Freeleaps Development with DevOps Guideline

1. Revision

Revision	Modifier	Modified At	Comments
0.0.1	@Zhenyu Sun	2025/03/05 (UTC+8)	Complete the drafting of the first version.

2. Introduction

This document describes the specification when development with Freeleaps DevOps System.

Every developer that needs to develop with Freeleaps DevOps System must follow this specification.

3. Development Workflow

The development workflows describe how the CI/CD pipeline works powered by Freeleaps DevOps System.

Whatever projects you are working on, the CI/CD always with same stages as:

- Commit message linting
- Check which codes need to be deploy through Git commit history
- Generate dynamic stages for specific code change set
    - Prepare build environment
    - Pre-caching dependencies for projects
    - Semantic releasing if trigger branch is master
    - Compilation and packaging
    - Build container image with specific builder (Docker in Docker or Kaniko)
    - Persistence new image version tag to Helm values file

As you can see, the project deployment configurations must be packed with Helm charts format to adapt with Freeleaps Kubernetes Cluster.

There have 2 concepts hidden with Freeleaps DevOps Pipeline (CI/CD Pipeline):

1. The master branch always used to deploy production environment.
2. The develop branch always used to deploy snapshot version to alpha/test/dev environment.

Which means we need to follow strict Git flow to make Freeleaps DevOps Pipeline work.

4. Strict Git Workflow

The strict Git workflow is used to adapt Freeleaps DevOps Pipeline.

Your Git repository must follow these rules:

- There always have 2 permanent branches:
    - develop: Represents on development application codes
    - master: Represents application codes snapshot working on production
- The master branch needs to be protected:
    - No one can push codes to master directly
    - Only pull requests from develop or hotfix are accepted
    - Only maintainer has authorized to merge pull requests from develop
- The develop branch needs to be protected:
    - No one can push codes to develop directly except maintainer
    - Only pull requests from other branches (except master) are accepted
    - Only maintainer has authorized to merge pull requests
- Every develop branch for features or any changes needs to fork from develop

- Hotfix branch could be forked from master if there has any urgent issue needs to be fix
- The master branch must be called to master
- The develop branch must be called to develop
- The develop branch for features or any changes must be prefixed with feature, there are some examples:
    - feature/add-user-api
    - feature/metrics-api
    - doc/guideline
- The hotfix branch must be prefixed with hotfix, there are some examples:
    - hotfix/npe
    - hotfix/oom
- Commit message must be follow rules described in section 5
- Hotfix branch must be merge into develop at same time when it's merged to master

5. Conventional Commit Message

The conventional commit message is a lightweight convention on top of commit messages. It provides an easy set of rules for creating an explicit commit history; which makes it easier to write automated tools on top of.

The commit message should be structured as follows:

```
<type>(optional scope): <description>

[optional body]

[optional footer(s)]
```

The commit contains the following structural elements, to communicate intent to the consumes of your projects:

1. fix: a commit of type fix patches a bug in your codebase (this correlates with PATCH in semantic releasing)
2. feat: a commit of type feat introduces a new feature to the codebase (this correlates with MINOR in semantic releasing)
3. BREAKING CHANGE: a commit that has a footer BREAKING CHANGE, or appends a ! After the type/scope, introduces a breaking API change (correlating with MAJOR in semantic releasing). A BREAKING CHANGE can be part of commits of any type
4. types other than fix and feat are allowed, for example @commitlint/config-conventional (based on the Angular Convention) recommends build, chore, ci, docs, style, refactor, perf, test and others.
5. footers other than BREAKING CHANGE may be provided and follow a convention similar to git trailer format

Additional types are not mandated by the conventional commits, and have no implicit effect in semantic releasing (unless they include a BREAKING CHANGE). A scope may be provided to a commit's type, to provide additional contextual information and is contained within parenthesis, e.g., feat(user): add ability to create user

Here are some examples:

Commit message with description and breaking change footer

feat: allow provided config object to extend other configs

BREAKING CHANGE: `extends` key in config file is now used for extending other config files

Commit message with ! to draw attention to breaking change

```
feat!: send an email to the customer when a product is shipped
```

Commit message with scope and ! to draw attention to breaking change

```
feat(api)!: send an email to the customer when a product is shipped
```

Commit message with both ! and BREAKING CHANGE footer

```
chore!: drop support for Node 6

BREAKING CHANGE: use JavaScript features not available in Node 6.
```

Commit message with no body

```
docs: correct spelling of CHANGELOG
```

Commit message with scope

```
feat(lang): add Polish language
```

For more about conventional commit message specification, please see: https://www.conventionalcommits.org/en/v1.0.0/