

Git & GitHub do Zero ao Envio no ClassHero

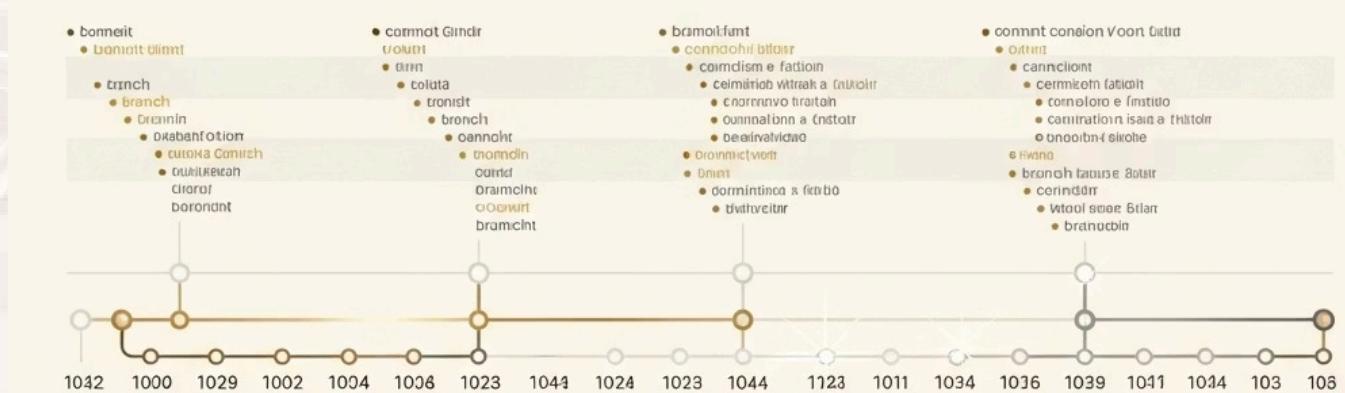
Este guia irá capacitá-lo a versionar projetos com Git, colaborar pelo GitHub e entregar repositórios pelo ClassHero com qualidade e segurança. Desde os conceitos básicos até as melhores práticas, você encontrará tudo o que precisa para dominar o controle de versão no desenvolvimento de software, mesmo começando do zero.

Por que usar Git e GitHub?

O controle de versão é uma habilidade essencial para qualquer desenvolvedor. Git e GitHub não são apenas ferramentas, mas parte fundamental do fluxo de trabalho moderno de desenvolvimento. Aqui estão as principais razões para adotá-los:

- **Salvar histórico completo:** cada *commit* é um ponto no tempo com o estado exato do projeto, permitindo voltar a qualquer momento se necessário.
- **Trabalhar em equipe** sem sobrescrever o trabalho do outro, através de branches e pull requests que organizam o fluxo de colaboração.
- **Desfazer erros com segurança**, voltando a commits anteriores quando algo dá errado.
- **Mostrar seu portfólio:** o GitHub funciona como um currículo técnico vivo, demonstrando suas habilidades para empregadores.

Git Version Control



No contexto acadêmico, usaremos Git para versionar nossos projetos localmente e GitHub para hospedar o código remotamente. A entrega final no **ClassHero** será o link do repositório e, quando solicitado pelo professor, o link de uma *release* específica.

Além dos benefícios técnicos, o uso dessas ferramentas desenvolve habilidades valorizadas no mercado de trabalho e prepara você para colaborar em projetos de qualquer tamanho.

Conceitos essenciais (sem jargão)

Repositório (repo)

É a pasta do seu projeto que contém todos os arquivos e o histórico completo de alterações. Pode existir localmente (na sua máquina) e remotamente (no GitHub).

Commit

Uma "fotografia" do estado do projeto em um momento específico, acompanhada de uma mensagem explicativa da mudança. Cada commit tem um identificador único (hash).

Branch

Uma linha de desenvolvimento paralela. A branch principal geralmente se chama "main" (ou "master" em repositórios mais antigos). Branches permitem trabalhar em recursos sem afetar o código principal.

Merge

O processo de unir as mudanças de uma branch em outra (ex.: incorporar um novo recurso da branch feature/login para a branch main).

Remote

Um servidor que hospeda seu repositório. GitHub é o mais popular, mas existem outros como GitLab e Bitbucket. O apelido "origin" costuma se referir ao remote principal.

Clone

Baixar uma cópia completa de um repositório remoto para sua máquina local, incluindo todo o histórico.

Push / Pull / Fetch

Push: enviar commits locais para o repositório remoto. Pull: trazer commits remotos e mesclar localmente. Fetch: apenas baixar as referências remotas sem mesclar.

.gitignore

Arquivo que lista padrões de arquivos e pastas que devem ser ignorados pelo Git (ex.: node_modules, arquivos de build, arquivos de configuração local).

Compreender esses conceitos fundamentais é essencial para usar o Git efetivamente. Ao longo deste guia, veremos como esses elementos se conectam no fluxo de trabalho diário e em situações específicas como a entrega de projetos no ClassHero.

Instalação e configuração inicial

Instale o Git

Escolha o método adequado para seu sistema operacional:

- **Windows:** Baixe e instale do gitforwindows.org (aceite a instalação do "Git Bash").
- **macOS:** Execute xcode-select --install no Terminal ou baixe em git-scm.com.
- **Linux:** Use sudo apt install git (Debian/Ubuntu) ou sudo dnf install git (Fedora).

No Windows, configure a conversão automática de final de linha com:

```
git config --global core.autocrlf true
```

Configure sua identidade

O Git precisa saber quem você é para associar seus commits:

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

Use o mesmo e-mail que você usará no GitHub para garantir que seus commits sejam corretamente vinculados à sua conta.

Crie uma conta no GitHub

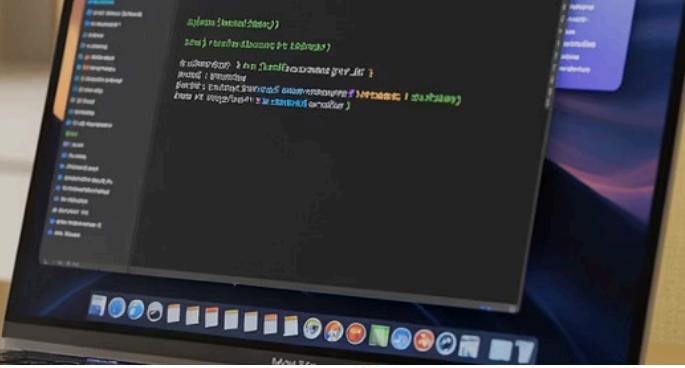
Acesse github.com e clique em "Sign up" para criar sua conta.

Recomendamos fortemente ativar a autenticação de dois fatores (2FA) para proteger sua conta. Acesse "Settings" → "Password and authentication" e siga as instruções para configurar o 2FA via aplicativo autenticador.

Configurando autenticação para fazer push

Para enviar código ao GitHub, você precisará configurar um método de autenticação. Existem duas opções principais:

Com essas configurações, você estará pronto para começar a usar Git e GitHub de forma segura e eficiente em seus projetos.



Opção A – SSH (Recomendado)

```
# gere sua chave (ed25519)
ssh-keygen -t ed25519 -C "seu_email@exemplo.com"
```

```
# pressione ENTER para aceitar o caminho sugerido
# e defina uma passphrase (opcional)
```

1

```
# inicie o agente e adicione a chave
ssh-add ~/.ssh/id_ed25519
```

```
# copie a chave pública
cat ~/.ssh/id_ed25519.pub
```

No GitHub: acesse "Settings" → "SSH and GPG keys" → "New SSH key" → cole a chave pública e salve.

Opção B – HTTPS + Token Pessoal (PAT)

No GitHub, acesse "Settings" → "Developer settings" → "Personal access tokens" → "Fine-grained token".

2

Configure um token com as permissões mínimas necessárias (normalmente apenas "repo") e defina um período de expiração adequado.

Ao fazer seu primeiro push, use este token como senha quando solicitado. Sua senha GitHub normal não funcionará para comandos Git.

Criando seu primeiro repositório

Existem dois caminhos principais para iniciar um projeto com Git: começar localmente e depois enviar para o GitHub, ou criar primeiro no GitHub e depois clonar para sua máquina. Vamos explorar ambos.

Do zero (local → remoto)

Quando você já tem um projeto local ou quer começar um do zero:

```
mkdir hello-git && cd hello-git  
echo "# Hello, Git" > README.md  
git init  
git add README.md  
git commit -m "chore: initial commit"
```

</>

Agora, crie um repositório vazio no GitHub (sem README para evitar conflito), copie a URL SSH ou HTTPS e execute:

```
# SSH  
git remote add origin git@github.com:SEU_USUARIO/hello-git.git  
# ou HTTPS  
# git remote add origin https://github.com/SEU_USUARIO/hello-git.git  
git branch -M main  
git push -u origin main
```

Do GitHub para sua máquina (remoto → local)

Quando você quer começar um projeto a partir de um repositório existente no GitHub:

No GitHub: clique em "New repository" → adicione nome, descrição, README e .gitignore se desejar.

Localmente, execute:

```
git clone git@github.com:SEU_USUARIO/SEU_REPO.git  
cd SEU_REPO
```

Esta abordagem é mais simples, especialmente para iniciantes. O GitHub cria automaticamente os arquivos iniciais e configura o repositório remoto para você.

 **Dica:** Se você criar um README no GitHub e também localmente, faça `git pull --rebase origin main` antes do primeiro push para evitar conflito. O rebase colocará suas mudanças locais em cima das remotas, minimizando problemas de mesclagem.

Em ambos os casos, você terá um repositório Git funcionando localmente e conectado ao GitHub. Lembre-se de que o nome da branch padrão agora é "main" em novos repositórios, não mais "master" como era anteriormente.

Fluxo básico do dia a dia

A rotina diária com Git envolve alguns comandos essenciais que você usará constantemente. Entender este fluxo é fundamental para trabalhar eficientemente com controle de versão.

Verificar status

`git status`

Mostra arquivos modificados, adicionados ou removidos que ainda não foram commitados. Este comando é seu melhor amigo para entender o estado atual do repositório.

Publicar alterações

`git push origin main`

Envia seus commits para o repositório remoto, tornando-os acessíveis para outros colaboradores e visíveis no GitHub.

Adicionar alterações

`git add arquivo.css`

Seleciona mudanças específicas para serem incluídas no próximo commit. Use `git add .` para adicionar todas as mudanças de uma vez.

Criar commit

`git commit -m "feat: adiciona estilo do header"`

Cria um ponto de salvamento com as mudanças adicionadas e uma mensagem descritiva. Prefira commits pequenos e frequentes.

Atualizar repositório

`git pull --rebase origin main`

Baixa e integra mudanças do repositório remoto antes de enviar suas alterações. Evita conflitos de merge comuns.



Boas práticas para o fluxo diário

- **Commits pequenos e frequentes** são melhores que grandes pacotes de mudanças. Cada commit deve representar uma mudança lógica única e completa.
- Use **mensagens de commit claras e descritivas**. Siga o padrão Conventional Commits quando possível (veremos mais sobre isso na seção de boas práticas).
- **Sempre puxe as mudanças remotas** antes de enviar suas próprias alterações para evitar conflitos desnecessários.
- Faça git status frequentemente para ter certeza de que você está ciente do estado atual do repositório.
- Verifique suas alterações com git diff antes de adicioná-las ao commit para garantir que não há código indesejado.

□ Este fluxo básico é o fundamento do trabalho com Git, seja em projetos pequenos ou grandes. Domine-o bem antes de avançar para recursos mais complexos.

Branches: trabalhando em paralelo

Branches (ramificações) são uma das funcionalidades mais poderosas do Git, permitindo que você trabalhe em diferentes recursos, correções ou experimentos de forma isolada, sem afetar o código principal. Pense em branches como linhas de desenvolvimento paralelas.

Criando e trocando de branch

```
# Cria e entra na nova branch  
git checkout -b feature/login
```

```
# Volta para a main (Git ≥ 2.23)  
git switch main
```

```
# Alternativa em versões mais antigas  
# git checkout main
```

Mesclando com a branch principal

```
# Certifique-se de estar na branch de destino  
git switch main
```

```
# Atualize a branch com as mudanças remotas  
git pull --rebase origin main
```

```
# Mescle sua branch de feature  
git merge feature/login
```

```
# Envie as mudanças mescladas  
git push origin main
```

Rebase x Merge: Para iniciantes, prefira **merge**. O rebase reorganiza o histórico; é poderoso, mas exige cuidado e conhecimento mais avançado. O merge preserva o histórico completo e é mais seguro para começar.

Estratégias comuns de branching



Feature Branch

Crie uma branch para cada nova funcionalidade (ex.: `feature/login`, `feature/carrinho-compras`). Desenvolva, teste e então mescle com a `main`.



Bugfix Branch

Para correções rápidas, use branches como `fix/login-erro`. Resolva o problema específico e mescle rapidamente.



Documentation Branch

Para mudanças apenas na documentação, use `docs/atualiza-readme`. Isso separa mudanças no código das mudanças na documentação.

Trabalhar com branches torna-se especialmente importante em projetos com múltiplos colaboradores, pois permite que todos trabalhem simultaneamente sem interferir no trabalho uns dos outros. Mesmo em projetos individuais, branches ajudam a organizar o desenvolvimento e manter o histórico de alterações mais limpo e compreensível.

- ⚠ Evite trabalhar diretamente na branch `main` em projetos colaborativos. Sempre crie uma branch específica para sua tarefa, mesmo que seja uma mudança pequena.

Colaboração com GitHub (fork, PR, review)

O GitHub facilita a colaboração entre desenvolvedores através de mecanismos como forks, pull requests e code reviews. Entender esses conceitos é essencial para trabalhar efetivamente em equipe.

Fork (se necessário)

Se você não tem acesso de escrita direto ao repositório original, crie um fork (uma cópia pessoal) clicando no botão "Fork" no GitHub. Isso criará uma cópia do repositório na sua conta GitHub.

Um fork é especialmente útil para contribuir com projetos de código aberto ou quando você precisa fazer alterações experimentais sem afetar o repositório original.

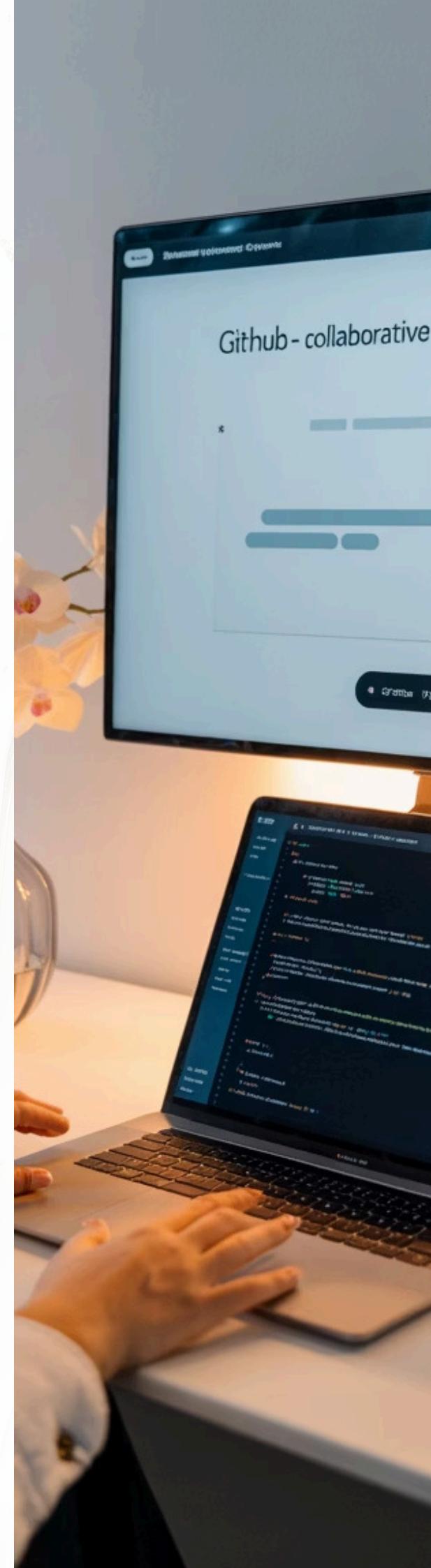
Clone e configuração do remote

```
# Clone seu fork (que se torna o 'origin')
git clone git@github.com:SEU_USUARIO/REPO-FORK.git
cd REPO-FORK
```

```
# Adicione o repositório original como 'upstream'
git remote add upstream
git@github.com:ORG/REPO-ORIGINAL.git
```

```
# Verifique os remotes configurados
git remote -v
```

Esta configuração permite manter seu fork atualizado com o repositório original.



Manter seu fork atualizado

```
# Busque as atualizações do repositório original  
git fetch upstream
```

```
# Certifique-se de estar na branch main  
git switch main
```

```
# Incorpore as mudanças do repositório original  
git merge upstream/main
```

```
# Atualize seu fork no GitHub  
git push origin main
```

Processo de Pull Request

Criar branch de feature

Crie uma branch para sua contribuição, faça seus commits e envie para seu fork:



```
git checkout -b feature/nova-funcionalidade  
# Faça suas alterações  
git add .  
git commit -m "feat: implementa nova funcionalidade"  
git push origin feature/nova-funcionalidade
```

Abrir Pull Request



No GitHub, vá até seu fork, selecione sua branch e clique em "Compare & pull request". Preencha uma descrição detalhada do que sua contribuição faz e por que ela é necessária.

Code Review



Os mantenedores do projeto revisarão seu código e poderão sugerir mudanças. Responda aos comentários, faça os ajustes necessários e envie novos commits para sua branch.

Merge



Após aprovação, seu PR será mesclado ao repositório original. Em projetos com permissões restritas, os mantenedores farão o merge; em outros, você pode receber permissão para fazê-lo.

- Em trabalhos acadêmicos, normalmente você **não precisa** de fork: crie o repositório no seu GitHub e convide o professor/avaliador como colaborador **ou** deixe o repositório público. O processo de PR ainda é útil para organizar o trabalho, mesmo em repositórios onde você tem acesso direto.

Entrega pelo ClassHero (passo a passo)

A entrega de trabalhos via ClassHero requer atenção a detalhes específicos para garantir que seu repositório esteja organizado, acessível e com instruções claras para avaliação. Siga este guia para maximizar suas chances de uma avaliação positiva.

1

Nome do repositório

Siga rigorosamente o padrão fornecido em aula. Exemplos:

- poo-2025-02-trabalho-01-nomesobrenome
- web-2025-01-projeto-final-nomesobrenome

O padrão geralmente inclui a sigla da disciplina, ano/semestre, tipo de trabalho e seu nome.

2

README completo

Crie um README.md detalhado que contenha, no mínimo:

- Nome e descrição do projeto
- Requisitos técnicos (versões de linguagens, frameworks, etc.)
- Instruções passo a passo para execução
- Estrutura de pastas explicada
- Exemplos de uso (quando aplicável)
- Autoria e licença

Um bom README é o cartão de visitas do seu projeto e facilita muito a avaliação.

3

.gitignore adequado

Configure corretamente o .gitignore para não enviar arquivos desnecessários:

- Pastas de dependências (node_modules, .dart_tool)
- Arquivos de build (bin/, obj/, build/)
- Arquivos de configuração local (.env, appsettings.Development.json)
- Arquivos temporários e logs

4

Licença e autoria

Inclua um arquivo LICENSE (geralmente MIT para projetos acadêmicos) e certifique-se de que seu nome e informações de contato estejam claramente indicados no README.

5

Visibilidade

Escolha uma das opções:

- **Repositório público:** qualquer pessoa pode ver, mas apenas colaboradores podem modificar
- **Repositório privado:** adicione o professor/avaliador como **collaborator** nas configurações do repositório

6

Release (opcional)

Se solicitado pelo professor, crie uma tag e uma Release:

```
git tag v1.0.0
git push origin v1.0.0
```

No GitHub, acesse "Releases" → "Create a new release", selecione a tag e adicione um changelog resumido das principais funcionalidades implementadas.

Checklist antes de entregar



O projeto roda do zero seguindo apenas as instruções do README?

Teste em um ambiente limpo ou peça a um colega para verificar se suas instruções são completas e claras.



Arquivos sensíveis ou pastas pesadas foram excluídos?

Verifique se o .gitignore está funcionando corretamente e se não há senhas, tokens ou arquivos grandes desnecessários no repositório.



Existem commits distribuídos ao longo do tempo?

Avaliadores valorizam um histórico que demonstre desenvolvimento progressivo, não apenas um grande commit final.



Há demonstrações visuais do funcionamento?

Screenshots, GIFs ou vídeos curtos no README ajudam muito na compreensão rápida do projeto.

- Para a entrega final no ClassHero, cole o link do repositório (e da Release, se exigido) no campo de entrega da atividade. Certifique-se de que o link está correto e acessível pelo professor.

Boas práticas: commits, mensagens e .gitignore

Conventional Commits

O padrão Conventional Commits torna o histórico do projeto mais legível e facilita a automação de changelog e versionamento semântico. A estrutura básica é:

tipo(escopo opcional): descrição

Onde tipo deve ser um dos seguintes:

feat

Nova funcionalidade adicionada ao projeto

fix

Correção de bug ou problema

docs

Alterações apenas na documentação

chore

Tarefas rotineiras como build, dependências

refactor

Mudanças no código que não alteram funcionalidade

test

Adição ou correção de testes

style

Mudanças que não afetam o significado do código (espaços, formatação)

perf

Melhorias de performance

Exemplos de bons commits

feat: implementa endpoint de criação de usuários

fix: corrige validação de email no registro

docs: atualiza instruções de execução no README

chore: atualiza dependências para versões seguras

refactor: simplifica lógica de autenticação

test: adiciona testes para serviço de pagamento

style: aplica formatação consistente em arquivos JS

perf: otimiza consultas ao banco de dados

.gitignore para diferentes tecnologias



Node/Nest/React

```
# Dependências  
node_modules/  
npm-debug.log  
yarn-error.log
```

```
# Ambiente  
.env  
.env.local  
.env.development  
.env.test  
.env.production
```

```
# Build  
dist/  
build/  
.next/  
out/
```

```
# Testes  
coverage/
```

```
# Logs  
logs/  
*.log
```

```
# Outros  
.DS_Store  
.idea/  
.vscode/  
*.swp  
*.swo
```



C#/.NET

```
# Binários e objetos  
bin/  
obj/
```

```
# Arquivos de usuário  
*.user  
*.userosscache  
*.rsuser  
*.suo  
*.userprefs
```

```
# VS e Rider  
.vs/  
.idea/  
*.sln.iml
```

```
# Configurações  
appsettings.*.json  
!appsettings.example.json
```

```
# Logs  
logs/  
*.log
```

```
# Ferramentas  
_ReSharper*/  
*.resharper  
*.DotSettings.user
```

```
# Testes  
TestResults/  
[Tt]est[Rr]esult*
```



Flutter/Dart

```
# Dart/Flutter
.dart_tool/
.flutter-plugins
.flutter-plugins-dependencies
.packages
.pub-cache/
.pub/
build/
pubspec.lock

# Android
**/android/**/gradle-wrapper.jar
**/android/.gradle
**/android/captures/
**/android/local.properties
**/android/key.properties

# iOS
**/ios/Pods/
**/ios/.symlinks/
**/ios/Flutter/App.framework
**/ios/Flutter/Flutter.framework
**/ios/Flutter/Generated.xcconfig
**/ios/Flutter/ephemeral/
**/ios/*.mode1v3
**/ios/*.mode2v3
**/ios/*.pbxuser
**/ios/Flutter/flutter_export_environment.sh

# Outros
.DS_Store
.idea/
**/GeneratedPluginRegistrant.*
```



- ⚠️** Ajuste sempre o `.gitignore` conforme as necessidades específicas do seu projeto. **Nunca** faça commit de arquivos sensíveis como `.env`, chaves privadas, dumps de banco de dados ou arquivos binários grandes.

Segurança: segredos, 2FA e tokens

A segurança é um aspecto crítico ao trabalhar com Git e GitHub. Práticas inadequadas podem levar a vazamentos de dados sensíveis ou comprometimento de contas. Aqui estão as principais considerações de segurança:

Proteção de segredos

- **Nunca** faça commit de senhas, chaves API, tokens ou outras credenciais diretamente no código.
- Use arquivos `.env` para variáveis de ambiente e sempre adicione-os ao `.gitignore`.
- Utilize `.env.example` como template, mostrando quais variáveis são necessárias sem revelar valores reais.
- Para projetos maiores, considere ferramentas de gerenciamento de segredos como Vault ou AWS Secrets Manager.

Autenticação de dois fatores (2FA)

- Ative 2FA na sua conta GitHub para uma camada adicional de segurança.
- Prefira autenticadores baseados em aplicativos (como Google Authenticator ou Authy) em vez de SMS.
- Mantenha seus códigos de recuperação em um local seguro caso perca acesso ao seu dispositivo de autenticação.
- A partir de agosto de 2023, o GitHub tornou 2FA obrigatório para contas que contribuem com código.

Tokens de acesso pessoal (PAT)

- Ao usar HTTPS, prefira tokens de acesso pessoal (PAT) com escopo mínimo necessário.
- Configure tokens com tempo de expiração adequado ao uso.
- Nunca compartilhe seus tokens, nem mesmo com colegas de equipe.
- Revogue imediatamente tokens que possam ter sido comprometidos.

O que fazer em caso de vazamento de credenciais

Identifique o problema

Determine exatamente quais credenciais foram expostas, quando e onde (em qual commit, por quanto tempo, se foi em um repositório público ou privado).

Remova do histórico

Para situações críticas, considere limpar o histórico com ferramentas como `git filter-branch` ou `BFG Repo Cleaner`. Lembre-se que isso reescreve a história e pode afetar colaboradores.

Para repositórios GitHub comprometidos, entre em contato com o suporte do GitHub para ajuda adicional.

Revogue as credenciais

Imediatamente revogue os tokens, chaves ou credenciais expostas nas respectivas plataformas (GitHub, AWS, etc). Não basta apenas removê-las do código.

Implemente medidas preventivas

Configure verificações pré-commit para detectar padrões de credenciais, use ferramentas de análise de segurança de repositório, e estabeleça um processo rigoroso de revisão de código.

- ✖ Lembre-se: mesmo que você remova um segredo de um commit posterior, ele ainda existe no histórico do Git e pode ser acessado. Uma vez que algo é commitado, considere-o potencialmente exposto.

Usando Git no VS Code e no GitHub Desktop

VS Code

O Visual Studio Code oferece uma integração robusta com Git, facilitando operações comuns através de uma interface gráfica:

- **Aba Source Control** (ícone de ramificação na barra lateral): mostra arquivos modificados, permite staging, commit, push e pull.
- **Visualização de diff**: clique em um arquivo modificado para ver as mudanças lado a lado.
- **Gutter indicators**: linhas coloridas na margem do editor indicam adições, modificações e remoções.

- **Branch atual:** visível na barra de status inferior, permitindo trocar de branch com um clique.
- **Comandos Git:** acessíveis via paleta de comandos (Ctrl+Shift+P ou Cmd+Shift+P).

Extensões úteis para Git no VS Code

- **GitLens:** adiciona funcionalidades avançadas como blame, histórico de linha, comparação de branches e muito mais.
- **Git Graph:** visualiza o histórico de branches e commits em um gráfico interativo.
- **Markdown All in One:** útil para editar arquivos README.md com preview em tempo real.

GitHub Desktop

GitHub Desktop é uma aplicação standalone que simplifica o uso do Git para quem prefere interfaces gráficas:

- **Interface simplificada:** foca nas operações mais comuns, ideal para iniciantes.
- **Fluxo visual:** clone, branch, commit, push/pull e PR com poucos cliques.
- **Histórico claro:** visualize facilmente a linha do tempo de commits.
- **Resolução de conflitos:** interface amigável para resolver conflitos de merge.
- **Integração com GitHub:** criação de issues e PRs diretamente do aplicativo.

□ Embora ferramentas gráficas como VS Code e GitHub Desktop facilitem o trabalho com Git, é recomendável aprender também os comandos básicos via terminal. Isso fornece uma compreensão mais profunda e flexibilidade em situações onde interfaces gráficas não estão disponíveis.

Qual escolher?



Iniciantes

GitHub Desktop oferece a curva de aprendizado mais suave, focando apenas nos fluxos mais comuns e escondendo complexidades.

Desenvolvedores

VS Code é ideal se você já o utiliza para codificar, permitindo integrar o fluxo de desenvolvimento com o controle de versão.

Avançados

Combine CLI para operações complexas com ferramentas visuais para visualização de histórico e diff, aproveitando o melhor dos dois mundos.

Independentemente da ferramenta escolhida, o importante é encontrar um fluxo de trabalho confortável que incentive boas práticas de controle de versão. Para projetos acadêmicos, tanto VS Code quanto GitHub Desktop são excelentes opções que simplificam o processo de entrega no ClassHero.

Resolvendo conflitos de merge

Conflitos de merge ocorrem quando dois desenvolvedores modificam a mesma parte de um arquivo e o Git não consegue determinar automaticamente qual versão deve prevalecer. Resolver conflitos é uma habilidade essencial para trabalhar efetivamente com Git.

Anatomia de um conflito

Quando um conflito ocorre, o Git marca os arquivos problemáticos com indicadores especiais:

```
<<<<< HEAD  
Código da sua versão atual (HEAD)  
=====  
Código da versão que está sendo mesclada  
>>>>> feature/nova-funcionalidade
```

Estes marcadores mostram as duas versões conflitantes do código:

- <<<<< HEAD até =====: sua versão atual (HEAD)
- ===== até >>>>> feature/nova-funcionalidade: a versão da branch que está sendo mesclada

Passos para resolver conflitos

Identificar arquivos conflitantes

Quando um git pull ou git merge resulta em conflito, o Git informa quais arquivos estão conflitantes. Use git status para ver a lista completa.

```
git status  
# Você verá mensagens como:  
# "both modified: arquivo.js"
```

Editar os arquivos conflitantes

Abra cada arquivo conflitante e edite manualmente para:

- Escolher uma das versões (sua ou da outra branch)
- Combinar as duas versões de forma que faça sentido
- Reescrever completamente se necessário

Remova os marcadores de conflito (<<<<< HEAD, =====, >>>>> feature/nova-funcionalidade) ao editar.

Testar a solução

Após resolver todos os conflitos, certifique-se de que o código ainda funciona corretamente:

- Compile o projeto (se aplicável)
- Execute testes automatizados
- Teste manualmente funcionalidades relacionadas às áreas modificadas

Finalizar a resolução

Depois de resolver todos os conflitos e testar, finalize o processo:

```
git add . # Marca todos os conflitos  
como resolvidos  
git commit -m "merge: resolve conflito  
em arquivo.js"  
git push # Se necessário
```

Usando ferramentas para resolver conflitos

VS Code

O VS Code oferece uma interface visual para resolução de conflitos com botões "Accept Current Change", "Accept Incoming Change", "Accept Both Changes" ou "Compare Changes" diretamente no editor.

GitHub Desktop

Abre automaticamente seu editor configurado para resolução de conflitos e marca os arquivos que precisam ser resolvidos.

Ferramentas dedicadas

Ferramentas como meld, kdiff3 ou p4merge podem ser configuradas como ferramentas de merge do Git para uma experiência mais visual:

```
git config --global merge.tool meld
```

Use git mergetool para iniciar a ferramenta configurada.

 Sempre faça um `git pull` antes de fazer `git push` para minimizar a ocorrência de conflitos. Quanto mais tempo uma branch existir separadamente da main, maior a chance de conflitos complexos.

Tags, versões e releases

Tags são referências estáticas para pontos específicos no histórico do Git, geralmente usadas para marcar versões ou releases importantes. Diferente das branches, tags não mudam com novos commits, servindo como "âncoras" permanentes para estados específicos do código.

Criando e gerenciando tags

```
# Criar uma tag leve
```

```
git tag v1.0.0
```

```
# Criar uma tag anotada (recomendado)
```

```
git tag -a v1.0.0 -m "Versão 1.0.0 - Release inicial"
```

```
# Listar todas as tags
```

```
git tag
```

```
# Ver detalhes de uma tag específica
```

```
git show v1.0.0
```

```
# Publicar tags no repositório remoto
```

```
git push origin v1.0.0
```

```
# Publicar todas as tags de uma vez
```

```
git push origin --tags
```

```
# Checkout para uma tag específica
```

```
git checkout v1.0.0
```

Tags anotadas contêm metadados adicionais como o autor, data e uma mensagem, sendo preferíveis para marcar releases oficiais.

Versionamento Semântico

Um padrão comum para numeração de versões é o SemVer (Semantic Versioning):

MAJOR.MINOR.PATCH

- **MAJOR**: mudanças incompatíveis com versões anteriores
- **MINOR**: adições de funcionalidades compatíveis
- **PATCH**: correções de bugs compatíveis

Exemplo: 2.4.1 → 2.4.2 (correção de bug)

Criando Releases no GitHub

Crie uma tag localmente

```
git tag -a v1.0.0 -m "Versão 1.0.0 - Release inicial"  
git push origin v1.0.0
```

Acesse a página de Releases

No GitHub, navegue até seu repositório e clique na aba "Releases" ou acesse diretamente "<https://github.com/seu-usuario/seu-repo/releases>".

Crie uma nova Release

Clique em "Draft a new release" ou "Create a new release". Selecione a tag que você criou, adicione um título e uma descrição detalhada das mudanças (changelog).

Anexe binários (opcional)

Se aplicável, você pode fazer upload de arquivos compilados, documentação em PDF ou outros artefatos que fazem parte da release.

Publique a Release

Clique em "Publish release" para finalizar. A release estará visível na página do repositório e os usuários poderão baixar o código-fonte daquele ponto específico.

Benefícios para entregas acadêmicas

Estado definido

Tags e releases definem exatamente qual versão do código está sendo entregue, sem ambiguidades mesmo se você continuar desenvolvendo após a entrega.

Documentação clara

O changelog na release explica o que foi implementado, facilitando a avaliação e demonstrando profissionalismo.

Histórico organizado

Múltiplas entregas ao longo de um semestre ficam organizadas como releases sequenciais, criando um histórico visual do desenvolvimento.

- Para entregas no ClassHero, quando solicitado, inclua tanto o link do repositório quanto o link específico da release. Um exemplo seria:
["https://github.com/seu-usuario/projeto-poo/releases/tag/v1.0.0"](https://github.com/seu-usuario/projeto-poo/releases/tag/v1.0.0)

Markdown rápido para README

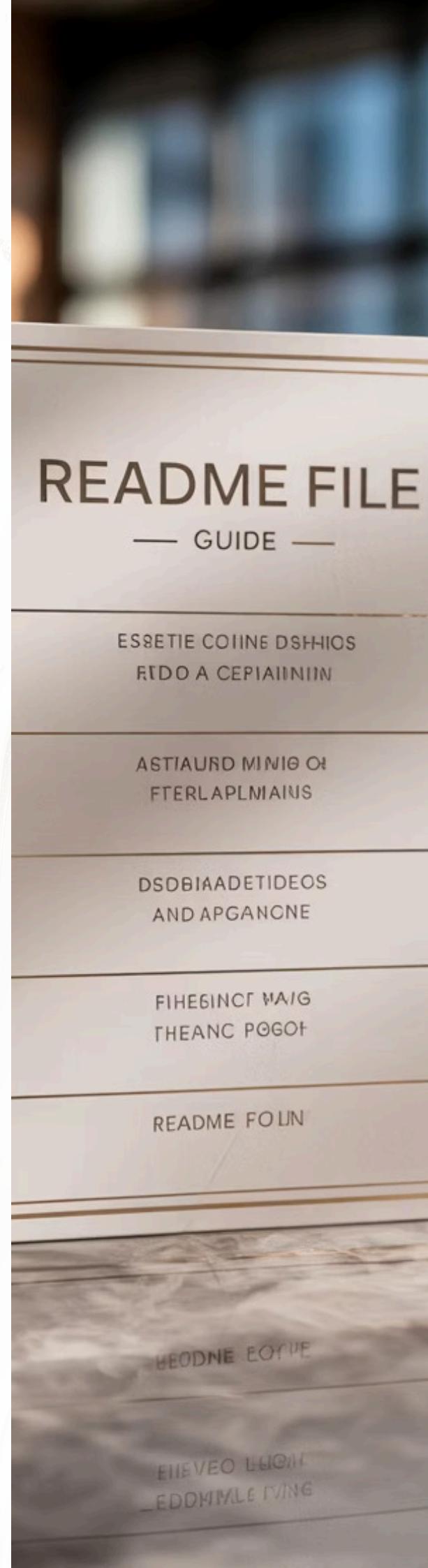
O README é a porta de entrada do seu projeto - a primeira coisa que as pessoas (incluindo avaliadores) verão. Um bom README deve ser claro, conciso e fornecer todas as informações necessárias para entender, instalar e usar o projeto.

Elementos essenciais de um README

- Título e descrição:** O que o projeto faz e por que é útil
- Requisitos:** Dependências e pré-requisitos técnicos
- Instalação:** Passo a passo detalhado
- Uso:** Como utilizar o projeto, com exemplos
- Estrutura:** Organização de pastas e arquivos principais
- Contribuição:** Como contribuir (para projetos colaborativos)
- Licença:** Termos de uso e distribuição
- Autoria:** Quem desenvolveu o projeto

Dicas para um README efetivo

- Use **screenshots** ou GIFs para demonstrar visualmente o projeto
- Mantenha o tom **profissional** mas acessível
- Organize com **títulos hierárquicos** para facilitar a navegação
- Forneça **exemplos de código** ou comandos em blocos destacados
- Inclua **badges** relevantes (versão, status, cobertura de testes)



Sintaxe básica de Markdown

Título principal (H1)

Subtítulo (H2)

Subseção (H3)

Texto em negrito ou também negrito

Texto em itálico ou também itálico

- Item de lista não ordenada

- Outro item

- Sub-item (com 2 espaços)

1. Item de lista ordenada

2. Segundo item

[Link para site](<https://www.exemplo.com>)

![Texto alternativo da imagem](caminho/para/imagem.jpg)

> Bloco de citação

> Continua aqui

`Código inline` para trechos curtos

```javascript

// Bloco de código com syntax highlighting

function exemplo() {

  return "Olá mundo";

}

```

Tabela simples:

Coluna 1	Coluna 2
----------	----------

-----	-----
-------	-------

Dado 1	Dado 2
--------	--------

Exemplo de estrutura para projetos acadêmicos

```
# Nome do Projeto
```

Descrição curta e objetiva do que o projeto faz e qual problema ele resolve. Este parágrafo deve ser conciso e ir direto ao ponto.

```
## Requisitos
```

- Node.js >= 20.0.0
- MongoDB >= 5.0
- Docker (opcional)

```
## Como executar
```

```
```bash
```

```
Clone o repositório
git clone https://github.com/seu-usuario/seu-projeto.git
cd seu-projeto
```

```
Instale as dependências
```

```
npm install
```

```
Configure as variáveis de ambiente
```

```
cp .env.example .env
```

```
Edite o arquivo .env com suas configurações
```

```
Execute o projeto
```

```
npm run dev
```

```
```
```

O servidor estará disponível em `http://localhost:3000`.

```
## Testes
```

```
```bash
```

```
Execute os testes unitários
npm test
```

```
Execute os testes com cobertura
```

```
npm run test:coverage
```

```
```
```



Estrutura de pastas

...

src/

```
  ├── controllers/ # Controladores da aplicação  
  ├── models/    # Modelos e esquemas de dados  
  ├── routes/    # Definição de rotas  
  ├── middlewares/ # Middlewares personalizados  
  ├── utils/     # Funções utilitárias  
  └── app.js      # Ponto de entrada da aplicação
```

...

Funcionalidades

- Autenticação de usuários
- CRUD completo de recursos
- Validação de dados
- API RESTful documentada

Licença

Este projeto está licenciado sob a licença MIT - veja o arquivo LICENSE para detalhes.

Autor

João Silva - RA: 123456 - joao.silva@email.com



O GitHub interpreta automaticamente o arquivo README.md na raiz do repositório e o exibe na página principal. Invista tempo para criar um README completo e bem formatado - isso demonstra profissionalismo e facilita muito a avaliação do seu trabalho.

Solução de problemas comuns

Mesmo usuários experientes do Git ocasionalmente encontram erros e situações desafiadoras. Aqui estão algumas das mais comuns e como resolvê-las:



fatal: not a git repository

Problema: Este erro ocorre quando você tenta executar um comando Git fora de um repositório Git inicializado.

Solução:

- Verifique se você está na pasta correta usando `pwd` ou `dir`
- Execute `git status` para confirmar se está em um repositório Git
- Se for um projeto novo, initialize-o com `git init`
- Se deveria ser um repositório, verifique se a pasta `.git` existe (pode estar oculta)



Updates were rejected because the remote contains work...

Problema: Ocorre quando você tenta enviar alterações, mas o repositório remoto tem commits que você não possui localmente.

Solução:

- Atualize seu repositório local primeiro: `git pull --rebase origin main`
- Resolva quaisquer conflitos que aparecerem
- Faça push novamente: `git push origin main`
- Alternativamente, use apenas `git pull` e depois `git push` se preferir merges explícitos



Permission denied (publickey)

Problema: Sua chave SSH não está configurada corretamente ou não foi adicionada à sua conta GitHub.

Solução:

- Verifique se sua chave SSH está registrada: `ssh-add -l`
- Adicione sua chave se necessário: `ssh-add ~/.ssh/id_ed25519`
- Teste a conexão SSH: `ssh -T git@github.com`
- Verifique se a chave pública está adicionada corretamente no GitHub em `Settings → SSH and GPG keys`
- Como último recurso, gere uma nova chave e adicione-a ao GitHub



Everything up-to-date, mas o GitHub não atualiza

Problema: Git informa que tudo está atualizado, mas as mudanças não aparecem no GitHub.

Solução:

- Verifique se você commitou suas mudanças: `git status` deve mostrar "nothing to commit"
- Confira se está na branch correta: `git branch -vv`
- Verifique se o remote está configurado corretamente: `git remote -v`
- Confirme se fez push para o remote e branch corretos: `git push origin nome-da-branch`
- Limpe o cache do navegador ou verifique em uma guia anônima



detected dubious ownership in repository

Problema: Comum em ambientes Linux/WSL, ocorre quando os arquivos do repositório têm permissões ou propriedade consideradas inseguras pelo Git.

Solução:

```
git config --global --add safe.directory  
/caminho/completo/do/projeto
```

Isso informa ao Git que o diretório é seguro mesmo com as permissões atuais. Use com cautela e apenas para diretórios que você confia.



Problemas com CRLF/LF (quebra de linha)

Problema: Diferenças de quebra de linha entre sistemas operacionais causando conflitos ou mudanças desnecessárias.

Solução:

- No Windows: `git config --global core.autocrlf true`
- No macOS/Linux: `git config --global core.autocrlf input`
- Para projetos específicos, crie um arquivo `.gitattributes` na raiz:

```
* text=auto eol=lf  
*.{cmd,[cC][mM][dD]} text eol=crlf  
*.{bat,[bB][aA][tT]} text eol=crlf
```

Estratégias gerais para resolução de problemas



- Use `git status` e `git log` frequentemente para entender o estado atual.
Estes comandos fornecem informações valiosas sobre o que está acontecendo no seu repositório.



- Antes de tentar soluções drásticas, faça backup do seu trabalho.
Use `git stash` para salvar temporariamente alterações não commitadas ou crie uma branch de backup.



- Consulte a mensagem de erro completa.
O Git geralmente fornece dicas sobre como resolver o problema na própria mensagem de erro.



- Mantenha o Git atualizado.
Versões mais recentes frequentemente corrigem bugs e melhoram mensagens de erro.

- Se encontrar um problema que não consegue resolver, a comunidade Git é muito ativa. Sites como Stack Overflow, a documentação oficial do Git e fóruns do GitHub são excelentes recursos para encontrar soluções para problemas específicos.

Glossário mínimo

Este glossário contém os termos essenciais do Git e GitHub que todo desenvolvedor deve conhecer. Dominar esse vocabulário facilitará a comunicação em equipe e a compreensão da documentação.



HEAD

Ponteiro que referencia o commit atual em que você está trabalhando. Geralmente é o último commit da branch atual.

Stash

"Guardar no bolso" mudanças temporariamente. Útil quando você precisa trocar de branch mas não quer commitar mudanças em andamento.

```
git stash      # salva as mudanças  
git stash pop  # aplica e remove do stash  
git stash list # lista stashes salvos
```

Rebase

Processo de reaplica commits sobre outro ponto da história. Alternativa ao merge que resulta em histórico mais linear, mas reescreve o histórico.

```
git rebase main # reaplica commits da branch atual sobre main
```

Cherry-pick

Aplica mudanças de commits específicos para a branch atual. Útil para portar correções entre branches.

```
git cherry-pick abc123 # aplica o commit abc123 à branch atual
```

Upstream/Origin

Upstream geralmente refere-se ao repositório original; Origin é o apelido padrão para o seu fork ou repositório remoto principal.

CI/CD

Continuous Integration/Continuous Delivery - automação de testes e deploy. GitHub Actions, Jenkins e Travis CI são ferramentas comuns para implementar CI/CD.

Detached HEAD

Estado em que HEAD aponta diretamente para um commit específico em vez de uma branch. Ocorre ao fazer checkout de um commit ou tag.

Force Push

Substituir o histórico remoto com o local. Potencialmente perigoso em branches compartilhadas.

```
git push --force # use com extrema cautela
```

Termos específicos do GitHub



Fork

Cópia pessoal de um repositório de outra pessoa ou organização. Permite experimentar mudanças sem afetar o projeto original.



Pull Request (PR)

Solicitação para que o código de uma branch seja revisado e incorporado em outra (geralmente a main). Ponto central para code review e discussão.



Issues

Sistema de rastreamento de tarefas, bugs e melhorias. Cada issue pode ser associada a PRs e atribuída a colaboradores.



Actions

Sistema de automação integrado do GitHub. Permite criar workflows personalizados para teste, build, deploy e outras tarefas.

Conceitos avançados

Reflog

Registro de todas as ações que modificaram HEAD. Útil para recuperar estados perdidos após operações como reset.

Bisect

Ferramenta para encontrar qual commit introduziu um bug usando busca binária no histórico.

Submodules

Repositórios Git dentro de outros repositórios Git. Útil para gerenciar dependências externas.

Hooks

Scripts que são executados automaticamente em certos eventos Git (pre-commit, post-commit, etc).

Squash

Combinar múltiplos commits em um único commit, geralmente usado durante rebases interativos.

Este glossário não é exaustivo, mas cobre os termos mais frequentemente usados no dia a dia com Git e GitHub, especialmente no contexto de projetos acadêmicos e entregas pelo ClassHero.

Cheat sheet (cola rápida)

Iniciar e conectar ao GitHub

```
# Inicializar um repositório local  
git init  
  
# Conectar a um repositório remoto  
git remote add origin git@github.com:SEU_USUARIO/SEU_REPO.git  
  
# Verificar status atual  
git status  
  
# Adicionar arquivos ao staging  
git add arquivo.js    # arquivo específico  
git add .             # todos os arquivos  
  
# Criar commit  
git commit -m "mensagem clara e objetiva"  
  
# Atualizar repositório local  
git pull --rebase origin main  
  
# Enviar commits para o GitHub  
git push -u origin main  # primeira vez (-u configura tracking)  
git push                 # próximas vezes
```

Branches

```
# Criar e trocar para nova branch  
git checkout -b feature/nova-funcionalidade  
  
# Alternativa moderna (Git ≥ 2.23)  
git switch -c feature/nova-funcionalidade  
  
# Listar branches  
git branch      # locais  
git branch -r   # remotas  
git branch -a   # todas
```

```
# Trocar de branch  
git switch main      # forma moderna  
git checkout main    # forma tradicional  
  
# Mesclar branch à atual  
git merge feature/nova-funcionalidade  
  
# Deletar branch  
git branch -d feature/concluida  # local (seguro)  
git branch -D feature/problema   # local (força)  
git push origin --delete feature/x # remota
```

Visualizar histórico e mudanças

```
# Ver histórico de commits  
git log           # completo  
git log --oneline # resumido  
git log --graph --oneline # com gráfico  
git log -p         # com diff  
  
# Ver mudanças específicas  
git diff          # mudanças não staged  
git diff --staged  # mudanças staged  
git diff HEAD~1 HEAD # entre o commit atual e o anterior  
git diff main..feature/x # entre branches
```

Tags e releases

```
# Criar tag  
git tag v1.0.0        # tag leve  
git tag -a v1.0.0 -m "Mensagem" # tag anotada  
  
# Listar tags  
git tag  
  
# Publicar tags  
git push origin v1.0.0    # tag específica  
git push origin --tags     # todas as tags
```

Desfazer mudanças

```
# Descartar mudanças não commitadas  
git restore arquivo.js      # arquivo específico (Git ≥ 2.23)  
git restore .               # todos os arquivos  
git checkout -- arquivo.js  # forma antiga  
  
# Desfazer adição ao staging  
git restore --staged arquivo.js # Git ≥ 2.23  
git reset HEAD arquivo.js     # forma antiga  
  
# Modificar o último commit (antes de push)  
git commit --amend -m "Nova mensagem"  
  
# Voltar para um commit específico  
git reset --soft HEAD~1       # mantém mudanças staged  
git reset --mixed HEAD~1      # mantém mudanças, mas não staged  
git reset --hard HEAD~1       # descarta mudanças (cuidado!)
```

Stashing (guardar temporariamente)

```
# Guardar mudanças em andamento  
git stash
```

```
# Listar stashes  
git stash list
```

```
# Recuperar e remover último stash  
git stash pop
```

```
# Recuperar stash específico  
git stash apply stash@{1}
```

```
# Limpar todos os stashes  
git stash clear
```

- 💡 Esta cheat sheet cobre os comandos mais comuns para o dia a dia com Git. Mantenha-a à mão durante suas primeiras semanas usando Git e GitHub. Com o tempo, esses comandos se tornarão naturais. Para comandos mais avançados, consulte a documentação oficial do Git ou use `git help [comando]`.



Apêndices úteis

A) Workflow sugerido para a disciplina

Criar repositório



Cada entrega deve ter **um repositório dedicado** com nome padronizado conforme as orientações do professor.

Configure adequadamente o `.gitignore` desde o início para evitar problemas posteriores.

Trabalhar em branches



Mesmo em projetos individuais, trabalhe em branches `feature/...` e abra PRs para `main`.

Isso simula um ambiente profissional e cria oportunidades para auto-revisão do código.

Criar tag e release



Antes de entregar, crie uma tag `v1.0.0` e uma Release no GitHub com changelog resumido.

Isso "congela" o estado do código no momento da entrega, mesmo que você continue desenvolvendo.

Documentar



Preencha um README completo com todas as informações necessárias para rodar e entender o projeto.

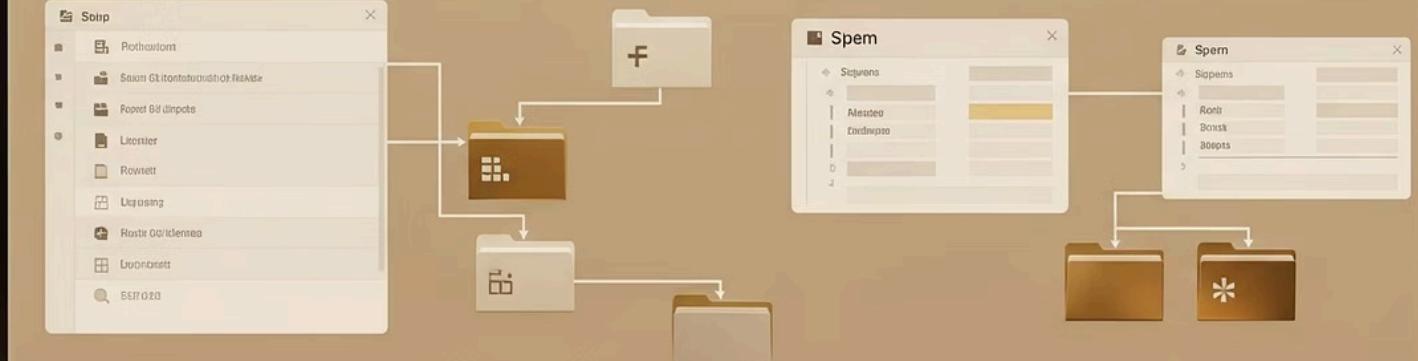
A qualidade da documentação frequentemente influencia a avaliação tanto quanto o código.

Entregar



Envie o link do repositório (e da Release, se solicitado) no ClassHero dentro do prazo.

Verifique o acesso caso o repositório seja privado.



B) Estrutura mínima recomendada do projeto

```
.project-root
├── README.md      # Documentação principal
├── .gitignore     # Arquivos/pastas a ignorar
├── LICENSE        # Licença (geralmente MIT)
├── .env.example   # Template de variáveis ambiente
├── package.json   # Manifesto (projetos Node)
└── src/           # Código-fonte
    ├── main.js      # Ponto de entrada
    ├── modules/     # Módulos do sistema
    |   └── [feature]/ # Organizado por funcionalidade
    └── shared/      # Código compartilhado
        ├── components/ # Componentes reutilizáveis
        └── utils/       # Funções utilitárias
```

Esta estrutura é uma base que pode ser adaptada conforme as necessidades específicas de cada projeto e tecnologia utilizada. O importante é manter uma organização lógica e bem documentada.

Dicas para organização de projetos

- Prefira organizar por **funcionalidade** ou **domínio** em vez de tipo de arquivo (evite pastas genéricas como "controllers", "models")
- Mantenha arquivos relacionados próximos para facilitar a navegação
- Use nomes descritivos e consistentes para arquivos e pastas
- Inclua um README em subpastas complexas para explicar seu propósito
- Mantenha a estrutura o mais plana possível, evitando aninhamentos excessivos
- Separe claramente código de produção de código de teste

⚠️ Lembre-se de que uma boa estrutura facilita tanto o desenvolvimento quanto a avaliação do projeto. Dedique tempo no início para planejar uma organização coerente.

C) Template de mensagens de commit

1

Commits de feature

feat: adiciona autenticação de usuários

Implementa o sistema de login e registro com validação de email e senha. Inclui recuperação de senha.

Resolve #42

Use para novas funcionalidades que agregam valor ao usuário.

2

Commits de correção

fix: corrige validação de CPF

Resolve o problema onde CPFs válidos eram rejeitados pelo sistema devido a formatação.

Closes #123

Use para correções de bugs e problemas.

3

Commits de documentação

docs: atualiza instruções de instalação

Adiciona passo a passo para configuração em ambiente Windows e macOS com screenshots.

Use para mudanças que afetam apenas documentação.

D) Checklist final de qualidade



O repositório segue a convenção de nomenclatura da disciplina?



O README contém todas as informações necessárias para rodar o projeto?



O projeto está organizado de forma lógica e bem estruturada?



Existe um histórico de commits significativo ao longo do tempo?



Arquivos sensíveis e pastas de dependências estão no .gitignore?



Quando aplicável, o projeto possui testes automatizados?



Há uma tag e release criadas para a entrega final?



O código segue boas práticas e está bem comentado quando necessário?

Próximos passos e recursos adicionais

Agora que você tem uma compreensão sólida de Git, GitHub e o processo de entrega pelo ClassHero, é hora de aplicar esse conhecimento na prática e continuar aprofundando seu aprendizado. Aqui estão algumas sugestões para seus próximos passos:



Pratique regularmente

Use Git para todos os seus projetos, mesmo os pequenos. A prática regular é a melhor maneira de desenvolver fluência com os comandos e fluxos de trabalho do Git.

Crie um repositório de "playground" para experimentar comandos mais avançados sem medo de quebrar nada importante.



Contribua com projetos open source

Procure projetos com a tag "good first issue" no GitHub para encontrar oportunidades adequadas para iniciantes.

Contribuir com projetos reais é uma excelente maneira de praticar o fluxo completo de fork, PR e code review.



Explore recursos avançados

Aprenda sobre hooks do Git, rebasing interativo, git bisect, e outras ferramentas poderosas que facilitam o trabalho em projetos maiores.

Experimente interfaces gráficas diferentes como GitKraken ou SourceTree para encontrar o que funciona melhor para você.



Simule trabalho em equipe

Em projetos acadêmicos, mesmo individuais, pratique fluxos de trabalho colaborativos criando diferentes branches para funcionalidades.

Faça code reviews dos seus próprios PRs, analisando criticamente seu código antes de mesclar à main.

Recursos adicionais para aprofundamento

Documentação oficial

- [Git Reference Manual](#) - Documentação completa e oficial do Git
- [Documentação do GitHub](#) - Guias e referências para todos os recursos do GitHub
- [Pro Git Book](#) - Livro completo disponível gratuitamente em português

Ferramentas de aprendizado interativo

- [Learn Git Branching](#) - Tutorial visual interativo sobre branches e merges
- [Atlassian Git Tutorials](#) - Série de tutoriais bem explicados
- [GitHub Skills](#) - Cursos interativos direto do GitHub

Ferramentas úteis

- [gitignore.io](#) - Gerador de arquivos .gitignore para qualquer tecnologia
- [Commitlint](#) - Ferramenta para garantir que mensagens de commit sigam padrões
- [Coleção de .gitignore](#) - Templates oficiais do GitHub para diversas linguagens

Lembre-se

Git é uma ferramenta poderosa que, como qualquer habilidade, melhora com a prática. Não tenha medo de cometer erros - eles são parte do processo de aprendizado. A beleza do Git é que, na maioria dos casos, você pode desfazer ou corrigir problemas.

Este guia foi projetado para dar a você um ponto de partida sólido, mas a verdadeira maestria vem da aplicação regular desses conceitos em projetos reais.