Git & Github

Introdução Engenharia Informática

Mário Antunes

27 de Outubro de 2025

Universidade de Aveiro

Um Guia Prático de Controlo de Versões e Colaboração

O Caos Antes do Controlo de Versões

Imagine que está a escrever um grande ensaio ou projeto de programação. A sua pasta provavelmente ficaria assim:

- `Projecto_v1.c`
- `Projecto_v2_corrigido.c`
- `Projecto_final.c`
- `Projecto_final_AGORA_VAI.c`
- `Projecto_final_APROVADO_v3.c`

Isto é confuso, propenso a erros e impossível de escalar. Não tem um registo claro de *o que* mudou, *porquê* mudou, ou *quando* mudou.

O Problema da Colaboração

1. **Método 1: Pastas Partilhadas (ex: Dropbox, Google Drive)**

- Estas são ferramentas de **sincronização de ficheiros**, não ferramentas de controlo de versões.
- **Problema:** *Bloqueio de ficheiros (File locking).* Se duas pessoas editarem o mesmo ficheiro, obtém `O_Meu_Ficheiro (Cópia Conflituosa).doc`. A última pessoa a guardar "ganha", e o trabalho é perdido. Apenas sincroniza a *última* versão.

2. **Método 2: Enviar Ficheiros por Email**
   - `Projecto_v5_alteracoes_Mario.zip`
   - `Projecto_v5_feedback_Ana.zip`
   - **Problema:** Como é que junta (merge) estas alterações? Este é um processo manual e caótico que garante o fracasso.

A Solução: Um Sistema de Controlo de Versões (VCS)

Um VCS é um sistema que regista alterações a um ficheiro ou conjunto de ficheiros ao longo do tempo. É uma **máquina do tempo** para o seu projeto.

Permite-lhe:

- Ver quem alterou o quê, e quando.
- Reverter para qualquer versão anterior.
- Comparar alterações ao longo do tempo.
- Trabalhar em equipa de forma segura sem sobrepor o trabalho dos outros.

Tipos de VCS: Centralizado vs. Distribuído

1. **Centralizado (CVCS) - ex: Subversion (SVN)**
   - Existe **um único servidor central** que contém todo o histórico do projeto.
   - Os programadores fazem "check-out" da versão mais recente, trabalham, e fazem "check-in" das suas alterações.
   - **Ponto fraco:** É um ponto único de falha. Se o servidor falhar, ninguém pode colaborar ou guardar o seu histórico.

2. **Distribuído (DVCS) - ex: Git, Mercurial**

- **Cada programador** tem uma cópia local completa (um "clone") de **todo o repositório**, incluindo o seu histórico completo.
- O "servidor" é apenas um outro repositório com o qual todos concordam em sincronizar.
- **Ponto forte:** Pode trabalhar offline, e o histórico está seguro em dezenas de máquinas.

A Origem do Git

- **Quem:** Linus Torvalds (o criador do Kernel do Linux).
- **Quando:** 2005.
- **Porquê:** A equipa do Kernel do Linux usava um DVCS proprietário chamado BitKeeper. Uma alteração no licenciamento forçou-os a parar de o usar.
- **O Problema:** Nenhum outro VCS na altura conseguia lidar com a escala (velocidade, tamanho e número de contribuidores) do projeto do Kernel do Linux.
- **A Solução:** Linus criou o **Git** em cerca de uma semana. Foi desenhado desde o início para ser distribuído, rápido e para garantir a integridade dos dados.

Como o Git “Pensa”: Snapshots, Não Diffs

Muitas ferramentas VCS mais antigas (como o SVN) armazenam as alterações como *deltas* ou *diffs* (uma lista do que mudou, linha por linha).

O Git não faz isso. O Git “pensa” no seu histórico como um **fluxo de snapshots (instantâneos)**.

Quando faz **commit** (guarda uma versão), o Git tira uma “fotografia” de como todos os seus ficheiros estão nesse momento e armazena uma referência a esse snapshot. Se um ficheiro não mudou, o Git apenas aponta para a versão anterior desse ficheiro.

O Conceito Central: Os 3 Estados

Esta é a parte mais crucial, e por vezes confusa, do Git. Os seus ficheiros existem num de três estados:

1. **Working Directory (Diretório de Trabalho):** Todos os seus ficheiros e pastas no sistema de ficheiros do seu computador. Esta é a sua "secretária desarrumada".
2. **Staging Area (Index) (Área de Preparação):** Uma área de "rascunho". É aqui que monta o seu snapshot. Usa `git add` para mover ficheiros *do* Working Directory *para* aqui.
3. **Repository (.git) (Repositório):** A base de dados permanente e imutável de todos os snapshots (commits) do seu projeto. Este é o "armário de arquivo".

Criar um Repositório: `git init`

Existem duas formas de iniciar um projeto com Git:

1. `git clone`: (Veremos isto mais tarde) Copiar um repositório *existente* de um servidor.
2. `git init`: Criar um *novo* repositório de raiz.

`git init` é o comando que executa dentro de uma pasta de projeto para a transformar num repositório Git.

```
$mkdir o-meu-novo-projeto$ cd o-meu-novo-projeto
$ git init
Initialized empty Git repository in /caminho/para/o-meu-novo-projeto/.git/
```

Este comando cria uma sub-pasta oculta chamada `.git`. Esta pasta `.git` é o "cérebro" do seu repositório—contém todos os snapshots, branches e histórico.

O Fluxo de Trabalho Central: `add` & `commit`

1. Modifica ficheiros no seu **Working Directory**.
2. Executa `git status` para ver o que mudou.
3. Usa `git add <nome-do-ficheiro>` para mover as alterações desejadas do Working Directory para a **Staging Area**.
4. Usa `git commit -m "A minha mensagem"` para pegar em tudo o que está na Staging Area, criar um **snapshot** permanente (um commit) e guardá-lo no seu **Repositório**.

A mensagem de commit é vital. Deve explicar *porquê* fez a alteração, não *o que* alterou (o código mostra o que).

O Que Faz uma *Boa* Mensagem de Commit?

Uma mensagem de commit é um registo para o seu eu futuro e para os seus colegas. Uma boa mensagem dá contexto e responde *porquê* uma alteração foi feita. O padrão da comunidade segue a regra “50/72”:

- **Assunto:** Um breve resumo, 50 caracteres ou menos.
- (Deixar uma linha em branco)
- **Corpo:** Uma explicação detalhada, com quebra de linhas aos 72 caracteres.

As 7 Regras de uma Ótima Mensagem de Commit

1. **Use o modo imperativo no assunto.**
   - **Bom:** `Add login page` (Adiciona página de login)
   - **Mau:** `Added login page` ou `Adding login page` (Pense como um comando: “Este commit irá...”)
2. **Separe o assunto do corpo com uma linha em branco.**
3. **Limite a linha do assunto a 50 caracteres.**
4. **Não termine a linha do assunto com um ponto.**
5. **Comece a linha do assunto com letra maiúscula.**
6. **Faça quebra de linha do corpo aos 72 caracteres.**
7. **Use o corpo para explicar *o quê* e *porquê* vs. *como*.** O código mostra *como*.

Exemplo: Bom vs. Mau

Commit Mau: `git commit -m "corrigir coisas"`

Commit Bom:

```
git commit -m "Fix: Corrige lógica de autenticação do utilizador" -m "

A função de login anterior falhava ao fazer o hash da password antes
de a comparar com a base de dados, resultando numa vulnerabilidade
de segurança crítica.

Este commit aplica a função de hashing SHA-256 ao input do
utilizador antes da consulta à base de dados. Isto resolve a falha
de segurança."
```

O Poder do Git: `branch`

Um **branch** (ramo) é simplesmente um ponteiro leve e móvel para um dos seus commits. O branch principal é tipicamente chamado `main` ou `master` (descontinuado ultimamente).

Porquê usar branches? Para trabalhar em novas funcionalidades ou corrigir bugs em **isolamento**, sem estragar o código estável que está no `main`.

- `git branch <nome>`: Cria um novo branch.
- `git checkout <nome>`: Muda o seu Working Directory para esse branch.
- `git checkout -b <nome>`: Um atalho que cria e muda num só passo.

Visualizar Branches & Merging

Este diagrama mostra a relação entre diferentes branches.

- O trabalho começa no **Main branch** (retângulos).
- Um novo `branch` é criado para trabalhar numa funcionalidade (círculos).
- Quando a funcionalidade está completa, é feito `merge` de volta para o branch principal.

{ width=85% }

Ver o Histórico: `git log`

Assim que tem commits, precisa de os ver.

- `git log`: Mostra o histórico de commits completo, com autores, datas e mensagens.
- `git log --oneline`: Mostra uma visão compacta, de uma linha, do histórico.
- `git log --graph --oneline`: Mostra o histórico com arte ASCII representando os branches e merges.

Combinar Trabalho: `merge`

Depois de terminar o seu trabalho num branch de funcionalidade (ex: `feature/login`), precisa de o integrar de volta no `main`.

Um `merge` (fusão) junta os históricos de dois branches.

1. Mude para o branch que quer atualizar: `git checkout main`
2. Execute o merge: `git merge feature/login`

O Git irá criar um novo "merge commit" que une os dois históricos.

O Inevitável: Conflitos de Merge!

Um conflito de merge acontece quando tenta fazer merge de dois branches que **editaram a mesma linha no mesmo ficheiro**. O Git não sabe qual a alteração correta, por isso para e pede-lhe para corrigir manualmente.

1. O Git irá marcar o ficheiro com `<<<<<<<` e `>>>>>>>` para lhe mostrar ambas as versões conflituantes.
2. Tem de abrir o ficheiro, apagar os marcadores e editar o código para ficar correto.
3. De seguida, faz `git add` ao ficheiro corrigido e `git commit` para finalizar o merge.

Alternativa ao Merging: `rebase`

Um `rebase` é uma forma de "reescrever o histórico" para o manter limpo e linear.

Em vez de um "merge commit" confuso, o `rebase` pega nos commits do seu branch de funcionalidade e **re-aplica-os, um por um,** em cima da versão mais recente do `main`.

- **Resultado:** Um histórico limpo, numa única linha.
- **Atenção:** Esta é uma ferramenta poderosa que reescreve o histórico. **NUNCA** faça rebase em branches públicos que outros colegas estejam a usar.

Colaboração - Git & GitHub: `remote` & `origin`

Até agora, tudo tem sido local. Como partilha o seu trabalho?

- Um `remote` é uma ligação nomeada a um repositório Git noutro local (ex: num servidor).
- `origin` é o nome convencional padrão para o seu `remote` principal (o servidor de onde clonou ou para onde quer enviar o seu trabalho).

Os Principais Comandos de Colaboração

- `git clone [url]`: Descarrega uma cópia completa (um clone) de um repositório remoto para a sua máquina e configura a ligação `origin`.
- `git pull`: ("Puxa") Vai buscar as alterações do `origin` e faz merge para o seu branch local. É `git fetch` + `git merge`.
- `git push`: ("Empurra") Envia os seus commits locais (que o remote não tem) para o `origin`.

Git vs. GitHub

Esta é uma distinção crítica.

- **Git** é a **ferramenta**. É o VCS distribuído, de linha de comandos, que instala no seu computador.
- **GitHub** é um **serviço**. É uma empresa baseada na web (fundada em 2008, agora propriedade da Microsoft) que **aloja** repositórios Git.

O GitHub adiciona uma "camada social" por cima do Git, adicionando funcionalidades como gestão de issues, wikis e Pull Requests.

O Fluxo de Trabalho Open-Source: `fork`

Não pode simplesmente fazer `push` das suas alterações para o repositório de outra pessoa (como o repositório oficial do Python).

Um `fork` é uma **cópia pessoal, do lado do servidor,** do repositório de outra pessoa. Fica na sua conta GitHub, e tem controlo total sobre ela. Este é o primeiro passo para contribuir.

O Coração da Colaboração: `pull request`

Um **Pull Request (PR)** é um pedido formal para que o dono de um projeto "puxe" (faça merge) das suas alterações (do seu *branch* ou *fork*) para o branch `main` dele.

Um PR é o início de uma **conversa**. *Não* é apenas um comando. É uma página web no GitHub onde:

- Descreve *porquê* fez as alterações.
- A sua equipa pode fazer **revisão de código (code review)**, linha a linha.
- Podem discutir melhorias.
- Testes automatizados podem ser executados.
- O dono do projeto pode aprovar e fazer merge do seu código.

Um Fluxo de Trabalho Git Típico (Resumo)

1. `git clone [url]`: Obter o projeto de um servidor remoto (como o GitHub).
2. `git checkout -b nova-funcionalidade`: Criar um novo branch para trabalhar isolado.
3. ... *Escreve o seu código, faz as suas alterações* ...
4. `git add .` : Adicionar os seus ficheiros alterados à Staging Area.
5. `git commit -m "Adiciona funcionalidade de login"`: Guardar um snapshot do seu trabalho.

6. `git push origin nova-funcionalidade`: Enviar o seu branch para o servidor remoto.
7. **Ir ao GitHub:** Abrir um **Pull Request** para propor as suas alterações.
8. **Discutir / Rever:** A sua equipa revê o seu código.
9. **Merge:** Um responsável pelo projeto faz merge do seu PR no `main`.
10. `git checkout main`: Voltar para o seu branch `main` local.
11. `git pull origin main`: Atualizar o seu `main` local com o código que acabou de ser integrado.

Fluxo de Trabalho Avançado: “GitFlow”

Embora o seu fluxo de trabalho típico seja ótimo para projetos pequenos, projetos maiores usam frequentemente um modelo mais estruturado e formal como o **GitFlow**.

- `main`: Apenas contém lançamentos (releases) oficiais, etiquetados (tagged). Nunca faz commit diretamente aqui.
- `develop`: O branch de integração principal para todas as novas funcionalidades.
- `feature` branches: Criados a partir do `develop` e integrados (merge) de volta no `develop`.
- `release` branches: Criados a partir do `develop` para preparar um novo lançamento (correções finais de bugs).
- `hotfix` branches: Criados a partir do `main` para corrigir bugs urgentes em produção.

{ width=85% }

Marcar Versões: `tag` & `release`

Quando o seu projeto atinge um ponto estável (ex: `v1.0.0`), quer marcá-lo.

- `git tag v1.0.0`: Uma "tag" (etiqueta) no Git é um ponteiro permanente que aponta para um commit específico. Ao contrário de um branch, uma tag não se move. É uma âncora no seu histórico.
- **GitHub Releases:** Um "Release" (lançamento) no GitHub é uma funcionalidade construída em cima de uma *tag*. É uma página web formal para o seu lançamento que lhe permite:
  - Escrever "notas de lançamento" (um *changelog*).
  - Anexar ficheiros binários (como `.exe` ou `.zip` instaladores).
  - Marcá-lo como um "pré-lançamento".

Esta é a forma oficial de apresentar uma nova versão aos

Resumo: Git vs. GitHub

- **Git** é a **ferramenta** distribuída no seu computador para seguir alterações (snapshots).
  - `init`, `add`, `commit`, `branch`, `merge`, `pull`, `push`
- **GitHub** é o **serviço** web social que aloja os seus repositórios e facilita a colaboração.
  - `Fork`, `Pull Request`, `Issues`, `Releases`
- **Fluxo de Trabalho Principal:** `Branch` ➡️ `Add` ➡️ `Commit` ➡️ `Push` ➡️ `Pull Request` ➡️ `Merge`
- **Regra de Ouro:** Trabalhe isolado em branches. Apenas integre (merge) trabalho limpo e finalizado no `main`.

Leitura Adicional e Recursos 📚

- **Livro Pro Git:** O guia definitivo para o Git, disponível online gratuitamente (em inglês).
  - https://git-scm.com/book/
- **Guia Hello World do GitHub:** Um tutorial simples de 10 minutos para começar.
  - https://docs.github.com/en/get-started/quickstart/hello-world
- **Learn Git Branching (Interativo):** Um tutorial interativo, semelhante a um jogo, para aprender a usar branches.
  - https://learngitbranching.js.org/
- **Git Cheat Sheet (Atlassian):** Uma ótima folha de consulta de uma página para os comandos mais comuns.
  - https://www.atlassian.com/git/tutorials/atlassian-git-cheatsheet