Git: Best Practises

|  |  |
| --- | --- |
| Author: | Ian Addison |
| Document Name: | Git: Branching Strategies |
| Document Location: | [\\lukpc210\Projects\Product Projects\Operations\Vitic 2017.1\Project Documents\POC\VXRW-272](file:///\\lukpc210\Projects\Product%20Projects\Operations\Vitic%202017.1\Project%20Documents\POC\VXRW-272) Git Best Practises.docx |
|  |
| Date Created: | 03/03/2017 |
| Date Last Modified: | 10/03/2017 |
| Status: | Draft |
| Confidentiality: | <Select/enter confidentiality status> |
| Circulation: | <Detail circulation list> |
| Document Version: | 0.2 |

Contents

[1. Overview 2](#_Toc476917159)

[2. Using Git 3](#_Toc476917160)

[3. Adoption of Git in Lhasa 4](#_Toc476917161)

[3.1 Repository Structure 4](#_Toc476917162)

[3.2 Branching Strategy 4](#_Toc476917163)

[3.3 Committing and Pull Requests 5](#_Toc476917164)

[3.4 Rebasing 6](#_Toc476917165)

[References 9](#_Toc476917166)

# Overview

This document has been written to identify the best practises which should be adopted along with Git in order to implement it successfully. Part of implementing Git is learning how it works and the useful commands. This will not be covered in depth in this document as there are lots of tutorials available, however section 2 of this document gives a list of useful links for those new to Git.

In order to adopt Git effectively and get the most from it, Git should not be seen as a drop in replacement for SVN. Instead there are a number of best practises and guidelines which should be adopted in conjunction with adopting Git that were not necessarily required when using SVN. These are covered in section 3 of this document, along with the benefits they bring.

# Using Git

There is a lot of information available on the web around Git, this section lists the books, documents, articles and tutorials that we feel are most useful for those who are new to Git.

**Git Immersion:** <http://gitimmersion.com/>

A Git tutorial which covers using Git from the point of view of a developer.

**Pro Git Book:** <https://git-scm.com/book/en/v2>

Covers Git from a developer’s perspective, but also goes explains how Git works internally. The book is available online using the link above.

**Definitions of “fast forward”, “rewind” and “replay”:** <https://jason.pureconcepts.net/2017/02/git-rebase/>

Contains definitions of terms such as fast forward, rewind and replay.

# 3. Adoption of Git in Lhasa

## 3.1 Repository Structure

A separate document [1] has been written to cover migrating an SVN repository to Git. This will migrate the existing repository to Git keeping the SVN structure. In order for branching, merging and Continuous Integration to be successful when using Git, the proposed structure for a project in Git moves away from a single repository containing the project and sub projects, to a repository for each sub project as follows:

* org.lhasalimited.vitic.acceptance.test: This repository contains the acceptance tests
* org.lhasalimited.vitic.api.test: This repository contains the API tests
* org.lhasalimited.vitic.backend.web: This repository contains the backend code and unit tests. The backend may be split into several sub projects but there will be a single repository containing the backend project(s)
* org.lhasalimited.vitic.frontend.web: This repository contains the frontend code and unit tests
* org.lhasalimited.vitic.performance.test: This repository contains the performance tests

The advantage of splitting the project into several repositories is explained in section 3.2.

## 3.2 Branching Strategy

One user story may be split into several subtasks. Assume we have the following user story and subtasks:

* VXRW-1 (user story): Search for SMILES string
  + VXRW-2 (subtask): API tests
  + VXRW-3 (subtask): Acceptance tests
  + VXRW-4 (subtask): Frontend
  + VXRW-5 (subtask): Backend
  + VXRW-6 (subtasks): Performance tests

Here a new branch would be created **for each** **subtask** from the mainline of development in that repository. All commits on the branch should include the Jira ticket number. When the work is completed on **the user story** a pull request can be created for each branch so the code can be merged into the main line of development. The pull requests should be merged into master together. This process is detailed in section 3.3.

As there is a separate repository for each sub project, when a subtask is started a branch would be created only in the repository corresponding to that subtask. In the example above, we would end up with the following:

|  |  |
| --- | --- |
| **Repository** | **Branch** |
| org.lhasalimited.vitic.acceptance.test | acceptance/VXRW-3 |
| org.lhasalimited.vitic.api.test | api/VXRW-2 |
| org.lhasalimited.vitic.backend.web | backend/VXRW-5 |
| org.lhasalimited.vitic.frontend.web | frontend/VXRW-4 |
| org.lhasalimited.vitic.performance.test | performance/VXRW-6 |

Note the naming convention of the branch. This in the form <PROJECT>/<JIRA\_REF>. A short description could be given after the Jira ticket number if required. Please be aware that if using Git branches as SonarQube project names (from Jenkins), some characters are not supported in SonarQube project names, which may make the project not be displayed in SonarQube.

If there was a single Git repository containing all sub projects, it would mean that in order to work on the acceptance tests (VXRW-3), a branch would need to be created which would contain all sub projects and not just the sub project relevant to the ticket being worked on.

Another advantage of having a repository per sub project is that merge conflicts are easier to handle. For example, using the single Git repository approach may result in merge conflicts across multiple sub projects (e.g. backend and API tests). A single developer would need to resolve these merge requests but a developer responsible for backend work would not necessarily be the best person to resolve a conflict on the API tests (and vice versa).

The branching strategy defined in this section is the simplest and takes into account working in an Agile environment (using Scrum and involving the test team throughout the sprint), however there is not a one size fits all approach with regards to branching. There are other branching strategies available which could be used where this approach does not fit.

## 3.3 Committing and Pull Requests

When all work on a ticket (branch) is completed, a pull request should be created so the branch can be merged into the main line of development. Nothing should ever be committed directly to the main line of development. The reason for this is a peer review is performed on the pull request, and only once that peer review has given the ok can the code be merged. This helps prevent substandard code making its way into the main line of development.

Before creating a pull request, all commits on the branch should be squashed into a single commit. There are two ways of achieving this:

* Commit squash [2]: This allows the developer to commit their changes in several commits and then squash them into a single commit just before creating the pull request. The command git rebase –i (interactive) is used to achieve this.
* Commit amend: For the initial commit on a branch, the developer commits using git commit. Subsequent commits are made using git commit --amend which merges the changes into the previous commit rather than creating a new commit.

Both methods achieve the same result; a single commit. The reason for squashing multiple commits into a single commit is two-fold:

* The commit history on the mainline of development is kept concise (there is a single commit for each Jira ticket)
* If the full history of commits on branches were ported over to the main line of development, the commits related to a specific ticket would be interspersed with the commits on other branches

A pull request might highlight issues with the code that need fixed. At this stage there is already a single commit on the branch which has been pushed to remote, and fixing the issues and committing would result in an additional commit. This can be alleviated by performing the following actions when making changes in light of a pull request review:

* Make code changes and commit using git commit --amend
* When pushing the changes to the remote, use git push --force

Please be aware that if other people are working on this branch their revision history will conflict with the new one, so it is bad practise to do this on a branch if you are not the sole developer.

This scenario is not just specific to pull requests. If a developer has been committing during normal development and has squashed their commit then pushed this to the remote, then any subsequent commits will need to be pushed using git push -–force so that they are squashed into a single commit.

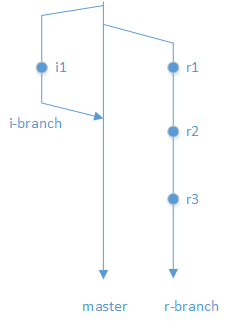
## 3.4 Rebasing

One usage of rebase was mentioned in section 3.3, where it can be used in interactive mode to squash a number of commits into a single commit. This section explains a different reason for using rebase.

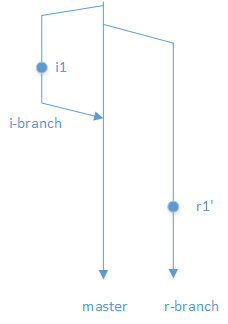
When working on a branch, there will be several other branches being worked on in parallel by other developers. Before creating a pull request, it is good practise to use the command git rebase master branch (where branch is the branch you are working on). This does the following:

* Makes a note of the commits made on the branch and un-applies them
* Applies the commits made on master to the branch
* Applies the commits made on the branch back to the branch

A detailed explanation of this process can be found in [3].

The following diagrams highlight the process. Figure 1 shows two branches being worked on in parallel. Here i-branch was created from master before the r-branch was created and is merged back to master whilst development is still ongoing on r-branch. There have been 3 commits on the r-branch (r1, r2 and r3).

Figure

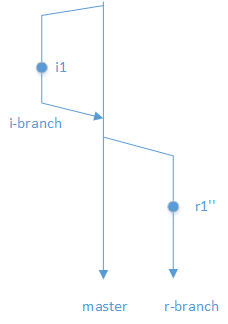
The developer working on r-branch decides their work is complete and wants to create a pull request to merge the r-branch to master. Firstly, the commits on the r-branch should be squashed into a single commit. Squashing commits into a single commit was covered in section 3.3. Figure 2 shows the result of squashing the commits. Now the r-branch has a single commit (r1’), this is a different commit object to r1, r2 and r3.

Figure

Had the developer working on r-branch not squashed the commits, the commit history (once r-branch is merged to master) could end up as follows:

* r1
* i1
* r2

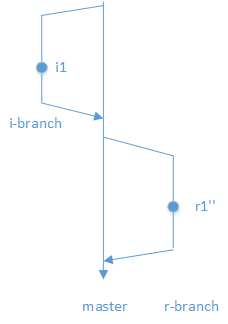
Note how the commits are on the r-branch are interspersed with commits on the other branches making the commit history less readable. This highlights why it is important to squash commits (as mentioned in section 3.3).

After the developer working on r-branch has squashed their commits, but before they push to master, they should ensure any changes merged into master in the meantime are pulled into the branch. This prevents the likelihood of merge conflicts (especially where branches are long running). This can be achieved using the command git rebase master (when the r-branch is checked out ) which will apply the changes from master then replay the commit (r1’ above) as a new commit (r1’’) as shown in Figure

Figure

Given that branches are continuously being merged to master, it is good practise to use the command git rebase master on at least a daily basis to ensure that the branch does not diverge too much and to resolve potential merge conflicts early on. This also ensures that you run all tests on all of the code before merging into the main line of development, ensuring that you don’t break the build with code that has not been tested together.

Figure 4 shows how the flow would look after the merge of r-branch to master.



Figure

# References

[1] SVN to Git Migration

\\lukpc210\Projects\Product Projects\Operations\Vitic 2017.1\Project Documents\POC\sprint\_bride\_of\_chucky\SVN to Git Migration v0.2.docx

[2] Squashing Git Commits

<https://makandracards.com/makandra/527-squash-several-git-commits-into-a-single-commit>

[3] Git Rebase

<https://jason.pureconcepts.net/2017/02/git-rebase/>