Proposed GIT Workflow and Instructions for Developers

For TCS Pfizer Drupal Team

Table of Contents

About this Document

This document proposes a workflow for use by the TCS Pfizer Mumbai Team. The approach taken is a mix of Feature Branch and GitFlow approaches. The reasons behind the deviations from any of the standard workflows, are available in the document "GITWorkflowforTCSPfizerMumbaiTeam.pdf" made available to the Leadership Team.

Proposed Workflow

The Workflow is described in words below. More Detailed examples, screenshots and commands are provided in the latter section of this document.

Branches, Environments

The new workflow shall have the following branches

- **dev** (This is pointed to the dev environment). This shall be the default branch in the repo. It is more or less equivalent to master branch. But considering that the dev branch is pointed to the dev environment, and that changing the Jenkins scripts is not feasible, master branch has been removed. This is also the branch on which Integration happens.
- **stage** (This is pointed to the stage environment).

- **test** (This is pointed to the stage-2 environment). This is not always present. This branch is used for testing hotfixes by the maintenance team.
- **Feature Branches**

Repositories

The repositories involved

- **Central** Github Repo
- **Local** Repos on Developer machines
- **Acquia Cloud** Git Repo [This is out of scope for the current workflow proposed. The code from this repo powers the Dev, Stage, Test and Prod Envs. But pushing to this repo, and deployment are handled by Jenkins scripts and the TCS team should not worry about this repo for all practical purposes]

People Involved

- **Developer –** Any one interacting with the repository other than the Code Reviewer and Release Manager. Includes associates from Support and Maintenance teams.
- **Code Reviewer –** A code reviewer is identified for each project or a set of 2-3 projects. The code reviewer accepts Pull Requests allowing code to flow from Feature Branches to Dev Branch. He is the only gateway through which any change can be made to the dev branch (Except in the case of a hotfix)
- **Release Manager –** A Release Manager is identified for each project or a set of Projects. He is responsible for the stage branch. He reviews and accepts pull requests from Code Reviewer whenever the code Reviewer raises Pull Request for a release. He is the only gateway in which any change can be made onto the stage branch. (Except in the case of a hotfix)

Workflow

1. Developer pulls **dev** branch to make sure he has the latest code from the dev branch. Developer creates a new branch locally for the feature he is working on, called **Feature Branch** (say **FB1**), from dev branch.
2. Developer adds/modifies files on his local repo, on the FB1 branch. He, however, keeps pushing the FB1 branch to the central repo frequently, whenever he has committed significant amount of code to the local repo. This will ensure that all his local code changes are backed up frequently on the central repo.
3. Developer completes his changes and raises a pull request to the code reviewer requesting his Feature Branch FB1 to be merged into dev branch. **Code Reviewer(CoRe)** (pre-assigned person at a project level) is the police responsible for not just reviewing code, but also the only gateway for any code (other than hotfixes, discussed later) that enters the dev branch. The CoRe reviews the code on the **Pull Request**.

   Pull Request is a good way to handle code reviews, considering the very good UI that GitHub provides to review code and provide line level comments.

4. The CoRe can choose to either
    a. **Accept** the pull request. (This will result in the changes from the FB being merged into the dev branch)
    b. **Make Suggestions** on the pull request. (The dev branch remain the same as earlier). In this case, the developer continues with making changes on his FB1 and will re-initiate a pull request again with more commits.
5. When the Pull Request is accepted, the changes from FB1 are now on the dev branch. The Jenkins scripts would have already pushed this code to the corresponding branch on Acquia Repo and hence the Dev Env can be used now to test the changes from FB1.
6. The developer as well as the QA team now test the site with dev branch on the dev environment. There are 2 cases here:
    a. Dev Env looks good. Probably some bugs could be found when tested. The usual happy path.
    b. Dev Env is broken and can not be tested.
7. In the case of 6b (Dev Env is broken), dev branch is reverted to earlier state before FB1. (In case the code from FB had actually performed any DB changes, then a DB restore is also required. Ex - modules with hook_schema and any DB operations in hook_install or hook_update)
   Challenge: This could lead to some time lapse as the TCS team does not have access to restoring an old db to dev environment.
8. In case of 6a (Dev Env looks fine), QA team tests the Dev Env, **logs defects** say (**Iss#1**, **Iss#2**). Any Developer working on Iss#1 will start by checking out dev and creating a Feature Branch called (say **FBIss1**) and follows the same prcedure as described from steps 1 to 6 above. Consider that in the meanwhile the dev branch received some pull requests for some fixes of earlier issues (say **Iss#0**) and the pull request was accepted and hence the dev branch now includes changes from branches **FB1**, **FBIss1**, **FBIss2**, **FBIss0**. Note : QA should be proactive here to see that the fix for Iss#0 included in dev now does not break the already passed test cases for FB1, on the dev environment i.e, Regression Testing to ensure all cases for this release are covered.
9. The Dev Env is tested to see that there are no known issues and now dev branch is merged onto stage. The Stage Branch is used by the QA team or Pfizer to do one round of User Acceptance testing. There are 2 cases here:
    a. No issues found.
    b. Stage Environment broken or even smaller defects found, which can not go into production.
10. In case of 9b, Stage Branch is immediately reverted to its earlier state. Issue is logged (say **Iss#3**). The developer assigned starts by checking out stage branch into a new Feature Branch (FBIss3). So **dev** is now in the state of (FB1+FBIss1+FBIss2+FBIss0). **Stage** branch is behind dev, at the same state as where dev was, in step 1. ***It is important to note here that TCS team should treat Stage branch as good as a production environment. Any code with known issues that is not going into production, should NOT stay on the Stage Branch. Once dev is merged onto Stage, it should either be taken to production by notifying the Pfizer team, or it should be rolled back to earlier state.***

Challenge: Reverting could lead to some time lapse as the TCS team does not have access to restoring db to stage environment.

11. In case of 9a, Stage now has (ideally) bug-free code that is now ready to be pushed to production. TCS team notifies Pfizer Ops team who deploys the code to production.

Hotfixes and Maintenance/Support Requests

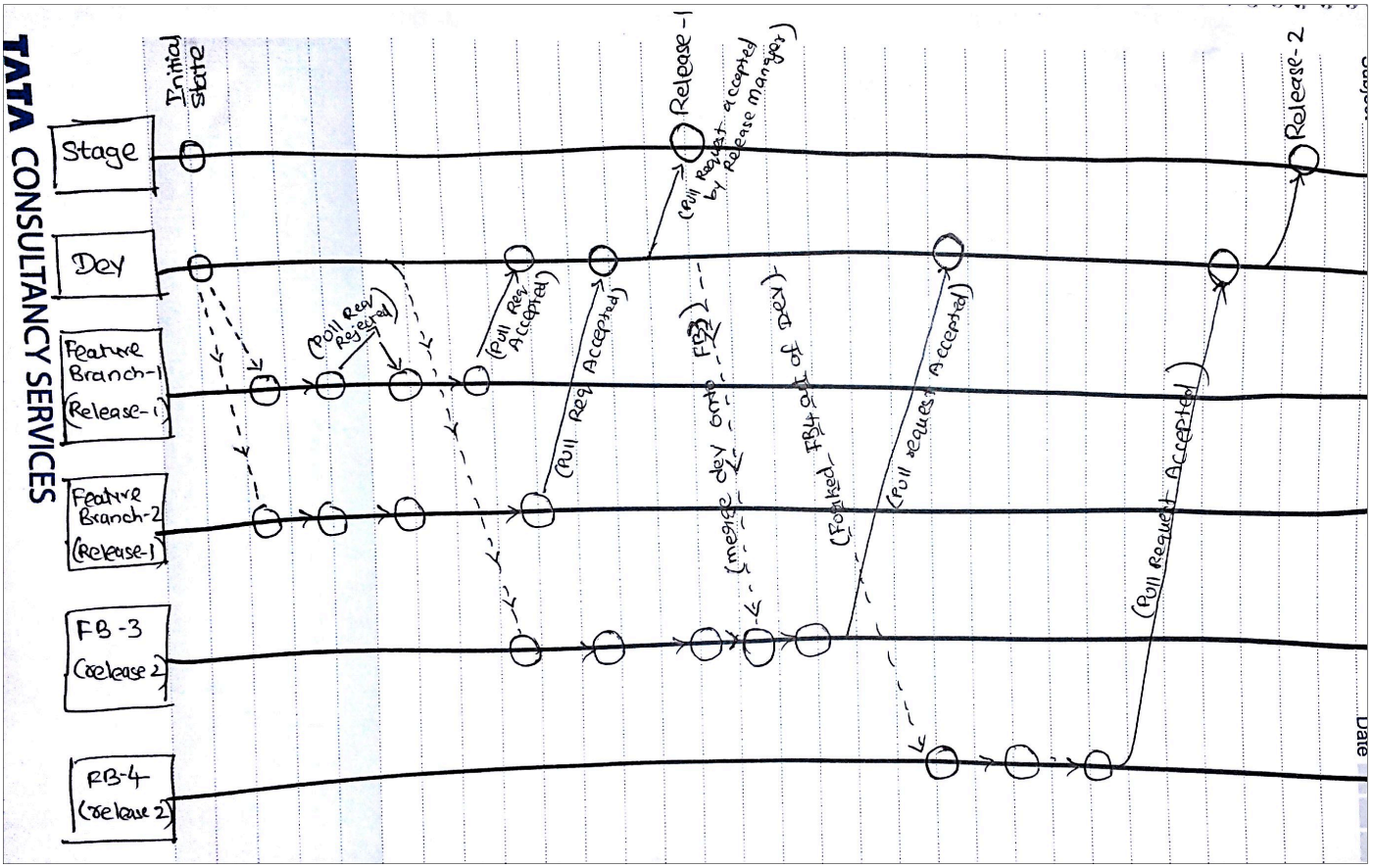
All Support Requests that are **not** hotfixes (usually small changes that should be implemented on production with high priority) - shall follow the same procedure to deploy code - ie using a Feature Branch forked from dev and merging code ino dev branch via a pull request.

In case of a hotfix:

1. Developer checks out **Stage** Branch into a hotfix branch (say HF1)
2. Developer makes changes on HF1 locally and pushes the branch to Central Repository as often as possible for back-up reasons.
3. Once the changes are complete and all commits are made and pushed to Central Repo, developer now merges this HF1 branch onto **test** branch (if stage-2 environment is available). The hotfix is tested on the stage-2 environment. This step can be skipped if a stage-2 environment is not available for the project.
4. The changes from hotfix branch HF1 are now merged into the **stage** branch and tested on the Stage Environment.
5. Additionally, the changes are merged to **dev** branch as well. **This is important**. This procedure of allowing hotfixes to be forked from and commit to Stage directly has the inherent problem of making the stage and dev branches going out of sync. The developer who worked on the hotfix should take the onus of merging the hotfix branch changes onto the dev branch as well.

The workflow – Snapshot of a couple of releases

The term “Release“ here denotes merging some code from master onto stage and notifying Pfizer that the code from stage branch can be pushed to Production.



Developers Dev1,Dev2,Dev3,Dev4 are working on Features on Feature Branches FB1, FB2, FB3 and FB4. FB1 and FB2 are intended to be released in Release 1. FB3 and FB4 are intended to be released in Release 2.

Timeline of the 4 developers:

Dev 1 forks FB1 out of Dev Branch. He work on it. Submits a Pull Request. Code Reviewer makes some comments. Further commits are made and another pull request is made. And this time the pull request was accepted and his changes were merged into Dev branch. His changes go to stage in Release 1.

Dev 2 forks FB2 out of Dev Branch during the same release cycle. Very similar to Dev 1, his changes were also merged into dev. Very similar to Dev 1, his changes go to Stage in the first release.

Dev 3 forks FB3 out of Dev Branch. But what is to be noticed here is that he is working on features that are going to be released in Release-2. But he forked out of dev prior to Release-1.

So, after release one, he merges latest dev (pulled from Central Repo) onto his FB3, so that he is in sync with Dev, though he branched out of dev long ago in the previous cycle. His changes are included in release-2.

Dev-4 does a simple job, like dev-1. He forks a branch out of Dev after Release-1. Makes changes, raised a pull request, which is accepted. His changes are published in Release-2 onto Stage.

The above story does not denote how a hotfix is handled. Refer to “Example – 5” on this document for details about the workflow involved in handling a hotfix.

Tutorials, Example Stories, Commands for Developers

Preparing Existing Repos to make them fit into the workflow

To make any repo suit the workflow, it should have the following branches:
dev (default branch)
stage
It should not have a master branch. And Dev should be the default branch.

Create the required branches

If you should take code out of master, into dev and stage, for an existing project with latest code on master branch:

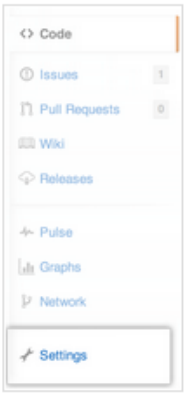
```
Tanays-MacBook-Air:gitdemo tanaysai$ git branch dev master
Tanays-MacBook-Air:gitdemo tanaysai$ git branch stage master

Tanays-MacBook-Air:gitdemo tanaysai$ git push origin --all
Total 0 (delta 0), reused 0 (delta 0)
To git@github.com:saitanay/gitdemo.git
 * [new branch]       dev -> dev
 * [new branch]       stage -> stage
 * [new branch]       test -> test
```

Default Branch

Set **dev** to the default branch
https://help.github.com/articles/setting-the-default-branch

1 Open the repository page

2 Click the Settings button



3 Change the default branch

Delete the master branch

```
Tanays-MacBook-Air:gitdemo tanaysai$ git branch -d master
Deleted branch master (was 42e4f89).
Tanays-MacBook-Air:gitdemo tanaysai$ git push --del origin master
To git@github.com:saitanay/gitdemo.git
 - [deleted]         master
```

Initial Status of Branches

Now you have the same code on dev and stage branches to start with.

Branch Name Convention

Ex: dev-123-slideshow

- The branch name will have 3 parts, separated by 2 hyphens
- The first part is the name of the parent branch that it was forked from. Allowed values - dev, stage
- Second part is the issue #, can be Alphanumeric based on the Issue tracker that you are using. Characters Allowed - [A-Z][a-z][0-9]
- Third part is a small summary of the issue in words, not exceeding 10 characters and 3 words. No separator between words

Example-1 : Dev-1 works on a feature, builds it, and publishes it (to dev branch) Dev-2 works on another feature, builds it and publishes it (to dev branch again)

Dev1 starts working on a feature, say adding slideshow, corresponding to a ticket #123. He starts by creating a feature branch out of dev.

```
git branch dev-123-slideshow dev
```

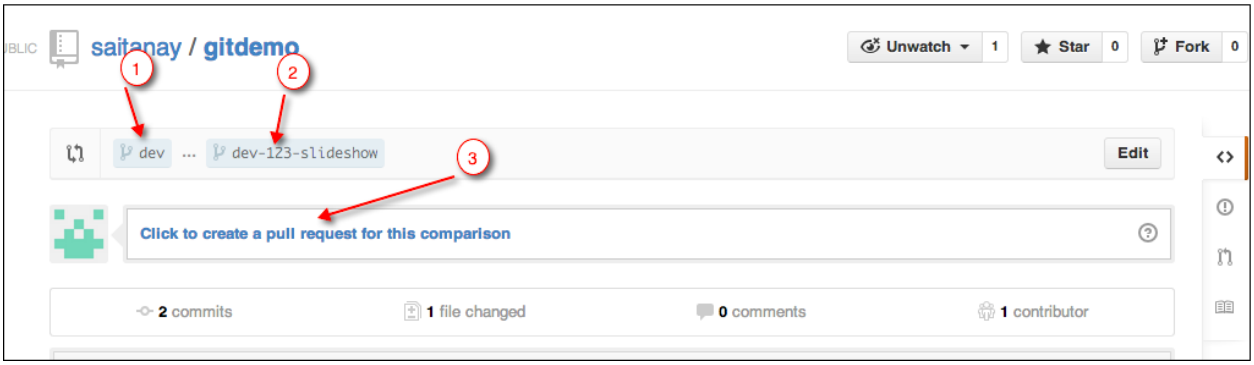
Developer makes changes to his branch locally and commits them.

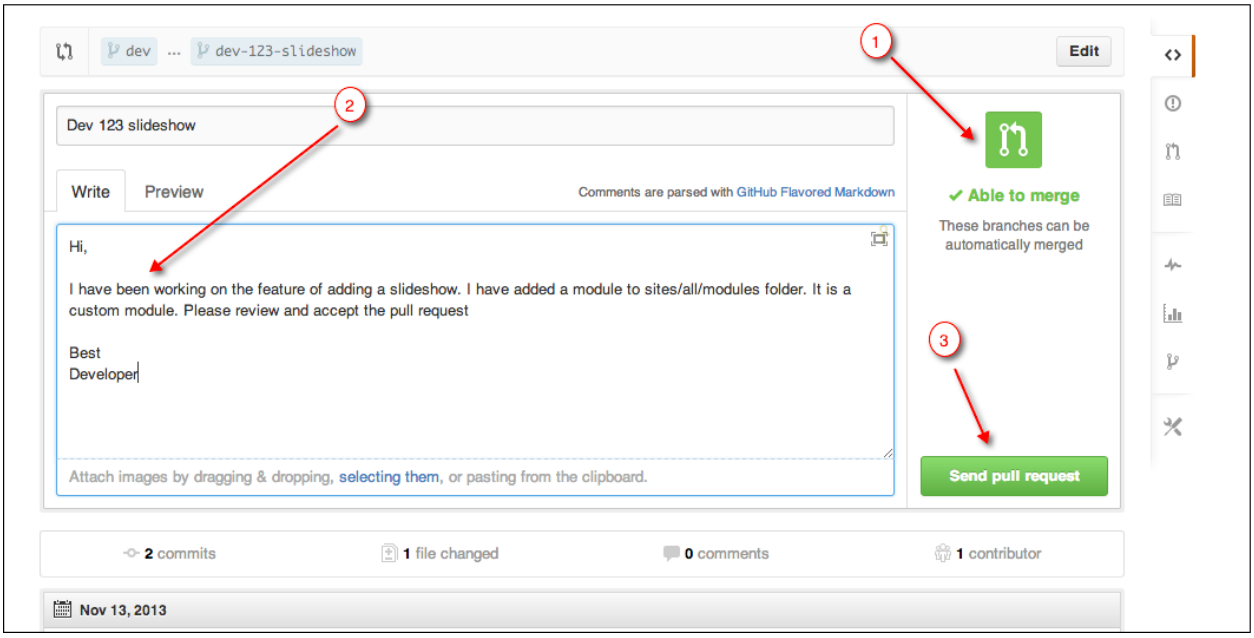
```
Tanays-MacBook-Air:gitdemo tanaysai$ git add .
Tanays-MacBook-Air:gitdemo tanaysai$ git commit -m "added slide”

Tanays-MacBook-Air:gitdemo tanaysai$ git add .
Tanays-MacBook-Air:gitdemo tanaysai$ git commit -m "added show"
```

The developer will keep pushing the feature branch to central repository often to keep his changes backed up.

```
git push origin dev-123-slideshow
```

Once Done with developing the feature and testing the code on his local build, developer raises a pull request asking code reviewer to merge his changes into dev branch.



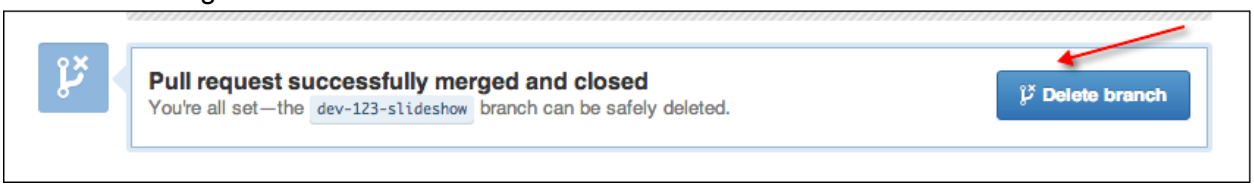


The developer and code reviewer communicate on the discussion below the pull request.

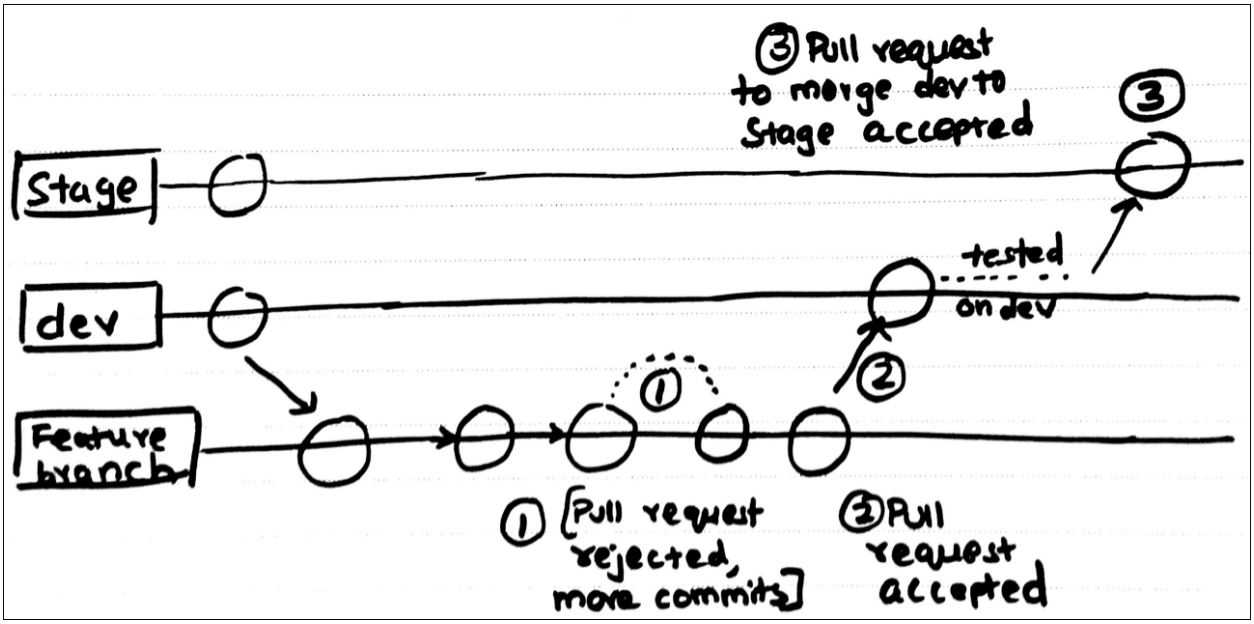
More commits can be added to the pull requested by the developer by committing to the same branch on which pull request was requested (In this case dev-123-slideshow) .

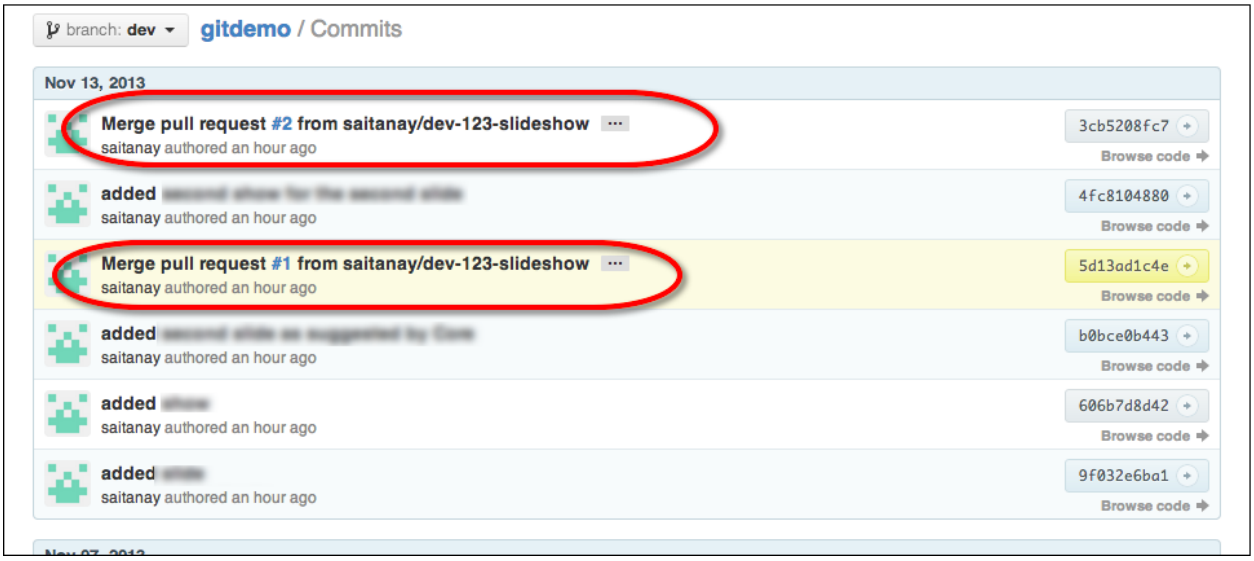
Once the Code Reviewer is satisfied with the code, he accepts the pull request.

GitHub shows and option to the Code Reviewer to delete the Feature Branch on the Central Repository. But it is advisable to retain the branch till the changes from the Feature Branch are released to Stage.



The history of the dev branch now shows the individual commit messages from the branches where the pull requests were accepted from, as well as the commit message pertaining to the pull request itself.





Now the dev environment (which points to the dev branch) has the code with the Features or issues fixes from pull requests that have been accepted by the code reviewer.

To add some complexity, let us say there is another developer just starting to work on another Feature Branch - **dev-124-dropdownmenu**

The second developer about to work on the Feature “Drop Down Menu” will fork from dev branch. But he should make sure that he pulls the dev branch before he starts so that he has the latest changes that were merged into dev recently.

```
Tanays-MacBook-Air:gitdemo tanaysai$ git pull origin dev
From github.com:saitanay/gitdemo
 * branch            dev        -> FETCH_HEAD
Updating 42e4f89..3cb5208
Fast-forward
 human.module | 4 ++++
 1 file changed, 4 insertions(+)
Tanays-MacBook-Air:gitdemo tanaysai$ git hist
*   3cb5208 2013-11-13 | Merge pull request #2 from saitanay/dev-123-slideshow
|\
| * 4fc8104 2013-11-13 | added second show for the second slide (origin/dev-12
* |   5d13ad1 2013-11-13 | Merge pull request #1 from saitanay/dev-123-slidesh
|\ \
| |/
| * b0bce0b 2013-11-13 | added second slide as suggested by Core [Tanay Sai]
| * 606b7d8 2013-11-13 | added show [Tanay Sai]
| * 9f032e6 2013-11-13 | added slide [Tanay Sai]
```

```
Tanays-MacBook-Air:gitdemo tanaysai$ git branch dev-124-dropdownmenu dev
Tanays-MacBook-Air:gitdemo tanaysai$ git co dev-124-dropdownmenu
Switched to branch 'dev-124-dropdownmenu'
Tanays-MacBook-Air:gitdemo tanaysai$ vi human.module
Tanays-MacBook-Air:gitdemo tanaysai$ git add .
Tanays-MacBook-Air:gitdemo tanaysai$ git commit -m "added dropdown menu"
[dev-124-dropdownmenu f72d068] added dropdown menu
 1 file changed, 2 insertions(+)
```

```
Tanays-MacBook-Air:gitdemo tanaysai$ git push origin  dev-124-dropdownmenu
Counting objects: 5, done.
Delta compression using up to 4 threads.
Compressing objects: 100% (2/2), done.
Writing objects: 100% (3/3), 301 bytes | 0 bytes/s, done.
Total 3 (delta 1), reused 0 (delta 0)
To git@github.com:saitanay/gitdemo.git
 * [new branch]      dev-124-dropdownmenu -> dev-124-dropdownmenu
```

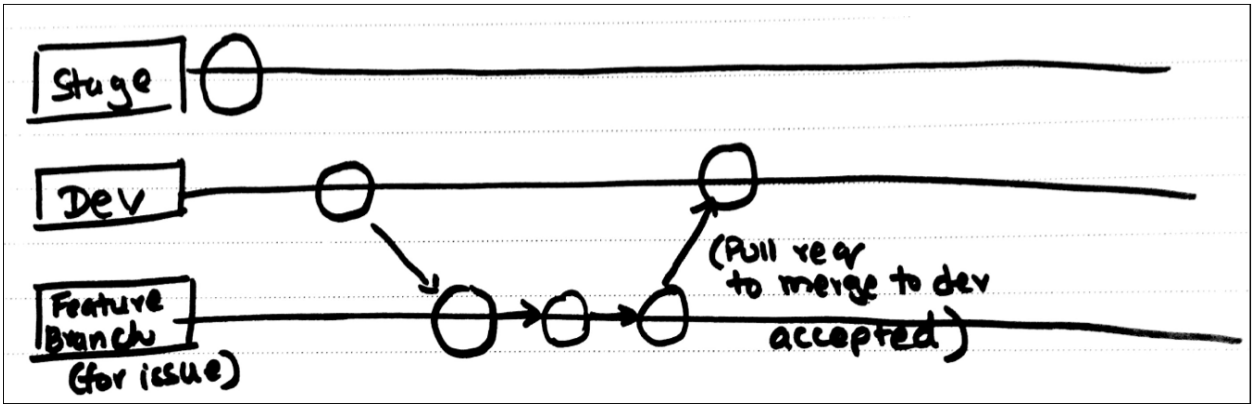
Now the developer working on the feature branch dev-124-dropdownmenu raises a pull request for his code to be merged into dev branch and it is accepted and merged by the code reviewer.

So now, the dev branch has the changes from both the feature branches dev-124-dropdownmenu and dev-123-slideshow. These changes can now be tested on the dev environment url.

Example-2: Defect found on the dev environment

If any issue is found on the dev environment, the following workflow is suggested:

1. Raise an issue# for the bug
2. Assign it to a developer
3. Developer makes a feature branch with the issue# forking from dev branch
4. Developer makes changes to fix the issue and commits them to the feature branch
5. Developer raises a pull request
6. Code Reviewer accepts the pull request
7. The dev env is re-tested for the issue and the issue # is closed



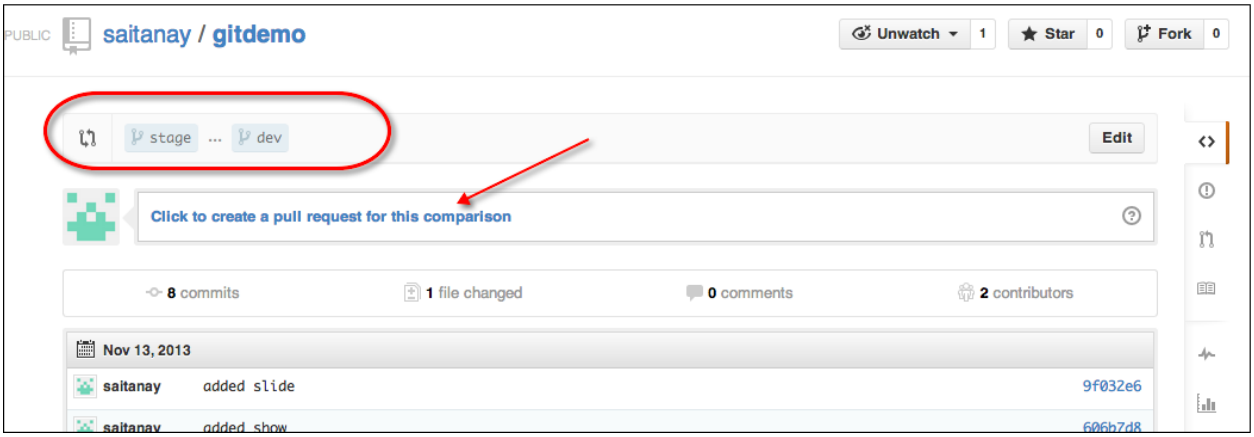
In the above pic, note that initially the dev is ahead of stage, since could have been some integrations (accepting pull requests) onto dev, after the earlier release onto stage.

Currently the stage branch is in the same state since the beginning of this story. It does not have the changes from the feature branches or issues fixed. Now that we have a stable release on the dev environment, with the 2 features and 1 bug fix, Code Reviewer will now raise a pull request (to the Release Manager) asking the Code from dev Branch to be merged to Stage.

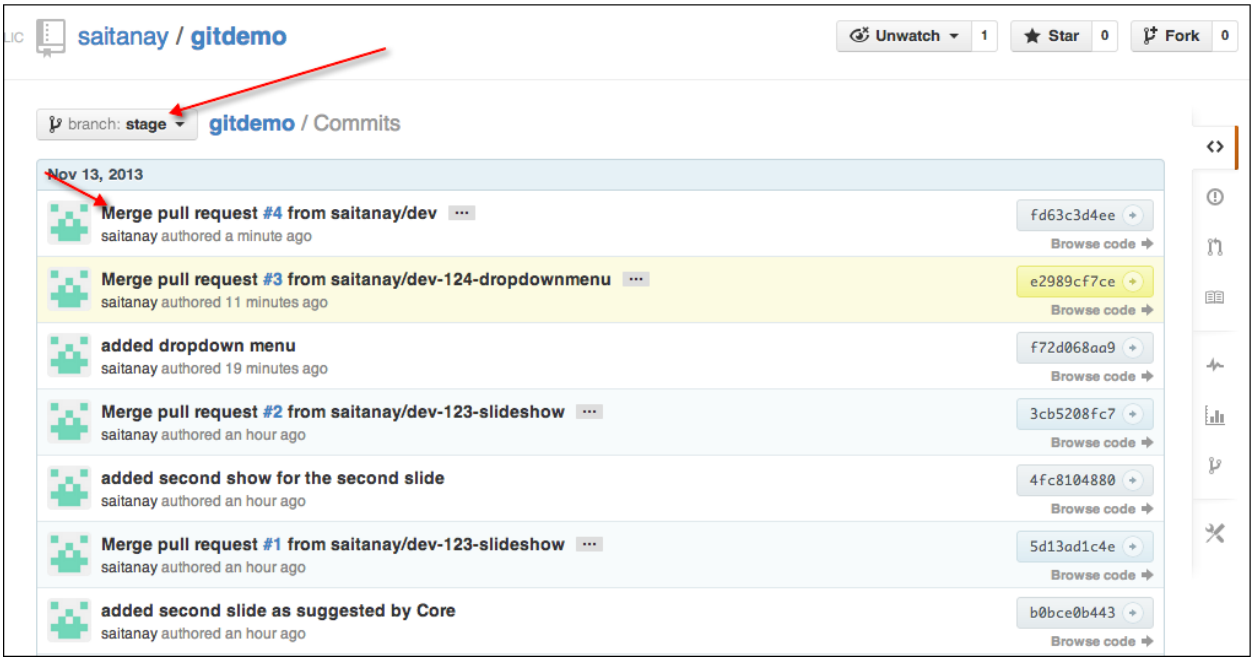
Till the merge is complete and tested, no Pull Requests shall be accepted by the Code Reviewer.

The Release Manager now verifies the Pull Request and accepts the request. Thus causing all the code from dev branch to be patched onto Stage branch.

The dev branch is now free and is available for new integrations (ie accepting new pull requests by the code manager).



The stage branch now has all the code changes and history from the 2 Feature Branches and the Bug fix that were included in this release.



The staging environment is now tested for acceptance and Pfizer is notified. Pfizer deploys the changes from stage to production.

TIP: To see statuses of all branches on local and remote, try this

```
Tanays-MacBook-Air:gitdemo tanaysai$ git remote show origin
* remote origin
  Fetch URL: git@github.com:saitanay/gitdemo.git
  Push  URL: git@github.com:saitanay/gitdemo.git
  HEAD branch: dev
  Remote branches:
    dev                                    tracked
    refs/remotes/origin/dev-123-slideshow  stale (use 'git remote prune' to remove)
    refs/remotes/origin/dev-124-dropdownmenu stale (use 'git remote prune' to remove)
    stage                                  tracked
    test                                   tracked
  Local refs configured for 'git push':
    dev   pushes to dev   (up to date)
    stage pushes to stage (local out of date)
    test  pushes to test  (up to date)
```

TIP: To Clean references to branches deleted on remote, on your local machine

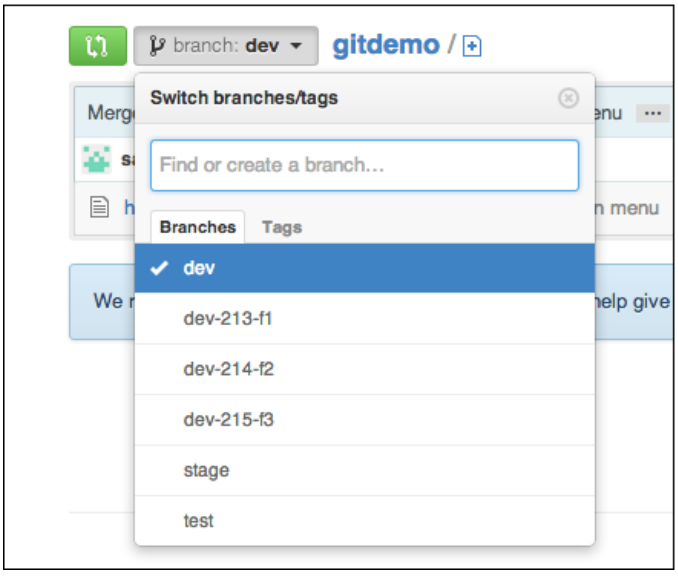
```
Tanays-MacBook-Air:gitdemo tanaysai$ git remote prune origin
Pruning origin
URL: git@github.com:saitanay/gitdemo.git
 * [pruned] origin/dev-123-slideshow
 * [pruned] origin/dev-124-dropdownmenu
```

Example-3 : Three Features were planned for the release - F1,F2,F3. The changes from feature branches of F1, F2,F3 were integrated into dev. On testing on dev, there was a major bug found in F2. So F2 changes should now be removed from dev and F1,F2 should be released to stage.

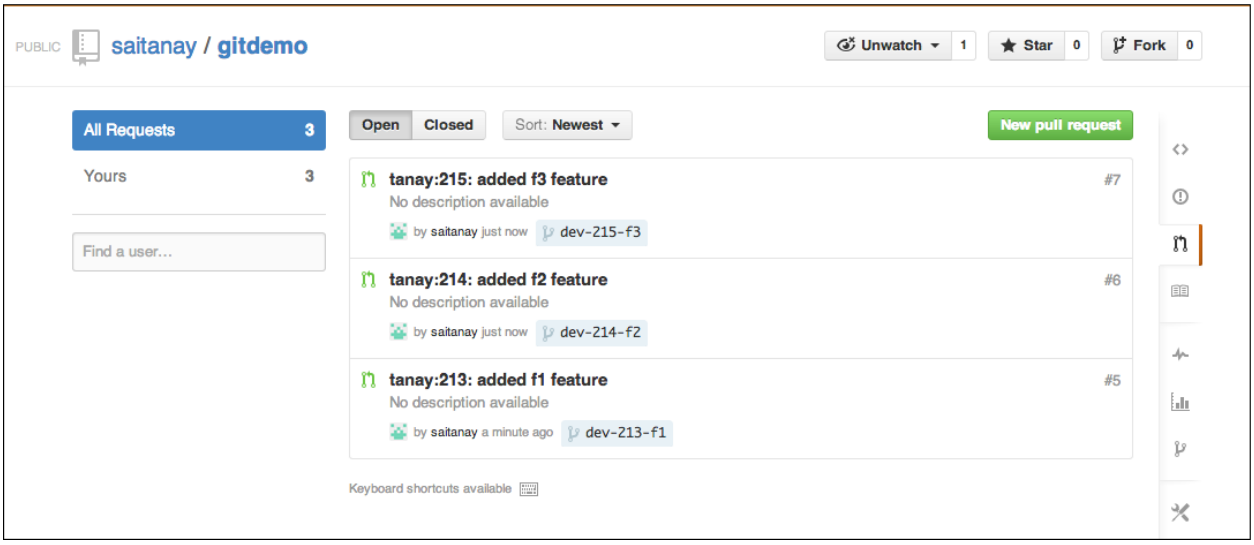
The developer working on F1, branches out of dev, works on the feature and commits to his feature branch.

```
Tanays-MacBook-Air:gitdemo tanaysai$ git pull origin dev
From github.com:saitanay/gitdemo
 * branch            dev        -> FETCH_HEAD
Already up-to-date.
Tanays-MacBook-Air:gitdemo tanaysai$ git branch dev-213-f1 dev
Tanays-MacBook-Air:gitdemo tanaysai$ git co dev-213-f1
Switched to branch 'dev-213-f1'
Tanays-MacBook-Air:gitdemo tanaysai$ vi human.module
Tanays-MacBook-Air:gitdemo tanaysai$ git add .
Tanays-MacBook-Air:gitdemo tanaysai$ git commit -m "tanay:213: added f1 feature"
[dev-213-f1 e942482] tanay:213: added f1 feature
 1 file changed, 2 insertions(+)
Tanays-MacBook-Air:gitdemo tanaysai$ git push origin dev-213-f1
Counting objects: 5, done.
Delta compression using up to 4 threads.
Compressing objects: 100% (2/2), done.
Writing objects: 100% (3/3), 290 bytes | 0 bytes/s, done.
Total 3 (delta 1), reused 0 (delta 0)
```

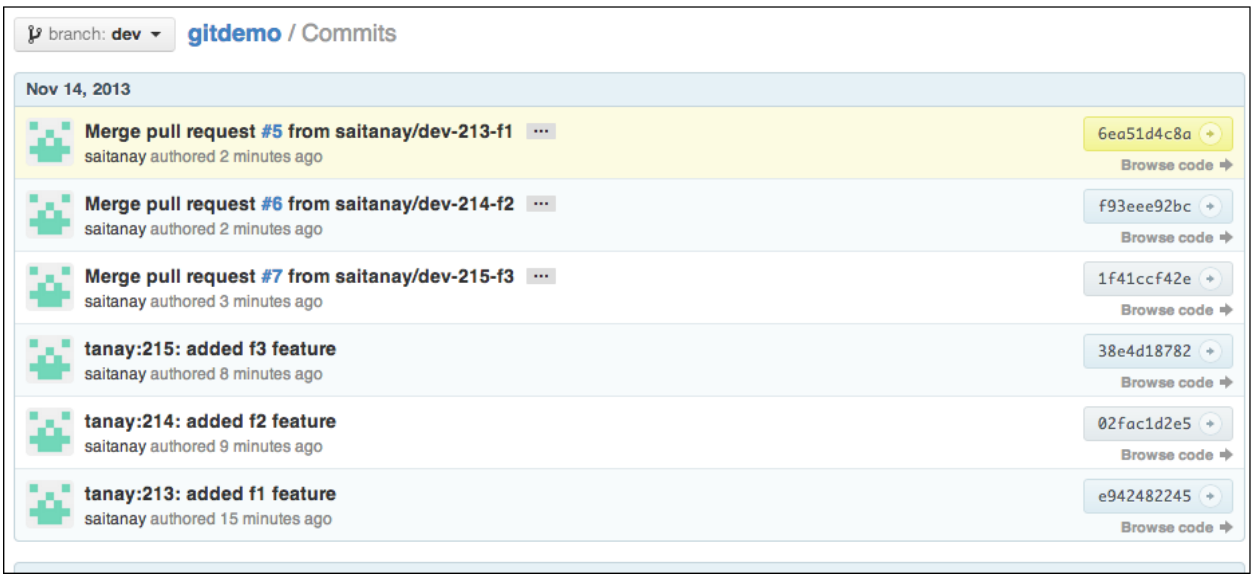
Developers working on F2 and F3 also make similar branches from dev, and push them to the remote repository.



All the three developers make pull requests, which are accepted by the code reviewer.



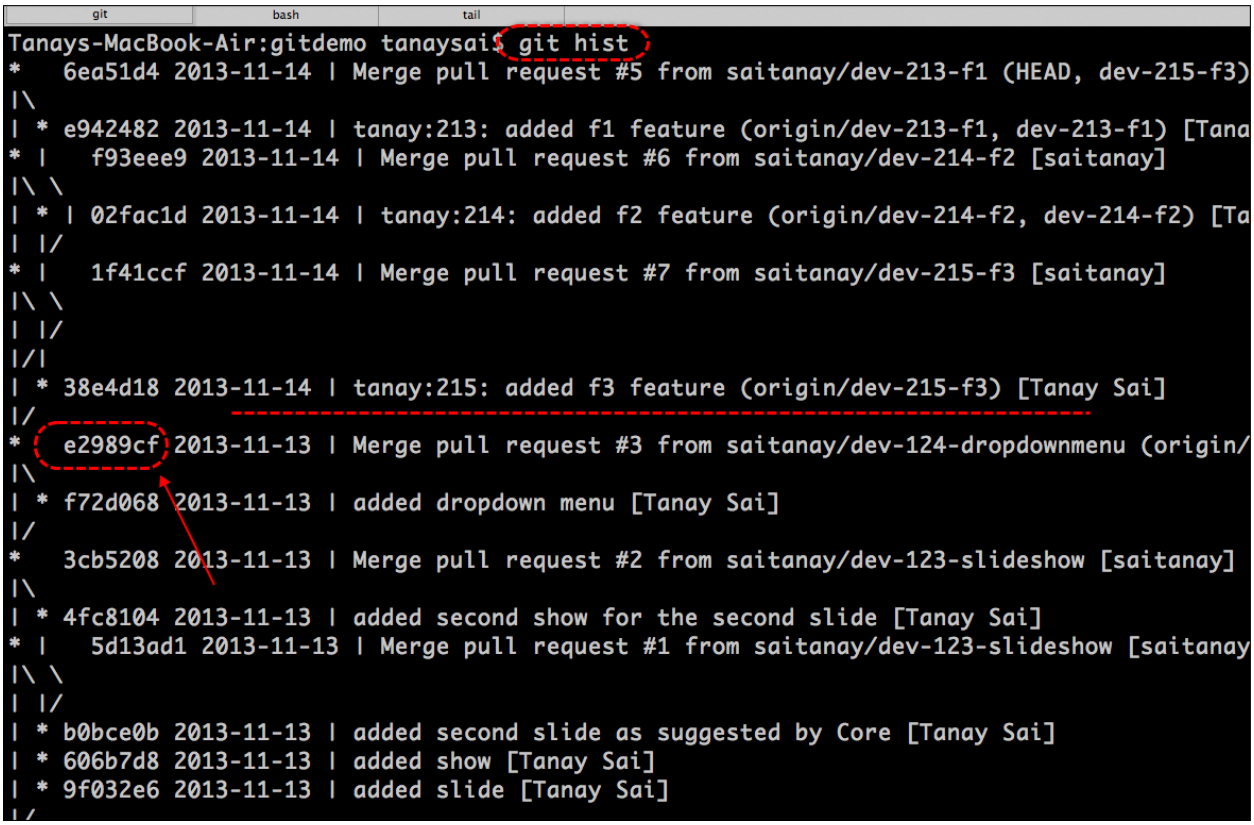
Consider the code reviewer accepted all the 3 pull requests. The status of the dev branch on the central repo looks like below.



Now while testing the dev environment, something was found seriously wrong with F2 feature's code. It will take a while to fix it. But F1 and F3 should go ahead to production. So we should now get dev branch to the state of F1+F3 without F2.

An easy way here would be to revert the dev branch to the state where it was before any of F1/F2/F3 were merged. Then merge F1 and F3 onto dev branch. The Code Reviewer should perform this operation, as he is the only one who can make any changes to the dev branch.

```
Tanays-MacBook-Air:gitdemo tanaysai$ git pull origin dev
remote: Counting objects: 15, done.
remote: Compressing objects: 100% (5/5), done.
remote: Total 7 (delta 2), reused 0 (delta 0)
Unpacking objects: 100% (7/7), done.
From github.com:saitanay/gitdemo
 * branch            dev        -> FETCH_HEAD
Updating 38e4d18..6ea51d4
Fast-forward
 human.module | 3 +++
 1 file changed, 3 insertions(+)
```



The encircled commit above points to the status of the dev branch before any of F1, F2,F3 were merged onto dev. ie, it is the status of the stable dev branch from earlier release.

Code reviewer reverts local branch to earlier state.

```
Tanays-MacBook-Air:gitdemo tanaysai$ git reset --hard e2989cf
HEAD is now at e2989cf Merge pull request #3 from saitanay/dev-124-dropdownmenu
Tanays-MacBook-Air:gitdemo tanaysai$ git hist
*   e2989cf 2013-11-13 | Merge pull request #3 from saitanay/dev-124-dropdownmenu (HEAD, or
|\
| * f72d068 2013-11-13 | added dropdown menu [Tanay Sai]
|/
*   3cb5208 2013-11-13 | Merge pull request #2 from saitanay/dev-123-slideshow [saitanay]
|\
| * 4fc8104 2013-11-13 | added second show for the second slide [Tanay Sai]
* |   5d13ad1 2013-11-13 | Merge pull request #1 from saitanay/dev-123-slideshow [saitanay]
|\ \
```

Revert the Remote Repo's Dev Branch also to earlier state

```
Tanays-MacBook-Air:gitdemo tanaysai$ git push origin dev --force
Total 0 (delta 0), reused 0 (delta 0)
To git@github.com:saitanay/gitdemo.git
 + 6ea51d4...e2989cf dev -> dev (forced update)
```

Now, merge F1 and F3 onto local dev branch and then push it to remote repo. Make sure that you (code reviewer) are on dev branch.

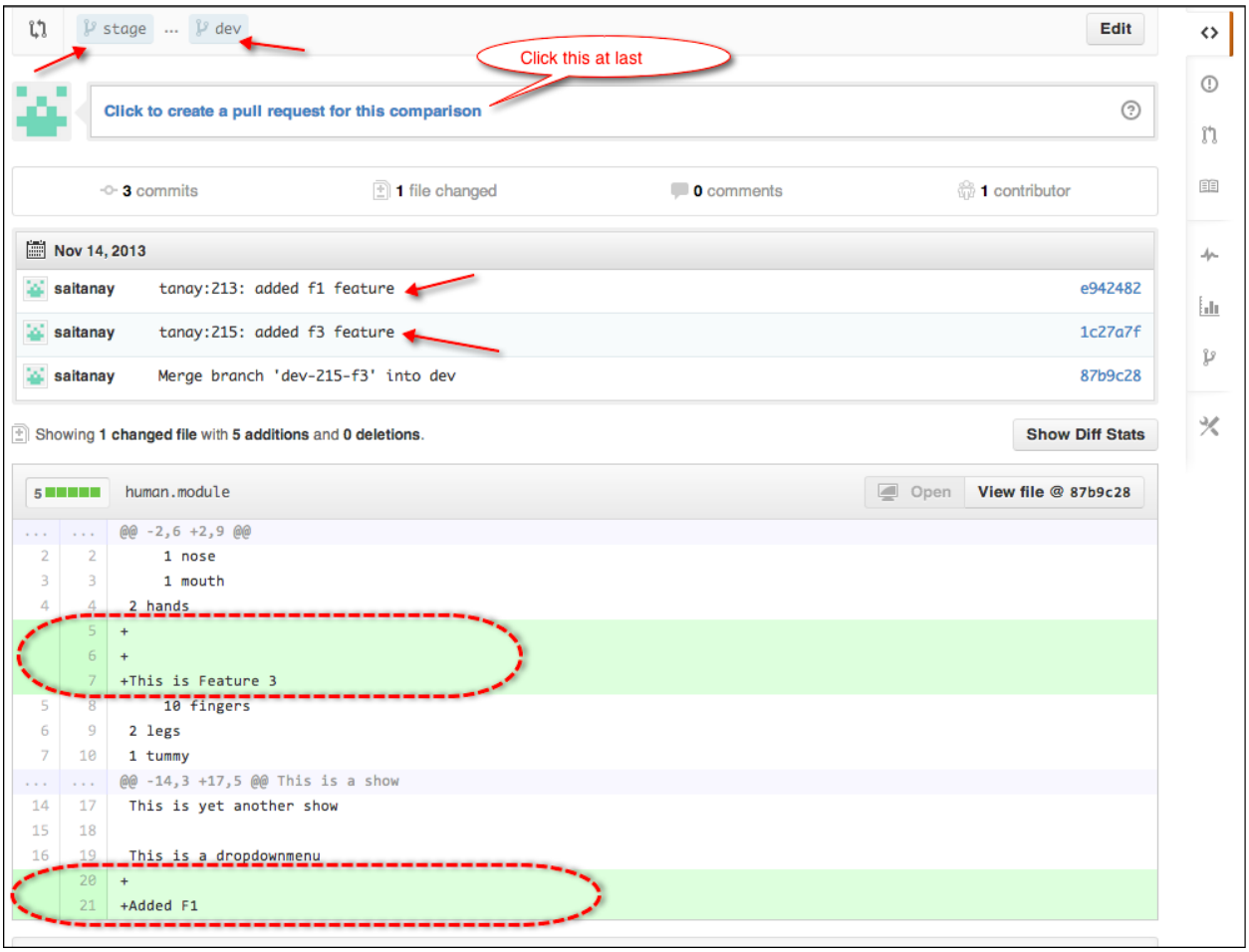
```
Tanays-MacBook-Air:gitdemo tanaysai$ git merge dev-213-f1
Updating e2989cf..e942482
Fast-forward
 human.module | 2 ++
 1 file changed, 2 insertions(+)
Tanays-MacBook-Air:gitdemo tanaysai$ git merge dev-215-f3
Auto-merging human.module
Merge made by the 'recursive' strategy.
 human.module | 3 +++
 1 file changed, 3 insertions(+)
```

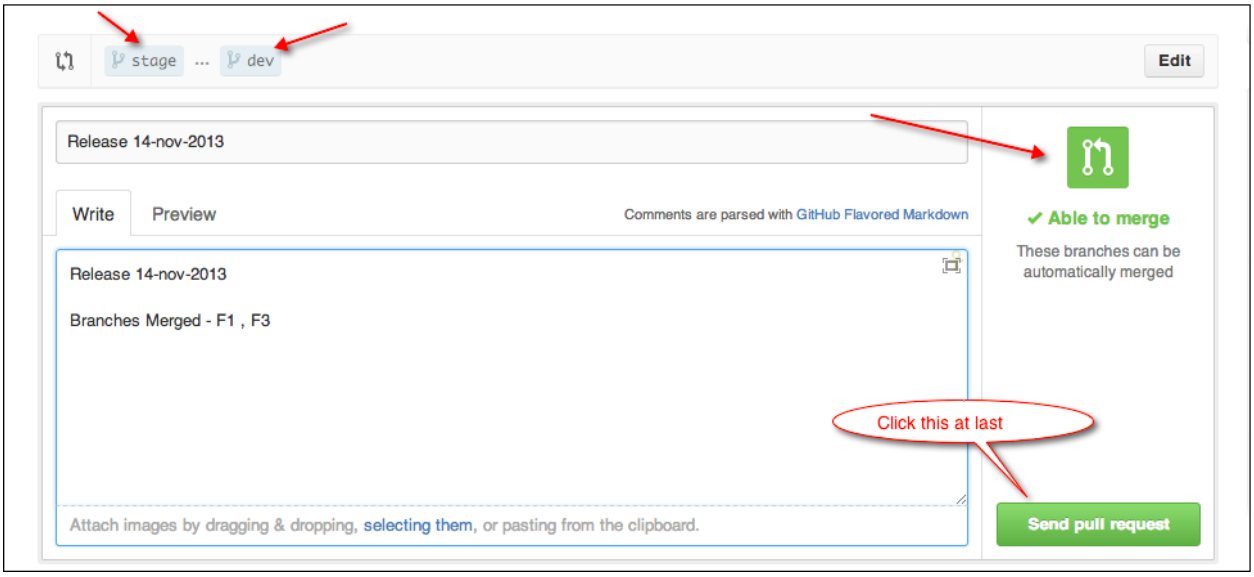
```
Tanays-MacBook-Air:gitdemo tanaysai$ git push origin dev
Counting objects: 7, done.
Delta compression using up to 4 threads.
Compressing objects: 100% (2/2), done.
Writing objects: 100% (3/3), 328 bytes | 0 bytes/s, done.
Total 3 (delta 1), reused 0 (delta 0)
To git@github.com:saitanay/gitdemo.git
   e2989cf..87b9c28  dev -> dev
```

Thus, the code reviewer can take out any integrated feature (F2) and rebuild dev with the required features (F1 and F3).

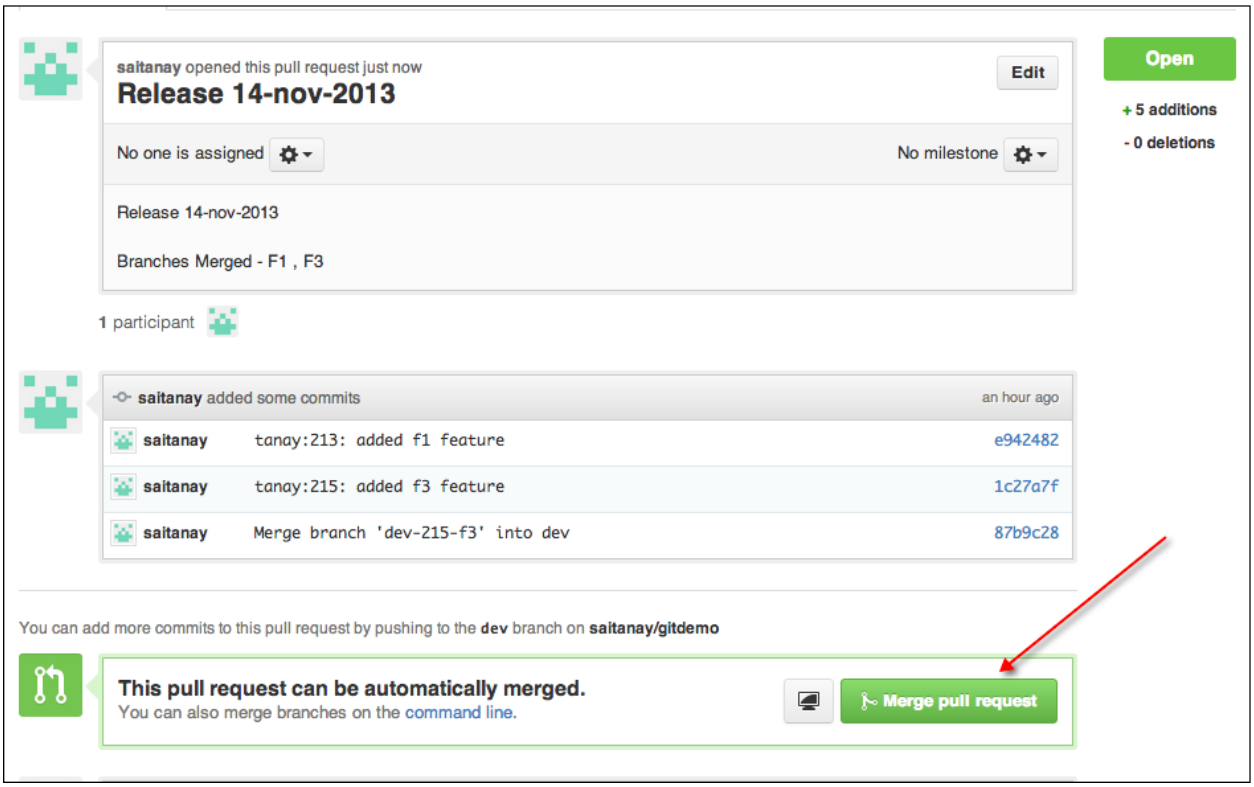
Example-4 : Making a release to the Staging Branch when the dev branch is stable and no known issues (that can go into prod)

This will usually be done by the Release Manager in the project. The Release Manager raises a Pull Request for himself on GitHub, requesting dev branch to be merged onto stage.





Next the Release Manager accepts the pull request that he created.



The stage branch now has the changes from F1 and F2 features that were released.

If the Stage Environment is tested and found to be good, Pfizer can be notified to deploy the stage branch code to production.

Example - 5 : Code was released onto Stage branch from dev (as shown in Example 3)

Say the Stage branch now has F1 and F2. Release manager has not yet notified Pfizer to deploy code since QA is testing the stage environment.

While testing the site on Staging Environment, QA reported to release manager that the build can not go to production because of the issues that it has got. The release is to be polished a bit, which could take a while. But since we treat Stage branch to be as good as production, we can not leave the broken things on stage. So the release manager now had to revert the stage branch to the status of its earlier release.

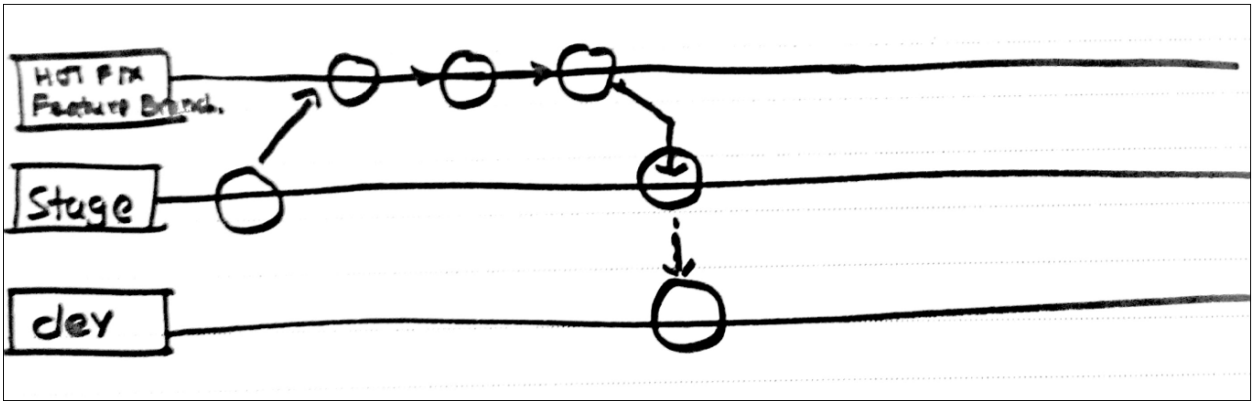
```
Tanays-MacBook-Air:gitdemo tanaysai$ git pull origin stage
Tanays-MacBook-Air:gitdemo tanaysai$ git log
      From git hist, or git log, it was identified that the earlier
      stable release on stage can be referred by the commit hash
      e2989cf
Tanays-MacBook-Air:gitdemo tanaysai$ git reset --hard e2989cf
Tanays-MacBook-Air:gitdemo tanaysai$ git push origin stage --force
```

Example - 6: Making a hotfix on Staging Environment

Consider that Maintenance team has been asked to make a quick small change on the production environment. Going through the full procedure of usual feature addition might not be feasible considering that the fix has to be deployed real soon.

The developer working on the hotfix,

- Creates a feature branch for the hotfix from **stage** branch, and names the branch **stage-888-contactustypo** (888 = issue#, contactustypo = small description)
- He tests the stage-888-contactustypo locally.
- Then he merges stage-888-contactustypo into stage branch and tests the staging environment. He also notifies Release Manager, who is usually responsible for all changes on stage.
- Once tested, the developer merges his hotfix branch onto stage branch.
- After testing on stage environment, Pfizer is notified to deploy the stage branch to production.
- Additionally, the developer now has to take the onus of merging his changes back to dev branch. **This is important**.




```
Tanays-MacBook-Air:gitdemo tanaysai$ git pull origin stage
Tanays-MacBook-Air:gitdemo tanaysai$ git branch stage-888-
contactustypo stage
Tanays-MacBook-Air:gitdemo tanaysai$ git co stage-888-
contactustypo
Make the changes
Tanays-MacBook-Air:gitdemo tanaysai$ git add .
Tanays-MacBook-Air:gitdemo tanaysai$ git commit -m "tanay:888:
hotfix: fixed something something"
Tanays-MacBook-Air:gitdemo tanaysai$ git push origin stage-888-
contactustypo
Tanays-MacBook-Air:gitdemo tanaysai$ git pull origin dev
Tanays-MacBook-Air:gitdemo tanaysai$ git co dev
Tanays-MacBook-Air:gitdemo tanaysai$ git merge stage-888-
contactustypo
Tanays-MacBook-Air:gitdemo tanaysai$ git push origin dev
```

Appendix - Git Short Commands

Some of the git commands shown on the screenshots in this document like **git co, git hist** may cause ambiguity since they are not the standard commends. They are nothing but short codes for git commands configured on the .gitconfig file. Below is the list:

```
co = checkout

ci = commit

st = status

br = branch

hist = log --pretty=format:'%h %ad | %s%d [%an]' --graph --date=short

type = cat-file -t

dump = cat-file -p
```