Trunk Based Development

Trunk-based development is a software development practice where all developers work on a single branch, often referred to as the "trunk" or "main" branch. This approach contrasts with other branching strategies, such as Git Flow, where developers create multiple long-lived branches for features, releases, or hotfixes.

Key Characteristics of Trunk-Based Development

- **Single Branch**: Developers commit their changes directly to the main branch. This reduces the complexity of managing multiple branches and merges.
- **Frequent Commits**: Developers are encouraged to commit small, incremental changes frequently. This helps in identifying issues early and reduces the risk of large, complex merges.
- **Continuous Integration**: Trunk-based development is often paired with continuous integration (CI) practices. Automated tests are run on every commit to ensure that the codebase remains stable.
- **Feature Flags**: To manage incomplete features without affecting the main branch's stability, developers use feature flags. This allows them to toggle features on or off without deploying separate branches.
- **Short-lived Feature Branches**: If feature branches are used, they are short-lived and merged back into the trunk as soon as possible. This minimizes the divergence from the main branch.

References

- Trunk-Based Development
- Trunk-based development - Atlassian

Roles and Responsibilities for Developers

- **Frequent Commits**: Commit small, incremental changes frequently to the main branch. This helps in identifying issues early and reduces the risk of large, complex merges.
- **Continuous Integration**: Ensure that every commit triggers automated tests to verify the stability of the codebase. This practice helps catch integration issues early.
- **Use Feature Flags**: Implement feature flags to manage incomplete features. This allows you to merge code into the trunk without exposing unfinished features to users.
- **Short-Lived Branches**: If feature branches are necessary, keep them short-lived. Merge them back into the trunk as soon as possible to minimize divergence and potential conflicts.
- **Automated Testing**: Maintain a robust suite of automated tests, including unit tests and integration tests, to ensure code quality and functionality.
- **Code Reviews**: Conduct regular code reviews to maintain code quality, share knowledge, and ensure adherence to coding standards. This also helps in catching potential issues early.
- **Communication and Collaboration**: Foster open communication within the team to ensure everyone is aligned with the project goals and aware of ongoing changes. This helps in avoiding duplicate work and conflicts.

- **Maintain a Releasable Trunk**: Keep the trunk in a releasable state at all times. This practice supports continuous deployment and allows for quick responses to market changes.
- **Cultivate a CI Culture**: Encourage a culture of continuous integration where developers are committed to integrating their work frequently and validating changes through automated builds.
- **Manage Feature Toggles**: Regularly review and retire unused feature toggles to prevent them from cluttering the codebase and complicating the development process.

Prefixes, Branches and tags name

We will use the following **prefixes** following with a Slash `/`

- **ftr/**, for Feature branch.
- **bgfx/**, for Bug Fix branch.
- **htfx/**, for Hot Fix branch.
- **refactor/**, for Refactoring branches.
- **doc/**, for Documentation branches.
- **enh/**, for Enhancement branches.
- **dev/**, for Experimental branches - for POCs, trying new library, new implementation of existed feature - .

Branch name should contains, **Jira Ticket Number** and a **short descriptor** of the task with **hyphens (-)** as separators

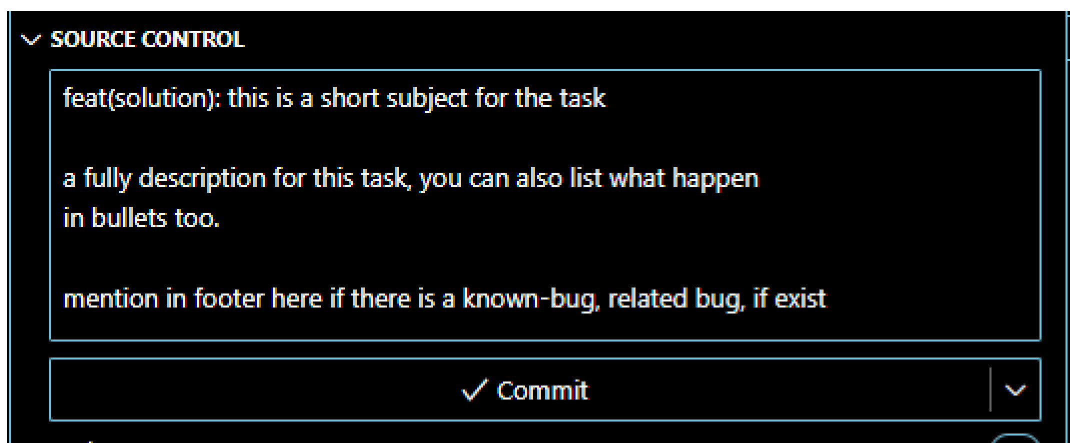
> Example: `ftr/WCA-78453-create-login-form`, this is important to have the Jira Ticket Number in the branch name to make it easier to identify the branch and the task. and also to reflect in the Jira Platform.

Tags name should contains only version number `v1.0.2`,

How to write a commit [ref]

Frontend solution use commitlint as a checker for the commit message. You can commit through two ways only

1- **VSCode**



2- **command-line**

```
D:\WakeCap\FRONTEND_PROJ (master -> origin)
λ |
```

Here is how to write a commit message, and you cannot push your commit unless you follow the pattern

```
feat(scope): short subject

body

footer
```

Copy

The **type** value must be one of: [**build**, **chore**, **ci**, **docs**, **feat**, **fix**, **perf**, **refactor**, **revert**, **style**, **test**]

The **scope** value must be one of:

```
[
  "docs",
  "solution",
  "web_root",
  "web_nav",
  "web_admin",
  "web_pm",
  "web_mt",
  "shared_ui",
  "shared_tui",
```

```
    "shared_icons",
    "shared_helpers",
    "shared_comps"
  ]
```

> **Scope** will has a new values if we have a new projects in the future, for each new project we should add a new value for the scope in `.commitlint.json` file.

How to write a PR

While you are creating, GitHub will load for you a list of templates, you can use them to create a PR. Select the template you want to use.

Husky

Now we have husky configuration for the branch name and commit message, it will check if the branch name and commit message are valid before you commit or push your code.

- For **branch name**, it will check if the branch name is valid and if it contains the Jira Ticket Number and a short descriptor of the task with **hyphens (-)** as separators.
- For **commit message**, it will check if the commit message is valid and if it contains the type, scope, short subject, body, and footer.

Merging Strategy

We will use the **rebase strategy** for merging the branch to the main branch. Also you have to **squash your commits** before merging to the main branch.