Guia de Versionamento e Fluxo

Git Flow e SemVer

CI/CD - BIRD

Sumário

1 Introdução

Nesse guia padronizaremos a forma como o time desenvolve, integra e entrega software. Usaremos o **Git Flow** para organizar as ramificações e o **Versionamento Semântico** para comunicar o impacto das mudanças.

2 Configuração do Ambiente

Teremos uma organização do Bird no github, com os pesquisadores adicionados como colaboradores, onde serão criados repositórios específicos para cada projeto.

3 Versionamento Semântico (SemVer)

Vamos usar o padrão **Major.Minor.Patch** (ex: `v1.2.0`). A alteração dos números depende do impacto das mudanças feitas no código:[1]

3.1 Estrutura da Versão (X.Y.Z)

- **MAJOR (X.0.0) - Quebra de Compatibilidade:** Incrementada quando há mudanças drásticas. É geralmente quando **acaba a retrocompatibilidade**.
- **MINOR (0.Y.0) - Nova Funcionalidade:** Incrementada quando tem **novas funcionalidades** adicionadas, mas que ainda são compatíveis com versões anteriores. Exemplo: Adicionar um novo botão na interface.
- **PATCH (0.0.Z) - Correção de Bug:** Incrementada para correções de falhas simples que não alteram funcionalidades. Exemplo: Corrigir um erro de digitação, ajustar uma cor CSS ou corrigir um cálculo.

4 O Modelo Git Flow

A estrutura do repositório vai ser composta pelas seguintes branches principais:[2]

4.1 Branches Permanentes

- `main`: Representa a **Produção**. **Não recebe commit direto**. Só recebe código via Merge de `release` ou `hotfix`. Cada commit deve ter uma **Tag** de versão.
- `develop`: Representa o **Desenvolvimento Contínuo**, é a branch de integração. Ela **contém as funcionalidades completas para a próxima versão**.

[1] Se quiserem ler mais sobre versionamento semântico podem acessar a especificação.

[2] Se quiserem ler mais sobre o modelo do Git Flow, podem encontrar o artigo original do Vincent Driessen (2010), "A successful Git branching model" ou esse em português.

4.2 Branches Temporárias

4.2.1 Feature Branches (`feature/*`)

- **Objetivo:** Desenvolver uma nova funcionalidade.
- **Nasce em:** `develop`.
- **Morre em:** `develop`.
- **Fluxo:** O dev cria a branch, trabalha nela e abre um PR para a `develop`. Após o merge, a branch local pode ser apagada.

4.2.2 Release Branches (`release/*`)

- **Objetivo:** Congela o código para testes de QA e preparação final (documentação, versão), aqui acontece o `Staging`. **Ela é exclusiva para isso, nenhuma feature nova entra aqui.**
- **Nasce em:** `develop` (quando o time decide que vai lançar uma versão).
- **Morre em:** Dois lugares. Ao finalizar a release, ela é mergeada na:
  1. `main`: Para atualizar a produção.
  2. `develop`: Para garantir que correções de bugs feitas durante a fase de release voltem para o desenvolvimento.
- O nome da branch deve seguir o SemVer (ex: `release/v1.2.0`).

4.2.3 Hotfix Branches (`hotfix/*`)

- **Objetivo:** Resolver bugs críticos em produção.
- **Nasce em:** `main`.
- **Morre em:** Assim como a release, ela é mergeada na:
  1. `main`: Para corrigir o erro imediatamente (gera nova Tag Patch).
  2. `develop`: Para garantir que o erro não volte a aparecer na próxima release.
- Geralmente incrementa o **Patch** (ex: `hotfix/v1.2.1`).

5 Exemplo:

Por exemplo, imagine que estamos na **versão v1.1.0**.

1. **Início do Trabalho:** O dev quer criar um "Modo Escuro". Ele cria a branch `feature/dark-mode` a partir da `develop`.
2. **Integração:** Ele termina, abre PR e mergeia na `develop`. Outros devs também mergeiam suas features.

3. **Corte da Release:** O time decide lançar. É criada a branch `release/v1.2.0` a partir da `develop`.
4. **Fase de QA:** O QA testa a `release/v1.2.0`. Encontra um bug no CSS.
5. **Correção na Release:** O dev corrige o bug na branch `release/v1.2.0` (commit de fix).
6. **Lançamento:** A release é aprovada.
    - Mergeiam a `release/v1.2.0` na `main` → Cria-se a Tag `v1.2.0`.
    - MErgeiam a `release/v1.2.0` na `develop` (o bug é corrigido na develop também).
7. **Hotfix:** No dia seguinte, descobrem que o login parou de funcionar na produção (`main`).
8. **Correção do Hotfix:**
    - Criam `hotfix/v1.2.1` a partir da `main`.
    - Corrigem o erro.
    - Mergeiam na `main` (Tag `v1.2.1`) e na `develop`.

6 Glossário de Comandos

Aqui temos um glossário dos comandos que mais usaremos no git, caso alguém não se lembre ou não esteja acostumado. Pode também rodar o comando `git --help` direto no terminal, acessar a documentção do Git ou o glossário da Atlassian.

6.1 Configuração Inicial

```
git clone https://github.com/Brain-BIRDs/seu-projeto.git
cd seu-projeto
```

6.2 Operações Diárias

Fluxo Básico para Feature:

```
# 1. Garanta que esta atualizado
git checkout develop
git pull origin develop

# 2. Crie sua branch
git checkout -b feature/minha-tarefa

# ... Trabalho sendo feito ...

# 3. Salve e envie
git add .
git commit -m "feat: Adiciona nova tela"
git push origin feature/minha-tarefa
```

6.3 Boas Práticas de Commit

Para que seja fácil entender e encontrar o que desejamos, é bom seguir padrões de commit:

- **feat:** Nova funcionalidade.
- **fix:** Correção de bug.
- **docs:** Alteração em documentação.
- **style:** Formatação (ponto e vírgula, espaços).
- **refactor:** Melhoria de código sem mudar funcionalidade.