se com GitHub, GitLab e Bitbucket;
- Possibilita bloqueio de arquivos (*lock*) para evitar conflitos.

5.1.4 Limitações

- O armazenamento LFS pode ter **cotas e custos adicionais** em serviços remotos;
- Necessita de **instalação e configuração** local (`git lfs install`);
- Requer que o **servidor remoto suporte LFS**;
- Para repositórios antigos, pode ser necessário **migrar o histórico**.

5.1.5 Exemplo de Uso

```
# Instala o Git LFS no sistema
git lfs install

# Define tipos de arquivo que serão rastreados pelo LFS
git lfs track "*.zip"
git lfs track "*.psd"

# Adiciona e versiona normalmente
git add .gitattributes
git add arquivo.zip
git commit -m "Adiciona arquivo grande com Git LFS"
git push origin main
```

5.1.6 Boas Práticas

- Configure o Git LFS **antes de adicionar arquivos grandes**;
- Use o comando `git lfs track` para definir padrões de arquivos;
- Verifique o status com `git lfs status`;
- Evite rastrear arquivos pequenos em grande quantidade;

- Monitore o uso de armazenamento e transferências.

6 Commits, Merges e Pull Requests

6.1 Introdução

Commits, merges e pull requests são elementos centrais no fluxo de trabalho com Git e plataformas como GitHub. Um **commit** é uma unidade lógica de alteração no repositório; um **merge** integra mudanças de uma branch em outra; e um **pull request** (PR) é uma solicitação formal de revisão e integração de uma branch normalmente usada em colaboração para revisar código, executar checks automáticos (CI) e registrar a decisão de integrar.

6.2 Commits

6.2.1 O que é um commit

Um commit registra o estado do diretório de trabalho (os arquivos staged) em um nó do histórico do Git. Cada commit tem um identificador (SHA), metadata (autor, data) e uma mensagem que descreve a mudança.

6.2.2 Boas práticas de commits

- Faça commits atômicos: cada commit deve representar uma *única* mudança lógica.
- Mensagens claras: use assunto imperativo curto (~50 caracteres) + linha em branco + corpo explicativo se necessário (72 caracteres por linha).
- Inclua referência a issues quando relevante (ex.: “Closes #42”).
- Evite commitar arquivos gerados (binários, dependências) use `.gitignore`.
- Assine commits quando necessário: `git commit -S -m "..."` (GPG).

6.2.3 Fazendo commits passo a passo

1. Verificar o estado dos arquivos:

```
git status
```

2. Ver as mudanças não staged:

```
git diff          # diferenças não adicionadas ao stage
git diff --staged # diferenças preparadas para commit
```

3. Adicionar alterações ao stage (todo arquivo ou interativo):

```
git add caminho/arquivo.txt # adiciona arquivo específico
git add .                   # adiciona tudo (cuidado)
git add -p                  # adiciona parcialmente (patch)
```

4. Criar o commit com boa mensagem:

```
git commit -m "Assunto curto em imperativo"
# ou para mensagem longa (editor):
git commit
```

Exemplo de mensagem:

```
Corrige cálculo de juros
```

```
Ajusta a fórmula de cálculo para considerar juros compostos
quando o período é maior que 12 meses. Testes unitários
adicionados para cobrir casos de fronteira.
```


5. Ver histórico resumido:

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

6.2.4 Editar o último commit / corrigir mensagens

```
# alterar o conteúdo do último commit (já staged)
git commit --amend
```

```
# alterar apenas a mensagem do último commit
git commit --amend -m "Nova mensagem"
```

Atenção: se o commit já foi enviado ao remoto, evite `--amend` sem combinar com a equipe ele reescreve o histórico e exigirá push forçado.

6.2.5 Desfazer / alterar staging

<pre>git restore --staged arquivo.txt</pre>	<pre># remove do stage, mantém # alteração no working tree</pre>
<pre>git restore arquivo.txt</pre>	<pre># descarta alteração no # working tree (se não commitada)</pre>
<pre>git reset --soft HEAD~1</pre>	<pre># desfaz último commit, mantendo # mudanças staged</pre>
<pre>git reset --mixed HEAD~1</pre>	<pre># desfaz último commit, mantendo # mudanças no working tree # (unstaged)</pre>
<pre>git reset --hard HEAD~1</pre>	<pre># desfaz último commit e # descarta mudanças (CUIDADO)</pre>

6.2.6 Padrões de Commits

Manter um padrão consistente nas mensagens de commit é fundamental para garantir um histórico de versões claro, fácil de compreender e rastrear. Um bom padrão de commit permite que outros desenvolvedores entendam rapidamente **o que foi alterado, por que foi alterado e qual impacto a mudança traz**.

Existem diferentes convenções adotadas por equipes e comunidades. Veja a tabela 4 na página 90 para ver padrões populares.

Acesse também as referencias:

- [Adorno 2021](#);
- [Iuricode 2023](#);

Boas práticas ao escrever commits

- Use o modo **imperativo** (ex.: “adiciona“, “corrige“, “remove“);
- Mantenha a linha de assunto com no máximo **50 caracteres**;
- Separe título e corpo com uma **linha em branco**;
- Descreva **o motivo da mudança** no corpo, não apenas o que foi alterado;
- Use **referências a issues** quando aplicável (ex.: “Closes #123“);
- Escreva mensagens em português ou inglês, mas mantenha um idioma único no projeto.

Exemplo de commit completo

```
feat(api): adiciona endpoint para criação de pedidos
```

Adiciona o endpoint POST /orders para permitir o cadastro de novos pedidos. Inclui validação de campos obrigatórios e testes unitários. Closes #42.

Vantagens de seguir um padrão

- Histórico limpo e fácil de entender;
- Facilita revisão de código e auditorias;
- Permite geração automática de changelogs;
- Ajuda em pipelines de CI/CD e versionamento semântico;
- Melhora colaboração em equipes e projetos open source.

6.3 Merges

6.3.1 Tipos de merge

Fast-forward Se a branch destino não avançou desde que a feature foi criada, o Git apenas avança o ponteiro (sem novo commit de merge).

Merge commit Cria um commit de merge que documenta a união de dois históricos (útil para preservar contexto de branch).

Rebase Reaplica commits de uma branch sobre outra, produzindo um histórico linear (reduz “ruído” dos merges, mas reescreve histórico).

6.3.2 Merge local com merge commit (passo a passo)

1. Atualize a branch principal:

```
git checkout main  
git pull origin main
```

2. Mudar para a branch de feature (se necessário):

```
git checkout feature/minha-feature  
git pull origin feature/minha-feature
```

3. Voltar para a branch de destino e fazer merge:

```
git checkout main
git merge --no-ff feature/minha-feature
# --no-ff força um commit de merge,
# preservando histórico da branch
```

4. Caso não haja conflitos, o Git criará o commit de merge automaticamente. Em caso de conflitos, siga a seção "Resolver conflitos" abaixo.

5. Envie as mudanças para o remoto:

```
git push origin main
```

6.3.3 Rebase (passo a passo) para um histórico linear

1. Atualize a base:

```
git checkout main
git pull origin main
```

2. Rebase da feature sobre a main:

```
git checkout feature/minha-feature
git rebase main
```

3. Resolva conflitos (se aparecerem), use `git rebase -continue` e, ao final:

```
# force push (com cuidado) após reescrever histórico
git push --force-with-lease origin feature/minha-feature
```

4. **Observação:** reescrever histórico (rebase + push forçado) requer coordenação com outros colaboradores.

6.3.4 Resolver conflitos passo a passo

1. Ao encontrar conflito durante merge/rebase, o Git mostra arquivos conflitantes:

```
CONFLICT (content): Merge conflict in caminho/arquivo.txt
```

2. Abra o arquivo e localize os marcadores:

```
<<<<<< HEAD
conteúdo na branch atual (main)
=====
conteúdo vindo da branch feature/minha-feature
>>>>>> feature/minha-feature
```

3. Edite o arquivo para a versão desejada (manter, combinar ou reescrever o trecho).
4. Marque o conflito como resolvido:

```
git add caminho/arquivo.txt
# se for rebase:
git rebase --continue
# se for merge:
git commit # caso o Git não tenha criado o commit automaticamente
```

5. Se quiser abortar a operação:

```
git merge --abort # durante um merge
git rebase --abort # durante um rebase
```

6.3.5 Squash e reescrita de commits (passo a passo)

1. Iterative rebase para combinar commits locais:

```
git checkout feature/minha-feature  
git rebase -i main
```

2. No editor que abre, marque squash (ou s) nos commits que deseja unir ao commit anterior. Salve e feche.

3. Após o rebase, force push com segurança:

```
git push --force-with-lease origin feature/minha-feature
```

4. Use `-force-with-lease` em vez de `-force` quando possível - ele evita sobrescrever pushes alheios.

6.4 Pull Requests (PR)

6.4.1 O que é um Pull Request

Um PR é uma solicitação para integrar as mudanças de uma branch em outra (normalmente de uma branch de feature para `main` ou `develop`) e serve como ponto central para revisão de código, execução de pipelines de CI e documentação da mudança.

6.4.2 Fluxo básico criando um PR (via web)

1. Crie uma branch local para a sua feature:

```
git checkout -b feature/minha-feature
```

2. Faça commits locais e envie a branch para o remoto:

```
git push -u origin feature/minha-feature
```

3. No repositório do GitHub, acesse `Pull requests` → `New pull request`.
4. Selecione a **branch de origem** (feature/minha-feature) e a **branch de destino** (ex.: main).
5. Preencha o título e a descrição: explique o **porquê** e o **o que** foi alterado. Use referências a issues (ex.: “Closes #123”).
6. Configure revisores, labels, milestone e assignees.
7. Se ainda não está pronta, crie como **Draft pull request** (rascunho).
8. Aguarde revisão, corrija comentários fazendo novos commits na mesma branch e push eles serão anexados automaticamente ao PR.
9. Quando aprovado, escolha a estratégia de merge (merge commit / squash and merge / rebase and merge) e realize o merge.

6.4.3 Criar e gerenciar PRs via GitHub CLI (passo a passo)

```
# autentique (uma vez)
```

```
gh auth login
```

```
# depois de push da branch
```

```
gh pr create --base main --head feature/minha-feature --title "Título do PR"
```

```
# abrir PR como draft:
```

```
gh pr create --draft --base main --head feature/minha-feature --fill
```

```
# listar PRs locais:
```

```
gh pr list
```

```
# fechar ou mesclar via CLI:
```

```
gh pr merge <numero-ou-url> --squash --delete-branch
# ou
gh pr merge <numero> --merge          # cria commit de merge
gh pr merge <numero> --rebase         # rebase and merge
```

6.4.4 Checklist para revisão de Pull Request

- O PR tem um título e descrição claros (o *porquê* e o *o que*);
- Testes automatizados adicionados/atualizados e pipeline CI passando;
- Código segue padrões de lint e estilo;
- Mudanças pequenas e focadas (um PR por responsabilidade);
- Documentação e comentários quando necessário;
- Evidências visuais (screenshots) quando há alterações de UI.

6.4.5 Depois do merge limpeza e sincronização

```
# atualizar a branch principal local
git checkout main
git pull origin main
```

```
# remover branch remota (após merge)
git push origin --delete feature/minha-feature
```

```
# remover branch local
git branch -d feature/minha-feature
```

```
# caso a branch não possa ser deletada por não estar totalmente mesclada
git branch -D feature/minha-feature
```

Também é útil rodar:

```
git fetch --prune
```


para remover referências remotas deletadas.

6.5 Boas práticas e recomendações finais

- Mantenha PRs pequenos e revisáveis; grandes PRs demoram mais para receber feedback.
- Automatize checks com CI (testes, lint, análise estática) e impeça merge enquanto falharem (branch protection).
- Use `-force-with-lease` quando precisar reescrever histórico; nunca force sem checar se colegas não empurraram commits.
- Documente o fluxo do time (merge strategy preferida ex.: squash para commits limpos, merge commit para histórico preservado).
- Escreva mensagens de commit úteis e mantenha um padrão no time (ex.: Conventional Commits) para facilitar geração automática de changelogs.

6.6 Exemplos rápidos de comandos úteis

```
# ver status e diferenças
```

```
git status
```

```
git diff
```

```
git diff --staged
```

```
# histórico e log
```

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

```
# adicionar parcialmente
```

```
git add -p
```

```
# rebase interativo (últimos 3 commits)
```

```
git rebase -i HEAD~3
```

```
# desfazer mudanças locais (trabalhe com cuidado)
git restore arquivo.txt
git reset --hard HEAD
```

```
# forçar push com segurança
git push --force-with-lease origin minha-branch
```

7 GitHub Pages e GitHub Actions

7.1 O que é GitHub Pages?

GitHub Pages é uma ferramenta gratuita oferecida pelo GitHub para hospedagem de sites estáticos diretamente a partir de um repositório. Essa funcionalidade permite que desenvolvedores publiquem páginas web com tecnologias como **HTML**, **CSS** e **JavaScript**, sem a necessidade de servidores adicionais ou configuração complexa.

Entre as principais características do GitHub Pages, podemos destacar:

- **Hospedagem gratuita:** todo repositório público pode gerar uma página web sem custos.
- **Suporte a domínios personalizados:** é possível usar seu próprio domínio, além do subdomínio padrão do GitHub (`username.github.io`).
- **Atualização automática:** sempre que você faz um *push* no repositório, o site é atualizado automaticamente.
- **Compatibilidade com geradores de site estático:** ferramentas como *Jekyll*, *Hugo* e *Eleventy* podem ser integradas facilmente.

Exemplo de uso: um repositório com um arquivo `index.html` na branch `main` pode ser publicado acessando o GitHub Pages nas configurações do repositório, sem qualquer configuração adicional.

7.2 O que é GitHub Actions?

O GitHub Actions é uma plataforma de automação que permite criar fluxos de trabalho (*workflows*) para automatizar tarefas de desenvolvimento. Ele funciona diretamente no GitHub, sem necessidade de servidores externos, e pode ser usado para:

- **Testes automatizados:** executar testes sempre que houver alterações no código.

- **Compilação e *build* de projetos:** gerar executáveis, bibliotecas ou pacotes para diferentes plataformas.
- **Publicação automática:** enviar versões de aplicativos ou páginas web para ambientes de produção.
- **Integração contínua (CI) e entrega contínua (CD):** garantir que o código enviado para o repositório esteja sempre funcional.

Estrutura de um workflow: Um workflow é definido por arquivos YAML dentro da pasta `.github/workflows/` do repositório e consiste basicamente em:

- `name`: nome do workflow.
- `on`: eventos que disparam o workflow, como `push`, `pull_request`, etc.
- `jobs`: conjunto de tarefas a serem executadas.
- `steps`: etapas dentro de cada job, que podem incluir instalação de dependências, execução de scripts, testes, builds e deploy.

7.3 Integração entre GitHub Pages e GitHub Actions

A integração entre GitHub Pages e GitHub Actions permite automatizar a publicação de sites sempre que o código for atualizado. Esse processo envolve:

1. Configuração do repositório para hospedar o site na branch `gh-pages` ou na pasta `/docs`.
2. Criação de um workflow no GitHub Actions, geralmente disparado pelo evento `push` na branch principal (`main`).
3. Etapas do workflow típicas:
 - **Instalação de dependências:** por exemplo, instalar Node.js e pacotes necessários.
 - **Compilação do site:** gerar os arquivos finais (HTML, CSS, JS).

- **Publicação:** enviar os arquivos para a branch ou pasta configurada para GitHub Pages.

4. Verificação: após o deploy, o site é atualizado automaticamente, podendo ser acessado pelo domínio configurado.

Exemplo de arquivo de workflow simples (deploy.yml):

```
name: Deploy GitHub Pages

on:
  push:
    branches:
      - main

jobs:
  build:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v3
      - name: Setup Node.js
        uses: actions/setup-node@v3
        with:
          node-version: '18'
      - name: Install dependencies
        run: npm install
      - name: Build site
        run: npm run build
      - name: Deploy to GitHub Pages
        uses: peaceiris/actions-gh-pages@v3
        with:
          github_token: ${ secrets.GITHUB_TOKEN }
          publish_dir: ./dist
```

Benefícios da integração:

- Atualização automática do site sem intervenção manual.
- Garantia de que apenas código validado e testado seja publicado.
- Possibilidade de incluir etapas adicionais, como otimização de imagens, minificação de CSS/JS e execução de testes automatizados antes do deploy.

8 Assinaturas de Commits com chave GPG

8.1 O que é?

A assinatura de commits com **GPG (GNU Privacy Guard)** é um recurso do Git que permite garantir a autenticidade e a integridade das alterações feitas em um repositório. Quando um commit é assinado, outras pessoas podem verificar que aquele commit foi realmente feito por você, evitando alterações fraudulentas ou commits não autorizados.

8.1.1 Importância das assinaturas GPG

- **Segurança:** Confirma que o commit foi feito pelo autor legítimo.
- **Integridade:** Permite verificar se o commit não foi alterado após sua criação.
- **Transparência:** Em projetos open source, facilita identificar contribuições confiáveis.

8.2 Como usar?

8.2.1 Passo 1: Instalar o GPG

Antes de assinar commits, você precisa instalar o GPG no seu sistema.

- **Linux/Debian:**

```
sudo apt update  
sudo apt install gnupg
```

- **Windows:** Instale o Gpg4win (<<https://www.gpg4win.org/>>)
- **MacOS:**

```
brew install gnupg
```

8.2.2 Passo 2: Gerar uma chave GPG

Para criar uma chave GPG pessoal, use o comando:

```
gpg --full-generate-key
```

O terminal fará algumas perguntas:

1. Tipo de chave: selecione `RSA and RSA (default)`.
2. Tamanho da chave: recomendo `4096 bits` para maior segurança.
3. Validade da chave: escolha o período de validade ou `0` para sem expiração.
4. Nome e e-mail: use o mesmo e-mail configurado no Git (`git config user.email`).
5. Senha: defina uma senha segura para proteger sua chave.

8.2.3 Passo 3: Listar chaves e copiar o ID da chave

Para ver as chaves criadas:

```
gpg --list-secret-keys --keyid-format LONG
```

O resultado terá um formato como:

```
sec    rsa4096/ABCDEF1234567890 2025-01-01 [SC]
        Key fingerprint = 1234 5678 9ABC DEF0 1234
        5678 9ABC DEF0 1234 5670
uid                               Seu Nome <seuemail@example.com>
```

O que você precisa é do **ID da chave**, que no exemplo acima é `ABCDEF1234567890`.

8.2.4 Passo 4: Configurar o Git para usar a chave GPG

Diga ao Git qual chave usar para assinar commits:

```
git config --global user.signingkey ABCDEF1234567890
git config --global commit.gpgsign true
```

8.2.5 Passo 5: Adicionar a chave GPG ao GitHub

Para que o GitHub reconheça seus commits assinados:

1. Copie a chave pública:

```
gpg --armor --export ABCDEF1234567890
```

2. Entre no GitHub: Settings > SSH and GPG keys > New GPG key
3. Cole a chave pública e salve.

8.2.6 Passo 6: Fazer commits assinados

A partir de agora, todos os commits serão assinados automaticamente. Você também pode assinar commits individualmente:

```
git commit -S -m "Mensagem do commit"
```

No GitHub, os commits assinados aparecerão com a etiqueta **Verified**.

8.2.7 Passo 7: Verificar commits assinados

Para verificar um commit localmente, use:

```
git log --show-signature
```

O Git mostrará se o commit foi assinado corretamente e qual chave foi usada.

8.3 Dicas de segurança e boas práticas

- Proteja sua chave GPG com uma senha forte.
- Faça backup da chave privada em um local seguro.
- Não compartilhe sua chave privada.
- Rotacione suas chaves periodicamente, se necessário.

9 Exercícios Práticos

Observação: Todas as atividades devem seguir as boas práticas de commits, merges e pull requests.

9.1 Git e GitHub-CLI

9.1.1 Objetivo

Verificar se as ferramentas estão instaladas corretamente.

9.1.2 Passo a Passo Detalhado

1. **Abrir o terminal** (Prompt de Comando no Windows, Terminal no Mac/Linux)
2. **Verificar se o Git está instalado:**

```
git --version
```

Resultado esperado: Deve aparecer algo como `git version 2.xx.x`

3. **Verificar se o GitHub CLI está instalado:**

```
gh --version
```

Resultado esperado: Deve aparecer algo como `gh version 2.xx.x`

9.1.3 Problemas Comuns e Soluções

- Se algum comando não for reconhecido, reinstale a ferramenta
- No Windows, talvez seja necessário reiniciar o computador após a instalação

9.2 Criar um repositório no GitHub via CLI

9.2.1 Objetivo

Criar um repositório público com descrição.

9.2.2 Pré-requisito

Fazer login no GitHub CLI:

```
gh auth login
```

9.2.3 Passo a Passo

1. Criar o repositório:

```
gh repo create meu-primeiro-repo --public  
--description "Meu primeiro repositório" --clone
```

2. Entrar na pasta do repositório:

```
cd meu-primeiro-repo
```

9.2.4 README.md

9.2.4.1 O que é README.md

- É a "cara" do seu projeto no GitHub
- Explica o que seu projeto faz, como usar, etc.
- Usa uma linguagem chamada Markdown (por isso o .md)

9.2.4.2 Como Criar

1. Criar o arquivo:

```
echo "# Meu Primeiro Projeto" >> README.md
```

2. Adicionar conteúdo:

```
# Meu Primeiro Projeto
```

```
Este é meu primeiro repositório no GitHub!
```

```
## O que este projeto faz?
```

- Aprender Git e GitHub
- Praticar comandos
- Compartilhar conhecimento

```
## Como usar?
```

1. Clone este repositório
2. Siga as instruções
3. Aprenda!

3. Salvar e enviar para o GitHub:

```
git add README.md  
git commit -m "docs(readme): Adiciona README com  
a descrição do projeto"  
git push origin main
```

Em `git push origin main`, pode ser colocado a flag `-u`, ou seja, `git push -u origin main` isso precisará ser feito apenas uma vez os próximos pushes poderão apenas se fazer com `git push`. A flag `-u` faz com que o git se torne um colaborador do repositório, ou seja, ele não precisa mais digitar `origin` e `main`, ele já sabe que é o repositório principal.

9.2.5 LICENSE

9.2.5.1 Por que usar LICENSE

- Define como outras pessoas podem usar seu código
- Protege seus direitos autorais
- Torna seu projeto mais profissional

9.2.5.2 Como Adicionar Licença MIT

1. Criar arquivo LICENSE:

```
touch LICENSE
```

2. Adicionar conteúdo da licença MIT:

```
1 MIT License
2
3 Copyright (c) [ano] [seu nome]
4
5 Permission is hereby granted, free of charge, to any person
6 obtaining a copy of this software and associated
7 documentation
8 files (the "Software"), to deal in the Software without
9 restriction, including without limitation the rights to use,
10 copy, modify, merge, publish, distribute, sublicense, and/or
11 sell copies of the Software, and to permit persons to whom
12 the Software is furnished to do so, subject to the following
13 conditions:
```

```
14 The above copyright notice and this permission notice shall
15 be included in all copies or substantial portions of the
    Software.
16
17 THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY
    KIND,
18 EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE
    WARRANTIES
19 OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND
20 NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT
21 HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY,
22 WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING
23 FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE
    OR
24 OTHER DEALINGS IN THE SOFTWARE.
```

Listing 9.1 – Licença MIT

3. Substituir [ano] e [seu nome] pelos seus dados

4. Salvar e enviar:

```
git add LICENSE
git commit -m "chore(licensing):Adiciona licença MIT"
git push origin main
```

9.3 Clonando um repositório do GitHub

9.3.1 Objetivo

Aprender a baixar repositórios existentes.

9.3.2 Passo a Passo

1. Repositório alvo: <https://github.com/ronidomingues/github-capacitation>
2. Copiar a URL do repositório

3. No terminal, clonar:

```
git clone https://github.com/ronidomingues/  
github-capacitation.git
```

4. Entrar na pasta criada:

```
cd github-capacitation
```

5. Verificar o conteúdo:

```
ls -la
```

9.4 Github Pages

9.4.1 Objetivo

Publicar um site gratuitamente.

9.4.2 Passo a Passo

1. Criar novo repositório:

```
gh repo create meu-site --public  
--description "Meu primeiro site" --clone  
cd meu-site
```

2. Copiar os arquivos do jogo (HTML, CSS, JS) para a pasta do repositório

3. Verificar estrutura:


```
ls -la
```

4. Adicionar, commitar e enviar:

```
git add .  
git commit -m "feat(jogo): adiciona arquivos do jogo"  
git push origin main
```

5. Ativar GitHub Pages:

- No GitHub, vá em **Settings Pages**
- Em **Source**, selecione **main branch**
- Clique **Save**

6. Acessar seu site:

- URL será: `https://seu-usuario.github.io/meu-site`

9.5 Github Actions

9.5.1 Objetivo

Automatizar execução de código Python.

9.5.2 Passo a Passo

1. Criar repositório para o código Python:

```
gh repo create meu-script-python --public  
--description "Script Python com GitHub Actions" --clone  
cd meu-script-python
```

2. Copiar o arquivo Python fornecido para o repositório

3. Criar pasta para workflows:

```
mkdir -p .github/workflows
```

4. Criar arquivo de workflow:

```
touch .github/workflows/python.yml
```

5. Adicionar conteúdo ao workflow - Preencher o que falta:

Há um modelo com `fortran 90`, disponível na pasta `materials`.

```
name: "Executar script Python e Commitar resultado"

on:
  push:
    branches: [ main ]
  workflow_dispatch:

jobs:
  build-run:
    runs-on: ubuntu-latest
    steps:
      # 1 Faz o clone do repositório para a VM Ubuntu;
      # 2 Configura o Python a ser usado pela VM;
      -name: Instalar Python3
      uses: actions/setup-python@v5
      with:
        python-version: '3.x'
      # 3 Executa o script Python;
      # 4 Cria um commit com o resultado;
      -name: Commitar PDFs gerados
      run: |
```

```
git config user.name "github-actions[bot]"
git config user.email "github-actions
[bot]@users.noreply.github.com"
git add materials/*.pdf
git commit -m "Atualizar PDFs compilados
automaticamente [skip ci]" || echo "Nenhuma
alteração para commitar"
git push
env:
  GITHUB_TOKEN: ${ secrets.GITHUB_TOKEN }
```

9.5.3 Uso de [skip ci] no GitHub Actions

No GitHub Actions, um workflow normalmente é disparado por eventos como:

```
on:
  push:
    branches:
      - main
```

Ou seja, **cada git push** aciona o workflow.

Quando o próprio workflow realiza um commit e push automaticamente (por exemplo, atualizando arquivos gerados ou listas), isso poderia disparar o workflow novamente, criando um **loop infinito**.

Para evitar esse problema, é possível incluir no commit uma anotação especial:

```
git commit -m "Atualiza lista automática [skip ci]"
```

O código [skip ci] instrui o GitHub Actions (e outros sistemas de CI, como GitLab CI ou Travis CI) a **ignorar este commit**, ou seja, não disparar nenhum workflow.

Dessa forma, o workflow pode atualizar arquivos ou fazer commits automaticamente sem reiniciar seu próprio processo indefinidamente.

Observação: Além de `[skip ci]`, também é possível usar `[ci skip]`, que possui a mesma função.

6. Adicionar, commitar e enviar tudo:

```
git add .
git commit -m "feat:Adiciona script Python e GitHub Actions"
git push origin main
```

7. Verificar execução:

- No GitHub, vá em **Actions** para ver o workflow rodando

9.6 Merge

9.6.1 Objetivo

Aprender a juntar alterações de diferentes origens.

9.6.2 Passo a Passo

1. Fazer alteração REMOTA:

- No GitHub, edite o README.md online
- Adicione uma linha no final
- Commit a alteração

2. Fazer alteração LOCAL:

```
# No seu computador, no mesmo repositório
echo "Alteração local" >> arquivo-local.txt
git add arquivo-local.txt
```

```
git commit -m "Adiciona arquivo local"
```

3. Tentar enviar alteração local:

```
git push origin main  
# VAI DAR ERRO! Porque tem alteração remota  
# que você não tem localmente
```

4. Fazer merge:

```
git pull origin main  
# Isso baixa as alterações remotas e faz merge  
# com suas alterações locais
```

5. Resolver conflitos (se houver):

- Se Git não conseguir juntar automaticamente, ele pedirá para resolver manualmente
- Abra os arquivos com conflitos, resolva e depois:

```
git add .  
git commit -m "Resolve conflitos de merge"  
git push origin main
```

9.7 Pull Request

9.7.1 Objetivo

Contribuir para projetos de outras pessoas.

9.7.2 Passo a Passo

1. Fork do repositório original:

- No GitHub, vá para o repositório <<https://github.com/ronidomingues/github-capacitation>>
- Clique em **Fork** (canto superior direito)
- Isso cria uma cópia em sua conta

2. Clonar SEU fork:

```
git clone https://github.com/seu-usuario/  
repositorio-forkado.git  
cd repositorio-forkado
```

3. Criar branch para sua feature:

```
git checkout -b minha-feature
```

4. Fazer suas alterações:

Entre na pasta `presences` e adicione um arquivo `.txt` com o seu nome, por exemplo `roni.txt`, esse arquivo não precisa ter nenhum conteúdo, mas se quiser deixar uma avaliação de tudo até aqui será ótimo ; –).

```
echo "Minha avaliacao" >> presences/meu-nome.txt
```

5. Commit e push:

```
git add .  
git commit -m "<tipo>(escopo): <descrição>"  
git push origin minha-feature
```

6. Criar Pull Request:

- No GitHub, vá para SEU fork
- Clique em **Pull Request** **New Pull Request**
- Selecione: base (repositório original) compare (sua branch)
- Descreva suas alterações
- Clique **Create Pull Request**

9.8 Materiais de Apoio

9.8.1 Checklist para Cada Exercício

- ☐ Comandos executados sem erro
- ☐ Arquivos criados corretamente
- ☐ Commits com mensagens descritivas
- ☐ Push realizado com sucesso
- ☐ Resultado verificado no GitHub

9.8.2 Comandos Úteis para Consulta

```
# Status do repositório  
git status
```

```
# Ver histórico de commits  
git log --oneline
```

```
# Ver diferenças  
git diff
```

```
# Ver configuração  
git config --list
```

9.8.3 Dicas para Boas Práticas

- Commits frequentes e pequenos
- Mensagens de commit claras e descritivas
- Sempre fazer pull antes de push
- Testar localmente antes de enviar
- Revisar código antes de criar PR

10 Conclusão

Ao longo desta capacitação, foram abordados os principais conceitos e ferramentas que compõem o ecossistema do GitHub, desde a criação e configuração de repositórios até a realização de operações complexas como merges, rebases, pull requests e a automação de pipelines com GitHub Actions.

O domínio dessas habilidades não apenas facilita a colaboração em projetos de software, mas também promove a adoção de boas práticas de desenvolvimento, como commits semânticos, revisão de código e integração contínua. A utilização de recursos como Git LFS para arquivos grandes e a assinatura de commits com chaves GPG reforça a segurança e a integridade do versionamento.

Por fim, a realização dos exercícios práticos propostos consolida o aprendizado e prepara o participante para atuar em ambientes reais, contribuindo de forma eficiente e profissional em projetos individuais e em equipe. Espera-se que este material sirva como referência contínua e incentive a adoção de um fluxo de trabalho organizado, colaborativo e alinhado com as melhores práticas do mercado.

Referências

Adorno 2021 ADORNO, R. *Padrões de Commits (Commit Patterns)*. 2021. <<https://dev.to/renatoadorno/padroes-de-commits-commit-patterns-41co>>. Acesso em: 8 out. 2025.

Iuricode 2023 IURICODE. *Padrões de Commits*. 2023. <<https://github.com/iuricode/padroes-de-commits>>. Repositório GitHub. Acesso em: 8 out. 2025.

Microsoft 2025 MICROSOFT. *Gerenciador de Pacotes do Windows (winget)*. 2025. <<https://learn.microsoft.com/pt-br/windows/package-manager/winget/>>. Acesso em: 8 out. 2025.

Apêndices

APÊNDICE A – Comandos Git

Este apêndice apresenta uma referência completa dos principais comandos Git organizados por categoria e funcionalidade.

Comandos Git

Tabela 2 – Comandos do Git

Comando	Categoria	Explicação Detalhada
git init	Configuração/Setup	Transforma o diretório atual em um repositório Git, criando o diretório.git. Pode ser executado com segurança em um diretório existente sem sobrescrever configurações.
git config	Configuração/Setup	Lê ou define variáveis de configuração em nível de sistema, global ou local. Essencial para definir a identidade (user.name, user.email) do autor do commit.
git clone [url]	Configuração/Setup	Cria uma cópia local de um repositório remoto. Configura automaticamente a referência 'origin' e faz o checkout da branch principal.
git add [file]	Snapshotting Básico	Move alterações de um arquivo do Working Tree para o Index (Staging Area), preparando-o para o próximo commit.
git status	Snapshotting Básico	Exibe o estado da Working Tree e do Index, listando arquivos modificados, staged ou não rastreados.
Continua na próxima página		

Continuação da Tabela: Comandos do Git

Comando	Categoria	Explicação Detalhada
git diff	Snapshotting Básico	Mostra as diferenças entre o Working Tree e o Index (alterações não staged).
git diff --staged	Snapshotting Básico	Mostra as diferenças entre o Index (Staging Area) e o último commit (HEAD).
git commit -m "[msg]"	Snapshotting Básico	Salva o conteúdo atualmente no Index como um novo snapshot permanente (commit) na história.
git commit --amend	Manipulação Histórico	Altera o commit anterior, seja modificando sua mensagem ou adicionando/removendo arquivos. Isso reescreve o histórico, gerando um novo SHA.
git rm [file]	Gerenciamento Arquivo	Remove um arquivo do Working Tree e do Index. O uso de --cached remove apenas do Index, mantendo o arquivo local.
git mv [old][new]	Gerenciamento Arquivo	Move ou renomeia um arquivo de forma rastreada pelo Git.
git clean	Gerenciamento Arquivo	Remove arquivos não rastreados (untracked files) do Working Tree.
git reset --soft [hash]	Manipulação Histórico	Move o ponteiro HEAD para o commit, mas mantém o Index e o Working Tree intactos (alterações permanecem staged).
git reset --mixed [hash]	Manipulação Histórico	(Padrão) Move o HEAD para o commit e reseta o Index (desencena arquivos), preservando o Working Tree.
git reset --hard [hash]	Manipulação Histórico	Move o HEAD e reseta o Index e o Working Tree, descartando todas as mudanças locais desde o hash. Altamente destrutivo.
Continua na próxima página		

Continuação da Tabela: Comandos do Git

Comando	Categoria	Explicação Detalhada
git branch	Branching/Navegação	Gerenciamento de branches: lista, cria ou deleta branches locais.
git checkout	Branching/Navegação	Comando legado multi-uso. Alterna entre branches ou restaura arquivos antigos/-commits, podendo resultar em 'detached HEAD'.
git switch	Branching/Navegação	Comando moderno focado em alternar branches. Atualiza a Working Tree e o Index. Utilizado para criar novas branches de forma segura.
git merge [branch]	Integração/Merge	Integra alterações de uma branch na atual, criando um 'merge commit' se houver divergência. Operação não-destrutiva.
git rebase [base]	Integração/Rebase	Move ou reaplica commits para uma nova base, reescrevendo o histórico para mantê-lo linear. Ideal para branches locais e não publicadas.
git rebase -i [base]	Integração/Rebase	Modo interativo do rebase, permitindo squash (combinação), edição ou reordenação de commits.
git cherry-pick [hash]	Integração/Portabilidade	Aplica as alterações introduzidas por um único commit específico na branch atual, criando um novo commit equivalente.
git revert [hash]	Manipulação Histórico	Cria um novo commit que desfaz as alterações introduzidas por um commit anterior. Usado para desfazer mudanças em histórico compartilhado de forma segura.
Continua na próxima página		

Continuação da Tabela: Comandos do Git

Comando	Categoria	Explicação Detalhada
git fetch	Sincronização Remota	Baixa dados (objetos e refs) de um repositório remoto para o repositório local, sem alterar o Working Tree ou Index (operação segura).
git pull	Sincronização Remota	Equivalente a git fetch seguido por uma integração (default: merge). Pode alterar o estado local e causar conflitos imediatamente (operação menos segura).
git push [remote][branch]	Sincronização Remota	Carrega commits locais para um repositório remoto. Exige uma operação fast-forward, a menos que <code>-force</code> seja utilizado.
git push --tags	Sincronização Remota	Envia tags locais para o repositório remoto.
git remote	Sincronização Remota	Gerencia os repositórios remotos rastreados (e.g., listar, adicionar, remover).
git log	Auditoria/Inspeção	Exibe o histórico de commits.
git shortlog	Auditoria/Inspeção	Fornece um resumo conciso do git log, agrupando commits por autor.
git show	Auditoria/Inspeção	Exibe informações detalhadas sobre um objeto Git (commit, tag, etc.).
git reflog	Auditoria/Recuperação	Registra as atualizações locais no HEAD e em outras referências, agindo como uma rede de segurança para recuperar commits perdidos após resets ou rebase.
git tag	Marcação/Utilitários	Cria, lista, deleta ou verifica objetos de tag, usados para marcar pontos estáticos (releases) no histórico.
Continua na próxima página		

Continuação da Tabela: Comandos do Git

Comando	Categoria	Explicação Detalhada
git tag -a [name]	Marcação/Utilitários	Cria uma tag anotada (com metadados e mensagem), preferida para releases públicas.
git stash	Utilitários de Contexto	Salva temporariamente o Working Directory e o Index (alterações não comitadas) para permitir a troca de contexto.
git stash pop	Utilitários de Contexto	Aplica o último stash salvo e o remove da lista de stashes.
git stash apply	Utilitários de Contexto	Aplica o último stash salvo, mas o mantém na lista.
git submodule	Utilitários Avançados	Inicializa, atualiza ou inspeciona submódulos (repositórios aninhados).
git worktree	Utilitários Avançados	Gerencia múltiplas Working Trees (checkouts) do mesmo repositório, permitindo trabalhar em várias branches simultaneamente.
gitk	Utilitários Avançados	O navegador de repositório Git (ferramenta GUI).
scalar	Utilitários Avançados	Ferramenta projetada para gerenciar repositórios Git de grande escala (Large Git Repositories).
git sparse-checkout	Utilitários Avançados	Reduz a Working Tree para um subconjunto de arquivos rastreados, otimizando o desempenho em repositórios massivos.

APÊNDICE B – Comandos GitHub CLI

Este apêndice apresenta uma referência dos principais comandos do GitHub CLI (gh) organizados por funcionalidade.

Comandos do GitHub-CLI (gh)

Tabela 3 – Comandos do GitHub-CLI (gh)

Comando Base	Subcomando	Explicação Funcional Detalhada	Exemplo de Sintaxe Chave
gh alias	set	Cria um alias para um comando gh, permitindo atalhos personalizados para comandos frequentes	<code>gh alias set prc "pr create"</code>
gh alias	list	Lista todos os aliases configurados no GitHub CLI	<code>gh alias list</code>
gh alias	delete	Remove um alias previamente configurado	<code>gh alias delete prc</code>
gh auth	login	Autentica o usuário no GitHub via navegador web ou token	<code>gh auth login</code>
gh auth	logout	Remove a autenticação do usuário atual	<code>gh auth logout</code>
<i>Continua na próxima página</i>			

Continuação da Tabela: Comandos do GitHub-CLI (gh)

Comando Base	Subcomando	Explicação Funcional Detalhada	Exemplo de Sintaxe Chave
gh auth	status	Exibe o status de autenticação atual e usuário conectado	gh auth status
gh auth	refresh	Renova a autenticação para um host específico	gh auth refresh --hostname github.com
gh auth	token	Exibe o token de autenticação atual	gh auth token
gh browse	-	Abre o repositório atual no navegador web	gh browse
gh browse	--branch	Abre uma branch específica no navegador	gh browse --branch feature-branch
gh browse	--commit	Abre um commit específico no navegador	gh browse --commit abc123
gh browse	--issue	Abre uma issue específica no navegador	gh browse --issue 42
gh browse	--pull-request	Abre um pull request específico no navegador	gh browse --pull-request 15
gh browse	--settings	Abre as configurações do repositório no navegador	gh browse --settings
gh browse	--wiki	Abre a wiki do repositório no navegador	gh browse --wiki
gh codespace	code	Abre um codespace no Visual Studio Code	gh codespace code
gh codespace	cp	Copia arquivos entre o sistema local e um codespace	gh codespace cp local.txt remote:./
<i>Continua na próxima página</i>			

Continuação da Tabela: Comandos do GitHub-CLI (gh)

Comando Base	Subcomando	Explicação Funcional Detalhada	Exemplo de Sintaxe Chave
gh codespace	create	Cria um novo codespace	gh codespace create
gh codespace	delete	Remove um codespace específico	gh codespace delete my-codespace
gh codespace	jupyter	Abre um codespace no JupyterLab	gh codespace jupyter
gh codespace	list	Lista todos os codespaces disponíveis	gh codespace list
gh codespace	logs	Exibe os logs de um codespace específico	gh codespace logs my-codespace
gh codespace	ports	Lista as portas encaminhadas de um codespace	gh codespace ports
gh codespace	ports forward	Encaminha uma porta do codespace para o local	gh codespace ports forward 3000:4000
gh codespace	ports visibility	Define a visibilidade de uma porta	gh codespace ports visibility 3000:public
gh codespace	ssh	Conecta-se a um codespace via SSH	gh codespace ssh
gh codespace	stop	Para um codespace em execução	gh codespace stop my-codespace
gh gist	create	Cria um novo gist a partir de arquivos ou entrada padrão	gh gist create script.py
gh gist	clone	Clona um gist específico para o sistema local	gh gist clone abc123
<i>Continua na próxima página</i>			

Continuação da Tabela: Comandos do GitHub-CLI (gh)

Comando Base	Subcomando	Explicação Funcional Detalhada	Exemplo de Sintaxe Chave
gh gist	delete	Remove um gist específico	gh gist delete abc123
gh gist	edit	Edita um gist existente	gh gist edit abc123
gh gist	list	Lista todos os gists do usuário	gh gist list
gh gist	view	Visualiza um gist específico no terminal	gh gist view abc123
gh issue	create	Cria uma nova issue no repositório	gh issue create --title "Bug"--body "Descrição"
gh issue	list	Lista issues do repositório com filtros opcionais	gh issue list --state open
gh issue	status	Mostra o status das issues relevantes para o usuário	gh issue status
gh issue	close	Fecha uma issue específica	gh issue close 42
gh issue	comment	Adiciona um comentário a uma issue	gh issue comment 42 --body "Comentário"
gh issue	delete	Remove uma issue específica	gh issue delete 42
gh issue	edit	Edita uma issue existente	gh issue edit 42 --title "Novo título"
gh issue	lock	Trava os comentários de uma issue	gh issue lock 42
gh issue	reopen	Reabre uma issue fechada	gh issue reopen 42
<i>Continua na próxima página</i>			

Continuação da Tabela: Comandos do GitHub-CLI (gh)

Comando Base	Subcomando	Explicação Funcional Detalhada	Exemplo de Sintaxe Chave
gh issue	transfer	Transfere uma issue para outro repositório	gh issue transfer 42 owner/repo
gh issue	view	Exibe detalhes de uma issue específica	gh issue view 42
gh project	copy	Copia um projeto para um novo repositório ou organização	gh project copy 1 --draft --target-owner novaorg
gh project	create	Cria um novo projeto	gh project create --title "Meu Projeto"
gh project	delete	Remove um projeto específico	gh project delete 1
gh project	edit	Edita as propriedades de um projeto	gh project edit 1 --title "Novo Título"
gh project	field	Gerencia campos personalizados do projeto	gh project field create 1 --name "Prioridade"
gh project	item	Gerencia itens dentro de um projeto	gh project item add 1 --url < https://github.com/owner/repo/issues/1 >
gh project	list	Lista projetos disponíveis	gh project list --owner owner
gh project	view	Visualiza detalhes de um projeto específico	gh project view 1
gh pr	checks	Exibe os status checks de um pull request	gh pr checks 15
gh pr	close	Fecha um pull request específico	gh pr close 15
<i>Continua na próxima página</i>			

Continuação da Tabela: Comandos do GitHub-CLI (gh)

Comando Base	Subcomando	Explicação Funcional Detalhada	Exemplo de Sintaxe Chave
gh pr	comment	Adiciona um comentário a um pull request	gh pr comment 15 --body "Comentário"
gh pr	create	Cria um novo pull request	gh pr create --title "Feature" --body "Descrição"
gh pr	diff	Exibe as diferenças introduzidas pelo pull request	gh pr diff 15
gh pr	edit	Edita propriedades de um pull request	gh pr edit 15 --title "Novo Título"
gh pr	list	Lista pull requests do repositório	gh pr list --state open
gh pr	merge	Mescla um pull request	gh pr merge 15 --squash
gh pr	ready	Marca um pull request como pronto para revisão	gh pr ready 15
gh pr	reopen	Reabre um pull request fechado	gh pr reopen 15
gh pr	review	Adiciona uma revisão a um pull request	gh pr review 15 --approve
gh pr	status	Mostra o status dos pull requests relevantes	gh pr status
gh pr	view	Exibe detalhes de um pull request específico	gh pr view 15
gh pr	checkout	Faz checkout da branch de um pull request	gh pr checkout 15
<i>Continua na próxima página</i>			

Continuação da Tabela: Comandos do GitHub-CLI (gh)

Comando Base	Subcomando	Explicação Funcional Detalhada	Exemplo de Sintaxe Chave
gh release	create	Cria um novo release	gh release create v1.0.0 --title "Versão 1.0.0"
gh release	delete	Remove um release específico	gh release delete v1.0.0
gh release	download	Baixa os assets de um release	gh release download v1.0.0
gh release	list	Lista todos os releases do repositório	gh release list
gh release	upload	Faz upload de assets para um release	gh release upload v1.0.0 arquivo.zip
gh release	view	Exibe detalhes de um release específico	gh release view v1.0.0
gh release	edit	Edita propriedades de um release existente	gh release edit v1.0.0 --title "Novo Título"
gh repo	archive	Arquiva um repositório	gh repo archive owner/repo
gh repo	clone	Clona um repositório para o sistema local	gh repo clone owner/repo
gh repo	create	Cria um novo repositório	gh repo create meu-repo --public
gh repo	delete	Remove um repositório	gh repo delete owner/repo
gh repo	edit	Edita propriedades de um repositório	gh repo edit --description "Nova descrição"
gh repo	fork	Cria um fork de um repositório	gh repo fork owner/repo
<i>Continua na próxima página</i>			

Continuação da Tabela: Comandos do GitHub-CLI (gh)

Comando Base	Subcomando	Explicação Funcional Detalhada	Exemplo de Sintaxe Chave
gh repo	list	Lista repositórios do usuário ou organização	gh repo list --limit 10
gh repo	rename	Renomeia um repositório	gh repo rename novo-nome
gh repo	sync	Sincroniza um fork com seu repositório upstream	gh repo sync
gh repo	view	Exibe detalhes de um repositório	gh repo view owner/repo
gh repo	deploy-key	Gerencia chaves de deploy do repositório	gh repo deploy-key add chave.pub --title "Servidor"
gh repo	secret	Gerencia secrets do repositório	gh repo secret set API_KEY --body "valor"
gh run	cancel	Cancela uma execução de workflow	gh run cancel 123456789
gh run	delete	Remove execuções de workflow	gh run delete 123456789
gh run	download	Baixa artifacts de uma execução	gh run download 123456789
gh run	list	Lista execuções de workflows	gh run list
gh run	rerun	Reexecuta um workflow falho	gh run rerun 123456789
gh run	view	Exibe detalhes de uma execução	gh run view 123456789
gh run	watch	Monitora uma execução em tempo real	gh run watch 123456789
<i>Continua na próxima página</i>			

Continuação da Tabela: Comandos do GitHub-CLI (gh)

Comando Base	Subcomando	Explicação Funcional Detalhada	Exemplo de Sintaxe Chave
gh search	code	Busca por código no GitHub	gh search code "função javascript"
gh search	commits	Busca por commits	gh search commits "fix bug"--author=user
gh search	issues	Busca por issues e pull requests	gh search issues "bug label:bug"
gh search	prs	Busca especificamente por pull requests	gh search prs "feature state:open"
gh search	repos	Busca por repositórios	gh search repos "topic:machine-learning"
gh search	users	Busca por usuários	gh search users "nome location:Brasil"
gh secret	list	Lista secrets disponíveis	gh secret list
gh secret	remove	Remove um secret específico	gh secret remove API_KEY
gh secret	set	Define ou atualiza um secret	gh secret set API_KEY --body "valor"
gh ssh-key	add	Adiciona uma chave SSH à conta	gh ssh-key add chave.pub --title "Laptop"
gh ssh-key	list	Lista chaves SSH da conta	gh ssh-key list
gh ssh-key	delete	Remove uma chave SSH	gh ssh-key delete 123
gh workflow	disable	Desabilita um workflow	gh workflow disable "CI Tests"
<i>Continua na próxima página</i>			

Continuação da Tabela: Comandos do GitHub-CLI (gh)

Comando Base	Subcomando	Explicação Funcional Detalhada	Exemplo de Sintaxe Chave
gh workflow	enable	Habilita um workflow	gh workflow enable "CI Tests"
gh workflow	list	Lista workflows disponíveis	gh workflow list
gh workflow	run	Executa um workflow manualmente	gh workflow run "CI Tests"
gh workflow	view	Exibe detalhes de um workflow	gh workflow view "CI Tests"

APÊNDICE C – Padrões de Commits

Este apêndice apresenta uma referência completa dos principais padrões de commits organizados por categoria e funcionalidade.

Padrões de Commits

Tabela 4 – Padrões de Commits Profissionais

Tipo	Descrição	Quando Utilizar	Exemplo
feat	Introduz uma nova funcionalidade ao projeto.	Quando adicionar novas capacidades ou funcionalidades.	feat: adicionar autenticação via OAuth2 feat(api): implementar endpoint de usuários
fix	Corrige um bug ou erro no código.	Quando resolver problemas ou defeitos no sistema.	fix: corrigir cálculo de impostos fix(auth): resolver loop infinito no login
docs	Alterações na documentação.	Quando atualizar README, comentários ou documentação.	docs: atualizar guia de instalação

Continua na próxima página

Continuação da Tabela: Padrões de Commits Profissionais

Tipo	Descrição	Quando Utilizar	Exemplo
			docs(api): adicionar exemplos de uso
style	Mudanças que não afetam o significado do código.	Ao ajustar formatação, espaços, vírgulas, etc.	style: corrigir indentação no CSS style: remover espaços em branco
refactor	Reestruturação do código sem alterar comportamento.	Quando melhorar a estrutura sem mudar funcionalidades.	refactor: extrair método para reduzir complexidade refactor(db): otimizar queries SQL
perf	Melhorias de performance.	Ao otimizar velocidade ou eficiência do código.	perf: otimizar algoritmo de ordenação perf: reduzir tempo de carregamento em 30%
test	Adiciona ou modifica testes.	Ao criar novos testes ou corrigir existentes.	test: adicionar testes unitários para UserService test: corrigir teste de integração
Continua na próxima página			

Continuação da Tabela: Padrões de Commits Profissionais

Tipo	Descrição	Quando Utilizar	Exemplo
build	Mudanças no sistema de build ou dependências.	Ao atualizar dependências, Webpack, Maven, etc.	build: atualizar React para v18 build: configurar Dockerfile
ci	Mudanças na configuração de CI/CD.	Ao modificar GitHub Actions, GitLab CI, Jenkins, etc.	ci: adicionar pipeline de deploy automático ci: configurar testes E2E no GitHub Actions
chore	Tarefas de manutenção e rotina.	Para atualizações de rotina que não se encaixam em outras categorias.	chore: atualizar versão do package.json chore: limpar dependências não utilizadas
revert	Reverte um commit anterior.	Quando necessário desfazer mudanças anteriores.	revert: "feat: adicionar feature X" revert: commit abc1234
hotfix	Correção crítica para produção.	Para bugs críticos que exigem correção imediata.	hotfix: corrigir vulnerabilidade de segurança

Continua na próxima página

Continuação da Tabela: Padrões de Commits Profissionais

Tipo	Descrição	Quando Utilizar	Exemplo
			hotfix: resolver falha no processamento de pagamentos
security	Correções relacionadas à segurança.	Ao abordar vulnerabilidades ou melhorar segurança.	security: atualizar bibliotecas com vulnerabilidades security: implementar sanitização de inputs
init	Commit inicial do projeto.	Para o primeiro commit de um novo projeto.	init: configuração inicial do projeto init: estrutura base da aplicação

Anexos

ANEXO A – Lista de Presença

A.1 Listab de dos mebros presentes na capacitação

Nome	Presente em
------	-------------