E-BOOK

Git e Github

Passo a passo

A importância do gerenciamento de código em DevOps e Cloud

Gerenciar código de maneira eficiente tornou-se uma prática que vai muito além do código feito pelos programadores. Hoje, temos que gerenciar infraestrutura declarada como código, manifestos do Kubernetes, Dockerfiles e muito mais. Por exemplo, onde antes a infraestrutura era muitas vezes configurada e gerenciada manualmente, agora é tratada como código, sujeita a revisões, testes e versionamento. Com a adoção crescente de DevOps e da computação em nuvem, o profissional que antes não precisava se preocupar com código, hoje precisa. Vamos examinar a importância do gerenciamento de código no contexto de Cloud e de DevOps:

O papel vital no ciclo DevOps No mundo do DevOps:

- **Velocidade e Integridade**: O gerenciamento de código permite que as equipes mantenham a rapidez necessária para lançamentos frequentes, ao mesmo tempo em que assegura a integridade do código ao longo do processo.
- **Colaboração:** Através do gerenciamento de código, várias partes da equipe - desde desenvolvedores, QA e operações - podem trabalhar simultaneamente em diferentes partes ou recursos do código sem interferir uns nos outros.
- **Rastreabilidade:** Em um ambiente onde a integração contínua é um princípio básico, rastrear cada mudança de código até sua origem e entender seu propósito é crucial. Isso não apenas ajuda na resolução de problemas, mas também na conformidade e auditorias.

Sustentando a flexibilidade da Computação em Nuvem

Quando se trata de computação em nuvem:

- **Gerenciamento de Múltiplos Ambientes:** A capacidade de gerenciar versões de código para diferentes ambientes (desenvolvimento, teste, produção) na nuvem é facilitada por práticas robustas de gerenciamento de código.
- **Automatização de Implantações:** Na nuvem, as implantações frequentemente precisam ser automatizadas, reproduzidas e distribuídas. O gerenciamento de código assegura que o código correto está sendo implantado na versão certa, sempre.
- **Escalabilidade com Integridade:** A nuvem permite escalar rapidamente, e um gerenciamento de código eficiente garante que, mesmo em escala, a integridade do código seja mantida.

Segurança no código

Em ambos, DevOps e Cloud, a segurança tornou-se uma preocupação fundamental. Com um gerenciamento de código adequado:

- **Identificação de vulnerabilidades:** Ferramentas modernas podem escanear o código em busca de vulnerabilidades conhecidas, garantindo que elas sejam tratadas antes da implantação.
- **Rastreamento de dependências:** Ao gerenciar e versionar o código de forma adequada, é mais fácil acompanhar as bibliotecas e dependências usadas, garantindo que versões desatualizadas ou inseguras possam ser identificadas e atualizadas.

Como funciona o versionamento de código

O versionamento de código, também conhecido como controle de versão, é uma ferramenta que registra alterações em um arquivo ou conjunto de arquivos ao longo do tempo, para que você possa recolher versões específicas mais tarde. Ele é uma ferramenta indispensável para desenvolvedores e equipes de infraestrutura, permitindo que múltiplos indivíduos trabalhem juntos e mantenham um histórico das alterações realizadas.

Por que o versionamento é necessário?

- **Colaboração:** Em grandes equipes ou projetos, muitos profissionais trabalham no mesmo código. O versionamento ajuda a garantir que as alterações feitas por alguém da equipe não interfira nas alterações feitas por outro.
- **Histórico:** Ter um registro completo de quem fez o quê e quando ajuda a identificar quando e onde os bugs foram inseridos.
- **Reversão:** Se uma versão do software tem um bug crítico, é fácil voltar para uma versão anterior que funcionava corretamente.

Como o Git se encaixa nisso?

Se você já teve contato com máquinas virtuais, provavelmente já criou um snapshot para a sua VM. A operação do Git pode ser vista da mesma forma. Assim como um snapshot captura o estado exato de uma máquina virtual em um ponto específico no tempo, no Git, cada vez que você realiza um commit, você está criando uma “fotografia” do estado atual do seu código. Esse registro detalhado permite que, se necessário, você possa reverter seu código para qualquer commit anterior, da mesma forma que pode reverter uma máquina virtual para um snapshot anterior. Esta capacidade do Git oferece um excelente ferramental para que você consiga gerenciar seu código e trabalhar de forma colaborativa, permitindo experimentações e testes sem medo de comprometer o trabalho já realizado.

O que é o Git

Git é um sistema de controle de versão distribuído, gratuito e de código aberto, projetado para lidar com tudo, desde projetos pequenos a muito grandes, com rapidez e eficácia. Foi criado por Linus Torvalds em 2005 para auxiliar no desenvolvimento do kernel do Linux. Hoje, é indiscutivelmente a ferramenta de gerenciamento de código mais popular e amplamente adotada no mundo.

Vamos entender algumas características importantes do Git:

Natureza Distribuída

Diferente de sistemas de controle de versão centralizados, que dependem de um único repositório central, o Git opera de forma distribuída. Cada membro da equipe possui uma cópia completa do histórico do projeto no seu próprio ambiente. Esta abordagem distribuída significa que as operações, como commits e consultas de histórico, são realizadas localmente. Isso não apenas torna as operações mais rápidas, mas também permite trabalhar offline e sem depender de um servidor central.

Integridade dos Dados

O Git prioriza a integridade dos dados. Antes de registrar qualquer arquivo, o Git cria uma “assinatura” (hash) para o arquivo, estabelecendo uma referência para aquele conteúdo específico. Esta funcionalidade assegura que o conteúdo do arquivo não será alterado sem o conhecimento do Git.

Por que o Git é essencial hoje?

O Git atende às necessidades das equipes que desejam trabalhar não só com desenvolvimento, mas também com gerenciamento de ambientes de Cloud, DevOps e infraestrutura. Sua rapidez, eficiência e arquitetura distribuída tornam a colaboração em projetos extremamente fluida. Dada a sua posição como a ferramenta de gerenciamento de código mais utilizada globalmente, o domínio do Git tornou-se uma habilidade praticamente mandatória para profissionais de tecnologia. Se você deseja se destacar no mercado atual, é fundamental dominar o Git.

Principais conceitos e o fluxo no Git

Ao trabalhar com o Git, é fundamental entender seus principais conceitos e como eles interagem entre si. Estes são a base do gerenciamento de código eficiente com Git.

Repositório

Um **repositório** (ou "repo") é um diretório que contém todos os arquivos do projeto, assim como o histórico de todas as alterações feitas a esses arquivos. Ele serve como a base para rastrear e colaborar em um projeto.

Commit

Um **commit** representa uma "fotografia" das alterações feitas nos arquivos, é um marco na linha do tempo daquele projeto. Cada commit possui um identificador único (uma "hash") que permite rastrear exatamente quais alterações foram feitas por quem e quando.

Branch

Uma **branch** (ou "ramo") é uma derivação independente da linha principal do projeto. Ela permite que as pessoas da equipe trabalhem em novas funcionalidades ou correções sem alterar a branch principal (frequentemente chamado de "main" ou "master"). Uma vez que as alterações em uma branch "alternativa" estejam prontas, elas podem ser incorporadas de volta na branch principal.

Merge

Merge é o processo de combinar as alterações de uma branch com outra. Pode ser de uma branch de funcionalidade para a branch principal, ou entre outras branches. Durante o merge, o Git tenta combinar as alterações automaticamente. Porém, em caso de conflitos entre as alterações (algo que pode ocorrer com frequência), é necessária intervenção manual para resolvê-los.

Repositórios Remotos

Repositórios remotos são versões do seu projeto hospedadas na internet ou em uma rede. Eles permitem que múltiplos membros da equipe colaborem no mesmo projeto, garantindo que todos tenham acesso às versões mais recentes do código.

Clone

Clonar é o ato de copiar um repositório remoto para a sua máquina local. Este processo permite que você tenha uma cópia local completa do projeto, permitindo que você trabalhe sem afetar o repositório remoto.

Pull

O comando **pull** é usado para buscar as últimas mudanças de um repositório remoto e integrá-las ao seu branch atual em sua cópia local.

Push

Depois de fazer alterações e commits em sua máquina local, **push** é o processo de enviar essas alterações de volta para o repositório remoto, para que outros possam acessar e colaborar com elas.

Instalação do Git

Como ponto de partida, você precisa instalar o Git no seu ambiente. A instalação pode variar de acordo com o sistema operacional que você está usando. Abaixo estão os passos para as plataformas mais comuns:

Windows

1. **Download do instalador:** Acesse o site oficial do Git git-scm.com e clique em "Download for Windows". Isso iniciará o download do instalador do Git para Windows.

2. **Execução do instalador:** Assim que o download estiver concluído, localize o arquivo baixado e dê um duplo clique para iniciar a instalação.

3. **Siga as instruções na tela:** A instalação é bastante direta, mas, se você não tiver certeza sobre alguma configuração específica, pode aceitar os padrões recomendados clicando em "Next" até concluir o processo.

MacOS

1. **Homebrew:** Se você já tem o Homebrew instalado, pode simplesmente executar o comando:

```
brew install git
```

2. **Download direto:** Caso não tenha o Homebrew, acesse o site oficial do Git git-scm.com e clique em "Download for Mac". Isso baixará o instalador do Git para MacOS. Siga as instruções na tela para concluir a instalação.

Linux

A instalação no Linux varia dependendo da distribuição. No entanto, para a maioria das distribuições populares, você pode usar o gerenciador de pacotes padrão.

- **Debian/Ubuntu:**

```
sudo apt update
sudo apt install git
```

- **Fedora:**

```
sudo dnf install git
```

- **Arch Linux:**

```
sudo pacman -S git
```

Depois de concluir a instalação em qualquer uma das plataformas, você pode abrir o terminal ou prompt de comando e digitar **`git --version`** para garantir que o Git foi instalado corretamente. Esta ação retorna a versão do Git que você instalou.

Configuração do Git e Iniciando um repositório Git Local

Depois de instalar o Git, é fundamental configurá-lo corretamente para garantir um fluxo de trabalho eficiente. Além disso, aprender a iniciar um repositório local é o primeiro passo para gerenciar seu código com Git.

Configurando sua identidade

Antes de fazer commits no Git, é importante definir quem você é, para que suas contribuições sejam corretamente atribuídas. Execute os seguintes comandos para configurar seu nome e endereço de e-mail:

```
git config --global user.name "Seu Nome"
git config --global user.email "seuemail@exemplo.com"
```

Configurações Adicionais

Existem outras configurações que podem melhorar seu fluxo de trabalho com Git:

Definindo a branch padrão como "main": Tradicionalmente, o Git usava "master" como nome padrão para a branch principal. Porém, a comunidade começou a adotar "main" para essa finalidade. Para garantir que seus novos repositórios usem "main" como padrão, use:

```
git config --global init.defaultBranch main
```

Configurando o editor padrão: Se você tem uma preferência de editor, como o "vim", pode configurá-lo como editor padrão para Git:

```
git config --global core.editor vim
```

Iniciando um novo repositório

Se você está começando um novo projeto e deseja usar o Git para gerenciar seu código, o primeiro passo é iniciar um novo repositório. Navegue até o diretório do seu projeto e execute:

```
git init
```

Inclusão e alteração de arquivos e commit

Trabalhar com o Git envolve um fluxo específico para rastrear e registrar alterações nos arquivos. Entender corretamente esse fluxo e as etapas, e principalmente como o Git lida com essas mudanças, é essencial para um gerenciamento eficaz do código.

Staged Area (Área de Preparação)

Antes de nos aprofundarmos no comando **add**, é vital entender um conceito central do Git chamado "Staged Area" ou "Área de Preparação". Essa é uma área intermediária onde o Git rastreia e armazena as alterações que você deseja commitar. Simplificando, é uma espécie de área de "pré-commit". As alterações que são adicionadas à área de preparação estão prontas para serem commitadas no próximo **git commit**.

Adicionando arquivos ao Git

Quando você cria ou modifica um arquivo no diretório do projeto, o Git nota essa mudança, mas não rastreia imediatamente. Para informar ao Git que você deseja rastrear essas alterações, você deve adicionar no "Staged Area" com o comando **add**:

```
git add nome_do_arquivo.txt
```

O comando acima prepara especificamente o **nome_do_arquivo.txt** para o próximo commit. Se você deseja adicionar todas as alterações no repositório para a área de preparação, você pode usar:

```
git add .
```

Removendo arquivos da Staged Area com `reset`

Pode acontecer de você adicionar arquivos na área de preparação por engano ou decidir que não deseja commitar essas mudanças naquele momento. Para remover arquivos da área de preparação, você pode usar o comando **reset**:

```
git reset nome_do_arquivo.txt
```

O comando acima irá remover as alterações feitas no **nome_do_arquivo.txt** da Staged Area, mas as alterações feitas no arquivo em si não serão descartadas. Se você quiser remover todas as alterações da área de preparação, simplesmente execute:

```
git reset
```

Commitando alterações

Depois de ter seus arquivos ou alterações na "Staged Area", você pode registrar (ou "commitar") essas alterações. Este comando captura um "instantâneo" das mudanças que você adicionou:

```
git commit -m "Adicionado arquivo inicial do projeto"
```

Aqui, o parâmetro **-m** permite que você adicione uma mensagem descritiva com seu commit

Alterando arquivos

Após rastrear um arquivo, o Git identificará se ele foi alterado desde o último commit. Para ver os arquivos que foram alterados e estão fora da "Staged Area", use:

```
git status
```

Depois de modificar um arquivo, o processo é parecido com o anterior. Primeiro, você adiciona o arquivo na área de preparação e, em seguida, commita as alterações:

```
git add nome_do_arquivo_modificado.txt
git commit -m "Atualizadas informações no arquivo"
```

Arquivo .gitignore

Em qualquer projeto, existem arquivos ou diretórios que você pode não querer rastrear com o Git, seja porque são gerados automaticamente, como arquivos de log, ou porque contêm informações sensíveis, como senhas. O Git oferece uma solução robusta para garantir que certos arquivos sejam ignorados quando você faz um commit. Isso é feito através do arquivo **`.gitignore`**.

O que é o .gitignore?

O `.gitignore` é um arquivo de texto simples que informa ao Git quais arquivos ou diretórios ele deve ignorar e não rastrear. Cada linha deste arquivo especifica um padrão que determina os arquivos/diretórios a serem ignorados.

Como configurar o .gitignore

1. **Crie o arquivo:** No diretório raiz do seu repositório, crie um arquivo chamado `.gitignore`.
2. **Adicione padrões:** Abra este arquivo em um editor de texto e comece a adicionar padrões.

Por exemplo:

```
# Ignora todos os arquivos .log
*.log

# Ignora todos os arquivos na pasta "temp"
temp/

# Ignora o arquivo config.php
config.php
```

3. **Salve e commit:** Depois de configurar seus padrões, salve o arquivo e faça um commit para que o Git comece a usá-lo.

Padrões comuns

Aqui estão alguns padrões comuns encontrados em arquivos `.gitignore`, especialmente úteis:

```
# Arquivos de backup
*~

# Arquivos binários
*.bin
*.db
*.dat
*.o
*.so

# Arquivos temporários
*.tmp
*.bak
*.swp
```

Repositórios de templates `.gitignore`

Existem diversos repositórios e ferramentas online que fornecem templates de `.gitignore` para diferentes linguagens de programação e frameworks. Um dos mais populares é o github/gitignore. Você pode usar estes templates como ponto de partida e ajustá-los de acordo com as necessidades específicas do seu projeto.

Como criar e gerenciar branches

Uma das características mais importantes do Git é a capacidade de criar e gerenciar branches. Isso permite que os profissionais de desenvolvimento e infraestrutura experimentem novas funcionalidades, correções e outros ajustes sem perturbar o código estável existente.

O que é uma Branch?

Uma **branch** (ou "ramo" em português) é uma linha de desenvolvimento independente. Pense nisso como uma forma de ter várias versões do seu código, cada uma representando diferentes funcionalidades ou ajustes. A branch padrão no Git é chamada de `main` (anteriormente conhecida como `master`).

Criando uma nova Branch

Para criar uma nova branch, use o comando `git branch`, seguido pelo nome que deseja dar à sua branch:

```
git branch nome_da_branch
```

Para começar a trabalhar nessa branch, você precisa "mudar" para ela usando o comando git checkout:

```
git checkout nome_da_branch
```

Dica rápida: Você pode criar e alternar para uma nova branch em um único comando usando:

```
git checkout -b nome_da_branch
```

Visualizando Branches

Para ver uma lista de todas as branches no seu repositório:

```
git branch
```

A branch atualmente ativa (ou seja, a que você está trabalhando) será destacada e marcada com um asterisco.

Mesclando Branches (Merge)

Após terminar o trabalho em uma branch e testar suas mudanças, você pode querer incorporar essas mudanças na branch `main` (ou qualquer outra branch de destino). Isso é feito com o comando `git merge`:

```
# Certifique-se de que você está na branch que deseja mesclar
as mudanças (geralmente a 'main')
git checkout main

# Mesclar a branch
git merge nome_da_branch
```

Deletando uma Branch

Depois de mesclar sua branch, se você desejar, pode deletá-la para manter seu repositório organizado:

```
git branch -d nome_da_branch
```

Use `-d` para garantir que a branch foi mesclada antes de ser deletada. Se você estiver certo de que deseja excluir uma branch sem mesclá-la, pode usar `-D`.

Git, GitHub e repositórios remotos

O Git permite rastrear mudanças no código, enquanto o GitHub, e outras plataformas similares, atuam como um centro para armazenar, compartilhar e colaborar no código. Vamos mergulhar em como usar de forma eficiente essas ferramentas.

Repositórios Remotos

Um **repositório remoto** é basicamente uma versão do seu projeto que fica armazenado na internet ou em qualquer outro servidor na rede. Ele age como uma espécie de "hub", permitindo que diversos colaboradores enviem ou peguem código, mantendo tudo sincronizado e atualizado.

GitHub: Mais que simples armazenamento

O GitHub.com é um dos lugares mais populares para hospedar repositórios remotos Git. Além de simplesmente armazenar o código, ele oferece ferramentas robustas para colaboração, como:

- **Issues:** Para rastrear bugs, tarefas e discussões.
- **Pull Requests:** Propor, revisar e discutir mudanças no código.
- **Actions**: Ferramentas de automação, desde testes até desdobramentos.

Configurando SSH para o GitHub

Antes de podermos enviar ou receber dados do GitHub sem precisar inserir nossa senha o tempo todo, podemos configurar o acesso SSH. O SSH é um protocolo de comunicação seguro, e o GitHub suporta a autenticação SSH para operações Git remotas.

1. **Gerar uma chave SSH:** No seu terminal, digite:

```
ssh-keygen -t rsa -b 4096 -C "seu_email@example.com"
```

2. **Copiar sua chave pública:** Use o comando abaixo para copiar.

```
cat ~/.ssh/id_rsa.pub
```

3. **Adicionar a chave ao GitHub:** Vá para as configurações SSH no GitHub, clique em "New SSH Key", dê um nome descritivo e cole sua chave no campo "Key".

Com isso, agora você pode interagir com seus repositórios no GitHub sem precisar autenticar a todo momento.

Vinculando seu repositório local a um remoto

Uma vez que você tem um repositório no GitHub, você pode conectá-lo ao seu local:

```
git remote add origin URL_DO_REPOSITORIO
```

Aqui, `origin` é apenas um nome padrão para seu repositório remoto principal.

Enviando e Recebendo Mudanças

Para enviar suas mudanças para o GitHub, use:

```
git push origin main
```

Para puxar mudanças do GitHub para seu local:

```
git pull origin main
```