Gitlabuserguide

What is GitLab?

GitLab is a web-based DevOps platform that provides a complete CI/CD pipeline, source code management (SCM), and collaboration tools. It integrates version control with CI/CD, issue tracking, and code review, allowing teams to develop, deploy, and monitor applications efficiently. GitLab helps streamline workflows, improve collaboration, and automate the development lifecycle.

Git Branching Model

Branches Overview

`dev` Branch

- **Purpose**: Key branch for development.
- **Features**: All feature branches are created from and merged back into `dev` .
- **Bug Fixes**: All bug fixes in `dev` code are branched off `dev` and merged back into it.
- **Maintenance**: This branch should always be kept up-to-date with the `stage` branch by the IN team lead.

`stage` Branch

- **Purpose**: For release-related activities.
- **Bug Fixes**: All release-related bug fixes should be done in the `stage` branch. No new features should be forked off this branch.
- **Tagging**:
  - Tag the branch with `*-alpha.n` for alpha testing.
  - Tag the branch with `*-beta.n` for beta testing.
- **Deployment**:
  - Once testing is complete, merge the code into the `main` branch.
  - Tag it with the respective version and deploy it to production manually.
  - Run automated and manual tests.

`main` Branch

- **Purpose**: The production branch.
- **Bug Fixes**: Any bugs found in production should be addressed directly in the `main` branch.
- **Maintenance**:
  - Once bugs are fixed, pull the changes back into the `stage` branch by the AF (Assignee/Team).
  - After updating the `stage` branch, pull the changes into the `dev` branch by TODO (To Do/Assignee).
  - Finally, pull the bug fixes from `dev` into the respective feature branches by the developers (devs).

Workflow Summary

Here, we have one OR more separate repositories to maintain the code based on requirements.

So far,the workflow we use is one of many branches. And these branches are designated to each correspondent server.

Server URL	Server Stakeholders	Branch Name
https://production-domain/	QA / Client	main
https://stage-domain/	QA / Client Review / UAT	stage
https://dev-domain/	QA / Developers	dev

`main` is main branch; only the merge touches it (more on this in a bit)

There are `stage` & `dev` branches, taken initially from `main` , that all developers work off. Instead of having a branch per developer, we make feature, or ticket, branches from `dev` .

1. **Feature Development**:
   - Branch off `dev` for new features.
   - Merge feature branches back into `dev` .
2. **Bug Fixes**:
   - Address bugs in `main` .
   - Pull fixes into the `stage` branch.
   - Update `dev` with fixes from `stage` .
   - Ensure fixes are pulled into feature branches.
3. **Release Management**:

- Perform bug fixes in `stage`.
- Tag and deploy from `stage` to production.
- Run tests post-deployment.

For every discrete feature (bug, enhancement, hotfix etc.), a new local branch is made from `main`. Developers don't have to work on the same branch, since each feature branch is scoped to only what that single developer is working on. This is where its cheap branching comes in handy.

Once the `dev` & `stage` branches are verified by QA, we will release the branch and merge it back into `main`. This is the only time we touch the `main`, ensuring that it is as clean as possible.

Therefore the next time each developer is on some task and does the pulls, they'll get all of the updated code which will include the stuff merged from `dev` & `stage`.

This may OR not be sound long winded but it's actually fairly simple and robust (every developer could also just merge locally from `main` but it's neater if one person does that, the rest of the team live in a simple world much like the single developer workflow).

Type of irregular Branches:

Suppose there is a ticket TICKET-XXXX for a new task/bug.

Type	Description	Branch Name
Feature/Issue/Bug	New features, Bug, Issue related task	TICKET-XXXX
Hotfix	Issue/Bug related task for PRODUCTION Server	HOTFIX-TICKET-XXXX

A bug is classified as hotfix. For hotfix, a hotfix branch will be cut directly from `main` branch.

Tags

- **Alpha Testing**: `*-alpha.n`
- **Beta Testing**: `*-beta.n`
- **Production**: Respective version tags

How to Test a Pull Request (PR)

Prepare the Local Folder for Testing PR

```
# Fetch all changes from the remote repository
git fetch --all

# Checkout the PR branch
git checkout <BRANCH_NAME>
# For example:
# git checkout 1000-fix-the-current-code

# Switch to the 'dev' branch
git checkout dev

# Merge the PR branch into 'dev' (without committing)
git merge --no-commit --no-ff <BRANCH_NAME>
# For example:
# git merge --no-commit --no-ff 1000-fix-the-current-code
```

File and Configuration Exclusions

Before pushing code to GitLab, ensure the following types of files and configurations are not included in the repository:

- **File/Folder Types**:
  - Image Files: `jpg`, `jpeg`, `png`, `gif` (Except core files, which never been changed)
  - Video Files: `mp4`, `avi`, `mov`
  - Audio Files: `mp3`, `wav`, `ogg`
  - Archive Files: `zip`, `tar`, `rar`
  - Compiled Binaries and Executables: `exe`, `dll`, `so`, `o`, `a`
  - Document Files: `pdf`, `doc`, `docx`, `xls`, `xlsx`
  - Auto-generated Files: Including `.lock` and `.tmp`
  - Personal configuration or IDE Files
  - Files containing sensitive information: `env`, `key`, `pem`, `crt`
  - Hidden System Files: `.DS_Store`, `Thumbs.db`
  - Others Files: `mkv`, `webm`, `svg`, `flv`, `sql`, `log`, `bk`, `db`, `sql`, `env`, `conf`, `tmp`, `lock`
  - Build Artifacts: `dist`, `build`, `target`
  - Dependency Folders: `node_modules`, `vendor`, `Pods`
  - Caches
- **Configuration Files**:

- **Database Configuration**: Files containing database connection strings, credentials, and sensitive configuration details (e.g., `database.yml` , `.env`).
- **API Keys and Secrets**: Configuration files that store API keys, tokens, or other sensitive information.
- **Credentials**: Any file containing user credentials, passwords, or security keys.

Best Practices:

1. **Use** `.gitignore` : Configure a `.gitignore` file to exclude the above file types and sensitive configurations from being tracked by Git.
2. **Review Changes**: Always review the list of files being committed to ensure no sensitive or unnecessary files are included.
3. **Environment Variables**: Store sensitive information in environment variables rather than in the codebase.
4. **Configuration Management**: Use configuration management tools or services to handle environment-specific settings securely.

Useful Commands:

Git add:

By using the command `git add` , you can prepare staged changes to push over the branch. This process involves a few different command line operations. [Refer here in depth].

Git commit:

Adding commits helps to keep track of your progress and changes as you work by using the command `git commit -m “WRITE-SOME-MESSAGE”` . Git will consider each commit a change point or "save point". It is a point in the project you can go back to if you want to find some code from there. [Refer here in depth].

Git pull:

If you want to update your working repository with a remote or any other local branch, this command `git pull origin branch-name` will help you to do that. [Refer here in depth].

Git push:

Use `git push origin branch-name` to push commits made on your local branch to a remote repository.

Let's Understand the Actual Scenarios:

Task Details: User’s Listing with Filters

1. Assigned to Dev:

Suppose, ticket (`TICKET-XXXX`) is assigned to (Dev - x). The x developer will make a new branch from the `main` branch. If it is a hotfix and need to accomodate it ASAP over production, consider `HOTFIX-TICKET-XXXX`)
The branch name should be the same as the Ticket number with the prefix `TICKET-` . Example: `TICKET-XXXX`

```
git checkout main
git pull origin main
git checkout -b TICKET-XXXX
```

Using this command, the `TICKET-XXXX` branch will be generated from the main branch and needs to be pushed to `TICKET-XXXX` .

```
git push origin TICKET-XXXX
```

2. Push Code:

The developer will make changes and push the code into the `TICKET–XXXX` .

Before pushing the code, it's essential to check the changes in the files appropriately. Ensure no unnecessary changes like spacing are included.

```
git add .
git commit -m “First commit message”
git push origin TICKET-XXXX
```

3. Deployed on Dev: (TICKET-XXXX to dev)

Once code review is done, the code can be merged with the following process:

```
git checkout dev
git pull origin dev
git pull origin TICKET-XXXX
```

Changes will be pulled into the `dev` branch automatically. If there are any conflicts, they need to be resolved:

```
git add .
git commit -m “Commit message”

```

```
git push origin dev
```

If there are no conflicts, just run:

```
git push origin dev
```

This will upload all the changes to the `dev` branch.

4. Deployed on Stage: (TICKET-XXXX to stage)

Once the ticket passes the QA process in `<dev-domain>` , we upload the changes to the stage server (`<stage-domain>`).

```
git checkout stage
git pull origin stage
git pull origin TICKET-XXXX
git push origin stage
```

5. Deployed on Live: (TICKET-XXXX to main)

Once the ticket passes the UAT/QA process in `<stage-domain>` , it is uploaded to the `<prod-domain>` .

```
git checkout main
git pull origin main
git pull origin TICKET-XXXX
git push origin main
```

Deployment Process:

While we use automation (Jenkins) for deployments #1 to #3, steps #4 and #5 are done manually by moving files folder-wise.

This is how we manage the project/feature/code deployment process. Along with GIT activities, some build processes are required for specific technologies, which are handled via automation (Jenkins).

By following these guidelines, you can maintain a clean and secure codebase.