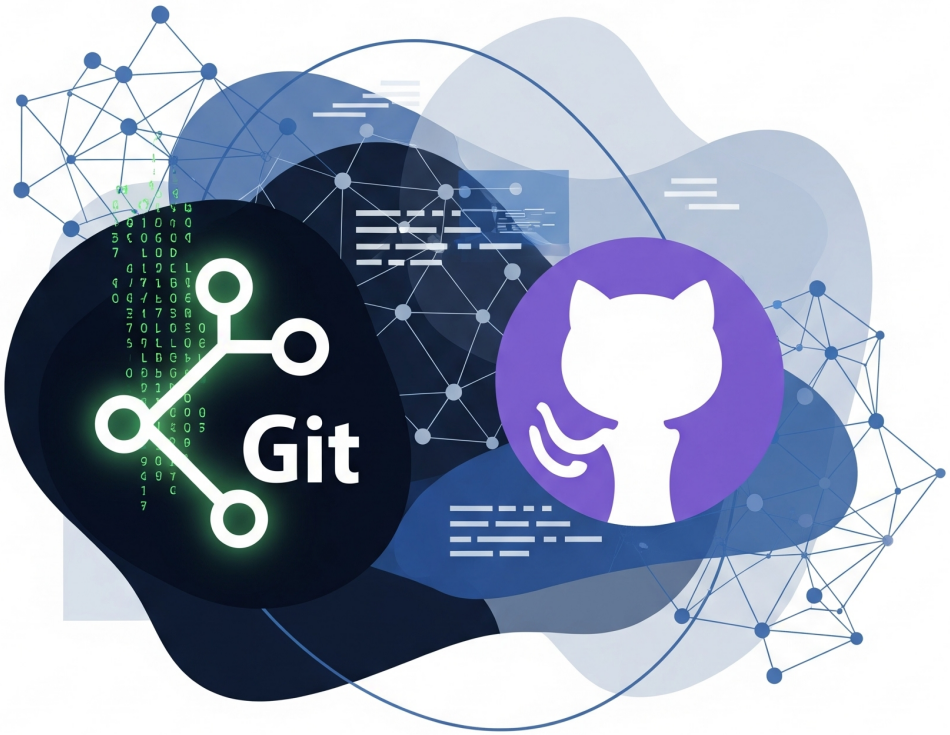
Utilizar Serviços Git e GitHub



Plano de Sessão Síncrona Detalhado: UC00617 - Utilizar Serviços Git e GitHub (2550 Horas)

João Carlos G. Soares e Silva

8 jul 2025

Resumo

Apresenta-se um plano detalhado para as sessões síncronas do curso "UC00617 - Utilizar Serviços Git e GitHub", com uma carga horária total de 25 horas. O plano estrutura os tópicos de aprendizagem, fornecendo definições operacionais e orientações metodológicas para o formador, e inclui exercícios práticos alinhados com os conteúdos de cada sessão. O objetivo pedagógico é potenciar a interação em tempo real, facilitar demonstrações práticas ao vivo e assegurar acompanhamento formativo contínuo, complementado por atividades autónomas para consolidação fora do horário de aula.

Introdução

Este plano foi otimizado para a modalidade síncrona, maximizando a interação em tempo real, as demonstrações ao vivo e o acompanhamento direto do formador. Os formandos farão a maioria dos exercícios **durante** as sessões, com alguns trabalhos para consolidar fora de horas.

—

Sessão Síncrona 1: Desmistificando o Controlo de Versões e Primeiros Passos com Git (5 Horas)

Resumo da Sessão

Esta sessão inicial estabelecerá a base. Começaremos por compreender a importância do controlo de versões, focando no **Git**. Os formandos serão guiados passo a passo na instalação e configuração do Git, e realizarão os seus primeiros **commits** em repositórios locais.

Conteúdo a Lecionar (Com Definições e Explicações para o Formador)

1.1. Introdução ao Controlo de Versões (1 hora)

- **O Que é Controlo de Versões (Version Control)?**
  - **Definição:** Um sistema que regista as alterações feitas a um ficheiro ou conjunto de ficheiros ao longo do tempo, permitindo recuperar versões específicas posteriormente.
  - **Analogia/Explicação:** "Pensem num 'máquina do tempo' para os vossos ficheiros. Em vez de terem 'Documento_final.docx', 'Documento_final_v2.docx', 'Documento_final_final.docx', o controlo de versões gere isto por vocês de forma inteligente."
  - **Por que é indispensável no Desenvolvimento de Software?**
    - **Rastreabilidade:** Saber quem fez o quê, quando e porquê. Essencial para auditoria e depuração.
    - **Colaboração:** Múltiplos desenvolvedores podem trabalhar no mesmo projeto simultaneamente sem sobrescrever o trabalho uns dos outros.
    - **Histórico Completo:** Acesso a qualquer versão anterior do código.
    - **Recuperação de Erros:** Reverter facilmente para uma versão funcional anterior em caso de introdução de bugs.
    - **Experimentação Segura:** Criar "ramificações"(**branches**) para testar novas funcionalidades sem afetar o código principal.
- **Sistemas de Controlo de Versões (VCS):**
  - **Centralizados (CVCS):**

- **Explicação:** Existe um servidor central que armazena todas as versões do código. Os desenvolvedores fazem "checkout" (obtêm) e "commit" (enviam) as alterações para esse servidor.
- **Exemplos:** Subversion (SVN), CVS.
- **Limitações:** "Um ponto único de falha"– se o servidor cair, ninguém consegue trabalhar. Dependência de rede constante.

– **Distribuídos (DVCS):**

- **Explicação:** Cada desenvolvedor tem uma cópia **completa** do repositório, incluindo todo o histórico de alterações.
- **Vantagens:** Trabalho offline, redundância de dados (cópia em cada máquina), maior velocidade (a maioria das operações é local), maior flexibilidade para fluxos de trabalho.
- **Exemplos: Git**, Mercurial, Darcs.

- **Por Que o Git?**
  - **Liderança de Mercado:** Amplamente adotado na indústria e em projetos open-source.
  - **Performance:** Extremamente rápido nas operações devido à sua natureza distribuída.
  - **Flexibilidade:** Suporta diversos fluxos de trabalho.
  - **Comunidade:** Grande ecossistema de ferramentas e vasto suporte online (Stack Overflow, documentação).

1.2. Instalação e Configuração do Git (2 horas)

- **Preparação:**
  - **Verificar requisitos:** Poucos, mas boa conexão à internet para download.
  - **Desinstalar versões antigas:** Se houver versões problemáticas.
- **Guia Interativo de Instalação (Formador partilha ecrã e guia passo a passo):**
  - **Windows:**
    - **Recurso:** Website oficial `git-scm.com/download/win`.
    - **Passos:** Executar o instalador. O formador deve guiar nas opções (componentes, editor padrão – sugerir VS Code ou Notepad++, ajustar o `PATH` – "Git from the command line and also from 3rd-party software").
  - **macOS:**
    - **Recurso:** Homebrew (`brew install git`) ou Xcode Command Line Tools.
    - **Passos:** Se já tiver Xcode, Git já pode estar presente. Se não, instalar Homebrew primeiro.
  - **Linux:**

- **Recurso:** Gestores de pacotes específicos (`sudo apt install git` para Debian/Ubuntu, `sudo yum install git` para Fedora/CentOS).
- **Passos:** Abrir terminal e executar o comando adequado.

- **Configuração Inicial Obrigatória (Global):**
  - **Formador explica:** Estas configurações são essenciais para identificar quem faz as alterações. Serão anexadas a CADA commit.
  - `git config --global user.name "O Seu Nome Completo"`: **Explicação:** Define o nome de autor que será gravado nos seus commits. Use o seu nome real ou um pseudónimo reconhecível.
  - `git config --global user.email "o.seu.email@exemplo.com"`: **Explicação:** Define o endereço de email associado aos seus commits. É importante que seja o mesmo email que usará no GitHub/GitLab para ligar os seus commits à sua conta.

- **Verificação da Instalação e Configuração:**
  - `git --version`: **Explicação:** Confirma se o Git está instalado e mostra a versão.
  - `git config --list`: **Explicação:** Lista todas as configurações do Git. Pedir para formandos procurarem as suas configurações `user.name` e `user.email`.

- **Resolução de Problemas Comuns:**
  - "Git not found"(erro no PATH).
  - Erros de permissão.
  - Problemas de conectividade de rede para download.

1.3. Primeiros Comandos Essenciais (2 horas)

- **O Ciclo de Vida de um Ficheiro no Git (Ilustrar com diagrama):**
  - **Working Directory (Área de Trabalho):**
    - **Definição:** É a pasta no seu computador onde os ficheiros do projeto estão e onde faz as edições.
    - **Estado:** Ficheiros podem ser "modified"(modificados, mas não adicionados para o próximo commit) ou "untracked"(novos ficheiros que o Git ainda não está a seguir).
  - **Staging Area (Área de Preparação/Index):**
    - **Definição:** Uma área intermédia onde você seleciona e organiza as alterações específicas que deseja incluir no seu próximo commit.
    - **Explicação:** "É como preparar o carrinho de compras antes de ir para a caixa. Só o que está no carrinho vai ser 'comprado' (commitado)."
  - **Local Repository (Repositório Local):**
    - **Definição:** A base de dados oculta (`.git` folder) que armazena todas as versões e histórico do seu projeto no seu computador.

- **Estado:** Ficheiros são "committed"(gravados no histórico).

- **Comandos Essenciais:**
  - `git init`:
    - **Definição:** Inicializa um novo repositório Git numa pasta existente. Cria a subpasta `.git`.
    - **Explicação:** "Isto transforma uma pasta comum num projeto Git rastreável."**Demonstração:** Criar uma pasta, entrar nela, `git init`, mostrar o `.git` oculto.
  - `git status`:
    - **Definição:** Mostra o estado atual da sua working directory e staging area. Indica quais ficheiros foram modificados, quais estão na staging area e quais não estão a ser rastreados.
    - **Explicação:** "É o vosso 'painel de controlo' para saber o que o Git 'vê'."**Demonstração:** Criar um ficheiro (untracked), modificar um existente (modified), `git add` (staged).
  - `git add <ficheiro>` / `git add .`:
    - **Definição:** Adiciona as alterações de um ficheiro (ou todos os ficheiros modificados e novos, se usar `.`) para a staging area.
    - **Explicação:** "Estou a dizer ao Git: 'Quero que estas alterações específicas sejam incluídas no meu próximo commit'."**Demonstração:** Fazer alterações, `git status`, `git add`, `git status` novamente.
  - `git commit -m "Mensagem do commit"`:
    - **Definição:** Grava todas as alterações que estão na staging area num novo snapshot (versão) no histórico do repositório local. Cada commit tem um SHA (identificador único), autor, data e mensagem.
    - **Explicação:** "Isto é o vosso 'guardar' permanente no histórico. Cada commit é um ponto seguro para o qual podem voltar."**Boas Práticas de Mensagens de Commit:**
      - **Regra de Ouro:** A primeira linha deve ser um sumário conciso (máx. 50-72 carateres), no imperativo (ex: "Adicionar funcionalidade", "Corrigir erro").
      - **Corpo (opcional):** Se necessário, uma linha em branco e depois um parágrafo mais detalhado sobre o *porquê* da alteração.
      - **Exemplos:** "feat: Adicionar autenticação de utilizador"vs. "Bug fix".

      Demonstração: Ciclo `edit -> add -> commit` várias vezes.
  - `git log`:
    - **Definição:** Exibe o histórico de commits do repositório.
    - **Explicação:** "Permite ver quem fez o quê e quando. Essencial para auditoria."**Demonstração:** Mostrar detalhes (autor, data, SHA). Introduzir `git log --oneline` para ver de forma mais compacta.

Exercício Prático (Durante a Sessão):

- **Exercício 1.1: O seu Primeiro Repositório Git Local:**
  - **Passo 1: Criar e Inicializar:**
    - No seu terminal (Git Bash/CMD/PowerShell/Terminal), crie uma pasta: `mkdir meu-primeiro-git-local`.
    - Entre na pasta: `cd meu-primeiro-git-local`.
    - Inicialize o repositório Git: `git init`.
    - *(Opcional: Mostre a pasta `.git` - pode ser necessário ativar "Mostrar ficheiros ocultos").*
  - **Passo 2: Adicionar o Primeiro Conteúdo:**
    - Crie um ficheiro de texto (pode ser com `touch info_pessoal.txt` ou pelo editor) chamado `info_pessoal.txt`.
    - Abra `info_pessoal.txt` no seu editor de texto (ex: VS Code) e adicione:

      ```
      Nome Completo: [O seu nome]
      Email: [O seu email]
      ```

    - Verifique o estado: `git status` (o ficheiro deve estar "untracked").
    - Adicione o ficheiro à staging area: `git add info_pessoal.txt`.
    - Verifique o estado novamente: `git status` (o ficheiro deve estar "changes to be committed").
    - Faça o seu primeiro commit: `git commit -m "feat: Adicionado ficheiro de informações pessoais"`.
  - **Passo 3: Fazer uma Alteração e um Segundo Commit:**
    - Abra `info_pessoal.txt` e adicione uma nova linha:

      ```
      Área de Interesse: Desenvolvimento Web
      ```

    - Crie um novo ficheiro chamado `hobbies.txt` e adicione alguns hobbies:

      ```
      1. Leitura
      2. Caminhadas
      3. Programar
      ```

    - Verifique o estado: `git status` (um modificado, um untracked).
    - Adicione ambos os ficheiros (modificado e novo) à staging area: `git add .` (usar `.` para adicionar tudo).
    - Faça o segundo commit: `git commit -m "feat: Adicionado hobbies e atualizado info pessoal"`.
  - **Passo 4: Ver o Histórico:**
    - Execute `git log` para ver os seus dois commits com detalhes.
    - Execute `git log --oneline` para uma visualização mais concisa.

—

Sessão Síncrona 2: GitHub e a Ponte para o Mundo Remoto (5 Horas)

Resumo da Sessão

Esta sessão concentra-se no **GitHub** como plataforma para repositórios remotos. Os formandos aprenderão a criar uma conta, a criar repositórios no GitHub e, crucialmente, a sincronizar o trabalho entre os seus repositórios locais e os repositórios remotos usando `git clone`, `git push` e `git pull`.

Conteúdo a Lecionar (Com Definições e Explicações para o Formador)

2.1. Introdução ao GitHub (1 hora)

- **O Que é o GitHub?**
  - **Definição:** A maior plataforma de hospedagem de repositórios Git do mundo. Funciona como uma rede social para desenvolvedores.
  - **Explicação:** "É onde os vossos projetos Git ganham vida online. Permite partilhar, colaborar e apresentar o vosso trabalho."
  - **Funcionalidades Chave:** Hospedagem de código, gestão de Pull Requests, Issues (gestão de tarefas/bugs), projetos (quadros Kanban), wikis, GitHub Pages (hospedagem de sites estáticos).
  - **Cenários de Uso:**
    - **Projetos Pessoais/Portfólio:** Mostrar o que fazem a potenciais empregadores.
    - **Projetos Open-Source:** Contribuir ou iniciar projetos de código aberto.
    - **Trabalho em Equipa:** Colaboração fluida com colegas.
- **Criação de Conta Gratuita no GitHub:**
  - **Recurso:** `github.com`.
  - **Passos:** Processo de registo, escolha de username, email, password. Resolver CAPTCHA.
  - **Configuração Inicial do Perfil:** Upload de foto, preencher bio, adicionar links pessoais/profissionais.
- **Interface Básica:** Visão geral da página inicial (feed de atividade), página de perfil, secção de repositórios.

2.2. Criar e Gerir Repositórios Remotos (2 horas)

- **Repositório Remoto:**
  - **Definição:** Uma cópia do seu repositório Git que vive num servidor remoto (neste caso, no GitHub), acessível via internet.

- **Explicação:** "É a versão 'na nuvem' do vosso projeto, o ponto central para a colaboração."

- **Criação de um Novo Repositório no GitHub:**
  - **Passos:** Navegar para o botão "New repository"ou `+` no canto superior direito.
  - **Opções Cruciais (Explicar cada uma):**
    - **Repository name:** Nome único para o seu projeto.
    - **Description (optional):** Breve sumário.
    - **Public/Private:**
      - **Público:** Visível para todos, ideal para open-source ou portfólio.
      - **Privado:** Apenas visível para si e colaboradores convidados.
    - **Initialize this repository with:**
      - `Add a README file`: **Importante:** Se for associar um repositório LOCAL JÁ EXISTENTE, **NÃO selecionar esta opção**, para evitar um conflito inicial. Se for começar um projeto DO ZERO no GitHub, pode selecionar.
      - `Add .gitignore`: **Explicação:** Ficheiro que diz ao Git para ignorar certos ficheiros ou pastas (ex: ficheiros temporários, segredos, dependências de pacotes).
      - `Choose a license`: Explicar brevemente que licenças definem como outros podem usar o seu código (abordado mais a fundo na Sessão 5).
  - **Após a criação:** Mostrar o ecrã com as instruções para ligar um repositório local.

- **Associar Repositório Local Existente a Remoto:**
  - `git remote add <nome_do_remote> <URL_do_repositorio>`:
    - **Definição:** Comando para adicionar um atalho (um "remote") para o URL de um repositório remoto ao seu repositório local.
    - **Explicação:** "Estamos a dizer ao Git: 'este URL representa o repositório remoto que vou usar'."
    - `origin` (**nome padrão**): Convenção para o remote principal de onde o projeto foi clonado ou para onde é empurrado.
  - `git remote -v`: **Definição:** Lista os remotes configurados (versão verbosa).
  - `git push -u origin main`:
    - **Definição:** Empurra (envia) os commits da sua branch local `main` para a branch `main` no remote chamado `origin`. O `-u` (ou `--set-upstream`) define o `origin/main` como o upstream padrão para a sua branch local `main`, facilitando futuros `push` e `pull`.
    - **Explicação:** "É o primeiro 'envio' do vosso código local para o GitHub."

- `git clone <URL_do_repositorio>`:
  - **Definição:** Cria uma cópia **completa** de um repositório remoto (incluindo todo o seu histórico) no seu computador.

- **Explicação:** "Se querem começar a trabalhar num projeto que já existe no GitHub, este é o comando que usam. Ele já configura o 'origin' para vocês."**Demonstração:** Clonar um repositório público de exemplo.

2.3. Sincronização de Repositórios (2 horas)

- `git push`:
  - **Definição:** Envia os commits que fez no seu repositório local (e que ainda não estão no remoto) para a branch correspondente no repositório remoto.
  - **Explicação:** "É o vosso 'upload' das alterações locais para a versão na nuvem, partilhando o vosso trabalho."
  - **Sintaxe:** `git push [remote_name] [branch_name]`. (Após o primeiro `push -u`, pode ser só `git push`).
  - **Cenários de Sucesso e Falha:** Mostrar um push bem-sucedido. Discutir o erro quando o remote tem commits que você não tem (leva a `git pull`).
- `git pull`:
  - **Definição:** Busca as alterações mais recentes do repositório remoto e integra-as automaticamente no seu repositório local. É um atalho para `git fetch` (obter as alterações) seguido de `git merge` (integrar as alterações).
  - **Explicação:** "É o vosso 'download' e 'atualização' do projeto com as últimas alterações dos colegas ou da versão principal."
  - **Sintaxe:** `git pull [remote_name] [branch_name]`. (Após o primeiro `push -u`, pode ser só `git pull`).
  - **Quando usar:** Antes de começar a trabalhar, ou se o `push` falhar por estar desatualizado.
- **Fluxo de Trabalho Básico de Sincronização (Ilustrar):**
  - `git pull` (para ter o mais recente) → Trabalhar → `git add` → `git commit` → `git push`.

Exercício Prático (Durante a Sessão):

- **Exercício 2.1: Publicando o seu Projeto no GitHub:**
  - **Passo 1: Criar Conta e Repositório Remoto:**
    - **Formador supervisiona:** Todos os formandos devem criar uma conta no GitHub durante a sessão.
    - No GitHub (interface web), cada formando criará um **novo repositório vazio** (sem `README.md` inicial) com o nome `portfolio-web-pessoal`. (Enfatizar que é "vazio"para este exercício).
  - **Passo 2: Preparar o Repositório Local (continuando o da Sessão 1):**
    - Vá para a pasta `meu-primeiro-git-local` (do Exercício 1.1).

- Crie um ficheiro `index.html` com uma estrutura HTML básica (ex: `<!DOCTYPE html><html><head><title>Meu Portfólio</title><link rel="stylesheet"href="Mundo!</h1></body></html>`).
- Crie um ficheiro `style.css` com uma regra básica (ex: `body { background-color: lightblue; }`).
- Adicione ambos os ficheiros: `git add .`.
- Faça o primeiro commit do projeto web: `git commit -m "feat: Estrutura inicial do portfolio web"`.

– **Passo 3: Ligar e Publicar:**

- No GitHub, na página do seu novo repositório `portfolio-web-pessoal`, copie o URL (HTTPS ou SSH, conforme preferência e configuração).
- No terminal (na pasta local), adicione o remote: `git remote add origin [URL_do_seu_repositorio]`.
- Publique o seu código: `git push -u origin main` (se a sua branch principal for `main`).
- **Verificação:** Peça aos formandos para abrirem o seu repositório no GitHub para confirmar que os ficheiros `index.html` e `style.css` apareceram.

• **Exercício 2.2: Clonar e Contribuir (Simulação de Colega):**

– **Passo 1: Clonar como se Fosse um Colega:**

- No seu ambiente de trabalho, crie uma **nova pasta separada** (ex: `meu-portfolio-clon`
- Entre nessa nova pasta: `cd meu-portfolio-clone`.
- Clone o **seu próprio** repositório `portfolio-web-pessoal` a partir do GitHub: `git clone [URL_do_seu_repositorio] .` (o `.` clona para a pasta atual).

– **Passo 2: Fazer uma Alteração no Clone:**

- Na pasta `meu-portfolio-clone`, abra o `index.html` e adicione um parágrafo logo abaixo do `<h1>`:

  ```
  <p>Este é o meu primeiro portfólio web desenvolvido com Git e GitHub!</p>
  ```

- Adicione a alteração: `git add .`.
- Faça commit: `git commit -m "feat: Adicionado parágrafo descritivo na página inicial"`.

– **Passo 3: Sincronizar as Alterações:**

- A partir da pasta `meu-portfolio-clone`, faça `git push origin main`.
- **Volte à sua primeira pasta original** (`portfolio-web-pessoal`).
- Execute `git pull origin main` para trazer as alterações feitas no "clone".
- **Verificação:** Abra o `index.html` na sua primeira pasta e confirme que o novo parágrafo foi adicionado.

—

Sessão Síncrona 3: Gerir o Histórico e Clientes Gráficos (5 Horas)

Resumo da Sessão

Esta sessão aprofunda a gestão do histórico de commits e introduz uma alternativa à linha de comando: os **clientes gráficos Git**. Os formandos aprenderão a usar `git log` de forma mais eficaz, a aplicar **tags** para marcar versões importantes, e a realizar operações básicas com um cliente gráfico.

Conteúdo a Lecionar (Com Definições e Explicações para o Formador)

3.1. Análise e Edição do Histórico de Commits (2 horas)

- **Aprofundamento de `git log`:**
  - **Formador explica:** `git log` é a vossa "câmera"do tempo. Estas opções ajudam a ver o que precisam.
  - `git log --oneline`: **Explicação:** Mostra cada commit numa única linha concisa (SHA abreviado e mensagem). Ideal para uma visão rápida.
  - `git log --graph`: **Explicação:** Desenha um gráfico ASCII do histórico de commits, útil para visualizar branches e merges.
  - `git log --all`: **Explicação:** Mostra o histórico de *todas* as branches, não apenas da atual.
  - `git log --decorate`: **Explicação:** Mostra as referências de branches e tags ao lado dos commits.
  - `git log --author="Nome do Autor"`: **Explicação:** Filtra commits feitos por um autor específico.
  - `git log -p`: **Explicação:** Mostra o "patch"(as diferenças linha a linha) introduzidas por cada commit. Útil para entender o que foi alterado.
- **Desfazer Alterações (Seguro):** `git revert`
  - **Definição:** `git revert <SHA_do_commit>` cria um *novo commit* que desfaz as alterações introduzidas pelo commit especificado.
  - **Explicação:** "É como 'anular' um commit, mas de forma segura, criando um novo registo no histórico em vez de apagar o original. Ideal para alterações já partilhadas."
  - **Diferença breve com `git reset` (apenas menção):** Explicar que `git reset` pode reescrever o histórico e é perigoso em repositórios partilhados, portanto não será abordado a fundo neste curso.
- `git tag`:
  - **Definição:** Uma "tag"é um marcador permanente e imutável que aponta para um commit específico no histórico. É frequentemente usado para marcar versões de lançamento (ex: `v1.0.0`).

- **Explicação:** "Pensem nisto como uma 'etiqueta' ou 'marco' importante no vosso código, para que seja fácil voltar a essa versão no futuro."
- **Tipos de Tags:** Ligeiramente mencionar "annotated tags"(com informação do autor, data, mensagem) vs. "lightweight tags"(apenas um ponteiro simples). Focar nas lightweight por simplicidade.
- **Comandos:**
  - `git tag <nome_da_tag>`: Cria uma lightweight tag no commit atual.
  - `git tag <nome_da_tag> <SHA_do_commit>`: Cria uma tag num commit específico.
  - `git tag`: Lista todas as tags locais.
  - `git tag -d <nome_da_tag>`: Apaga uma tag local.
  - `git push origin --tags`: Envia todas as tags locais para o repositório remoto (as tags não são enviadas automaticamente com `git push`).

3.2. Ficheiros `README.md` (1 hora)

- **A Importância do `README.md`:**
  - **Explicação:** "É a 'montra' e o 'manual de instruções' do vosso projeto. É a primeira coisa que alguém vê quando visita o vosso repositório no GitHub."
  - **Propósito:**
    - Apresentar o projeto (o que faz, para quem é).
    - Fornecer instruções de instalação/configuração.
    - Exemplos de uso.
    - Informações de contacto ou contribuição.
- **Sintaxe Básica de Markdown:**
  - **Definição:** Uma linguagem de marcação leve que permite formatar texto simples de forma fácil de ler e converter em HTML. Os ficheiros `README.md` usam Markdown.
  - **Elementos Essenciais (Demonstração e Prática Rápida):**
    - **Títulos:** `# H1`, `## H2`, `### H3` (um `#` para cada nível).
    - **Listas:** `* Item 1` ou `- Item 1` (não ordenada), `1. Item 1` (ordenada).
    - **Negrito:** `**texto**` ou `__texto__`.
    - **Itálico:** `*texto*` ou `_texto_`.
    - **Links:** `[Texto do Link](URL)`.
    - **Imagens:** `![Texto Alternativo](URL_da_Imagem)`.
    - **Blocos de Código:** Usar crases triplas (```linguagem_codigo```) para blocos de código (ex:```html ```).
    - `Inline Code`: Usar uma crase (`código`) para código no meio da frase.

3.3. Clientes Gráficos Git (2 horas)

- **O Que São e Por Que Usá-los?**
  - **Definição:** Aplicações com interfaces gráficas de utilizador (GUI) que simplificam a interação com o Git, abstraindo muitos comandos da linha de comando.
  - **Explicação:** "Se a linha de comando é como conduzir um carro com caixa manual, um cliente gráfico é como um carro automático. Mais fácil para quem está a começar, mas pode esconder alguns detalhes importantes."
  - **Vantagens:**
    - **Visualização:** Histórico de commits (grafos), branches, alterações de ficheiros.
    - **Facilidade de Uso:** Operações comuns com cliques (stage, commit, push, pull).
    - **Resolução de Conflitos Visual:** Ferramentas integradas para resolver conflitos mais facilmente.
  - **Desvantagens:**
    - Podem ocultar a complexidade subjacente do Git, dificultando a compreensão profunda.
    - Dependência da interface – se a ferramenta mudar, pode ser preciso reaprender.
    - Nem todas as operações avançadas estão sempre disponíveis via GUI.
- **Apresentação e Demonstração Prática:**
  - **Escolha:** O formador deve escolher **um ou dois** para demonstrar, para não sobrecarregar. Boas opções incluem:
    - **GitGraph (Extensão do VS Code):** Ótimo para quem já usa VS Code. Foca-se na visualização do histórico.
    - **GitKraken:** Interface muito intuitiva e visual. Gratuito para uso pessoal/open source.
    - **Sourcetree:** Popular, mais robusto, da Atlassian (criadores do Bitbucket, Jira).
  - **Demonstração de Funcionalidades Essenciais na GUI:**
    - Abrir um repositório existente.
    - Ver o `git status` graficamente.
    - Arrastar e soltar (ou usar botões) para `git add` (staging).
    - Escrever a mensagem do `git commit` e comitar.
    - Botões para `git push` e `git pull`.
    - Visualização da árvore de commits, branches e tags.
- **Configuração:** Apoiar os formandos para instalarem e configurarem o cliente gráfico da sua preferência.

Exercício Prático (Durante a Sessão):

- **Exercício 3.1: Gerir Histórico e Documentar:**
  - **Passo 1: Explorar o Histórico (Revisão):**
    - No seu repositório `portfolio-web-pessoal` (do Exercício 2.2), faça algumas pequenas alterações em diferentes ficheiros (ex: adicione um comentário HTML no `index.html`, mude uma cor no `style.css`).
    - Faça 2-3 commits com mensagens curtas (e propositadamente "menos informativas"como "Update", "Fix").
    - Use `git log --oneline --graph` para ver o histórico de forma concisa e visual.
    - **Discussão:** "O que acham destas mensagens de commit? São úteis para outros ou para vocês próprios no futuro?"
  - **Passo 2: Marcar uma Versão (Tag):**
    - Assuma que o estado atual do seu projeto é a "versão 1.0".
    - Adicione uma tag leve ao commit mais recente: `git tag v1.0.0`.
    - Liste as tags locais: `git tag`.
    - Envie a tag para o GitHub: `git push origin v1.0.0` (ou `git push --tags` para enviar todas as tags locais).
    - **Verificação:** Peça para os formandos verificarem a tag na página do seu repositório no GitHub (secção "Releases"ou "Tags").
  - **Passo 3: Criar um `README.md` Profissional:**
    - Crie (ou edite se já existir do Exercício 2.1) o ficheiro `README.md` na pasta raiz do seu repositório.
    - **Implementar os seguintes elementos usando Markdown:**
      - Um título principal (`# Meu Portfólio Web Pessoal`).
      - Uma breve descrição do que é o portfolio.
      - Uma secção "Como Usar"ou "Funcionalidades"com uma lista (ex: `* Página Inicial`, `* Contactos`).
      - Pelo menos um link (ex: para o seu perfil do GitHub).
      - Pelo menos um bloco de código (ex: exemplo de como clonar o repo: ```bash git clone [URL] ```).
    - Adicione (`git add README.md`) e faça commit (`git commit -m "docs: Adicionado README.md detalhado"`).
    - Faça `git push`.
    - **Verificação:** Pedir aos formandos para verem o seu `README.md` renderizado no GitHub.
- **Exercício 3.2: Experimentar o Cliente Gráfico:**
  - **Passo 1: Instalar e Abrir o Cliente Gráfico:**
    - Peça aos formandos para instalarem o cliente gráfico que preferirem/foi demonstrado (ex: GitKraken).
    - Abram o seu repositório `portfolio-web-pessoal` nesse cliente gráfico.

- **Passo 2: Realizar Operações Essenciais via GUI:**
  - Faça uma pequena alteração num ficheiro existente (ex: adicione mais um parágrafo no `index.html`).
  - **Usando a interface gráfica:**
    - Observe como o cliente gráfico mostra as alterações pendentes.
    - Clique para "staging"(adicionar) as alterações.
    - Escreva a mensagem do commit na caixa de texto apropriada e clique para comitar.
    - Clique no botão "Push"para enviar para o GitHub.
  - **Verificação:** Confirmar no GitHub que as alterações foram recebidas.
  - **Discussão:** "O que acharam da experiência com o cliente gráfico versus a linha de comando? Qual preferem para que tipo de tarefa?"

—

Sessão Síncrona 4: Branches e Colaboração (5 Horas)

Resumo da Sessão

Esta sessão aborda o conceito fundamental de **branches** para o desenvolvimento paralelo e colaborativo. Os formandos aprenderão a criar, alternar e fundir branches, simulando um fluxo de trabalho em equipa, e serão introduzidos ao processo de **Pull Requests** no GitHub.

Conteúdo a Lecionar (Com Definições e Explicações para o Formador)

4.1. Conceitos de Branches (2 horas)

- **O Que São Branches?**
  - **Definição:** No Git, uma branch é simplesmente um ponteiro leve e móvel para um dos commits. A branch principal (`main` ou `master`) é a linha de desenvolvimento padrão.
  - **Analogia/Explicação:** "Pensem numa branch como uma 'linha do tempo alternativa' para o vosso projeto. Permite-vos experimentar e desenvolver novas funcionalidades sem perturbar a 'linha principal' de produção."
  - **Visualização:** Utilizar diagramas para mostrar como as branches divergem e convergem do `main`.
- **Por Que Usar Branches?**
  - **Isolamento de Trabalho:** Desenvolver novas funcionalidades ou corrigir bugs em ambientes isolados.
  - **Colaboração Paralela:** Múltiplos desenvolvedores podem trabalhar em diferentes partes do projeto ao mesmo tempo.

- **Experimentação Segura:** Testar ideias sem o risco de quebrar o código principal.
- **Gestão de Versões:** Manter o código de produção estável enquanto o desenvolvimento continua.

- **Comandos de Gestão de Branches:**
  - `git branch <nome_da_branch>`: **Definição:** Cria uma nova branch, apontando para o commit atual. **Explicação:** "Não te moves para ela, apenas a crias."
  - `git branch`: **Definição:** Lista as branches locais. A branch atual é marcada com um asterisco.
  - `git checkout <nome_da_branch>`: **Definição:** Muda o seu "HEAD"(o ponteiro para a branch atual) para a branch especificada, atualizando a sua working directory para corresponder ao estado dessa branch. **Explicação:** "Movem-se para essa linha do tempo."
  - `git switch <nome_da_branch>`: **Definição:** É uma alternativa mais recente e mais clara ao `git checkout` para mudar de branch. (Sugerir o uso desta, mas mencionar `checkout` por ser mais comum em material antigo).
  - `git branch -d <nome_da_branch>`: **Definição:** Apaga uma branch local (só pode ser apagada se já tiver sido mergeada).
  - `git branch -D <nome_da_branch>`: **Definição:** Apaga uma branch local, mesmo que não tenha sido mergeada (uso cauteloso!).
  - `git push origin --delete <nome_da_branch_remota>`: **Definição:** Apaga uma branch do repositório remoto.

- **Fluxo de Trabalho Básico com Branches (Ilustrar e Demonstração ao Vivo):**

  `main` (estável) → Criar `feature-x` → Desenvolver e fazer commits em `feature-x` → Voltar para `main` → Mergear `feature-x` para `main`.

4.2. Fusão de Branches (`git merge`) (1.5 horas)

- **O Processo de Merge:**
  - **Definição:** Combina as alterações de uma branch para outra.
  - **Explicação:** "É como juntar duas linhas do tempo de volta numa só, integrando as alterações feitas em paralelo."

- **Tipos de Merge (Ilustrar com `git log --graph`):**
  - **Fast-Forward Merge:**
    - **Explicação:** Ocorre quando a branch de destino não teve novos commits desde que a branch de origem foi criada. O Git simplesmente move o ponteiro da branch de destino para o último commit da branch de origem, sem criar um novo commit de merge. O histórico permanece linear.
    - **Cenário:** `main` → `feature` (commits) → `checkout main` → `merge feature`.

- **3-Way Merge (ou Recursive Merge):**
  - **Explicação:** Ocorre quando ambas as branches (origem e destino) tiveram commits independentes desde um ponto comum ("base comum"). O Git cria um **novo commit de merge** que tem dois pais (os últimos commits de ambas as branches). O histórico forma um "diamante".
  - **Cenário:** `main` → `feature` (commits) E `main` (mais commits) → `checkout main` → `merge feature`.
  - **Importância:** É aqui que os conflitos podem acontecer.

4.3. Colaboração com Pull Requests (1.5 horas)

- **O Papel dos Pull Requests (PRs) no GitHub:**
  - **Definição:** Um Pull Request (ou Merge Request em outras plataformas como GitLab) é uma funcionalidade do GitHub que permite notificar os mantenedores de um repositório sobre alterações que você fez numa branch e deseja que sejam integradas na branch principal (ou outra).
  - **Explicação:** "É a forma mais comum de propor o vosso código para ser revisto e depois adicionado a um projeto em equipa."
  - **Benefícios:**
    - **Revisão de Código (Code Review):** Permite que outros desenvolvedores revejam o seu código, sugiram melhorias, encontrem bugs antes da integração.
    - **Discussão:** Espaço para comentários e perguntas sobre as alterações.
    - **Testes Automatizados:** Integração com CI/CD (Continuous Integration/Continuous Deployment) para testes automáticos antes do merge.
    - **Documentação:** O histórico de PRs serve como documentação das decisões de design.
- **Ciclo de Vida de um PR (Demonstração do Formador na Interface do GitHub):**
  - 1. **Desenvolver em Branch:** O desenvolvedor cria uma nova branch para a funcionalidade/correção e faz os seus commits.
  - 2. **Push da Branch:** A branch é enviada para o repositório remoto no GitHub (`git push origin <nome_da_branch>`).
  - 3. **Abrir o Pull Request:** No GitHub, aparece a opção "Compare & pull request". O desenvolvedor preenche o título, descrição, e adiciona revisores.
  - 4. **Revisão e Discussão:** Outros desenvolvedores (ou o formador) revisam o código, deixam comentários, pedem alterações.
  - 5. **Atualizações:** O autor do PR faz mais commits na sua branch (localmente) e faz `push` novamente – o PR é atualizado automaticamente.
  - 6. **Merge do PR:** Após a aprovação e testes, o PR é "merged"(fundido) na branch de destino (ex: `main`).
  - 7. **Apagar a Branch:** A branch de funcionalidade pode ser apagada no GitHub após o merge (opcional, mas boa prática).

Exercício Prático (Durante a Sessão):

- **Exercício 4.1: Desenvolvimento com Branches:**
  - **Passo 1: Criar uma Branch de Funcionalidade:**
    - No seu repositório `portfolio-web-pessoal` (certifique-se de que está na branch `main` e que fez `git pull origin main` para ter o mais recente), crie uma nova branch para adicionar uma secção "Projetos": `git branch feat/sec_projetos`.
    - Mude para essa nova branch: `git checkout feat/sec_projetos`.
  - **Passo 2: Desenvolver na Branch `feat/sec_projetos`:**
    - No `index.html` (dentro da branch `feat/sec_projetos`), adicione uma nova secção `<section id="projetos>` com um título `<h2>Os Meus Projetos</h2>` e 2-3 parágrafos de texto fictício ou descrições de projetos imaginários.
    - No `style.css`, adicione algumas regras CSS simples para estilizar essa nova secção (ex: `section#projetos { padding: 20px; background-color: #f0f0f0; }`).
    - Adicione as alterações: `git add .`.
    - Faça commit: `git commit -m "feat: Adicionada secção 'Projetos' ao portfólio"`.
    - Envie a branch para o GitHub: `git push origin feat/sec_projetos`.
  - **Passo 3: Criar uma Branch de Correção (Simultânea):**
    - Mude de volta para a branch `main`: `git checkout main`.
    - Crie uma nova branch para uma correção rápida: `git branch fix/corrigir_link_css`.
    - Mude para essa branch: `git checkout fix/corrigir_link_css`.
    - No `index.html`, simule uma correção: se o link para `style.css` estiver como `<link rel="stylesheet"href="style.css>`, altere-o para `<link rel="stylesheet"href="./style.css>` (ou vice-versa, para simular uma pequena alteração que poderia ser uma correção).
    - Adicione a alteração: `git add .`.
    - Faça commit: `git commit -m "fix: Corrigido caminho relativo do CSS no index"`.
    - Envie a branch para o GitHub: `git push origin fix/corrigir_link_css`.
- **Exercício 4.2: Fusão das Branches na `main`:**
  - **Passo 1: Voltar à `main` e Mergear a Correção:**
    - Mude de volta para a branch `main`: `git checkout main`.
    - Faça `git pull origin main` (É CRUCIAL para ter a `main` local atualizada antes do merge, evitando conflitos iniciais).
    - Funda a branch `fix/corrigir_link_css` na `main`: `git merge fix/corrigir_link_css`.
    - **Verificação:** O formador e os formandos devem observar o terminal para ver o tipo de merge (provavelmente "fast-forward").
    - Faça `git push origin main`.
  - **Passo 2: Mergear a Funcionalidade:**

- Na branch `main`, faça `git merge feat/sec_projetos`.
- **Verificação:** O formador e os formandos devem observar o terminal para ver o tipo de merge (se houve commits na `main` desde a criação da `feat`, será um 3-way merge).
- Faça `git push origin main`.

– **Visualização:** Peça para os formandos usarem `git log --graph --oneline --all` e/ou o cliente gráfico para visualizar como as branches foram fundidas no histórico.

—

Sessão Síncrona 5: Resolução de Conflitos e Boas Práticas Profissionais (5 Horas)

Resumo da Sessão

A última sessão concentra-se num aspeto crítico: a **resolução de conflitos**, que inevitavelmente surgem em ambientes colaborativos. Os formandos aprenderão a identificar e resolver conflitos. A sessão terminará com uma discussão sobre as boas práticas de colaboração e a importância das normas e da ética no uso de Git e GitHub em contexto profissional.

Conteúdo a Lecionar (Com Definições e Explicações para o Formador):

5.1. Resolução de Conflitos (3 horas)

- **O Que São Conflitos de Merge?**
  - **Definição:** Um conflito de merge ocorre quando o Git não consegue integrar automaticamente as alterações de duas branches porque ambas as branches modificaram as mesmas linhas do mesmo ficheiro, ou uma branch apagou um ficheiro que a outra modificou.
  - **Explicação:** "O Git é inteligente, mas não consegue ler mentes. Quando há um 'choque' nas alterações, ele pede a vossa ajuda para decidir como resolver."
  - **Cenários Comuns:**
    - Duas pessoas editam a mesma linha de código em branches diferentes.
    - Uma pessoa apaga um ficheiro, outra pessoa modifica-o.
    - Duas pessoas criam ficheiros com o mesmo nome.
- **Identificação de Conflitos:**
  - **Mensagem no Terminal:** O Git avisará com `CONFLICT (content): Merge conflict in <file>` e dirá que o merge falhou.
  - `git status`: Indicará "unmerged paths"e listará os ficheiros em conflito.
  - **Marcas de Conflito nos Ficheiros:** Os ficheiros em conflito serão modificados com marcadores especiais:

```
<<<<<<< HEAD
// Sua versão das alterações
=======
// A versão da outra branch
>>>>>>> nome-da-outra-branch
```

Explicação: "As linhas entre `<<<<<<< HEAD` e `=======` são as vossas alterações. As linhas entre `=======` e `>>>>>>>` são as alterações da branch que estão a tentar incorporar."

- **Processo de Resolução Manual (Linha de Comando):**
  - **Passo 1: Identificar:** Use `git status` para ver os ficheiros em conflito.
  - **Passo 2: Editar o Ficheiro:** Abra o ficheiro em conflito no seu editor de texto.
    - **Decisão:** Escolha qual versão quer manter (a sua, a do outro, ou uma combinação de ambas).
    - **Ação:** Apague as linhas das marcas de conflito (`<<<<<<<`, `=======`, `>>>>>>>`) e mantenha apenas o código final desejado.
  - **Passo 3: Marcar como Resolvido:** `git add <ficheiro_resolvido>`. **Explicação:** "Isto diz ao Git que você já tratou do conflito neste ficheiro e ele está pronto para ser commitado."
  - **Passo 4: Completar o Merge:** `git commit`. O Git irá pré-preencher uma mensagem de commit de merge. Pode aceitá-la ou editá-la para ser mais descritiva.
- **Resolução de Conflitos via Cliente Gráfico:**
  - **Explicação:** "Muitos clientes gráficos têm 'ferramentas de merge' visuais que tornam este processo muito mais fácil."
  - **Demonstração:** Mostrar como a ferramenta de merge do cliente gráfico (ex: GitKraken, VS Code) apresenta as alterações lado a lado e permite aceitar blocos de código com cliques.

5.2. Boas Práticas e Colaboração Eficaz (1 hora)

- **Princípios da Colaboração com Git:**
  - **Comunicação Clara:**
    - **Explicação:** "Git é uma ferramenta, mas a comunicação humana é a mais importante. Falem com os vossos colegas!"
    - **Exemplos:** Antes de iniciar uma funcionalidade, avisar a equipa. Discutir grandes alterações antes de as codificar.
  - **Frequência de Commits:**
    - **Explicação:** "Façam commits pequenos e focados. Cada commit deve resolver um problema ou adicionar uma pequena funcionalidade."
    - **Vantagens:** Mais fácil de depurar, reverter, e rever.

- `git pull` **Antes de Começar e Antes de** `git push`:
  - **Explicação:** "É uma regra de ouro: puxem as últimas alterações antes de começar a codificar numa branch e, crucialmente, antes de fazerem `push` para evitar conflitos."
- **Revisão de Código (Code Review):**
  Explicação: "Permitir que outros membros da equipa leiam, compreendam e ofereçam feedback sobre o vosso código através de Pull Requests. Melhora a qualidade, encontra bugs e partilha conhecimento."

- **Etiqueta no Desenvolvimento Colaborativo (especialmente Open Source):**
  - **Respeito:** Pelo trabalho dos outros, pela convenção de código do projeto.
  - **Concisão:** Nas discussões e Pull Requests.
  - **Responsabilidade:** Ser responsável pelas alterações que submetem.
  - **Documentação:** Manter o `README.md` atualizado, usar mensagens de commit informativas.

5.3. Normas e Regulamentos Aplicáveis (1 hora)

- **Propriedade Intelectual e Licenças de Software:**
  - **Definição de Propriedade Intelectual:** Direitos sobre criações da mente (código, design).
  - **Licenças de Software:**
    - **Definição:** Contratos que definem como o software pode ser usado, modificado e distribuído.
    - **Explicação:** "Se vocês colocam um projeto no GitHub, uma licença diz a outros o que eles podem (e não podem) fazer com o vosso código."
    - **Tipos de Licenças (Breve Visão Geral):**
      - **Permissivas (Ex: MIT, Apache):** Permitem quase tudo, incluindo uso em projetos comerciais, desde que a licença seja mantida.
      - **Copyleft (Ex: GPL):** Exigem que projetos derivados também usem a mesma licença, promovendo o open-source.
    - **Importância:** Proteger os seus direitos e informar os utilizadores do seu código.
  - **Como adicionar uma Licença:** Geralmente um ficheiro `LICENSE` na raiz do repositório.

- **Segurança e Privacidade:**
  - **Regra Crucial: NUNCA** commitar credenciais (passwords, chaves de API, tokens de acesso) ou dados sensíveis diretamente no repositório Git. São públicos!
  - **Uso de** `.gitignore`:
    - **Definição:** Um ficheiro de texto na raiz do seu repositório que lista ficheiros e pastas que o Git deve ignorar (não rastrear).

- **Exemplos:** `node_modules/`, `.log`, `.env` (para variáveis de ambiente/segredos), `.DS_Store` (Mac), `Thumbs.db` (Windows).
- **Explicação:** "É o vosso 'filtro' para manter coisas que não devem ir para o repositório, como segredos ou ficheiros temporários."

- **Respeito por Regras Definidas:**
  - **Contributing Guidelines (`CONTRIBUTING.md`):** Explicar que muitos projetos open-source têm este ficheiro com regras sobre como contribuir (formato de commits, estilo de código, etc.).
  - **Explicação:** "Quando contribuem para um projeto, sigam as regras deles. Isso facilita a integração do vosso trabalho."

Exercício Prático (Durante a Sessão):

- **Exercício 5.1: Desafio de Resolução de Conflitos (Em Pares/Grupos Pequenos se possível):**
  - **Passo 1: Preparar o Cenário de Conflito:**
    - Todos os formandos devem garantir que a sua branch `main` está atualizada: `git pull origin main`.
    - Cada formando criará uma nova branch: `git checkout -b desafio-conflito-[SeuNo`
    - No ficheiro `index.html`, **ambos os formandos (ou o formando e o formador)** irão editar **a mesma linha ou secção**, introduzindo conteúdos diferentes e conflitantes.
      - **Exemplo:**
      - Formando A, na sua `desafio-conflito-A`: Altera `<h1>Bem-vindo ao Meu Portfólio!</h1>` para `<h1>Meu Portfólio de [Área Profissional]</h1>`.
      - Formando B, na sua `desafio-conflito-B`: Altera `<h1>Bem-vindo ao Meu Portfólio!</h1>` para `<h1>Explore o Meu Trabalho Online!</h1>`.
    - Cada um faz commit da sua alteração na sua própria branch `desafio-conflito`: `git commit -m "feat: Nova versão do título principal"`.
    - Cada um faz `git push origin desafio-conflito-[SeuNome]`.
  - **Passo 2: Provocar e Resolver o Conflito:**
    - **Primeiro Formando a Mergear:**
      - Muda para `main`: `git checkout main`.
      - Faz `git pull origin main` (para ter a `main` mais recente).
      - Faz `git merge desafio-conflito-[SeuNome_dele]`. Este deverá ser um merge sem conflito (fast-forward, se a `main` não tiver sido alterada entretanto).
      - Faz `git push origin main`.
    - **Segundo Formando a Mergear (irá criar o conflito):**
      - Muda para `main`: `git checkout main`.
      - **CRÍTICO:** Faz `git pull origin main` (agora terá a alteração do primeiro formando na sua `main` local).

- Tenta fazer `git merge desafio-conflito-[SeuNome_dele]`. **O Git irá acusar um conflito!**
- **Resolução Guiada (Formador acompanha passo a passo):**
- `git status` (ver ficheiros em conflito).
- Abrir `index.html` no editor de texto e identificar as marcas de conflito.
- Decidir qual versão manter ou combinar as duas (ex: «h1¿Bem-vindo ao Meu Portfólio Online de [Área Profissional]!¡/h1¿").
- Remover as marcas (`<<<<<<<`, `=======`, `>>>>>>>`).
- Marcar o ficheiro como resolvido: `git add index.html`.
- Completar o commit de merge: `git commit`. (Aceitar a mensagem padrão ou editá-la para descrever a resolução).
- Fazer `git push origin main`.

◦ **Verificação:** Peça a todos para verem o histórico da branch `main` no GitHub (`git log --graph` ou cliente gráfico) para ver o commit de merge do conflito.

—

Trabalho Prático Final (Realizar Fora da Sessão)

- **Objetivo:** Consolidar todas as competências adquiridas num projeto prático.
- **Descrição:** Os formandos deverão criar um **novo repositório Git e publicá-lo no GitHub** para um "Mini-Projeto Pessoal"à sua escolha (ex: um pequeno site pessoal com 2-3 páginas, uma calculadora web simples, uma aplicação de lista de tarefas em HTML/CSS/JS, um diário de estudos markdown).
  - O projeto deve ter um **histórico de commits limpo e significativo**, demonstrando commits pequenos e com boas mensagens (`feat:`, `fix:`, `docs:`).
  - Deve utilizar **pelo menos duas branches** para o desenvolvimento de diferentes funcionalidades ou correções. As branches devem ser fundidas na branch `main` após a sua conclusão.
  - Deverá incluir um **ficheiro** `README.md` bem estruturado e informativo, usando Markdown, explicando o projeto e como ele funciona.
  - **Cenário Extra (Opcional/Desafio para os mais rápidos):** Tentar simular e resolver um pequeno conflito *propositadamente* no seu próprio projeto (ex: criar um conflito entre duas das suas próprias branches e resolvê-lo antes de fundir na `main`), para demonstrar a capacidade de o gerir de forma independente.
  - **Adicionar um** `.gitignore`**:** Incluir um ficheiro `.gitignore` para excluir ficheiros temporários, de cache ou de configuração (ex: `.env`, `node_modules/`, `.vscode/`).

  - **Adicionar uma Licença:** Escolher e adicionar um ficheiro de licença (ex: MIT License) ao repositório para definir como outros podem usar o código.

- **Entrega:** O projeto será entregue através do **link do repositório GitHub** do formando.