Components for building structured code versioning

Ensuring stability and efficiency

Jan Friebe

- 32 years old
- live in NRW

Full Stack developer with passion for Frontend stuff
JavaScript/TypeScript // Vue // React // CSS // Design

jan.friebe.de friebe jan_friebe

The main blocks at a glance

1. Branch strategy
2. Branch naming
3. Approval System
4. Semantic commit message
5. Tags
6. Lifecycle examples

One branch strategy

Dont get me wrong here, its totally fine to work with one branch if you are alone or your team have a very very strong communication (difficult)

Disadvantages to work with one branch

- ✕ Conflicts with simultaneous changes
- ✕ Difficulty in tracing changes
- ✕ Delays in deploying updated
- ✕ Lack of flexibility

Multiple branches strategy (feature branches)

- dev (developmet - feature, refactoring, bug(no hotfixes), docu changes…)
- main (customer build - snapshot like)

For all new features, bugs, tests, experiments and so on you should create an isolated new branch!

Categorize your branches

features bugfix release hotfix experiment username docs config

Prefix your branch names

- feature/user-authentication
- bugfix/123-fix-login
- release/v1.0
- hotfix/v1.0-security-fix
- experiment/user-interface-redesign
- docs/update-readme
- config/update-env-variables

Advantages to work with multiple branches

- ✓ Isolation of changes
- ✓ Collaboration
- ✓ Code reviews
- ✓ Feature development
- ✓ Bug fixing
- ✓ Experimentation

Approval System

A feature that allows devs to review and approve changes (pull request) made by others before merging them into the main codebase

Advantages to work with the approval system

- ✓ Code quality
- ✓ Knowledge sharing
- ✓ Consitency
- ✓ Risk mitigation
- ✓ (Alignment with requirements)
- ✓ Team collaboration
- ✓ Continuous improvement

Semantic commit messages

Commit messages follow a structured format that conveys meaningful information about the changes

Categorize your commit in msg types

feat fix docs style refactor test (adding or modifying test) core (other changes related to build e.g. maintenance tasks

Structure of a message

1. Type
2. Scope (context of the change e.g. module name)
3. Short summary (changes that have been made e.g. remove if clause)
4. Longer description (optional)

Message example

```
feat(user-authentication): Add OAuth2 login functionality
Implement OAuth2 authentication using Google Sign-In Api
- Configure OAth2 client Id and secret
- Add login button to the user
```

type = feat

scope = user-authentication

short summary = Add OAuth2 login func

Longer description = Detailed description & bullet points

Tags

Tags are like a snapshot feature to mark a specific point, denote a milestone or a delivered build. No more and no less. Easy.

Lifecycle of a feature branch

1. Create a feature branch
2. Implement the feature in code base
3. Commit your changes (semantic commit messages)
4. Iterate and refine (fix bugs, refine your code)
5. Test your feature (unit tests, manuel testing)
6. Review and collaboration via PR (feedback of others)
7. Merge the feature branch into main
8. Be happy
9. Cleanup - delete merged feature branches

Lifecycle of a release branch

1. Create a release candidate branch
2. Feature Freeze
3. Bug fixes and stabilization
4. Testing
5. Feedback and iteration
6. Documentation and release notes
7. Code review and approval
8. RC builds
9. Final testing
10. Release

END