Guia do Desenvolvedor

GIT PE

Histórico do Documento

Data	Versão	Descrição	Autor
26/09/2016	1.0	Versão inicial	Celso Sa

Guia do Desenvolvedor Git-PE

Este documento apresenta um guia de uso da plataforma Git-PE. O sistema está associado com a suíte de ferramentas da ATI, para atender as demandas do Estado de Pernambuco através do uso de serviços de TI.

Git

Git é um sistema de controle de versão distribuído e um sistema de gerenciamento de código fonte. É uma ferramenta utilizada principalmente para desenvolvimento de software.

GitLab

Gitlab é um gerenciador de repositórios baseados no Git, que possui issue tracking (controle de ocorrências) e páginas wiki (página web colaborativa do projeto).

Caso queira informações sobre o sistema controle de versão, acesse o link: https://git-scm.com/book/pt-br/v1/Primeiros-passos-Sobre-Controle-de-Versão.

1. Estrutura de papéis e responsabilidades do Git-PE

A figura 1 apresenta a estrutura de papéis e responsabilidades do sistema Git-PE, e como este deve estar distribuído:

Equipe de Administração Git-PE (ATI): Serão os responsáveis pela manutenção e gerenciamento do sistema no geral. Ações como integração com ferramentas ou melhoria do ambiente estarão relacionadas com a equipe de administração do sistema Git-PE.

As solicitações para criar repositórios, deverá ser através da plataforma csati (http://www.csati.ati.pe.gov.br).

Administrador do Repositório: Serão os responsáveis da fábrica de software contratada, para o gerenciamento do repositório do projeto. Nesse caso, o administrador do repositório poderá adicionar novos membros, além de alterar o nível de visibilidade, adicionar proteções as branches, entre outras atividades gerenciais. É possível que um repositório possa ser gerenciado por mais de um administrador, isso deverá ser acordado com a equipe de projetos.

Usuário do Git: Serão desenvolvedores, convidados ou qualquer usuário comum da fábrica de software contratada, que não possua privilégios no sistema para manutenção e gerenciamento. Esses serão os usuários finais do sistema, que utilizarão o sistema Git-PE para trabalhar em seus respectivos projetos.

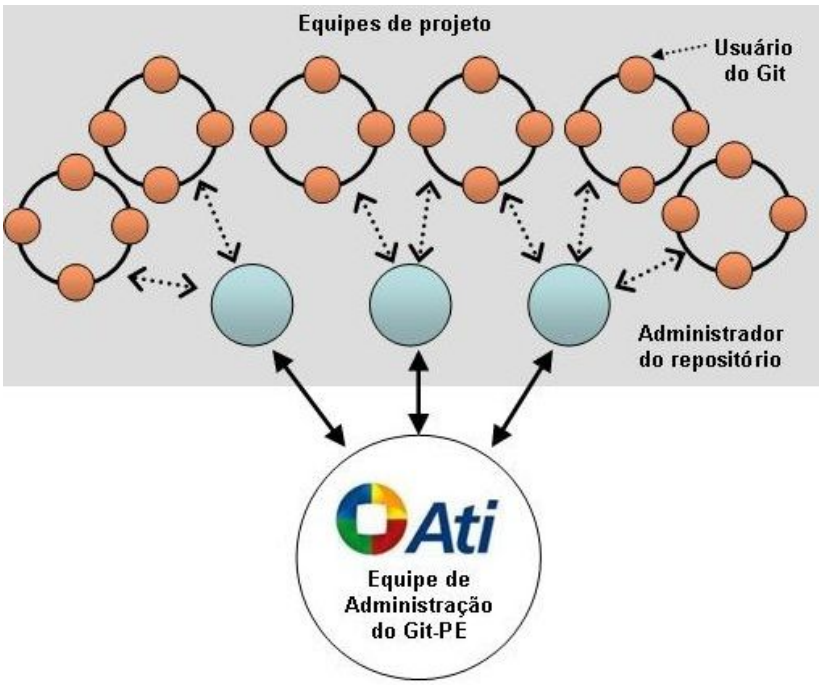

Figura 1: Estrutura de papéis e seus relacionamentos

2. Padrões e ações para as fábricas de Software

A administração do sistema Git-PE solicita alguns padrões e ações, que devem ser seguidas pelas fábricas de software. Esses padrões deverão estar de acordo com as fábricas de software que desenvolvem soluções para o governo.

2.1. Migração do SVN para o Git

Para manter a consistência do repositório, o Git disponibiliza o recurso de migração do SVN. Para isso, as seguintes instruções devem ser seguidas:

1. Criar arquivo para incluir os autores (ex.: autores.txt). O arquivo deve conter as seguintes informações: nome_usuário_git = nome <email>.

```
desenvolvedorA = Desenvolvedor A <desenv1@ati.pe.gov.br>
desenvolvedorB = Desenvolvedor B <desenv2@ati.pe.gov.br>
```

2. Deve seguir os seguintes comandos (utilizando git bash ou shell):

```
# criar uma pasta para o Git, ou iniciar projeto git normalmente
mkdir <pasta do novo projeto git>
cd <pasta do novo projeto git>

# iniciar o projeto SVN
git svn init http://<url svn> --no-metadata

# configurar os usuários do git
git config svn.authorsfile autores.txt
```

```
# sincroniza todas as informações do repositório SVN
git svn fetch

# adicionar a url remota do repositório git
git remote add origin https://<url do repositorio>.git

# enviar para o servidor origin
git push origin master
```

2.2. Branches de Produção, Homologação e Desenvolvimento

Para garantir a segurança e a qualidade, a equipe de administração do Git-PE disponibiliza duas branches que deverão ser utilizadas no desenvolvimento:

- **Master:** Branch de produção. Deve conter o código fonte da aplicação que está implantada no ambiente de produção. Essa versão deve ser validada pela fábrica de software juntamente com os gerentes do projeto. As versões geradas nessa branch deverão conter tags para a realização da implantação. A branch master só poderá ser atualizada mediante merge-request, que será autorizado pelo administrador do repositório.
- **Test:** Branch de Homologação. Deve conter as informações referentes ao ambiente de homologação. A branch pode ser implantada pelas fábricas de software. Todos os commits realizados nessa branch serão automaticamente implantados no sistema do ambiente de Homologação.
- **Dev:** Branch de Desenvolvimento. Deve conter as informações referentes ao ambiente de Desenvolvimento. A branch deve ser utilizada pela fábrica de software para integrar o código realizado pelos desenvolvedores.

Apenas as branches Master e Test serão analisadas pela equipe de administração do Git-PE, focando na qualidade e na consistência. Para isso, a ATI disponibiliza ferramentas para verificar a qualidade do código fonte e realizar testes durante cada implantação.

Portanto, é de extrema importância que as fábricas de software utilizem outras branches para manter o código em desenvolvimento. Para isso, é aconselhável criar branches relacionadas a um responsável ou features para que a fábrica possa trabalhar, sem comprometer as versões de homologação e produção. A figura 2 apresenta o esquema de branches que deverão estar contidas no repositório de um projeto.

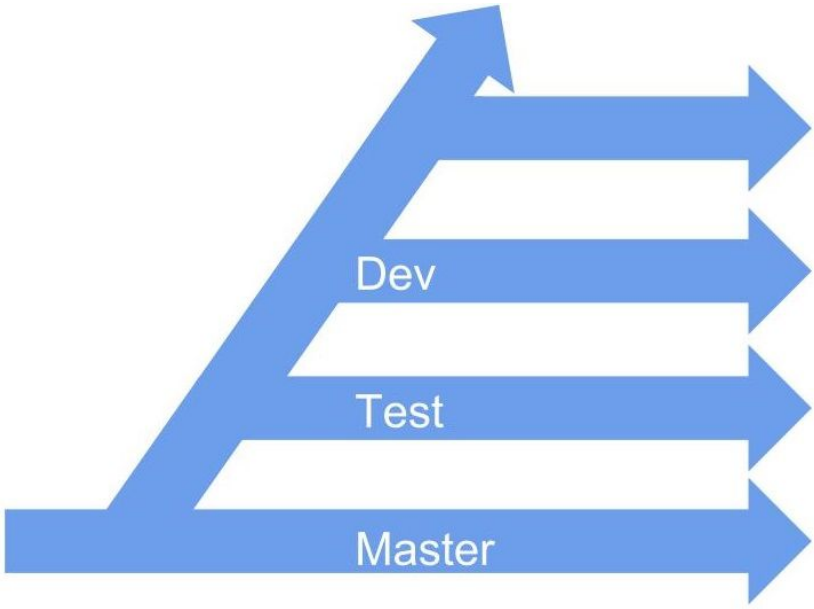

Figura 2: esquema de branches contida no repositório do projeto

3. Qualidade do Repositório de Código Fonte

A administração do sistema Git-PE não se envolve na utilização interna de cada repositório. Essa responsabilidade deve está relacionada com o administrador do repositório. Apenas sugerimos algumas boas práticas de trabalho com relação às características do Git.

3.1. Nível de visibilidade e controle de acesso a informação

Por padrão, todos os projetos devem ser privados, por motivos de segurança. Assim, o administrador do repositório poderá gerenciar quais usuários possuem acesso ao repositório, além de fornecer permissões aos usuários.

3.2 Uso das Branches e das Tags

O uso de branches é importante para manter a consistência e organização das informações de cada repositório. É possível criar diversas ramificações e assim os usuários poderão trabalhar de forma isolada. As branches devem ser organizadas de acordo com as necessidades do projeto, que podem ser utilizadas como ramificações por usuário, por ocorrências ou através de uma branch compartilhada.

O fato de trabalhar de forma isolada reduz bastante a possibilidade de haver conflitos. Porém, ainda há possibilidade de que os conflitos possam ocorrer quando um mesmo arquivo foi alterado, e existe uma nova versão no repositório remoto. Dessa forma, os conflitos entre os arquivos deverão ser tratados isoladamente e resolvidos pelo usuário.

As tags são recursos que ajudam a marcar um determinado commit. Assim, é possível estabelecer tags para possíveis versões do sistema (ex.: v1.0). A prática do

uso de tags facilita o entendimento da data e hora exata em que uma versão foi lançada. As tags podem conter caracteres e valores numéricos, que devem ser padronizadas pelo administrador do repositório.

3.3. Utilização do Merge-Request e branches protegidas

Algumas branches podem ser protegidas para evitar que qualquer usuário possa enviar um código que ainda não foi analisado. Para isso, o usuário pode proteger a branch e apenas os usuários escolhidos pelo administrador podem autorizar que os dados e informações sejam armazenados no repositório remoto. Além disso, o GitLab permite que conversas e trocas de informações sobre parte do arquivo possam ser realizadas.

Em projetos pequenos com equipes que possuem um bom nível de conhecimento, não é aconselhável a utilização de controles de commits de um projeto específico, pois isso torna uma atividade burocrática de forma desnecessária. A equipe pode perder performance devido ao overhead ou sobrecarga de um usuário com privilégios, para analisar cada commit realizado. Entretanto, para equipes heterogêneas e com vários usuários, essa prática se torna viável para evitar que commits indesejados sejam realizados.

3.4. Arquivo README e as páginas Wiki

É sempre importante manter a documentação do projeto atualizada. É de extrema importância que o arquivo README.md esteja atualizado, e de acordo com o projeto. Dessa forma, os usuários poderão ter acesso as informações relevantes do projeto. É comum que esses arquivos contenham informações sobre a configuração técnica do projeto e da equipe. O GitLab disponibiliza recursos de página Wiki para que os usuários possam incluir informações ténicas sobre o projeto. Assim, é de extrema importância manter a equipe sempre atualizada.

Algumas sugestões sobre as informações que devem conter no arquivo README.md: título e uma breve descrição do projeto; tecnologias utilizadas; estrutura da equipe técnica e seus respectivos contatos; informações de configuração e uso do projeto; informações sobre contribuição e organização do repositório, como estrutura de pastas, operações de merge-requests, branchs e padrões de mensagens de commits.

3.5. Uso do .gitignore

Deve-se evitar o máximo de arquivos desnecessários dentro do repositório. O Git conta com uma ferramenta para evitar que arquivos internos de uma pasta rastreada pelo Git sejam elegíveis a serem adicionados ao repositório. O uso do arquivo “.gitignore” permite que o Git desconsidere, arquivos ou pastas para evitar que estes possam ser adicionados ao repositório. Assim, o repositório se mantém limpo, apenas com as informações relevantes e necessárias do projeto.

3.6. Atenção aos Commits

É preciso ter bastante atenção com a prática do commit. Algumas dessas práticas precisam ser fiscalizadas para evitar possíveis problemas no projeto.

Commit frequentemente pequenos trechos: Isso ajuda a reduzir os conflitos entre a sincronização dos repositórios, como merge e rebase.

Não se deve realizar commits parciais: Evite commits de soluções parciais. Apenas realize esta operação quando tudo estiver estável e completo. Além disso, é sempre bom manter apenas uma solução por commit, pois se houver a necessidade de retornar uma versão, evita-se que outras soluções possam ser desfeitas por estarem contidas no mesmo commit.

Padrões de mensagens: As mensagens de cada commit devem ser padronizadas, claras e concisas, explicando o problema e a solução do que foi realizado. Deve-se explicar a solução para que todos da equipe possam entender o que foi solucionado, e o que está contido no commit.

Revise os arquivos que serão commitados: Antes de enviar os arquivos para o repositório, verifique se os padrões estão corretos, e se não há erros. Caso seja possível, realize testes no código antes de commitar e tenha bastante cautela na revisão desses arquivos.

3.7. Configure sua identidade

É preciso definir corretamente o seu nome de usuário e endereço de e-mail. Dessa forma, todos os commits deverão utilizar essas informações, que estarão imutavelmente anexadas.

3.8. Atenção com a operação Rebase

Atenção com a operação rebase. Faça rebase apenas de commits que nunca foram enviados publicamente. Se o rebase for realizado em commits que já foram enviados publicamente, podem haver pessoas que se basearam neles e problemas poderão ocorrer, pois haverá modificações duplicadas com commits diferentes.

Um caso de exemplo desse problema é representado através do link:

- https://git-scm.com/book/pt-br/v1/Ramificação-Branching-no-Git-Rebasing#Os-Perigos-do-Rebase