TIC em trilhas

Resumo do capítulo 1 sobre Controle de Versão

O controle de versão é a prática de gerenciar, rastrear e monitorar alterações em arquivos, pastas, imagens, vídeos e documentos em projetos individuais ou colaborativos. Ele permite que equipes de desenvolvimento aumentem a produtividade e a qualidade dos projetos, possibilitando o acesso a diferentes versões de um mesmo projeto.

Ferramentas de Controle de Versão

Existem várias ferramentas disponíveis no mercado para controle de versão, incluindo:

- **Git**
- **CVS (Concurrent Version System)**
- **SVN (Apache Subversion)**
- **Mercurial**
- **Monotone**
- **Bazaar**
- **TFS (Team Foundation Server)**
- **VSTS (Visual Studio Team Services)**
- **Perforce Helix Core**
- **IBM Rational ClearCase**
- **Revision Control System**
- **VSS (Visual SourceSafe)**

Entre essas, o Git é a ferramenta mais utilizada.

Plataformas de Hospedagem

O Git pode ser usado offline, mas é comumente utilizado em projetos colaborativos através de plataformas de hospedagem, como:

- **GitHub**
- **BitBucket**
- **SourceForge**
- **TaraVault**
- **AWS CodeCommit**
- **Beanstalk**
- **Phabricator**
- **Gitea**
- **Allura**

Diferença entre Git e GitHub

- **Git**: Ferramenta de versionamento de arquivos que permite monitorar e manter diferentes versões dos arquivos.

- **GitHub**: Plataforma de hospedagem de código-fonte que utiliza o Git para controle de versão, permitindo que vários usuários colaborem e mantenham sincronizadas as versões dos projetos.

Resumo do Capítulo 2 sobre Instalação das Dependências

Para demonstrar os conceitos práticos, é necessário instalar algumas ferramentas essenciais, como o Git e o Visual Studio Code.

Instalação do Git

O Git é fundamental para a compreensão teórica e prática dos conteúdos. Para instalá-lo:

1. Acesse git-scm.com e clique em 'Downloads'.
2. Escolha seu sistema operacional (macOS, Windows ou Linux/Unix) e clique no link correspondente.
3. No macOS, instale o Homebrew primeiro:
    - Acesse brew.sh, copie o comando de instalação e execute-o no terminal.
4. Após instalar o Git, verifique a instalação com `git --version` no terminal.

Instalação do Visual Studio Code

O Visual Studio Code (VS Code) é um Ambiente Integrado de Desenvolvimento (IDE) e editor de código. Para instalá-lo:

1. Acesse code.visualstudio.com e clique em 'Download' na página principal.
2. Após o download, abra o arquivo baixado para instalar o programa.

Dicas Importantes

1. **Verificação de Instalação**: Sempre verifique se as ferramentas foram instaladas corretamente usando comandos como `git --version`.
2. **Atualizações**: Mantenha suas ferramentas atualizadas para garantir que você tenha acesso às últimas funcionalidades e correções de bugs.
3. **Documentação**: Consulte a documentação oficial das ferramentas para resolver dúvidas e aprender novas funcionalidades.
4. **Plugins e Extensões**: No VS Code, explore e instale plugins que possam ajudar no seu fluxo de trabalho, como extensões para Git, depuração e linguagens de programação específicas.
5. **Configuração Inicial**: Após instalar o Git, configure seu nome e email com `git config --global user.name "<nome>"` e `git config --global user.email "<email>"`.

Essas dicas e passos de instalação garantirão que você esteja pronto para seguir com os conceitos práticos do módulo.

Resumo do Capítulo 3 sobre Funcionamento do Git

O Git é um sistema de controle de versão que utiliza repositórios para armazenar todas as informações de um projeto, como arquivos e pastas de maneira organizada.

Passos para Criar um Repositório

1. Crie uma nova pasta para o projeto.
2. Abra o terminal e navegue até essa pasta.
3. Digite `git init` para inicializar o repositório.
4. Verifique a criação do repositório com `ls -la` para ver o diretório `.git`.

Estrutura de Objetos do Git

O Git armazena quatro tipos de objetos:

- **Blob**: Armazena arquivos de qualquer extensão.
- **Tree**: Armazena informações sobre diretórios.
- **Commit**: Armazena diferentes versões do projeto.
- **Annotated Tag**: Um ponteiro que aponta para um commit.

Áreas de Trabalho do Git

1. **Working Directory**: Onde os arquivos são exibidos e editados.
2. **Staging Area**: Área de transição para preparar arquivos antes de 'commitá-los'.
3. **Git Repository**: Local onde os arquivos são armazenados, podendo ser local ou remoto.

Fluxo de Trabalho

- Arquivos são movidos do repositório para o Working Directory para edição.
- Após edição, são preparados na Staging Area.
- Realiza-se o commit para salvar as mudanças no repositório.

Comandos Básicos do Terminal

- `mkdir`: Cria uma pasta.
- `touch`: Cria um arquivo.
- `nano`: Edita um arquivo.
- `clear`: Limpa o terminal.
- `echo`: Exibe texto no terminal.
- `man`: Mostra ajuda sobre comandos.
- `cat`: Lista o conteúdo de um arquivo.
- `rm`: Remove arquivos e diretórios.
- `ls`: Lista arquivos e diretórios.
- `cd`: Navega entre diretórios.
- `>`: Escreve para um arquivo.

- `>>`: Anexa a um arquivo.
- `Tab`: Auto-completa comandos.

Dicas Importantes sobre Git

1. **Commits Frequentes**: Faça commits pequenos e frequentes.
2. **Mensagens Claras**: Escreva mensagens de commit descritivas.
3. **Uso de Branches**: Desenvolva novas funcionalidades em branches separados.
4. **Staging Area**: Utilize a Staging Area para agrupar mudanças antes de commitá-las.
5. **Repositórios Remotos**: Sincronize seu repositório local com repositórios remotos para colaborar com outros desenvolvedores.

Resumo do capítulo 4 sobre Operações Básicas do Git

Aqui está um resumo das operações básicas e algumas dicas importantes:

Configuração Inicial

Para começar a usar o Git, é necessário configurar o nome e o email do autor dos commits:

- **Configurar Nome**: `git config --global user.name "<nome>"`
- **Configurar Email**: `git config --global user.email "<email>"`

Essas configurações podem ser verificadas com `git config --list`.

Ciclo de Vida dos Arquivos

Os arquivos no Git podem ter quatro status principais:

1. **Untracked**: Arquivo não monitorado pelo Git.
2. **Unmodified**: Arquivo monitorado, sem modificações desde o último commit.
3. **Modified**: Arquivo monitorado, com modificações não preparadas.
4. **Staged**: Arquivo preparado para o próximo commit.

Comandos Básicos

- **Adicionar Arquivos à Staging Area**:
  - `git add <arquivo>`: Adiciona um arquivo específico.
  - `git add .`: Adiciona todos os arquivos modificados.
- **Fazer Commit**: `git commit -m "<mensagem>"`: Salva as mudanças preparadas no repositório local.
- **Verificar Status**: `git status`: Mostra o status dos arquivos no projeto.
- **Histórico de Commits**:
  - `git log`: Exibe o histórico de commits.
  - `git log --graph`: Exibe o histórico de commits de forma gráfica.

- **Remover Arquivos da Staging Area**: `git rm --cached <arquivo>`: Remove arquivos da área de preparação.

Dicas Importantes

1. **Configuração de Escopo**: Utilize as flags `--local`, `--global` e `--system` para definir o escopo das configurações de usuário.
2. **Commits Granulares**: Faça commits pequenos e frequentes para facilitar o rastreamento de mudanças.
3. **Mensagens de Commit**: Escreva mensagens claras e descritivas para cada commit.
4. **Uso da Staging Area**: Utilize a Staging Area para agrupar mudanças de forma significativa antes de commitá-las.
5. **Verificação de Status**: Use `git status` regularmente para monitorar o estado dos arquivos no projeto.

Resumo do Capítulo 5 sobre Branches, HEAD e Merging no Git

Branches

Branches são referências textuais para commits, permitindo que diferentes versões de um projeto sejam desenvolvidas em paralelo. A branch principal, criada automaticamente, é chamada de `main`. Branches são úteis para dividir o trabalho entre diferentes funcionalidades ou correções de bugs.

- **Criação de Branches**: Use `git branch <nome-da-branch>`
- **Troca de Branches**: Use `git checkout <nome-da-branch>`
- **Boas Práticas de Nomeação**: Utilize nomes descritivos como `feature/onboarding` ou `bugfix/corrigir-login`.

HEAD

O ponteiro `HEAD` indica a versão atual do projeto no Working Directory. Ele aponta para a branch ou commit que está sendo trabalhado no momento.

- **Mudar HEAD**: Use `git checkout <nome-da-branch>` ou `git checkout <SHA-hash>` para mudar a referência do `HEAD`.
- **Detached HEAD**: Ocorre quando o `HEAD` aponta para um commit específico, não associado a uma branch.

Merging

O processo de merging une o conteúdo de diferentes branches em uma única branch, permitindo integrar o trabalho de vários colaboradores.

- **Merge**: Use `git merge <nome-da-branch>` para unir uma branch à branch atual.
- **Resolução de Conflitos**: Durante o merge, pode ser necessário resolver conflitos manualmente se houver alterações conflitantes.

Tipos de Merge

1. **Fast-Forward Merge**:
    - Ocorre quando a branch que receberá as mudanças não tem novos commits após a criação da branch que será unida.
    - Passos:
        1. Realize o commit das mudanças em ambas as branches.
        2. Faça checkout para a branch que receberá as mudanças.
        3. Use `git merge <nome-da-branch>` para aplicar o merge.
    - Após o merge, a branch unida pode ser deletada com segurança.
2. **3-Way Merge**:
    - Utilizado quando a branch que receberá as mudanças já possui novos commits.
    - O Git cria um novo commit que combina as mudanças das duas branches.
    - Passos:
        1. Faça checkout para a branch que receberá as mudanças.
        2. Use `git merge <nome-da-branch>` para aplicar o merge.
    - Pode resultar em conflitos que precisam ser resolvidos manualmente.

Conflitos de Merge

- Conflitos ocorrem quando há alterações conflitantes no mesmo arquivo em diferentes branches.
- Para resolver conflitos:
    1. Use ferramentas como Visual Studio Code para visualizar e resolver conflitos.
    2. Após resolver os conflitos, adicione as mudanças à Staging Area e faça o commit.
    3. Se necessário, use `git merge --abort` para cancelar o merge e reverter ao estado anterior.

Comandos Básicos de Merge

- **Listar Branches**: `git branch`
- **Criar Branch**: `git branch <nome>`
- **Trocar de Branch**: `git checkout <nome>`
- **Deletar Branch**: `git branch -d <nome>`
- **Renomear Branch**: `git branch -m <antigo-nome> <novo-nome>`
- **Criar e Trocar de Branch**: `git checkout -b <nome>`

Dicas Importantes

1. **Uso de Branches**: Sempre crie branches para novas funcionalidades ou correções de bugs para manter o código principal estável.
2. **Commits Frequentes**: Faça commits frequentes em suas branches para facilitar o rastreamento de mudanças.
3. **Mensagens de Commit Claras**: Escreva mensagens de commit descritivas para facilitar a compreensão do histórico de mudanças.
4. **Merge Regular**: Realize merges regulares para integrar mudanças e evitar grandes conflitos de código.
5. **Verificação de Status**: Use `git status` frequentemente para monitorar o estado dos arquivos e branches.
6. **Commits Frequentes**: Realize commits frequentes para facilitar o rastreamento de mudanças.
7. **Mensagens Claras**: Escreva mensagens de commit descritivas.
8. **Uso de Branches**: Crie branches para novas funcionalidades ou correções de bugs.
9. **Merge Regular**: Realize merges regulares para integrar mudanças e evitar grandes conflitos.
10. **Ferramentas de Resolução de Conflitos**: Utilize ferramentas como Visual Studio Code para resolver conflitos de merge de forma eficiente.

Resumo do Capítulo 6 sobre Repositórios Remotos no Git

Repositórios remotos são essenciais para o desenvolvimento colaborativo, permitindo que vários colaboradores trabalhem juntos em um projeto, compartilhando arquivos e mudanças através de um serviço de hospedagem em nuvem.

Clonando Repositórios

Existem várias maneiras de clonar um repositório do GitHub:

1. **Via HTTPS**: Método mais comum, inclui a pasta `.git` com o histórico de commits e branches.
2. **Download do ZIP**: Baixa o projeto sem a pasta `.git`, portanto, sem histórico de commits.
3. **GitHub Desktop**: Abre o projeto diretamente na ferramenta.

Para clonar um repositório via HTTPS:

1. Acesse o repositório no GitHub.
2. Copie o link HTTPS.
3. No terminal, digite `git clone <link-do-repositório>`.

Criando Repositórios Remotos

Para criar um repositório remoto no GitHub:

1. Acesse o site do GitHub.

2. Clique em ‘Novo Repositório’.
3. Configure o repositório e adicione colaboradores.

Comandos Básicos para Repositórios Remotos

- **Clonar Repositório**: `git clone <link-do-repositório>`
- **Adicionar Repositório Remoto**: `git remote add origin <link-do-repositório>`
- **Verificar Repositórios Remotos**: `git remote -v`
- **Enviar Mudanças para o Repositório Remoto**: `git push origin <branch>`
- **Puxar Mudanças do Repositório Remoto**: `git pull origin <branch>`

Quando se trabalha com repositórios remotos, é essencial entender como enviar, buscar, atualizar e deletar mudanças entre o repositório local e o remoto. Aqui estão os principais pontos e dicas importantes:

Tipos de Repositórios

- **Repositório Local**: Armazenado no dispositivo do usuário.
- **Repositório Remoto**: Armazenado em um serviço de hospedagem na nuvem, como o GitHub.

Comandos Básicos

1. **Listar Repositórios Remotos**: `git remote`
    - Mostra os repositórios remotos configurados, com `origin` sendo o padrão.
2. **Verificar URLs de Push e Fetch**: `git remote -v`
    - Exibe os links usados para enviar (push) e buscar (fetch) informações.
3. **Buscar Atualizações**: `git fetch [repositório remoto]`
    - Copia todas as informações do repositório remoto para o local, sem atualizar as branches monitoradas.
4. **Puxar Atualizações**: `git pull`
    - Combina `fetch` e `merge`, copiando informações do repositório remoto e atualizando o Working Directory.
5. **Enviar Mudanças**: `git push`
    - Envia commits, branches e arquivos do repositório local para o remoto. Use `git push --set-upstream origin <branch>` para configurar uma nova branch remota.

Branches Monitoradas

- **Tracking Branch**: Uma branch no repositório remoto que está conectada a uma branch local, permitindo sincronização fácil.
- **Configurar Tracking Branch**:
    - Use `git push --set-upstream origin <branch>` ao criar uma nova branch.

- Alternativamente, use `git branch -u origin/<branch>` ou `git checkout <branch>` após criar a branch no remoto.

Dicas Importantes

1. **Sincronização Regular**: Use `git pull` frequentemente para manter seu repositório local atualizado com as mudanças do repositório remoto.
2. **Commits Claros**: Faça commits com mensagens descritivas para facilitar a colaboração.
3. **Branches para Funcionalidades**: Crie branches específicas para novas funcionalidades ou correções de bugs antes de integrá-las ao branch principal.
4. **Revisão de Código**: Utilize pull requests para revisar e discutir mudanças antes de integrá-las ao branch principal.
5. **Gerenciamento de Conflitos**: Esteja preparado para resolver conflitos de merge ao integrar mudanças de diferentes colaboradores.

Essas práticas e comandos ajudarão a gerenciar repositórios remotos de forma eficiente.