API – APRENDIZAGEM POR PROJETOS INTEGRADOS

Revisão: 2025-2

GUIA DO GITHUB

Sumário

1. Documentação no GitHub 2
2. Padrão de Mensagens dos Commits 3
3. Estratégia de Branch 4
4. Boas práticas para Uso do Git 5

1. Documentação no GitHub

Uma boa documentação no GitHub é importante para que outras pessoas (até mesmo o próprio autor) compreenda o seu objetivo, as dores trazidas pelo desafio do projeto e outros detalhes.

A seguir apresentam-se os itens fundamentais para compor um bom **README** do GitHub:

README (principal)

- Título do Projeto
- Descrição do Desafio (com a dor do Parceiro)
- Backlog de Produto
- Cronograma de evolução do Projeto (de forma visual)
- Tabela descritiva das Sprints com as colunas:
  . Período da Sprint
  . Link para Documentação da Sprint
  . Link para Vídeo no Youtube do Incremento entregue
- Tecnologias utilizadas
- Estrutura do Projeto
- Como executar, usar e testar o projeto
- Link para Pasta de Documentação
- Equipe (com nome completo, papel, foto, Link para GitHub e Link para LinkedIn)

Outros elementos obrigatórios de um bom projeto no GitHub:

- Pasta de Documentação
  . Checklist de DoR e DoD
  . DoR e DoD, em si, por Sprint
  . Manual de Usuário

Exemplos de bons GitHubs:

Curso-Turma / Período	Link do GitHub
6º ADS / 2025-1	https://github.com/BuzzTech-API/API_ADS_6SEMESTE_2025.1

2. Padrão de Mensagens dos Commits

A padronização de mensagens de commits é uma prática importante, pois, além de ajudar na compreensão do histórico de commits, facilita a criação de ferramentas automatizadas baseadas na especificação.

Existe um padrão convencional de mensagens (https://www.conventionalcommits.org/en/v1.0.0/).

Porém, visando simplificar:

<tipo> (<id_demanda1>, <id_demanda2>, ..., <id_demandaN>): <descrição da entrega feita no commit>

<tipo>	***Descrição***	***Exemplo***
<feat>	Quando da adição de um recurso, uma "feature" (funcionalidade)	feat (AB-1243, AB-56): Implementação dos repositories usados nas operações com as tabelas de variações climáticas
<fix>	Correção de um bug	fix (#45): Correção do componente de seleção de município
<docs>	Atualização de documentação	docs (#45): inclusão de diagrama de modelo de BD para a aplicação
<style>	Mudança de formatação, sem afetar o código	style (AB-1243, AB-56): ajuste de nomes de variáveis para o padrão camelCase
<refactor>	Refatoração do código, sem alterar funcionalidade	Seguir exemplos anteriores, alterando o tipo, IDs e descrições correspondentes
<test>	Adiciona ou modifica testes	Seguir exemplos anteriores, alterando o tipo, IDs e descrições correspondentes
<chore>	Atualizações menores que não impactam diretamente a funcionalidade do código	Seguir exemplos anteriores, alterando o tipo, IDs e descrições correspondentes

<id_demandaN> - Identificador da demanda criada na ferramenta de gestão de Stories/Tasks que o Time estiver usando (Github Issues, Jira Software, GitLab Issues, etc, podendo estar entre 1 e N.

<descrição da entrega feita no commit> - Descrição clara sobre o que está sendo entregue no commit criado e enviado para o Git.

3. Estratégia de Branch

Branches (em português ramificações) são um recurso que permite à uma equipe trabalhar em diferentes versões do código ao mesmo tempo, sem interferir no que já está estável e funcionando.

Podemos criar uma Branch para armazenar uma nova funcionalidade, uma outra para corrigir um bug, testar melhorias, etc. Por exemplo, poderíamos ter a Branch “feature/login”.

Excelente simulador e curso online para aprendizado de Git Branching: https://learngitbranching.js.org/?locale=pt_BR

Seja qual for a estratégia, precisa ser bem descrita e justificada para o seu projeto!

Existem algumas estratégias mais simples – seguem dois exemplos:

GitHub Flow

Funcionamento:

- A branch main (ou master) é sempre a versão estável.
- Cada funcionalidade ou correção é feita em uma **branch separada**.
- Após o desenvolvimento, abre-se um **Pull Request** para revisão e merge.

Exemplos de branches:

main

feature/login-form

bugfix/navbar-crash

Características:

Muito fácil de entender e funciona bem com qualquer tamanho de equipe.

Trunk-Based Development

Funcionamento:

Tudo gira em torno de Branch principal (*main*). Dev´s criam pequenas branches e integram rapidamente.

Características:

Menos branches, menos complexidade; incentiva integrações rápidas.

4. Boas práticas para Uso do Git

Trabalhar com Git vai muito além de fazer upload e versionamento de código! A seguir algumas dicas de boas práticas:

Antes de iniciar os commits no projeto GIT, crie um arquivo **.gitignore** que irá ignorar arquivos de logs, binários e arquivos com senhas. Você pode encontrar vários modelos na internet;

Nunca realize commit direto na Main. Essa prática poderá facilmente ocasionar em perda de código quando você atuar em conjunto com outros desenvolvedores. Utilize a criação de branchs;

Especifique a referência única (número da tarefa, nome da tarefa) em cada commit;

Realize **pequenos commits,** sempre mantendo a última versão mais estável para o commit atual;

Seja direto nos commits, pequenas modificações com pequenos comentários. Utilize o imperativo, por exemplo: “Adicionado box-shadow no botão da home”;

Ao criar um PR (*Pull Request*), **resumidamente descreva o que foi implementado**. Se não houver outras ferramentas que descrevam os testes, esse também é um ótimo espaço para descrever todos os cenários de teste;

Sempre utilize o ciclo de Kanban, ou seja, aguarde *code review* e análise do código. Preferencialmente não aprove o seu próprio PR (*Pull Request*), sempre que possível realize o processo completo.

Referência:

Disponível em: https://pt.linkedin.com/pulse/boas-pr%C3%A1ticas-para-git-leticia-coelho: