01

COMMITS SEMANTICOS

Tudo o que você precisa saber sobre commits semânticos

UHELLITON

E por que eu devo usar isso nos meus projetos?

Melhorar a legibilidade do histórico de versionamento do Git;

Aumentar a velocidade na hora de procurar por mudanças específicas no código; Entender, facilmente, quais mudanças estão sendo deployadas dentro de um pipeline;

Possibilitar a geração de um CHANGELOG ou de release notes de maneira totalmente automatizada;

Incentivar os desenvolvedores a realizarem commits de maneira pensada e específica, sem realizar commits grandes e cheios de mudanças;

Tipos

Uma visão completa dos tipo

- **feat:** Tratam adições de novas funcionalidades ou de quaisquer outras novas implantações ao código;
- **fix:** Essencialmente definem o tratamento de correções de bugs;
- **refactor:** Tipo utilizado em quaisquer mudanças que sejam executados no código, porém não alterem a funcionalidade final da tarefa impactada;
- **style:** Alterações referentes a formatações na apresentação do código que não afetam o significado do código, como por exemplo: espaço em branco, formatação, ponto e vírgula ausente etc.);
- **test:** Adicionando testes ausentes ou corrigindo testes existentes nos processos de testes automatizados (TDD);
- **docs:** referem-se a inclusão ou alteração somente de arquivos de documentação;
- **env:** utilizado quando se modifica ou adiciona algum arquivo de CI/CD.Exemplo: modificar um comando do Dockerfile ou adicionar um step a um Jenkinsfile.
- **build:** Alterações que afetam o sistema de construção ou dependências externas (escopos de exemplo: gulp, broccoli, npm),

Escopos

<tipo>[escopo opcional]: <descrição>

- feat(login/routes): novas configurações de rota para o login
- melhorando as configurações de login
- fix(AuthService): ajustando a url de autenticação
- correção url de autenticação
- refactor: padronizando logs em todo o serviços
- ajustando o log de serviços
- style: indentação e padronização no código usando o lint:fix
- padronização de código
- build: adicionando variável para rebuild automáticos nos containers docker
- ajustes no containers docker

- **feat(routes/settings): adjust settings to be called in any screen.**
- **feat: ajuste as configurações de rotas a serem chamadas em qualquer tela.**

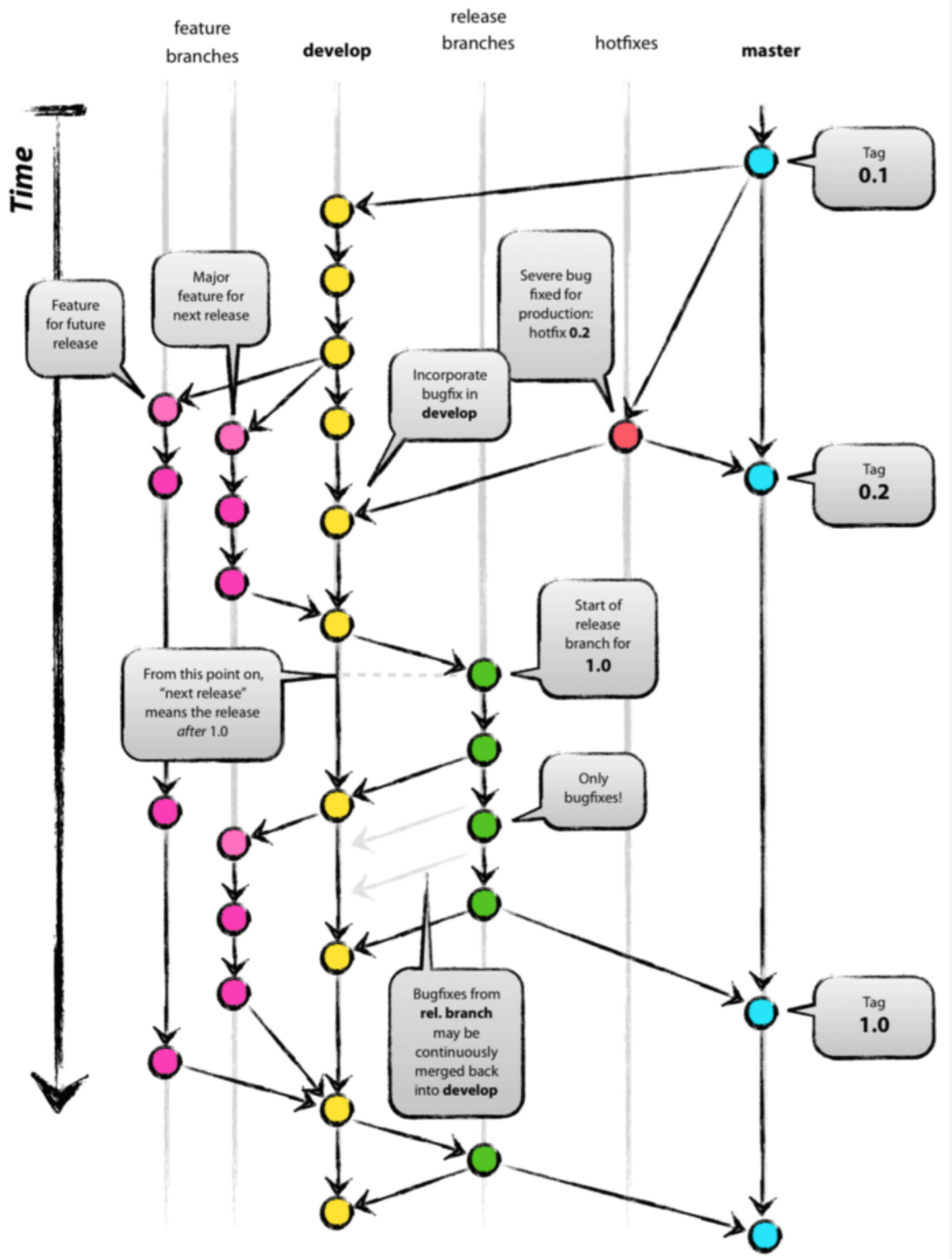
Git Flow

diretrizes para a organização dos nossos branches

Tipos

Uma visão completa dos tipo

- **master:** contém o nosso código de produção, todo o código que estamos desenvolvendo, em algum momento será “juntado” com essa branch
- **develop:** contém o código do nosso próximo deploy, isso significa que conforme as features vão sendo finalizadas elas vão sendo juntadas nessa branch para posteriormente passarem por mais uma etapa antes de ser juntada com a **master**
- **feature/*:** são branches para o desenvolvimento de uma funcionalidade específica, por convenção elas tem o nome iniciado por feature/, por exemplo: feature/cadastro-usuarios. Importante ressaltar que essas branches são criadas sempre à partir da branch **develop**
- **hotfix/*:** são branches responsáveis pela realização de alguma correção crítica encontrada em produção e por isso são criadas à partir da master. Importante ressaltar que essa branch deve ser juntada tanto com a master quanto com a **develop**
- **release/*:** tem uma confiança maior que a branch develop e que se encontra em nível de preparação para ser juntada com a master e com a **develop** (caso alguma coisa tenha sido modificada na branch em questão)



feature: para novas implementações

release: para finalizar o release e tags

fix: para resolver problema simples que pode esperar até o final da split ou lançamento da release

hotfix: para resolver problema crítico em produção que não pode esperar novo release

Neste caso, como já estamos na develop:

- **git checkout -b feature/novo-componente**
- **git checkout -b hotfix/login-jwt**
- **git checkout -b release/v1.0.1**
  -- merge feature/novo-componente
  -- mege hotfix/login-jwt