Guia - Ferramenta Git

SUMÁRIO

1. OBJETIVO

Descrever o fluxo de trabalho com a ferramenta de controle de versões Git.

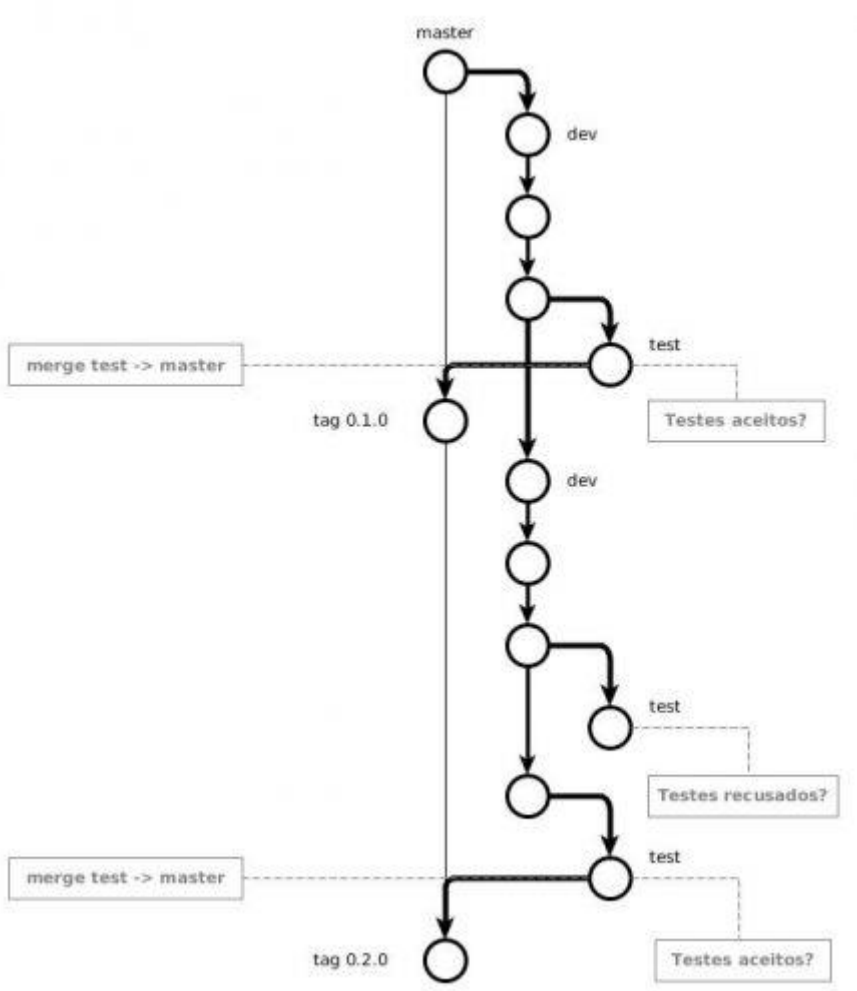
2. REFERÊNCIAS

https://mazer.dev/pt-br/git/introducao/git-workflow-simples/
https://www.alura.com.br/artigos/git-flow-o-que-e-como-quando-utilizar

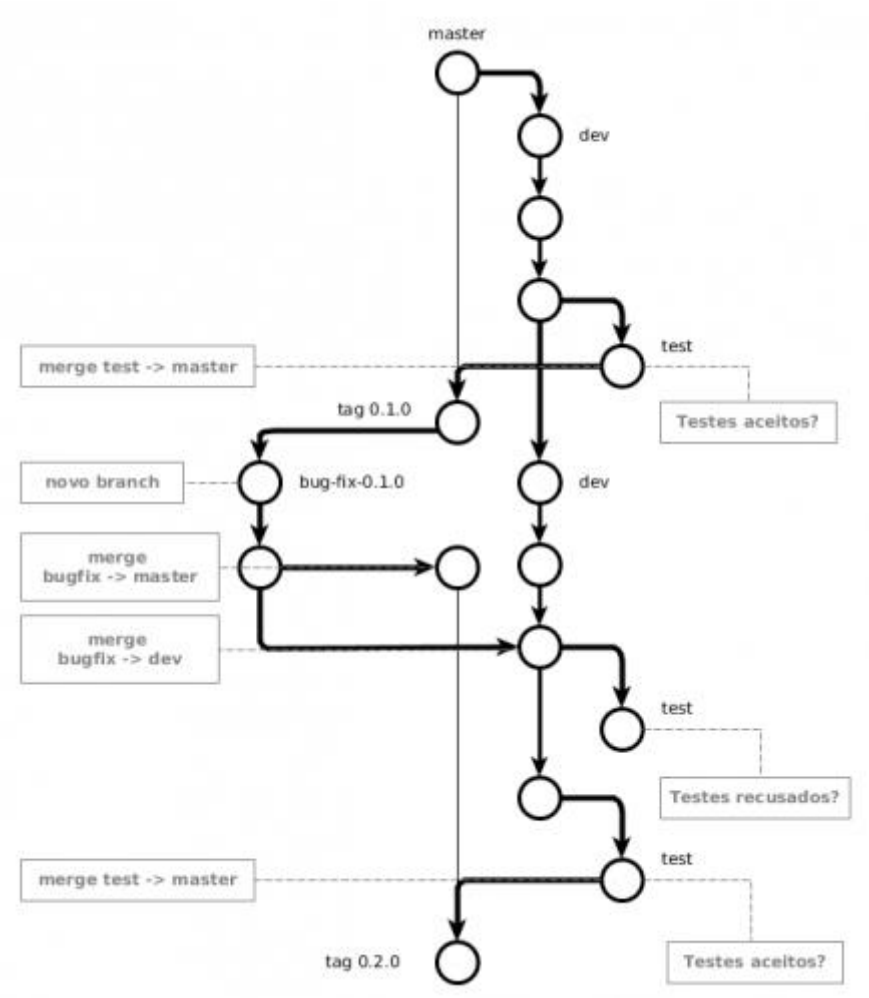
3. GIT FLOW SIMPLES

Este fluxo se baseia no desenvolvimento baseado em troncos (DBT). É prática de gerenciamento de controle de versão em que os desenvolvedores fazem o merge de atualizações pequenas e frequentes no “tronco” ou ramificação principal.

3.1. DBT



3.2. FLUXO PARA CORREÇÃO DE ERROS



3.3. PADRÕES, VERSIONAMENTOS E NOMENCLATURAS DE BRANCHES

Git Flow trabalha com duas branches principais, a **Develop** e a **Master**, que duram para sempre; e três branches de suporte, **Feature**, **Release** e **Hotfix**, que são temporários e duram até realizar o merge com as branches principais.

Então, ao invés de uma única branch **Master**, esse fluxo de trabalho utiliza duas branches principais para registrar o histórico do projeto. A branch **Master** armazena o histórico do lançamento oficial, e a branch **Develop** serve como uma ramificação de integração para recursos.

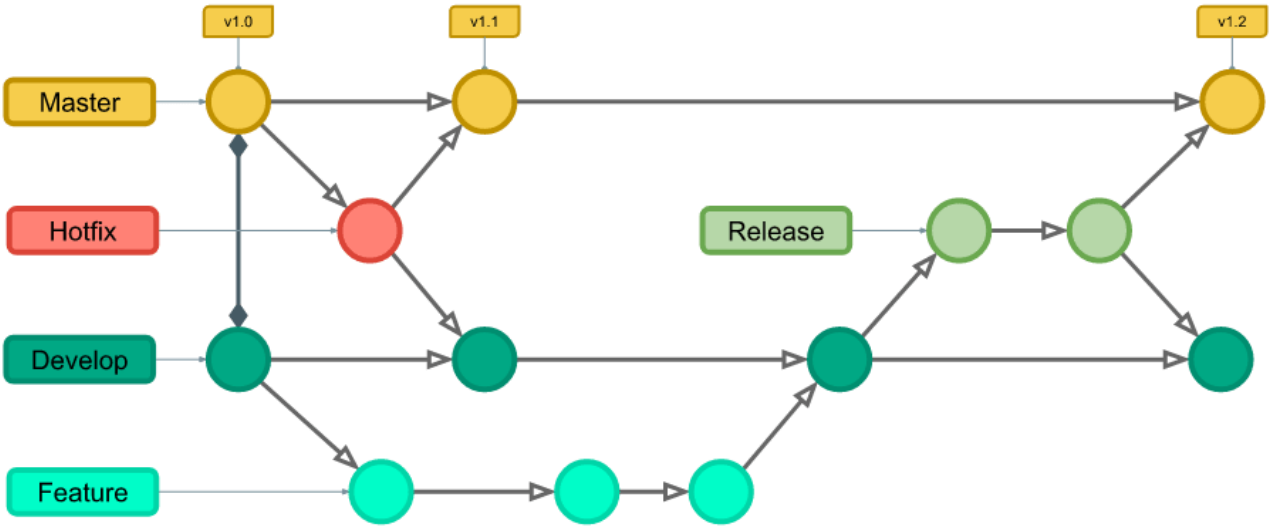
Todos os commits na branch **Master** devem ser marcados com um número de versão.

4.1 Versionamento Semântico (através de tag)

Major = O primeiro dígito informa a versão de compatibilidade e é alterado caso o software ou biblioteca sofra mudanças que a torne incompatível com outras versões. São as chamadas *breaking changes*, atualizações que possuem o potencial de "quebrar" códigos que utilizam versões anteriores.

Minor = O segundo dígito informa a versão da funcionalidade, onde uma nova função ou melhoria substancial é adicionada e não há problemas de incompatibilidade com outras versões.

Patch = O terceiro dígito informa a versão da correção de bugs, melhorias de desempenho ou alterações similares que não alteram as funcionalidades atuais e nem introduzem novas.



3.4. Branch Master/Main

Principal branch, aqui é onde temos todo o código de produção. Todas as novas funcionalidades que estão sendo desenvolvidas, em algum momento, serão mescladas ou associadas a **Master**. As formas de interagir com essa branch são através de uma **Hotfix** ou de uma nova **Release**.

3.5. Branch Develop

É a branch onde fica o código do próximo deploy. Ela serve como uma linha do tempo com os últimos desenvolvimentos, isso significa que ela possui funcionalidades que ainda não foram publicadas e que posteriormente vão ser associadas com a branch **Master**.

3.6. Branch Feature

São branches utilizadas para o desenvolvimento de funcionalidades específicas. Devem seguir a seguinte convenção de nome:

```
feature-<nome da nova funcionalidade>
```

Essas **Features** branches são criadas sempre a partir da branch **Develop**. Portanto, quando finalizada, elas são removidas após realizar o merge com a Branch **Develop**. Se tivermos dez funcionalidades a serem desenvolvidas, criaremos dez branches independentes.

É importante salientar que as branches de **Features** não podem ter interação com a branch **Master**, apenas com a branch **Develop**.

Para integrar com a **Develop** é necessário criar um Pull Request. Uma vez integrado deve-se criar uma tag.

3.7. Branch Hotfix

É uma branch criada a partir da **Master** para realizar correções imediatas encontradas no sistema em produção. Quando concluída, ela é excluída após realizar o merge com as branches **Master** e **Develop**.

Temos uma branch de **Hotfix** para cada hotfix que precisamos implementar!

A grande diferença entre **Feature** e **Hotfix** é que os Hotfix são criados a partir da **Master** e quando os finalizamos, eles são mesclados tanto na **Master** quanto na **Develop**. Isso ocorre porque o bug está em ambos os ambientes.

Além disso, quando fechamos um **Hotfix**, temos que criar uma **tag** com a nova versão do projeto. Isso porque cada mudança que fazemos na Branch **Master** precisa de uma tag que a represente.

3.8. Branch Release

Uma vez que uma etapa de desenvolvimento esteja concluída, teremos em nossa Branch **Develop** todas as **Features** e **Hotfix** mesclados. Então, se quisermos ter todas essas novas funcionalidades na Branch **Master**, teremos que criar uma Branch de **Release**.

A Branch **Release** serve como ponte para fazer o merge da **Develop** para a **Master**. Ela funciona como ambiente de homologação e é removida após realizar os testes do merge com a **Master**. Caso seja encontrado algum bug e haja alguma alteração, ela também deve ser sincronizada com a **Develop**.

Por fim, quando fechamos uma Branch **Release**, temos que criar uma tag com a nova versão de lançamento do software, para que possamos ter um histórico completo do desenvolvimento.

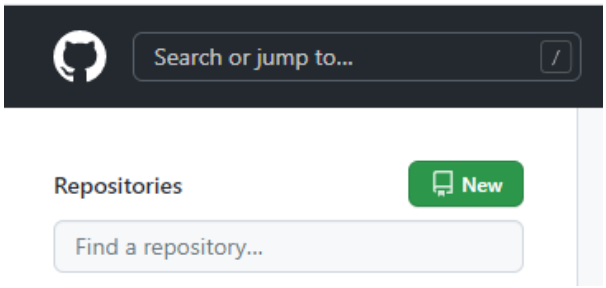
4. CONFIGURAÇÃO

4.1. Instalar o Git

1. Baixe o Git e Instale seguindo as instruções contidas em https://git-scm.com/downloads.

4.2. Criar conta no GitHub

1. Crie uma conta no GitHub seguindo as instruções em https://github.com/.
2. Crie o repositório Git em sua conta



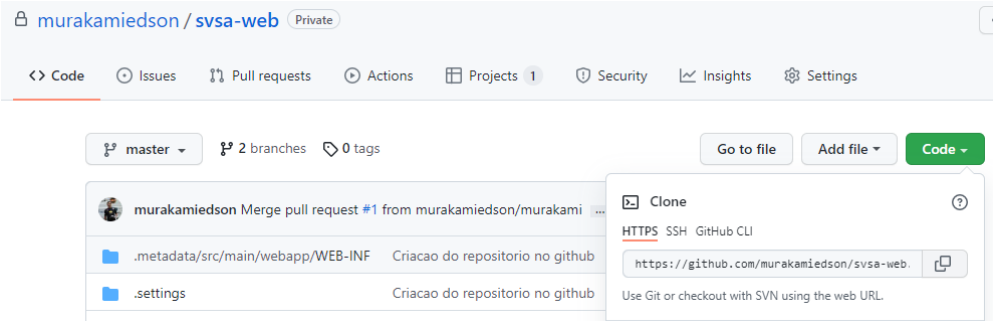
4.3. Criar a pasta do repositório raiz

Crie uma pasta chamada “repo” em um local do seu sistema de arquivos. Por exemplo:

```
C:\repo
```

4.4. “Clonar” o repositório na pasta raiz

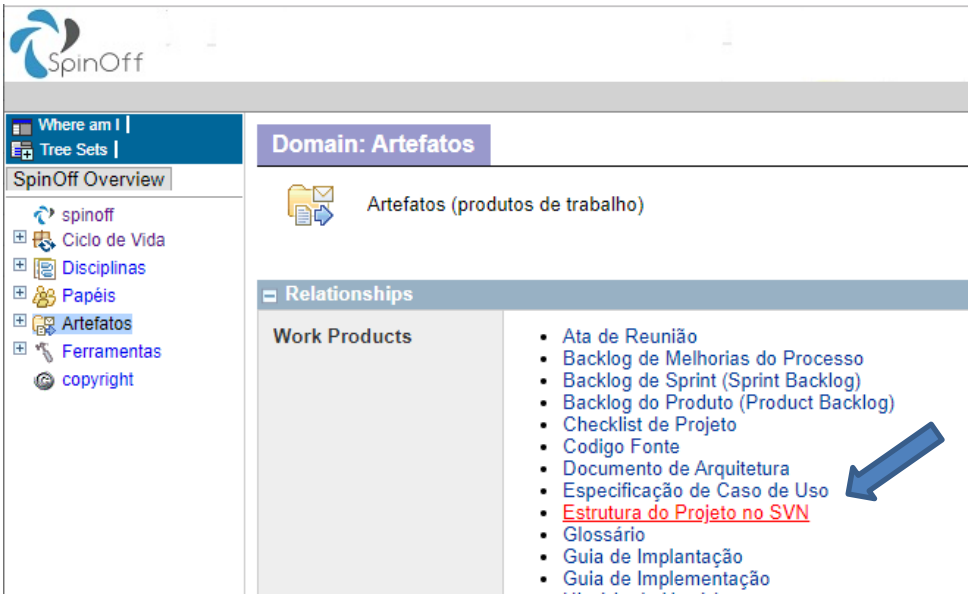
1. Obtenha a URL do repositório



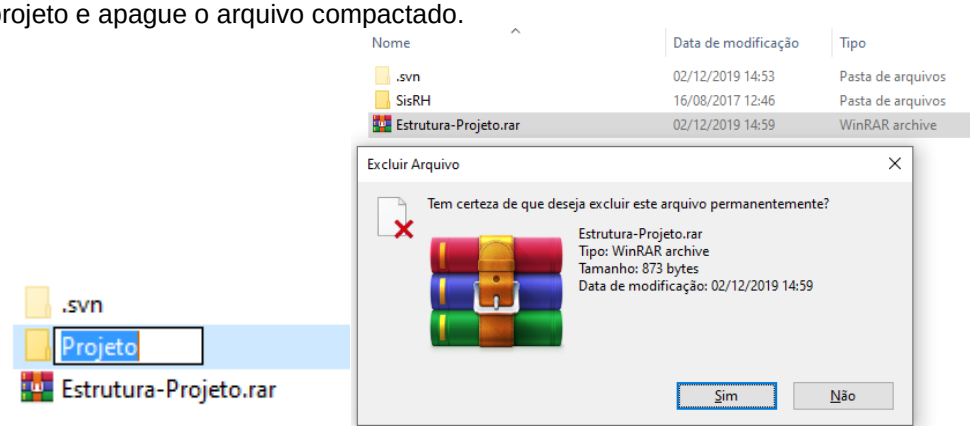
2. Clique com o botão direito sobre a pasta “repo” e abra o Git bash

```
git clone -b master https://github.com/<repo>/<projeto>.git
```

3. Baixe a estrutura de um projeto do SpinOff, conforme figura abaixo, dentro da pasta do repositório criada localmente.



4. Descompacte o arquivo na pasta do repositório, renomeie a pasta “Projeto” com a sigla do seu projeto e apague o arquivo compactado.



5. Clique com o botão direito e a dicione o seu projeto no repositório com a opção “Add”.
6. Depois use o comando “commit” para incluir o projeto no repositório local.
7. Sincronize com o repositório remoto usando o comando “Push”. (envia pastas e artefatos adiconados no repositorio local para o remoto).