Conclusions

 Git uses branches as the canonical solution to solve all the problems related to sharing and supporting multiple versions of the design. Major features supported are:

1. Lightweight Branching
   Git branches are much more lightweight than other version control systems. Designers are encouraged to create branches for each independent line of development and commit often to share code.

2. Better Merging
   Git can detect multiple merge scenarios and supports multiple strategies to deal with them. The various strategies supported are:
   - Fast Forward
     - If merging back into the original branch which has had no new commits, Git simply “fast forwards” the branch pointer.
   - 3-Way Merge
     - Used when both the branches have commits that are not in the other branch. Git creates a merge commit using the 2 latest commits and their common ancestor.
   - Rebase
     - Git allows the user to take a series of commits on a branch and replays them on top of another commit.

   - Cherry Pick Commits
     - Pick Specific Commits From Other Branches then commit them to the local branch.
   - Assigning Blame
     - Find out when the problem line was introduced in the repository.
   - Find Bad Commits Using Bisect
     - Repeatedly divide the commit history between the “good” commit and the “bad” commit to find the commit that introduced the bug.

4. Naming Convention For Branches
   We recommend creating the following infinitely living branches on the central server:
   - “origin/develop” that is the state of top of tree. All regressions are run on the tip of this branch and once passing will be a candidate for making a release.
   - “origin/master” as the release branch. Commits on this branch reflect the releases done by the projects. These should be tagged as per the release conventions.

For sharing code we recommend creating “Topic” branches that are temporary and eventually merged back.

References


Contact

Jeffery Scott (jeffery@juniper.net)
Sanjeev Singh (sanjeevs@juniper.net)