Aula 2 Parte 1: Branching e Merging

Objetivos: - Compreender e utilizar o modelo de branching do Git. - Aprender a mesclar mudanças e resolver conflitos. - Introdução ao Github

Conteúdos: - Comandos para branching e merging: git branch, git checkout, git merge. - Resolução de conflitos. - Trabalhando com Github

Pré-requisitos

Antes de prosseguir com esta aula, é recomendado ter conhecimento básico do Git e do controle de versão de software. Certifique-se de ter o Git instalado em seu sistema e estar familiarizado com os comandos básicos, como `git init`, `git add`, `git commit`, `git push` e `git pull`.

Além disso, é útil ter uma compreensão básica de como funciona o fluxo de trabalho de desenvolvimento de software e a importância do controle de versão.

Se você ainda não possui conhecimento prévio sobre esses tópicos, recomendo que você revise os conceitos básicos do Git e do controle de versão antes de prosseguir com esta aula.

Visão Geral

1. O que é branching e qual é o objetivo de utilizá-lo no controle de versão de software?
2. Como o branching permite que os desenvolvedores trabalhem em novas funcionalidades ou correções de bugs sem interferir no código principal?
3. Quais são os comandos do Git utilizados para criar e alternar entre branches?
4. O que é merging e qual é o seu propósito no controle de versão de software?
5. Como o merging é usado para unir o trabalho feito em uma branch de desenvolvimento de volta para a branch principal?
6. Quais são as vantagens de utilizar branches no Git?
7. Como os branches permitem experimentação segura e facilitam o desenvolvimento paralelo?
8. Como os branches facilitam o controle de versão eficiente e a revisão de código?
9. Quais são as desvantagens de utilizar branches no Git?
10. Como a complexidade de gerenciamento e os conflitos de merge podem ser desafios ao utilizar branches?
11. Como a desatualização de branches e a sobrecarga de processo podem ser desvantagens ao utilizar branches?
12. O que são conflitos no Git e como eles ocorrem durante o processo de mesclagem?
13. Como os conflitos de merge podem ser resolvidos manualmente?
14. Quais são as vantagens de utilizar o GitHub para hospedagem de código-fonte e colaboração?

15. Quais são os recursos avançados do Git e como eles podem ser explorados?
16. Quais são as ferramentas GUI para Git e como elas podem facilitar o trabalho com o controle de versão?
17. Como o Git pode ser integrado com IDEs, como o VSCode?
18. O que é o arquivo .gitignore e como ele é utilizado no controle de versão com Git?
19. Como o GitHub Desktop pode ser utilizado para facilitar a interação com o GitHub?
20. Quais são os benefícios de utilizar o GitHub para compartilhar projetos de código aberto e colaborar com outros desenvolvedores?

Explicando Branching

Branching, no contexto de controle de versão de software, é o processo de criar uma "cópia" do código que pode ser modificada independentemente do código principal. Isso permite que os desenvolvedores trabalhem em novas funcionalidades ou correções de bugs sem interferir no código principal.

Por exemplo, imagine que você está trabalhando em um software de gerenciamento de tarefas. A versão atual do software (v1.0) está funcionando bem e está em uso pelos usuários. No entanto, você quer adicionar uma nova funcionalidade que permita aos usuários compartilhar tarefas com outras pessoas.

Em vez de fazer essas alterações diretamente no código principal (e potencialmente interromper a versão atual do software), você pode criar um novo branch (por exemplo, "compartilhamento-de-tarefas") e fazer as alterações lá.

Uma vez que a nova funcionalidade esteja pronta e testada, você pode mesclar (merge) as alterações de volta para o código principal. Isso garante que o código principal permaneça estável e funcional enquanto você trabalha em novas funcionalidades ou correções de bugs.

branch

Quantos branches existem na figura?

Explicando Merging

Merging, no contexto de controle de versão de software, é o processo de combinar as alterações de uma branch em outra. Isso é comumente usado para unir o trabalho feito em uma branch de desenvolvimento de volta para a branch principal.

Por exemplo, imagine que você criou uma nova branch chamada "nova-funcionalidade" para trabalhar em uma nova funcionalidade para seu aplicativo. Depois de terminar o trabalho na nova funcionalidade e testá-la, você pode mesclar as alterações da branch "nova-funcionalidade" de volta para a branch principal.

O processo de mesclagem envolve comparar as alterações feitas em ambas as branches e combinar essas alter### Vantagens de Utilização de Branches no Git

Quantos merges houveram no branch main?

Vantagens de utilização de Branches no Git

1. **Isolamento de Código:** Cada branch serve como um ambiente isolado, permitindo que os desenvolvedores trabalhem em diferentes funcionalidades ou correções de bugs sem interferir uns nos outros.
2. **Experimentação Segura:** Os branches permitem que você experimente novas ideias sem arriscar o estado do código principal. Se a experimentação falhar, você pode simplesmente descartar o branch.
3. **Facilita o Desenvolvimento Paralelo:** Vários desenvolvedores podem trabalhar simultaneamente em diferentes branches, cada um focado em uma funcionalidade ou correção específica.
4. **Controle de Versão Eficiente:** Os branches facilitam o gerenciamento de diferentes versões de um software. Por exemplo, você pode ter um branch para a versão de produção do software e outro para o desenvolvimento de novas funcionalidades.
5. **Revisão de Código e Integração Contínua:** Ao usar branches, você pode facilmente configurar um fluxo de trabalho onde cada alteração é revisada (via pull request) e testada (via integração contínua) antes de ser mesclada no código principal.
6. **Desacoplamento de Releases e Desenvolvimento:** Você pode continuar a desenvolver novas funcionalidades em um branch de desenvolvimento enquanto prepara um release em um branch separado.

Desvantagens de Utilização de Branches no Git

1. **Complexidade de Gerenciamento:** Se não forem gerenciados corretamente, muitos branches podem tornar o repositório confuso e difícil de navegar.
2. **Conflitos de Merge:** Se dois branches modificarem a mesma parte do código, pode haver conflitos ao tentar mesclá-los. Esses conflitos precisam ser resolvidos manualmente.
3. **Desatualização de Branches:** Se um branch não for atualizado regularmente com as alterações do branch principal, pode se tornar significativamente desatualizado, tornando a mesclagem de volta ao branch principal mais difícil.
4. **Sobrecarga de Processo:** Dependendo do fluxo de trabalho, a criação de branches para cada pequena alteração pode levar a uma sobrecarga de

processo, com tempo gasto em revisões de código e integração contínua para cada branch.

5. **Divergência de Código:** Se os branches não forem mesclados de volta ao branch principal regularmente, o código pode começar a divergir significativamente, tornando mais difícil a integração futura.

Explicando conflitos no Git

Gerenciamento de conflitos no contexto de controle de versão com Git ocorre quando duas ou mais pessoas modificam a mesma parte do código em branches diferentes e depois tentam mesclar essas alterações. O Git não sabe qual alteração deve prevalecer, então ele marca o conflito e pede que você o resolva manualmente.

Por exemplo, imagine que você e um colega estão trabalhando no mesmo projeto. Você cria um branch chamado "funcionalidade-a" e seu colega cria um branch chamado "funcionalidade-b". Ambos modificam a mesma linha do mesmo arquivo.

Aqui estão os passos que vocês dois podem seguir:

Você e seu colega fazem commit das suas alterações em seus respectivos branches.

Seu colega mescla o branch "funcionalidade-b" de volta ao branch principal primeiro.

Agora, quando você tenta mesclar o branch "funcionalidade-a" de volta ao branch principal, o Git detecta que a mesma linha foi modificada em ambos os branches e não sabe qual alteração deve prevalecer. Ele marca essa linha no arquivo com conflito.

O arquivo com conflito ficará assim:

```
<<<<<<< HEAD
Código da funcionalidade-b
=======
Código da funcionalidade-a
>>>>>>> funcionalidade-a
```

Para resolver o conflito, você precisa editar o arquivo para decidir qual código deve prevalecer, ou se uma combinação dos dois é necessária. Depois de fazer isso, você faz um novo commit para finalizar a resolução do conflito.

Agora, o branch "funcionalidade-a" pode ser mesclado de volta ao branch principal sem conflitos.

Nota: Este laboratório pressupõe que você está trabalhando localmente e não está fazendo push das alterações para um repositório remoto.

Parte 2: Ferramentas de Controle de Versão Avançadas

Objetivos: - Explorar recursos avançados do Git. - Introdução ao uso de GUIs para Git.

Conteúdos: - Ferramentas GUI para Git: SourceTree, GitHub Desktop. - Integração do Git com IDEs (VSCode). - Trabalhando com Gitignore

Atividades: - Prática dos comandos avançados através de cenários complexos. - Exploração de uma ferramenta GUI Git em um projeto exemplo.

Introdução ao GitHub

O GitHub é uma plataforma de hospedagem de código-fonte e colaboração que permite que desenvolvedores trabalhem juntos em projetos de software. Ele fornece recursos avançados de controle de versão, gerenciamento de problemas, revisão de código e integração contínua.

Com o GitHub, você pode criar repositórios para armazenar seu código-fonte, colaborar com outros desenvolvedores por meio de pull requests, gerenciar problemas e tarefas, e implantar seu software em serviços de hospedagem.

Além disso, o GitHub oferece uma interface amigável e intuitiva, facilitando a navegação pelos repositórios, visualização do histórico de commits e revisão de alterações.

O GitHub também é amplamente utilizado pela comunidade de desenvolvedores para compartilhar projetos de código aberto, permitindo que outros desenvolvedores contribuam, aprendam e colaborem em projetos de software.

Se você é novo no GitHub, é altamente recomendável explorar seus recursos e aprender a usá-lo para melhorar sua produtividade e colaboração no desenvolvimento de software.

Introdução ao GitHub Desktop

O GitHub Desktop é uma ferramenta GUI (Interface Gráfica do Usuário) para o Git que facilita o controle de versão e a colaboração em projetos hospedados no GitHub. Ele fornece uma interface intuitiva e amigável para realizar tarefas comuns do Git, como clonar repositórios, criar branches, fazer commits, mesclar alterações e sincronizar com o repositório remoto.

Com o GitHub Desktop, você pode visualizar facilmente o histórico de commits, comparar alterações entre branches e resolver conflitos de mesclagem de forma visual. Ele também oferece recursos avançados, como o uso de submódulos, rebase interativo e cherry-pick.

Além disso, o GitHub Desktop é altamente integrado com o ecossistema do GitHub, permitindo que você crie pull requests, revise e comente em código, e acompanhe as atividades do repositório diretamente na interface.

Se você está começando com o Git ou prefere uma abordagem mais visual para o controle de versão, o GitHub Desktop é uma ótima opção para você. Ele está disponível para Windows e macOS e pode ser baixado gratuitamente no site oficial do GitHub.

Recursos do GitHub Desktop

O GitHub Desktop oferece uma variedade de recursos para facilitar o controle de versão e a colaboração em projetos. Alguns dos recursos principais incluem:

- Interface intuitiva e amigável: O GitHub Desktop possui uma interface fácil de usar, com menus e botões claros que facilitam a realização de tarefas comuns do Git.
- Visualização do histórico de commits: Você pode visualizar facilmente o histórico de commits do seu repositório, verificando as mensagens de commit, as alterações feitas e os autores.
- Comparação de alterações entre branches: O GitHub Desktop permite que você compare as alterações feitas em diferentes branches, facilitando a identificação de diferenças e conflitos.
- Resolução visual de conflitos de mesclagem: Quando ocorrem conflitos de mesclagem, o GitHub Desktop fornece uma interface visual para ajudar na resolução desses conflitos, permitindo que você escolha quais alterações devem prevalecer.
- Integração com o ecossistema do GitHub: O GitHub Desktop está integrado ao GitHub, permitindo que você crie pull requests, revise e comente em código e acompanhe as atividades do repositório diretamente na interface.
- Recursos avançados: Além das funcionalidades básicas do Git, o GitHub Desktop oferece recursos avançados, como o uso de submódulos, rebase interativo e cherry-pick.

Esses são apenas alguns dos recursos oferecidos pelo GitHub Desktop. Experimente a ferramenta e descubra como ela pode facilitar o seu trabalho com controle de versão e colaboração em projetos hospedados no GitHub.

Laboratórios: Instalando e Utilizando o GitHub Desktop

1. **Instalando o GitHub Desktop:**
   - Acesse o site oficial do GitHub Desktop em https://desktop.github.com.
   - Faça o download da versão adequada para o seu sistema operacional (Windows ou macOS).
   - Siga as instruções de instalação fornecidas pelo instalador.
   - Após a instalação, abra o GitHub Desktop.
2. **Configurando o Repositório:**
   - No GitHub Desktop, clique em "File" (Arquivo) e selecione "Add Local Repository" (Adicionar Repositório Local).
   - Navegue até o diretório do seu projeto e selecione a pasta que contém o repositório.
   - Clique em "Add Repository" (Adicionar Repositório) para adicionar o repositório ao GitHub Desktop.

3. **Configurando Nome de Autor e Email:**
   - No GitHub Desktop, clique em "File" (Arquivo) e selecione "Options" (Opções).
   - Na guia "Git", preencha os campos "Name" (Nome) e "Email" (Email) com as suas informações.
   - Clique em "Save" (Salvar) para salvar as configurações.
4. **Realizando Commits:**
   - No GitHub Desktop, você verá uma lista de arquivos modificados no seu repositório.
   - Selecione os arquivos que deseja incluir no commit, marcando as caixas de seleção ao lado deles.
   - Digite uma mensagem descritiva para o commit no campo "Summary" (Resumo).
   - Clique em "Commit to master" (Commitar para o master) para realizar o commit.
5. **Sincronizando com o Repositório Remoto:**
   - Após realizar commits locais, você pode sincronizar as alterações com o repositório remoto.
   - Clique em "Repository" (Repositório) e selecione "Push" (Enviar).
   - O GitHub Desktop enviará as alterações para o repositório remoto.

Agora você está pronto para utilizar o GitHub Desktop para configurar um repositório, definir suas informações de autor e email, e realizar commits. Lembre-se de sincronizar suas alterações com o repositório remoto regularmente para manter seu trabalho seguro e compartilhado com outros colaboradores.

Plugin Git do VSCode

O Visual Studio Code (VSCode) possui um poderoso plugin integrado para o controle de versão Git. Esse plugin oferece uma interface intuitiva e eficiente para realizar tarefas comuns do Git diretamente no editor.

Para utilizar o plugin Git do VSCode, siga os passos abaixo:

1. Abra o VSCode e navegue até o diretório do seu projeto.
2. Certifique-se de que o Git está instalado no seu sistema e configurado corretamente.
3. No VSCode, clique no ícone do Git na barra lateral esquerda ou pressione `Ctrl + Shift + G` para abrir a interface do Git.
4. A interface do Git exibirá as alterações pendentes no seu repositório. Você pode visualizar as alterações nos arquivos, adicionar arquivos ao commit, fazer commits, criar branches e muito mais.
5. Para fazer um commit, clique no ícone `+` ao lado dos arquivos modificados para adicioná-los ao commit. Em seguida, digite uma mensagem descritiva para o commit e pressione `Ctrl + Enter` para confirmar.

6. Para criar uma nova branch, clique no ícone de ramificação na parte superior da interface do Git. Digite o nome da nova branch e pressione `Enter`.

7. Para sincronizar com o repositório remoto, clique no ícone de sincronização na parte superior da interface do Git. Isso enviará as alterações locais para o repositório remoto e buscará as alterações remotas para o seu repositório local.

O plugin Git do VSCode também oferece recursos avançados, como visualização do histórico de commits, resolução de conflitos de mesclagem, criação de tags e muito mais. Explore a interface do Git no VSCode para descobrir todas as funcionalidades disponíveis.

Lembre-se de que o plugin Git do VSCode é altamente integrado com o editor e oferece atalhos de teclado e comandos de menu para facilitar o uso. Consulte a documentação oficial do VSCode para obter mais informações sobre o plugin Git e suas funcionalidades.

Casos de Uso do Arquivo .gitignore

O arquivo .gitignore é usado para especificar quais arquivos e diretórios devem ser ignorados pelo Git. Isso significa que o Git não rastreará as alterações nesses arquivos e diretórios, evitando que eles sejam incluídos nos commits e enviados para o repositório remoto.

Aqui estão alguns casos de uso comuns para o arquivo .gitignore:

1. Ignorar arquivos de compilação ou build:
   - Quando você está desenvolvendo um projeto, é comum gerar arquivos de compilação ou build, como arquivos executáveis, bibliotecas compiladas ou arquivos intermediários. Esses arquivos geralmente não são necessários para o controle de versão e podem ser ignorados usando o .gitignore.
2. Ignorar arquivos de configuração local:
   - Cada desenvolvedor pode ter suas próprias configurações locais, como arquivos de configuração do ambiente, chaves de API ou arquivos de configuração do IDE. Esses arquivos podem conter informações sensíveis ou específicas do ambiente local e devem ser ignorados para evitar que sejam compartilhados no repositório.
3. Ignorar arquivos gerados automaticamente:
   - Alguns frameworks ou ferramentas geram automaticamente arquivos durante o processo de desenvolvimento, como arquivos de cache, logs ou arquivos temporários. Esses arquivos podem ser ignorados para evitar poluir o repositório com arquivos desnecessários.
4. Ignorar arquivos de dependências:
   - Quando você está usando gerenciadores de pacotes, como npm, pip ou Maven, as dependências do projeto são baixadas e armazenadas em um

diretório específico. Essas dependências não precisam ser rastreadas pelo Git, pois podem ser facilmente baixadas novamente usando o gerenciador de pacotes. Portanto, é comum ignorar o diretório de dependências no .gitignore.

5. Ignorar arquivos sensíveis:
    - Em alguns casos, você pode ter arquivos que contêm informações sensíveis, como senhas, chaves de API ou dados confidenciais. Esses arquivos devem ser ignorados para evitar que sejam compartilhados acidentalmente no repositório.

Lembre-se de que o arquivo .gitignore é específico para cada repositório. Você pode criar um arquivo .gitignore na raiz do seu projeto e adicionar padrões de arquivos e diretórios que devem ser ignorados. É possível usar curingas, como asteriscos (*) e pontos de interrogação (?), para especificar padrões de correspondência.

Certifique-se de revisar e atualizar regularmente o arquivo .gitignore para garantir que os arquivos corretos sejam ignorados e que nenhum arquivo importante seja excluído acidentalmente.

Resumo

1. Branching é o processo de criar uma ramificação do código principal em um controle de versão de software. O objetivo é permitir que os desenvolvedores trabalhem em novas funcionalidades ou correções de bugs sem interferir no código principal.
2. O branching permite que os desenvolvedores trabalhem em novas funcionalidades ou correções de bugs sem interferir no código principal. Isso é possível porque cada branch é uma cópia independente do código principal.
3. Os comandos do Git utilizados para criar e alternar entre branches são:
    - `git branch`: cria um novo branch.
    - `git checkout`: alterna para um branch existente.
4. Merging é o processo de combinar as alterações de uma branch em outra. Seu propósito no controle de versão de software é unir o trabalho feito em uma branch de desenvolvimento de volta para a branch principal.
5. O merging é usado para unir o trabalho feito em uma branch de desenvolvimento de volta para a branch principal. Isso permite que as alterações feitas em uma branch sejam incorporadas ao código principal.
6. As vantagens de utilizar branches no Git são:
    - Isolamento de código.
    - Experimentação segura.
    - Facilita o desenvolvimento paralelo.
    - Controle de versão eficiente.

- Revisão de código e integração contínua.
- Desacoplamento de releases e desenvolvimento.

7. Os branches permitem experimentação segura e facilitam o desenvolvimento paralelo, pois cada branch é um ambiente isolado onde os desenvolvedores podem trabalhar em diferentes funcionalidades ou correções de bugs.

8. Os branches facilitam o controle de versão eficiente e a revisão de código, pois cada alteração pode ser revisada e testada antes de ser mesclada no código principal.

9. As desvantagens de utilizar branches no Git são:

   - Complexidade de gerenciamento.
   - Conflitos de merge.
   - Desatualização de branches.
   - Sobrecarga de processo.
   - Divergência de código.

10. A complexidade de gerenciamento e os conflitos de merge podem ser desafios ao utilizar branches. É importante ter um bom fluxo de trabalho e comunicação entre os desenvolvedores para lidar com essas questões.

11. A desatualização de branches e a sobrecarga de processo podem ser desvantagens ao utilizar branches. É importante manter os branches atualizados e ter um fluxo de trabalho eficiente para evitar esses problemas.

12. Conflitos no Git ocorrem quando duas ou mais pessoas modificam a mesma parte do código em branches diferentes e tentam mesclar essas alterações. Os conflitos precisam ser resolvidos manualmente, decidindo qual alteração deve prevalecer.

13. Os conflitos de merge podem ser resolvidos manualmente editando o arquivo em conflito e decidindo qual código deve prevalecer ou se uma combinação dos dois é necessária.

14. As vantagens de utilizar o GitHub para hospedagem de código-fonte e colaboração são:

    - Facilita o compartilhamento de projetos de código aberto.
    - Permite colaboração entre desenvolvedores.
    - Oferece recursos avançados de controle de versão.
    - Integração com outras ferramentas e serviços.

15. Os recursos avançados do Git podem ser explorados para melhorar o controle de versão, como o uso de tags, ramificações remotas, rebase, entre outros.

16. Existem várias ferramentas GUI para Git, como Sourcetree, GitKraken e GitHub Desktop, que facilitam o trabalho com o controle de versão, fornecendo uma interface gráfica para executar comandos do Git.

17. O Git pode ser integrado com IDEs, como o VSCode, por meio de extensões que fornecem recursos avançados de controle de versão diretamente na interface da IDE.

18. O arquivo .gitignore é utilizado no controle de versão com Git para especificar quais arquivos e diretórios devem ser ignorados pelo Git. Isso é útil para evitar que arquivos desnecessários sejam incluídos no repositório.

19. O GitHub Desktop é uma ferramenta que facilita a interação com o GitHub, permitindo que os desenvolvedores gerenciem repositórios, criem branches, façam commits e realizem outras operações diretamente em uma interface gráfica.

20. Os benefícios de utilizar o GitHub para compartilhar projetos de código aberto e colaborar com outros desenvolvedores incluem a facilidade de compartilhamento, a visibilidade da comunidade, a possibilidade de receber contribuições e a integração com outras ferramentas e serviços.