Source Code Management Practices

(Git/GitHub Version Control, Branching Strategy)

Contents

Introduction: [3](#__RefHeading___Toc79888832)

**Source Control Management Practices**[: 4](#__RefHeading___Toc79888833)

***Branching***[: 4](#__RefHeading___Toc79888835)

***Review***[***:***](#__RefHeading___Toc79888839) 6

***Conflict***[:](#__RefHeading___Toc79888842) 6

***Testing:***6

***Merging:*****6**

***Tagging:*****6**

**SCM practices with examples*:*****7**

**.gitignore File*:*****10**

**Introduction**

Source code management (SCM) is used to track modifications to a source code repository. SCM tracks a running history of changes to a code base and helps resolve conflicts when merging updates from multiple contributors. SCM is also synonymous with Version control.

As software projects grow in lines of code and contributor head count, the costs of communication overhead and management complexity also grow. SCM is a critical tool to alleviate the organizational strain of growing development costs.

## **The importance of source code management tools**

When multiple developers are working within a shared codebase it is a common occurrence to make edits to a shared piece of code. Separate developers may be working on a seemingly isolated feature, however this feature may use a shared code module. Therefore developer 1 working on Feature 1 could make some edits and find out later that Developer 2 working on Feature 2 has conflicting edits.

Before the adoption of SCM this was a nightmare scenario. Developers would edit text files directly and move them around to remote locations using FTP or other protocols. Developer 1 would make edits and Developer 2 would unknowingly save over Developer 1’s work and wipe out the changes. SCM’s role as a protection mechanism against this specific scenario is known as [Version Control](https://www.atlassian.com/git/tutorials/what-is-version-control).

SCM brought version control safeguards to prevent loss of work due to conflict overwriting. These safeguards work by tracking changes from each individual developer and identifying areas of conflict and preventing overwrites. SCM will then communicate these points of conflict back to the developers so that they can safely review and address.

This foundational conflict prevention mechanism has the side effect of providing passive communication for the development team. The team can then monitor and discuss the work in progress that the SCM is monitoring. The SCM tracks an entire history of changes to the code base. This allows developers to examine and review edits that may have introduced bugs or regressions.

## **The benefits of source code management**

In addition to version control SCM provides a suite of other helpful features to make collaborative code development a more user friendly experience. Once SCM has started tracking all the changes to a project over time, a detailed historical record of the projects life is created. This historical record can then be used to [‘undo’ changes](https://www.atlassian.com/git/tutorials/undoing-changes) to the codebase. The SCM can instantly revert the codebase back to a previous point in time. This is extremely valuable for preventing regressions on updates and undoing mistakes.

The SCM archive of every change over a project's life time provides valuable record keeping for a project's release version notes. A clean and maintained SCM history log can be used interchangeably as release notes. This offers insight and transparency into the progress of a project that can be shared with end users or non-development teams.

SCM will reduce a team’s communication overhead and increase release velocity. Without SCM development is slower because contributors have to take extra effort to plan a non-overlapping sequence of develop for release. With SCM developers can work independently on separate branches of feature development, eventually merging them together.

Overall SCM is a huge aid to engineering teams that will lower development costs by allowing engineering resources to execute more efficiently. SCM is a must have in the modern age of software development. [Professional teams use version control](https://bitbucket-marketing.atlassian.com/version-control-software) and your team should too.

**Source Control Management Practices:**

## **Git Branching Strategy for QA Automation**

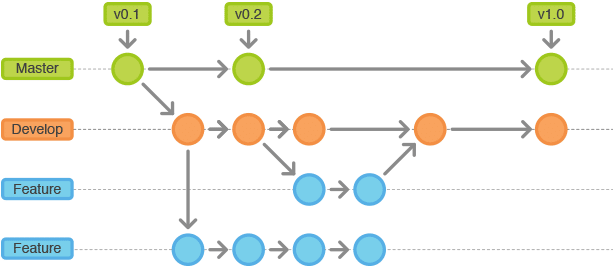
Let’s understand branching strategy with the help of an example, and here I take the opportunity to introduce [Agile](https://www.toolsqa.com/agile/what-is-agile/) & [Sprint](https://www.toolsqa.com/agile/scrum-framework/) as well. Let us say, there is a working product/website which has done a few releases in the past. A new release, let’s say Release-5, is in development and there are few sprints planned for it. The test team is supposed to deliver automated tests for every feature build in the Sprints. There can be multiple features in the Sprint as well, depends on its size. For this discussion, assuming there are.

I prefer using Master, Develop & Feature branches, so I will explain the above scenario according to this approach. It goes like this in a hierarchy:

**===> MASTER**

**===> DEVELOP**

**===> FEATURE**



### **Master Branch**

Origin/Master is the main branch where we have the production code.

To match it with our example above, Master will be only updated once, which is when the development work of Release-5 is finished and the new tests are running fine on Develop Branch. Now let's understand What is Develop Branch.

### **Develop/Release/Integration Branch:**

Origin/Develop is the main branch where the source code of HEAD always reflects a state with the latest delivered development changes for the current release.

In line with our example. this will always reflect the latest committed development work done for Release-5. Some would call this the “**integration branch**” or some "**release-branch**". Integrate-Branch because this branch contains all the merged code of developed features, here the Production code integrates with new development code. Release-Branch because it holds the code of the new release. The idea behind this branch is to make sure that we don’t mess up the production code branch, which is Master in this example.

### **Feature Branch**

This is a bit tricky I would say, just because people make use of this as per their understanding and their comfortability. Few people use this for per feature development by creating feature branches per feature. Some people use one feature branch for the whole team. Unlike the main branches, these branches always have a limited lifetime, since they will be removed eventually once the feature development is done.

With respect to our example, a team can have different feature branches for a per feature in a Sprint. But the purpose of the feature branches is to merge back to Develop branch ASAP. The guiding principle is to develop individual features in an as isolated manner as possible.

Assume you have multiple members checking in code for multiple features in the same Feature branch. It might happen that code for someone can break the build or introduce some bug in the feature branch which will hinder the work of other people working on the branch. To avoid such setbacks it is advisable that we have different feature branch for different features.

**Best Practices**

* While working on automation of any new Testcase clone the master branch so you will get latest code.
* Create a local branch with proper name. Use following naming conventions

## Use issue tracker Ids/User Story Id/Task Id in branch names

## Add a short descriptor of the task

## Use hyphens as separators

Example: 722-add-billing-module,

722-updating-billing-module,

722-fixing-billing-module.

Once all development is done in local branch push it to remote branch using relative push commands. While committing code keep follow these practices.

## **1. Make small commits**

It is a good practice to push code more often and not end up with a messy repo. Make small commits more frequently and avoid committing large chunks of code. This makes it easy to glance through the commit history and find what you are looking for. It is recommended that the use of git add . and git add -A should be in moderation and instead the focus should be on making frequent commits.

**2.** **Commit complete and well tested code**

Never commit incomplete code. This goes against the concept of committing. If you are working on a large task, try to break it down to smaller assignments and insure that each task is complete. Also, Get in the habit of testing your code prior to the commit stage.

###### **3. Write good commit messages**

Your commit log should tell a story. Therefore, Writing descriptive commit messages keeps your repository well managed and makes it easy to navigate through your commit log. Your commit message should be short, in present tense and explicitly say why you made the change.

Example: If applied, this commit will **fix a -- bug**

If applied, this commit will **remove -- method**

If applied, this commit will **update markup**

## **4. learn to use Git with command line.**

### **Create a Merge/Pull Request**

Now, you’ll need to share the code you’ve just written, so team can review it and it will become part of the work of the team. Depending on the code host, this will be a merge or pull request. Generally, this is done in the web interface of the code host.

**Review**

Code checked in by developer will be reviewed by team lead before committing into development branch

In doing a code review, you should make sure that:

* The code is well-designed.
* The functionality is good for the users of the code.
* Any changes are sensible and look good.
* Any parallel programming is done safely.
* The code isn’t more complex than it needs to be.
* The developer isn’t implementing things they might need in the future but don’t know they need now.
* Code has appropriate unit tests.
* Tests are well-designed.
* The developer used clear names for everything.
* Comments are clear and useful, and mostly explain why instead of what.
* The code conforms to our style guides.

**Conflict**

A merge conflict is an event that takes place when Git is unable to automatically resolve differences in code between two commits. Git can merge the changes automatically only if the commits are on different lines or branches.

There are a few steps that could reduce the steps needed to resolve merge conflicts in Git.

* The easiest way to resolve a conflicted file is to open it and make any necessary changes
* After editing the file, we can use the **git add a** command to stage the new merged content
* The final step is to create a new commit with the help of the **git commit** command
* Git will create a new merge commit to finalize the merge

**Testing**

Team lead would be responsible for testing the code on develop branch before merging

**Merging**

Only team lead will merge code into master branch.

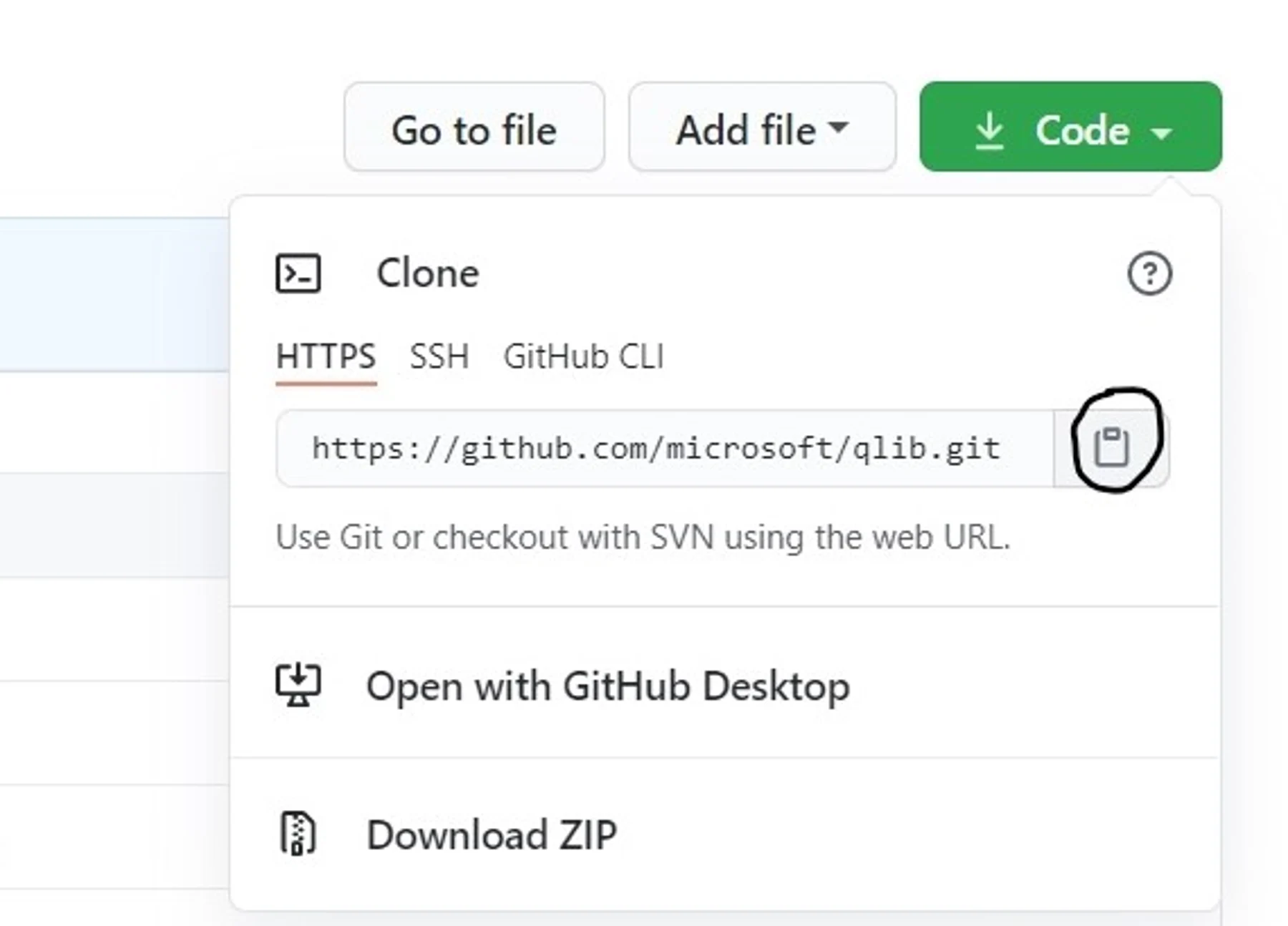
**Tagging**

Tags are ref's that point to specific points in Git history. Tagging is generally used to capture a point in history that is used for a marked version release (i.e. v1. 0.1). A tag is like a branch that doesn't change. Unlike branches, tags, after being created, have no further history of commits.

**Lets understand SCM practices with some examples and git commands**

1. When we create a repository/project on git or github master branch is created by default.

2. Clone the remote repository to local



click code and copy the URL and use following command to clone repository:

**git clone {repository URL}**

3. Create a develop/feature branch using the git branch command.

**git branch {feature name}**

4. Check out the branch you just created using the git checkout command.

**git checkout {feature name}**

5. List the branches you have locally using the git branch command. Branch name with \* is your current branch

**git branch**

6. Make any changes and testcase scripts to this branch.

7. Add those changes using following command. ‘.\*’ will add all the changes, if you want to add particular files write path of those file comma separated. git status command will give you list of file paths that has been changed.

**git add . \***

Note: your change isn't committed to the Git history yet it's in a "waiting" state.

8. Commit the change with a descriptive commit message.

**git commit -m 'added a new quote'**

Note: now the changes is part of the Git history as a single "commit"

**9.** Push that change to remote repository using the git push command

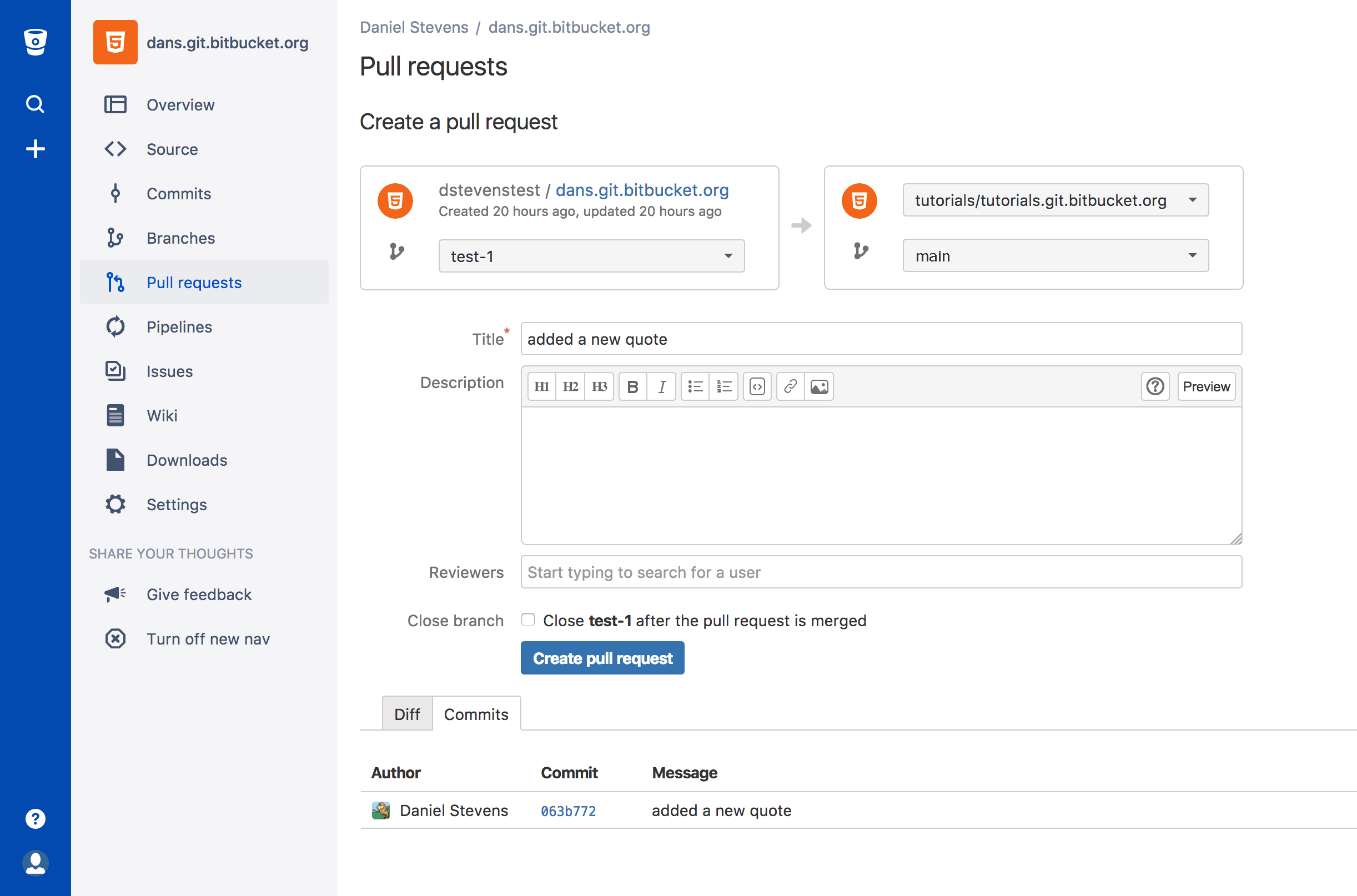
**git push origin {feature name}**

10. Open your remote repository and click Branches. You should now see both the main and the feature branches.

### **Push change and create a pull request**

Now it's time to get your first change reviewed and merge the branch.

1. Click **+> Create a pull request**. You can see your test-1 branch as the source branch and main in the destination branch.



1. You would also add reviewers on your team to the pull request and c**lick Create pull request**.
2. Make a comment in the pull request by selecting a line in the diff (the area displaying the change you made to the editme.html file).
3. Click **Approve** in the top left of the page. Of course in a real pull request you'd have reviewers making comments
4. Click **Merge**.
5. (Optional) Update the **Commit message** with more details.
6. Select the **Merge commit** Merge strategy from the two options:
   * **Merge commit**—Keeps all commits from your source branch and makes them part of the destination branch. This option is the same as entering git merge --no-ff in the command line.
   * **Squash**—Combines your commits when you merge the source branch into the destination branch. This option is the same as entering git merge --squash in the command line.

[Learn more](https://confluence.atlassian.com/x/cRNODQ" \l "Workwithpullrequests-Mergestrategies) for details on these two types of merge strategies.

1. Click **Commits** and you will see how the branch you just merged fits into the larger scheme of changes.

# **.gitignore File**

gitignore file is a text file placed in your git repositorythat tells git not to track certain files and folders that you don't want being uploaded to your master repository. It has a lot of uses, and you will almost always need to configure it if you're setting up a new repo.

**Always add .gitignore file in your project.**