Plano de Sessão Síncrona Detalhado: UC00617 Utilizar Serviços Git e GitHub (25 Horas)

O Seu Nome / Nome da Instituição

July 7, 2025

Conteúdo do Curso

- Sessão 1: Desmistificando o Controlo de Versões e Primeiros Passos com Git
- Sessão 2: GitHub e a Ponte para o Mundo Remoto
- Sessão 3: Gerir o Histórico e Clientes Gráficos
- Sessão 4: Branches e Colaboração
- Sessão 5: Resolução de Conflitos e Boas Práticas Profissionais
- Trabalho Prático Final

Sessão 1.1: O Que é Controlo de Versões?

A "Máquina do Tempo" do Código

- Um sistema que **regista e gere alterações** em ficheiros ao longo do tempo.
- Permite **recuperar qualquer versão anterior** do projeto.
- Evita a confusão de múltiplos ficheiros "final_final_v3.docx".

Sessão 1.2: Porquê é Indispensável?

Vantagens no Desenvolvimento de Software

- **Rastreabilidade:** Saber quem, o quê, quando e porquê em cada alteração.
- **Colaboração Eficaz:** Múltiplos desenvolvedores trabalham no mesmo código sem conflitos diretos.
- **Recuperação de Erros:** Voltar rapidamente a uma versão estável em caso de bugs.
- **Experimentação Segura:** Testar novas ideias isoladamente, sem afetar o código principal.
- **Histórico Completo:** Documenta a evolução do projeto.

Sessão 1.3: Tipos de Sistemas de Controlo de Versões

Centralizados vs. Distribuídos

Sistemas Centralizados (CVCS):

- Servidor central armazena todo o histórico.
- Exemplos: SVN, CVS.
- **Desvantagem:** Ponto único de falha; dependência de rede.

Sistemas Distribuídos (DVCS):

- Cada desenvolvedor tem uma **cópia completa** do repositório.
- **Exemplos: Git**, Mercurial.
- **Vantagens:** Trabalho offline, redundância, rapidez.

Sessão 1.4: O Ciclo de Vida no Git

Working Directory, Staging Area e Local Repository

- **Working Directory:** Onde os ficheiros estão e são editados. (*modified / untracked*)
- **Staging Area (Index):** Área intermédia para preparar as alterações que serão incluídas no próximo commit.
- **Local Repository:** Base de dados oculta (`.git`) com todo o histórico do projeto no seu computador. (*committed*)

Sessão 1.5: Instalação e Configuração do Git

Preparando o Ambiente

- **Instalação:**
  - Faça o download em `git-scm.com/downloads`.
  - Siga as instruções específicas para o seu sistema operativo (Windows, macOS, Linux).
  - Verifique a instalação: `git --version`.
- **Configuração Inicial (Crucial!):**
  - Identifica suas alterações nos commits.
  - `git config --global user.name "Seu Nome Completo"`
  - `git config --global user.email "seu.email@exemplo.com"`
  - Verifique: `git config --list`

- `git init`: Inicializa um novo repositório Git na pasta atual.
- `git status`: Mostra o estado dos ficheiros (modificados, na staging area, não rastreados).
- `git add <ficheiro>` ou `git add .`: Adiciona alterações para a staging area.
- `git commit -m "Mensagem do commit"`: Grava as alterações da staging area no histórico.
  - Mensagens claras e concisas são uma boa prática!
- `git log`: Exibe o histórico de commits. Use `--oneline` para uma vista compacta.

Sessão 1: Exercício Prático

O seu Primeiro Repositório Git Local

- **Passo 1:** Crie e inicialize uma pasta como repositório Git (`git init`).
- **Passo 2:** Crie e modifique ficheiros (`info_pessoal.txt`, `hobbies.txt`).
- **Passo 3:** Adicione as alterações (`git add .`) e faça múltiplos commits com mensagens descritivas (`git commit -m "... Resultados..."`).
- **Passo 4:** Explore o histórico de commits com `git log` e `git log --oneline`.

Sessão 2.1: Introdução ao GitHub

A Plataforma de Colaboração Global

- **O que é o GitHub?**
  - A maior plataforma de hospedagem de repositórios Git.
  - Uma rede social para desenvolvedores e projetos open-source.
- **Funcionalidades Chave:**
  - Hospedagem de código (repositórios remotos).
  - Gestão de Pull Requests (Revisão de Código).
  - Issues (rastreamento de bugs/tarefas).
  - Wikis, GitHub Pages.

Figure: Logotipo do GitHub. Fonte: (Verifique e atualize a fonte se necessário).

Sessão 2.2: Repositórios Remotos

Seu Projeto na Nuvem

- **Definição:** Uma cópia do seu repositório Git que vive num servidor acessível pela internet (no caso, o GitHub).
- Serve como o **ponto central** para a colaboração em equipa.
- Permite **partilhar** o seu trabalho e **colaborar** com outros, independentemente da localização.

Figure: Ícone de sincronização na nuvem. Fonte: (Verifique e atualize a fonte se necessário).

Sessão 2.3: Criar Repositórios no GitHub

Configurações Iniciais

- Acesse `github.com` e faça login.
- Clique em "New repository" ou no sinal +.
- **Opções Importantes:**
  - **Repository name:** Nome único e descritivo.
  - **Public/Private:** Quem pode ver o seu código.
  - **Initialize this repository with:**
    - **NÃO selecione "Add a README file"** se for ligar um repositório local existente!
    - `Add .gitignore`: Ficheiro para ignorar certos tipos de ficheiros (abordado depois).

Sessão 2.4: Sincronização Local ↔ Remoto

Comandos Essenciais

- **Ligar Repositório Local ao Remoto:**
  - `git remote add origin <URL_do_repositorio>`: Adiciona um atalho para o repositório remoto.
  - `git remote -v`: Lista os remotes configurados.
- **Enviar para o Remoto:**
  - `git push -u origin main`: Envia os commits locais para o GitHub pela primeira vez.
  - `git push`: (depois do primeiro push -u) Envia commits posteriores.
- **Clonar um Repositório Remoto:**
  - `git clone <URL_do_repositorio>`: Cria uma cópia local completa de um repositório existente no GitHub.

Sessão 2.5: Puxar e Sincronizar

Manter o Repositório Atualizado

- **Obter Alterações do Remoto:**
  - `git pull`: Busca as alterações mais recentes do repositório remoto e integra-as automaticamente no seu repositório local.
  - É uma combinação de `git fetch` (obter) e `git merge` (integrar).
- **Fluxo de Trabalho Básico de Sincronização:**
  - `git pull` (ao iniciar o trabalho ou para atualizar)
  - Trabalhar e fazer commits localmente (`git add`, `git commit`)
  - `git push` (para partilhar o seu trabalho)

Sessão 3.1: Análise Detalhada do Histórico

Explorando o `git log`

- `git log --oneline`: Resumo conciso dos commits.
- `git log --graph`: Desenha um grafo ASCII do histórico (útil com branches!).
- `git log --all`: Mostra histórico de todas as branches.
- `git log -p`: Mostra as alterações (diff) de cada commit.
- `git log --author="Seu Nome"`: Filtra por autor.

Sessão 3.2: Desfazer Alterações de Forma Segura

O Comando `git revert`

- `git revert <SHA_do_commit>`:
  - Cria um **novo commit** que desfaz as alterações de um commit anterior.
  - **Vantagem:** Não reescreve o histórico, sendo seguro para alterações já partilhadas.
  - Útil para "anular" um erro sem apagar o registo original.

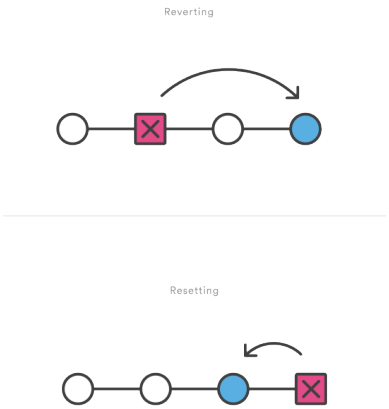

Figure: O `git revert` adiciona um novo commit que desfaz ações. Fonte: Wikimedia Commons, Autor: Git for Computer Scientists, Licença: Domínio Público.

Sessão 3.3: Marcar Versões Importantes

Usando `git tag`

- Uma **tag** é um marcador permanente e imutável para um commit específico.
- Ideal para marcar **versões de lançamento** (ex: `v1.0.0`, `v2.0.0-beta`).
- **Comandos:**
  - `git tag <nome_da_tag>`: Cria uma tag no commit atual.
  - `git tag`: Lista as tags locais.
  - `git push origin --tags`: Envia as tags locais para o repositório remoto.

Sessão 3.4: A Importância do `README.md`

O Cartão de Visitas do seu Projeto

- É a **primeira coisa que alguém vê** ao visitar seu repositório no GitHub.
- Fornece uma **descrição rápida** do projeto.
- Inclui **instruções essenciais**: instalação, uso, contribuição.
- É escrito em **Markdown**.

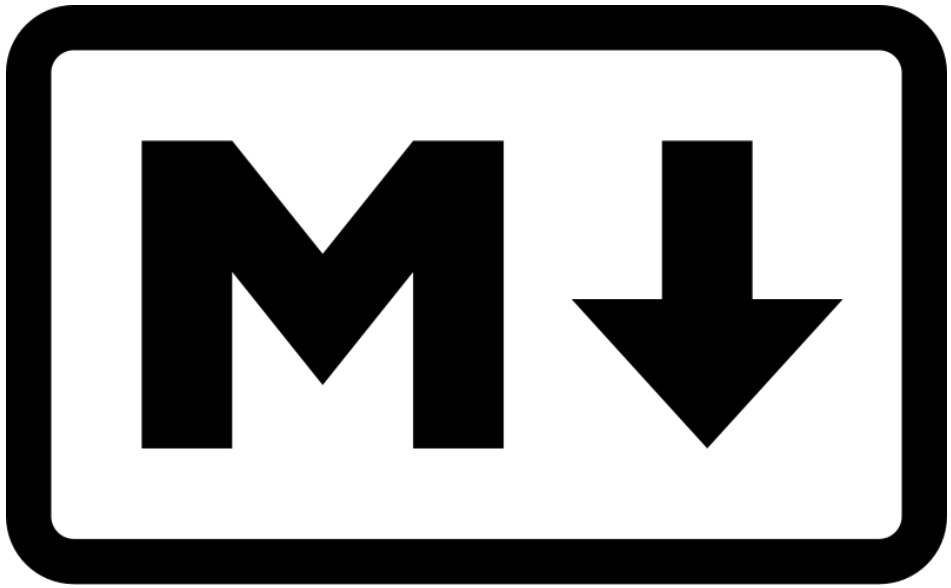

Figure: Um bom README é essencial para qualquer projeto. Fonte: Wikimedia Commons, Autor: Tio-tui, Licença: CC0 1.0 Universal.

Sessão 3.6: Clientes Gráficos Git (GUI)

Alternativas à Linha de Comando

- Aplicações com interface gráfica para interagir com o Git.
- **Vantagens:**
  - Visualização intuitiva do histórico (grafos de branches).
  - Operações comuns por cliques (stage, commit, push, pull).
  - Ferramentas visuais para resolução de conflitos.
- **Desvantagens:** Podem ocultar a complexidade do Git.
- **Exemplos:** GitKraken, Sourcetree, Git Extensions (VS Code).

Figure: Visão da branch principal no Sublime Merge. Fonte: (Verifique e atualize a fonte se necessário).

Sessão 3: Exercício Prático

Gerir Histórico e Documentar

- Explore o histórico do seu `portfolio-web-pessoal` com `git log --graph`.
- Marque uma versão (ex: `v1.0.0`) com `git tag` e envie para o GitHub.
- Crie/Edite o `README.md` usando Markdown (título, descrição, lista, link, bloco de código).
- Faça o commit e `git push` do `README.md`.
- **Desafio:** Instale e experimente um cliente gráfico Git, realizando um commit e push através dele.

Sessão 4.1: O Que São Branches?

Linhas do Tempo Alternativas

- No Git, uma branch é um **ponteiro leve para um commit**.
- A branch `main` (ou `master`) é a linha de desenvolvimento principal.
- **Analogia:** Pense como uma "ramificação" do seu projeto, onde pode trabalhar sem afetar a linha principal.

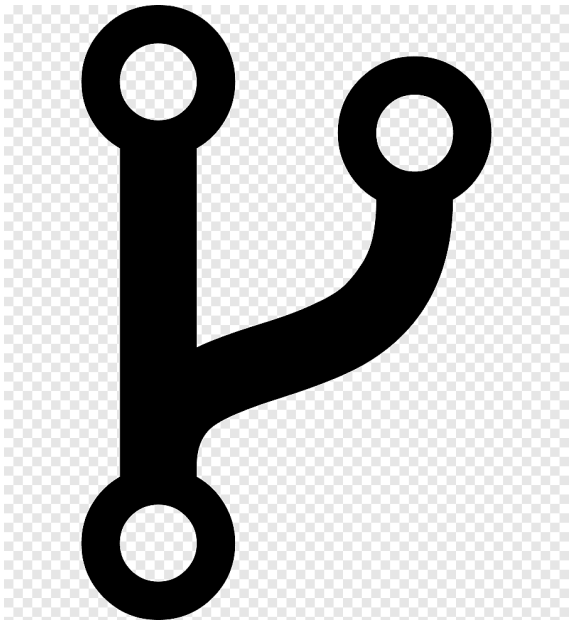

Figure: Diagrama de ramificação no Git. Fonte: (Verifique e atualize a fonte se necessário).

Sessão 4.2: Porque Usar Branches?

Desenvolvimento Paralelo e Seguro

- **Isolamento de Trabalho:** Desenvolva funcionalidades ou corrija bugs em ambientes separados.
- **Colaboração Eficaz:** Múltiplos desenvolvedores podem trabalhar em funcionalidades diferentes simultaneamente.
- **Experimentação Segura:** Teste ideias sem o risco de quebrar o código principal estável.
- **Gestão de Versões:** Mantenha o código de produção intocado durante o desenvolvimento de novas versões.

Sessão 4.3: Comandos de Gestão de Branches

Criação, Alternância e Remoção

- `git branch <nome_da_branch>`: Cria uma nova branch.
- `git branch`: Lista as branches locais (o asterisco indica a atual).
- `git checkout <nome_da_branch>`: Muda para a branch especificada.
- `git switch <nome_da_branch>`: Alternativa moderna e mais clara para mudar de branch.
- `git branch -d <nome_da_branch>`: Apaga uma branch local (se já foi mergeada).
- `git push origin --delete <nome_da_branch_remota>`: Apaga uma branch no repositório remoto.

Sessão 4.4: Fusão de Branches (`git merge`)

Combinando Linhas de Desenvolvimento

- **Definição:** Integra as alterações de uma branch para outra.
- **Tipos de Merge:**
  - **Fast-Forward Merge:** Se a branch de destino não teve novos commits, o Git apenas move o ponteiro. Histórico linear.
  - **3-Way Merge (Recursive):** Se ambas as branches tiveram commits independentes, o Git cria um **novo commit de merge** com dois pais. O histórico forma um "diamante".

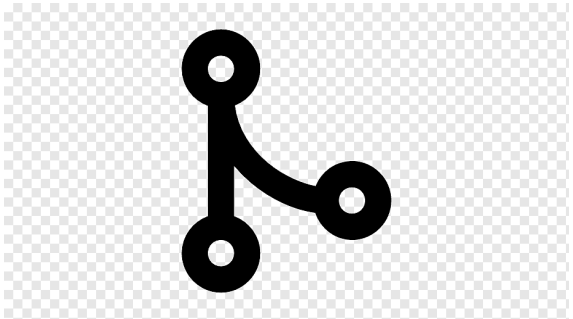

Figure: Diagrama de fusão (merge) no Git. Fonte: (Verifique e atualize a fonte se necessário).

Sessão 4.5: Colaboração com Pull Requests (PRs)

O Coração da Colaboração no GitHub

- **O Que São PRs?**
  - Um pedido para integrar as suas alterações numa branch principal de um repositório remoto.
  - Permite **revisão de código** e discussão antes do merge.
- **Benefícios:**
  - Melhora a qualidade do código através de feedback.
  - Deteta bugs antecipadamente.
  - Serve como documentação das decisões.
  - Permite testes automatizados (CI/CD).

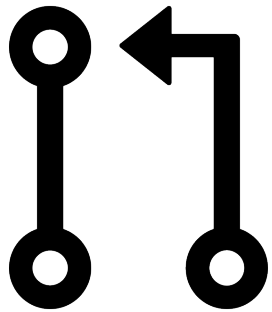

Figure: Diagrama de fusão (merge) no Git. Fonte: (Verifique e atualize a fonte se necessário).

Sessão 4.6: Ciclo de Vida de um Pull Request

Passos na Prática

1. Desenvolver em uma nova branch.
2. Fazer `git push` da branch para o GitHub.
3. Abrir o Pull Request no GitHub.
4. Revisão de Código e Discussão (comentários, pedidos de alteração).
5. Fazer mais commits/pushes na branch se necessário (PR atualiza-se).
6. Merge do PR após aprovação e testes.
7. Apagar a branch (opcional, mas boa prática).

Sessão 4: Exercício Prático

Desenvolvimento com Branches e Merges

- Garanta que sua `main` local está atualizada.
- Crie uma branch `feat/sec_projetos`. Faça alterações no `index.html` e `style.css`, comite e faça `git push`.
- Mude para a `main` e crie uma branch `fix/corrigir_link_css`. Faça uma pequena correção, comite e faça `git push`.
- Volte para `main`, faça `git pull`.
- Mergee `fix/corrigir_link_css` para `main`.
- Mergee `feat/sec_projetos` para `main`.
- Visualize o histórico final com `git log --graph --oneline --all`.

Sessão 5.1: O Que São Conflitos de Merge?

Onde o Git Precisa da Sua Ajuda

- Ocorrem quando o Git não consegue integrar alterações **automaticamente**.
- Geralmente, porque **duas branches modificaram as mesmas linhas** do mesmo ficheiro.
- Ou quando um ficheiro é apagado numa branch e modificado noutra.
- O Git irá parar o processo de merge e pedir a sua intervenção.

- **Terminal:** Mensagem `CONFLICT (content): Merge conflict in <file>`.
- `git status`: Indicará "unmerged paths".
- **Ficheiros em Conflito:** Contêm marcadores especiais:

Exemplo de Marcadores de Conflito

```
<<<<<<< HEAD // Seu código ======= // Código da outra branch >>>>>>> nome-da-outra-branch
```

Sessão 5.3: Processo de Resolução de Conflitos

Passos na Linha de Comando

1. **Identificar:** Use `git status`.
2. **Editar o Ficheiro:** Abra o ficheiro em conflito e decida qual versão manter (ou combine-as). **Remova todos os marcadores de conflito!**
3. **Marcar como Resolvido:** `git add <ficheiro_resolvido>`. Isto diz ao Git que o conflito foi tratado.
4. **Completar o Merge:** `git commit`. O Git irá pré-preencher uma mensagem de commit de merge.

Dica: Clientes gráficos (ou IDEs) oferecem ferramentas visuais para facilitar este processo!

Sessão 5.4: Boas Práticas de Colaboração

Trabalhando Eficazmente com a Equipa

- **Comunicação Clara:** Fale com sua equipa sobre as alterações antes de as fazer.
- **Commits Pequenos e Focados:** Cada commit deve resolver um problema ou adicionar uma pequena funcionalidade.
- `git pull` **Frequentemente:** Atualize seu repositório local antes de começar a trabalhar e antes de fazer `git push`.
- **Revisão de Código (Pull Requests):** Crucial para qualidade, feedback e partilha de conhecimento.
- **Mensagens de Commit Descritivas:** Útil para o futuro você e para a sua equipa.

Sessão 5.5: Propriedade Intelectual e Licenças

Definindo Como Seu Código Pode Ser Usado

- **Propriedade Intelectual:** Seus direitos sobre o código que cria.
- **Licenças de Software:**
  - Contratos que definem como seu software pode ser usado, modificado e distribuído.
  - **Permissivas (Ex: MIT, Apache):** Permitem uso amplo, inclusive em projetos comerciais.
  - **Copyleft (Ex: GPL):** Exigem que projetos derivados usem a mesma licença.
- Adicione um ficheiro `LICENSE` na raiz do seu repositório.

Sessão 5.6: Segurança e Privacidade

O Que NUNCA Commitar

- **Regra de Ouro: NUNCA commite credenciais** (senhas, chaves de API, tokens de acesso) ou dados sensíveis. São visíveis publicamente!
- **Uso de `.gitignore`:**
  - Um ficheiro de texto que lista ficheiros e pastas que o Git deve **ignorar** (não rastrear).
  - Essencial para manter o repositório limpo e seguro.

Figure: Proteja as suas credenciais! Fonte: Wikimedia Commons, Autor: Google Material Design Icons, Licença: Apache License 2.0.

Sessão 5: Exercício Prático

Desafio de Resolução de Conflitos

- **Passo 1:** Em pares ou individualmente, crie branches separadas (`desafio-conflito-SeuNome`).
- **Passo 2:** Modifique **a mesma linha** em `index.html` de forma diferente em cada branch e faça commit.
- **Passo 3:** Faça `git push` de ambas as branches.
- **Passo 4:** Mergee a primeira branch para `main` (`git pull` na `main` e `git merge`). Faça `git push main`.
- **Passo 5:** Tente mergear a segunda branch para `main`. **Irá ocorrer um conflito!**
- **Passo 6:** Resolva o conflito manualmente no ficheiro, adicione (`git add`) e complete o commit de merge (`git commit`). Faça `git push main`.
- **Verifique:** O histórico de commits na sua conta GitHub.

Trabalho Prático Final

Consolidação de Competências

- Crie um **novo repositório Git** para um "Mini-Projeto Pessoal" à sua escolha.
- **Publique-o no GitHub.**
- Demonstre um **histórico de commits limpo e significativo**.
- Utilize **pelo menos duas branches** para desenvolvimento, fundindo-as na `main`.
- Inclua um `README.md` **bem estruturado** (com Markdown).
- Adicione um ficheiro `.gitignore` para excluir ficheiros temporários/sensíveis.
- Escolha e adicione uma **Licença** (ex: MIT License) ao repositório.
- **Entrega:** O projeto será entregue através do **link do repositório GitHub**.
- **Desafio (Opcional):** Tente simular e resolver um pequeno conflito *propositadamente* no seu próprio projeto.

References / Sources

- **Title:** Just a Simple Guide for Getting Started with Git. No Deep Shit ;)
- **Author:** Roger Dudler
- **Contributors/Credits:** @tfnico, @fhd, Namics
- **Type:** Online Quick Start Guide
- **Available at:** `https://rogerdudler.github.io/git-guide/`
- **Last accessed:** July 5, 2025

This guide is also available in: deutsch, español, français, indonesian, italiano, nederlands, polski, português

References / Sources

- **Author:** Atlassian
- **Type:** Learn Git
- **Available at:** `https://www.atlassian.com/git/tutorials/undoing-changes/git-revert`
- **Last accessed:** July 5, 2025