

## Table of Contents

Help Repository Structure .....	1
Git Setup on your Local Host .....	2
Initial Setup .....	2
Add an SSH Key to your Local Host.....	3
Clone a Git Repository to your Local Host.....	3
Git Procedures.....	5
Status: Check the Status of your Local Repo .....	5
Add/Commit/Push: Save your Local Updates to Gitlab .....	8
Fetch/Rebase: Copy Gitlab Updates to your Local Repo.....	10
Good Practices for Git.....	12
Fast-Forwarding: Why it's Bad and How to Avoid It.....	14
Gitlab Setup ( <i>Advanced</i> ): Creating Groups and Projects .....	15
Create a group in Gitlab .....	15
Create a new Git project.....	15

## Help Repository Structure

This repository has the following folders and structure:

/

The root directory has the following files:

- \* Links to various useful files/subdirectories

NOTE—The following files should not be updated without notifying/coordinating with Engineering:

- \* help.json - defines the software-to-help links on the appliance
- \* .write\_tar - tells the builder to copy files over per alloy\_server\_help.spec (I think)
- \* alloy\_server\_help.spec - tells the builder to include HTML/ and help.json in this repo to the next software build .

/HTML

The help build to be loaded on the appliance. Also includes the help.json file that defines the software-to-help links. To publish a help build, the workflow is:

- \* Clear all contents from /HTML, except the help.json file
- \* Generate a new help build and copy the contents to /HTML
- \* Add, commit, push to Gitlab.
- \* Upon a new push, the next build will update the following on the appliance:

    /opt/npm/www/help/HTML

    /opt/npm/www/help/help.json

/techpubs\_private

Working directories for individual writers. The idea is that writer would keep their "in-progress" work here and then merge content into techpubs\_src when it's ready to go GA.

The key word here is "PRIVATE." Only [*writer\_name*] can push, pull, etc. content in the [*writer\_name*] directory. This will reduce the likelihood of merge conflicts, since you are the only person committing/pushing/fetching/etc. content in your private directory. Hopefully this will also encourage frequent backups of work in progress.

A \_\_tasks subdirectory includes descriptions of common writer tasks: commit, push, fetch, updating techpubs\_src/, publishing new help, etc.

/techpubs\_src

Frame source files for producing online help and PDFs. This directory should not include "in-progress" content. All the content in here should be GA-ready. This ensures that anyone can update help/PDFs on-demand. Because writers will update content only when it becomes GA-ready, this will reduce (but not eliminate) the risk of merge conflicts.

This directory has the following structure:

  \_templates\_and\_import\_files/

  alloy\_server\_doc/

- \* The root includes shortcuts to commonly-used books.
- \* groups of topic files start with underscores so they appear first alphabetically.
- \* files used to generate PDF manuals start with "pdf\_".

/ww\_production

WebWorks stationery and projects used to generate online help.

## Git Setup on your Local Host

Note—You need to perform these steps *on each local host* that you plan to use for git work.

### Initial Setup

Do the following:

1. Install git on your local host.
2. Create the following folder: `c:\git_repos`  
This is the parent directory for all your local git repositories.
3. Open your default web browser and bookmark <https://gitlab.lab.nbttech.com>
4. Open a Windows Explorer window and go to the root c: directory. Right-click on `c:\git_repos` and choose Git Bash Here (near the top of the right-click menu). The Git Bash CLI window opens.
5. Enter the following commands:

```
git config --global user.name "First Last"
```

```
git config --global user.email First.Last@riverbed.com
```

- If you don't have an SSH key for your profile, the following message will appear at the top of the project page:

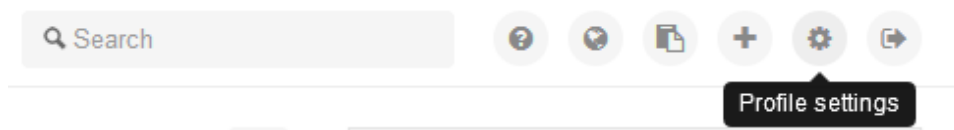
You won't be able to pull or push project code via SSH until you [add an SSH key](#) to your profile.

## Add an SSH Key to your Local Host

You need to add an SSH key on every machine that you plan to use for git. If you're using both a home and an office machine, for example, you'll need to add an SSH key on each machine.

Do the following on each host:

1. Open a web browser and go to the gitlab dashboard (<https://gitlab.lab.nbttech.com/>)
2. Click the Profile Settings toolbar button (top right).



3. Click SSH Keys in the left menu.
4. Click Add SSH Key (green button, top right).
5. In the Add an SSH Key page, click on the "SSH help page" link. The SSH help page provides step-by-step instructions for generating and adding an SSH key:
  - a. Git Bash: enter `ssh-keygen -t rsa -C "your-email.riverbed.com"`
  - b. Git Bash: enter `cat ~/.ssh/ida_rsa.pub`
  - c. Web browser: click Back to return to the Profile page
  - d. Copy and paste the key into the Add an SSH Key page

## Clone a Git Repository to your Local Host

**Note**—You must have a `c:\git_repos` parent directory for all git repositories on your local. Each repository will reside in a separate subdirectory. For example:

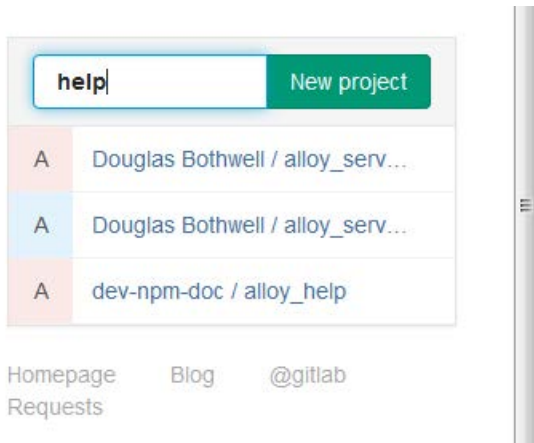
```
c:\git_repos (parent directory)
    \alloy_client_help(repository 1)
    \alloy_server_help(repository 2)
    \alloy_vm_help(repository 3)
```

Each subfolder houses a complete git repository. Each repo is separate independent from the others. Each repo has a `.git` database at the root level.

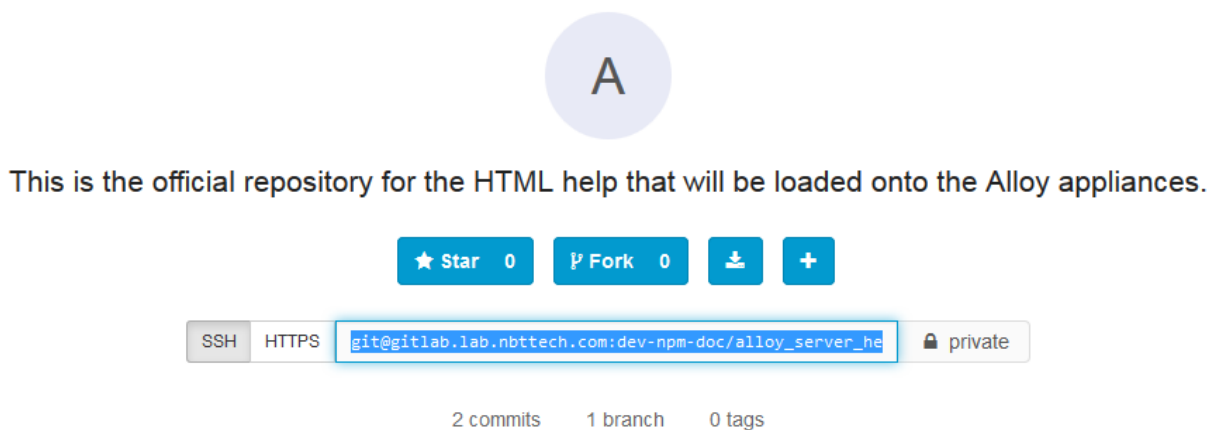
Do the following:

1. Open a web browser and navigate to the gitlab dashboard (<https://gitlab.lab.nbttech.com/>)

2. In the Gitlab dashboard (right column), search for the project you want and click the project link. Use the Filter by Name field ("help") to filter out irrelevant projects.



3. In the project home page, click the SSH button and select/copy the project URL.



4. Open a Windows Explorer window and navigate to c :
5. Right-click on c : \git\_repos and choose Git Bash Here.
6. Enter the following command:

```
git clone project_url
```

For example:


```
git clone git@gitlab.lab.nbttech.com:dev-npm-doc/alloy\_help.git
```

You now have an exact copy of the project (in its current state) on your local host.

## Git Procedures

### Branch: Verify that You are Working in the Correct Branch

**NOTE**—You should always be working in the same branch that Engineering is using for their official GA release. If you are working on doc for the Alloy Eagle release, for example, you should work in the same branch that Engineering is using for that release (in this case, **release\_eagle**). You can see the current branch in the Git Bash CLI:

A screenshot of a Git Bash terminal window. The title bar reads 'MINGW32:/c/git\_repos/alloy\_server\_help'. The prompt is 'dbothwell@DBOTHWELL-W7 MINGW32 /c/git\_repos/alloy\_server\_help (release\_eagle)'. The user has entered '\$ gitk -all' and the output shows the same prompt with '(release\_eagle)' in parentheses, indicating the current branch. A red arrow points from the bottom left towards the '(release\_eagle)' text in the output.

```
MINGW32:/c/git_repos/alloy_server_help
dbothwell@DBOTHWELL-W7 MINGW32 /c/git_repos/alloy_server_help (release_eagle)
$ gitk -all
dbothwell@DBOTHWELL-W7 MINGW32 /c/git_repos/alloy_server_help (release_eagle)
$ |
```

If you are unsure about which branch you should be working in, check with one of the Engineering POCs for the product release you are documenting.

**GOOD PRACTICE**—For each product release that will require documentation, the writer(s) should arrange a short meeting with the Engineering POC for that product to ensure that the documentation repository is set up correctly and is using the correct branch.

To switch to another branch, do the following:

- Close FrameMaker, ePublisher, and any other program that might have a file open in your repo. You want to be sure that all files are closed, since moving to another branch will “swap” out the current-branch versions of files with the new-branch versions.
- Open a Git Bash CLI window and navigate to the repos root directory (such as `c:\git_repos\alloy_server_help`)
- Enter the following command: **`git checkout branch-name`**

Here are some other useful branch commands:

- **`git branch -av`**  
This shows all the branches (both local and remote), the last commit message and tells you (for a branch that tracks a remote) if it is forward or behind (or both).
- **`git push origin HEAD:branch-name`**  
This pushes the latest commits (up to HEAD) in the Gitlab repo to **`branch-name`** repo

- **git checkout -b *branch-name* origin origin/*branch-name***  
This create a new branch fetches the latest commits (up to HEAD) in the Gitlab repo to the new ***branch-name*** repo

### Using Branches for Your Personal Work

Our Tech Pubs group has no hard policy against using branches (for example, a “my-special-feature” branch) and then merging the work into the main release branch. If you want to use branches in your own techpubs\_private folders, go for it. It’s your folder, it’s “private,” you can do what you want. However, keep in mind the following when you update the official source files:

- The recommended practice is to do your in-progress work in your techpubs\_private folder; then, when your content is ready to be published, merge the changes into the techpubs\_src folder. (In this context, “merge” means that you open both files in FrameMaker and copy/paste/update from your private file to the official file. It does NOT mean overwriting one file with another.) This enables you to compare directly your in-progress work with the GA-ready source files in techpubs\_src.
- FrameMaker source files are binary and proprietary. Git is most commonly used with plain text files. There is no easy way in git or FrameMaker to compare and merge binary files in one branch with the “same” files in another branch.
- When you checkout to a different branch, and the old/new branches have versions of the same file, git swaps out the file versions. topicA.fm (active branch) is visible in your local repo, while topicA.fm (inactive branch) is archived in a git snapshot/blob somewhere.
- To compare two versions of topicA.fm side-by-side, you need to
  - Save a copy of topicA.fm somewhere outside your repository (for example, c:\temp\topicA.my\_local\_branch\_copy.fm).
  - Switch over to the other branch.
  - Open the active-branch version and the temporary inactive-branch copy in FrameMaker and choose File > Utilities > Compare Documents.
- To merge changes from a temporary/personal branch to an official branch, the recommended practice is to
  - Create copies of the temporary-branch files
  - In the official branch, open each file and merge in changes from the temporary-file copy.
  - When all updates to all files are complete, add/commit the updates to your local repo and then push the updates to Gitlab.

### Status: Check the Status of your Local Repo

It is good practice to ensure that your local repo is in sync with the Gitlab repository. You should do this procedure regularly, at least once a day, and ALWAYS do this before you try to update any files in techpubs\_src/.

Do the following:

1. In Windows Explorer, navigate to c:\gitlab\_repos\, right-click on the repository directory, and choose Git Bash Here. The Git Bash CLI opens to the repository directory.
2. Enter the following: **git status**

The output tells the status of your local branch in relation to the Gitlab repo (*origin/branch\_name*). If your local branch is

- ***up-to-date with origin/branch\_name***: You're good, nothing to worry about.
- ***ahead of origin/branch\_name***: do a push right away to make sure that your local and remote repositories are in sync. See XREF.
- ***behind origin/branch\_name***: You're in a fast-forward situation. Fix it right away. See XREF.

## Add/Commit/Push: Save your Local Updates to Gitlab

Do the following:

1. Open a Git Bash session and navigate the CLI to the root folder of the repository. The easiest way to do this is:
  - o Open a Windows Explorer window
  - o Navigate to the `c:\gitlab_repos` folder
  - o Right-click on the repository directory (such as `alloy_server_help`) and choose Git Bash Here from the right-click menu.

2. Enter the following: **git status**

If your local branch is **behind origin/branch\_name**, you're in a fast-forward situation. Fix it and then return to step 1. [See Fetch/Rebase: Copy Gitlab Updates to your Local Repo](#) on page 10.

The `git status` output shows all the local files and directories that have changed since your last commit. For example:

```
$ git status
On branch branch_name
Your branch is up-to-date with 'origin/branch_name'.
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working
  directory)

        modified:   techpubs_src/alloy_doc/_monconfig/mifg_fm

Untracked files:
  (use "git add <file>..." to include in what will be committed)

        techpubs_src/alloy_doc/insets/
        HTML/
```

no changes added to commit (use "git add" and/or "git commit -a")

For our purposes, the difference between “unstaged” and “untracked” files is academic. The point is: These two lists show the files in your repository that have changed since your last commit. To commit any of these files (or all of them), you need to add them to the staging area via the **git add** command as described in the next step.

3. Run the **git add** command for each modified file that you want to commit to the repository. This basically says: "Include these files in the staging area so that, when I do a commit, they get updated in the local repo." You can copy-and-paste the filenames from the `git status` output (use the right-click menu in Git Bash). Examples:

```
$ git add techpubs_src/alloy_doc/_monconfig/mifg_fm
$ git add techpubs_src/alloy_doc/insets/
$ git add HTML/
$ git add --all           // 2 dashes, adds all updates shown in both lists
```

4. Enter a commit message according to the guidelines in [GOOD COMMIT MESSAGES](#) on page 12:



```
$ git commit -m "commit message summary
> [blank line]
> details line 1
> ...more detail lines...
> details line N, following by close-quote"
```

This command saves your staged files (via commit add) to your local repository. You will then see output that your commit went through. Example:

```
$ git commit -m "Bug-336699: Update MifG doc with info about capture jobs
>
> Updated techpubs_src/alloy_server_doc/__monconfig/mifg.fm:
> - New use case added to 'Use cases for MifGs'
> - New example and screenshot
> - New note: Refer to interfaces by parent MifG, not ifx name"
[djb-branch 06c4f1d] Bug-336699: Update MifG doc with info about capture jobs
1 file changed, 0 insertions(+), 0 deletions(-)
```

- NOTE--An easy way to enter the commit message is to type it up in Notepad and paste it in after the first double-quote.
  - WARNING--No double quotes within the commit message.
5. Once you commit your file changes in your local repository, you need to push your updates to the central Gitlab repository. Enter the following command:
- ```
git push origin HEAD:branch_name
```

The updates in your local repo are uploaded to the Gitlab repo. Your local staging area and repository are cleared for your next add/commit/push. Example output:

```
$ git push origin HEAD: branch_name
blah blah blah...

Counting objects: 6, done.
Delta compression using up to 4 threads.
Compressing objects: 100% (5/5), done.
Writing objects: 100% (6/6), 11.50 KiB | 0 bytes/s, done.
Total 6 (delta 4), reused 0 (delta 0)
To git@gitlab.lab.nbttech.com: dev-npm-doc/alloy_help.git
7c87f6e..0d1832c branch_name -> branch_name
```

**GOOD PRACTICE—**Push your updated repository to Gitlab immediately after you do a commit. This reduces the possibility of fast-forwarding situations.

## Fetch/Rebase: Copy Gitlab Updates to your Local Repo

The following steps outline a safe way to sync your local and Gitlab repos.

Do the following:

1. In Windows Explorer, navigate to `c:\gitlab_repos\`, right-click on the repository directory, and choose Git Bash Here.

The Git Bash CLI opens to the repository directory.

2. Enter the following: **git branch -av**

This shows the status of your local (*branch\_name*) and Gitlab (`remotes/origin/branch_name`) repositories. Specifically, it shows any commits in one repo that are not reflected in the other. In the following example, the local and remote repo's have two separate commits of two separate files:

```
$ git branch -av
* branch_name 14d768d updating merge-test.fm [sf1]
remotes/origin/branch_name 78b8f79 updating merge-test.fm [md2]
```

3. Enter the following: **git fetch origin**

This downloads the new(er) files from Gitlab.

4. Enter the following: **git log -p remotes/origin/branch\_name**

This shows the latest commits in the Gitlab repo and the files that were changed in each commit (bold):

```
$ git log -p remotes/origin/branch_name
commit 051036c5eaa637428842a0a3e41c332d6a24ff7d
Author: Bob Beshal ske <beshal.ske@riverbed.com>
Date: Mon Nov 2 14:45:45 2015 -0800

    test/ignore merge test: bob 1 in SF

diff --git a/techpubs_private/bbeshal.ske/merge-test.fm
b/techpubs_private/bbeshal.ske/merge-test.fm
index f498fa1..c7c061f 100644
Binary files a/techpubs_private/bbeshal.ske/merge-test.fm and
b/techpubs_private/bbeshal.ske/merge-test.fm differ

commit fd361b7968ffb9d6abd925b56de999fdf5b3190c
Author: Doug Bothwell <dbothwell@riverbed.com>
Date: Mon Nov 2 17:37:20 2015 -0500

    test/ignore merge test: doug 1 in MD

diff --git a/techpubs_private/dbothwell/merge-test.fm
b/techpubs_private/dbothwell/merge-test.fm
index d61427e..25bc8d9 100644
Binary files a/techpubs_private/dbothwell/merge-test.fm and
b/techpubs_private/dbothwell/merge-test.fm differ
```

5. Enter 'q' to quit vi mode.
6. Do the following if all of the following conditions are true:
  - o Remote commits indicate changes to files that you yourself have changed locally, AND

- You want to preserve your local changes in the Git repository (i.e., overwriting your local files with the Gitlab versions would wipe out work you want to preserve), AND
- **git status** indicates that your branch is behind `origin/branch_name`.

If all these conditions are true, you are in a fast-forward situation. Create a `local_temp` folder on your desktop and make copies of all files with local changes that you want to preserve. Append `LOCAL` to the filename (i.e., `filename.LOCAL.fm`). **The following step will overwrite the local files in your repository with the newer Gitlab versions.** You will then need to merge your changes from the `LOCAL` files to the overwritten files.

7. Enter the following: **git rebase origin/branch\_name**

This pulls all the commits/updates from the Gitlab file set to your local directory. In essence, this does a “copy/paste all” and overwrites any local files that have newer versions in Gitlab. Your local is now in sync with Gitlab.

```
$ git push origin HEAD: branch_name
Counting objects: 5, done.
Delta compression using up to 2 threads.
Compressing objects: 100% (5/5), done.
Writing objects: 100% (5/5), 88.59 KiB | 0 bytes/s, done.
Total 5 (delta 2), reused 0 (delta 0)
To git@gitlab.lab.nbttech.com: dbothwell/alloy_server_help_test.git
09cf2d2..78b8f79 HEAD -> branch_name
```

If you weren't in a fast-forward situation, you're finished.

If you were in a fast-forward situation, proceed.

8. For each local file with changes that you need to merge into the repository, do the following:
  - Open the repository file (`filename.fm`) and the local copy (`filename.LOCAL.fm`) in FrameMaker.
9. Compare the two documents:
  - Open both files in FrameMaker. Make sure your local copy is saved to disk.
  - Choose File > Utilities > Compare Documents.
  - Click Compare.
  - Go to the resulting `filenameCMP.fm` file.
  - In FrameMaker, choose View > Color > Views and select a view in which all colors are Normal or Cutout (i.e., not Invisible).
  - Carefully examine the insertions (SteelheadEX condition) and deletions (Deleted condition) and add the changes from `filename.LOCAL.fm` to `filename.fm`.
10. When you are finished merging in all your changes to the files in your repository, push your changes up to Gitlab as described in [Add/Commit/Push: Save your Local Updates to Gitlab](#) on page 8.

**GOOD PRACTICE**—Now that you've gone through the hassle of fixing a fast-forward situation, read [Error! Reference source not found.](#) (page 14) carefully so you can avoid this situation in the future.

## Good Practices for Git

This list is taken from previous sections in this document. The following practices will avoid needless hassles/complications and make the sailing smoother for you and others:

1. “Rebase” your local repository regularly—that is, sync your local repo with Gitlab—at least once a day, as described in [Fetch/Rebase: Copy Gitlab Updates to your Local Repo](#) on page 10.
2. Make sure that you are working in the correct branch for the product/release you are documenting.
3. At the outset of each new release that requires new doc, the writer(s) should meet with the Engineering POC to ensure that the writers have the repository and branch set up correctly.
4. Do your “in-progress” work in your `techpubs_private/` folder. Do not update the files in `techpubs_src/` until your contents has been tech- and peer-review and is ready for publication.
5. Do not use the same filenames in `techpubs_private` as in `techpubs_src`. If you want to copy/paste an official source file into your private folder, make sure you rename the file. This will minimize the risk of accidentally overwriting a file via copy/paste.
6. When you are ready to update `techpubs_src/`, rebase your local repository and check with other writers working on the project to let them know you’re about to update the Gitlab repository. The general sequence is this:
  - a. Writer A pings the other writer(s) and says: *Hey, I want to update the techpubs\_src folder. Let me know when the coast is clear.*
  - b. The other writers fetch the latest repo from Gitlab. Their local repo’s and Gitlab are in sync. Each writer replies to A: *OK, the coast is clear!*
  - c. A updates the files of interest in his local `techpubs_src`.
  - d. A adds, commits, and pushes his updates to Gitlab. A’s local repos and Gitlab are in sync; other local repo’s are one step behind.
  - e. A says: *OK, I’ve pushed my updates. Go ahead and fetch my updates.*
  - f. The other writers fetch the updated `techpubs_src` from Gitlab. Now all local repos are in sync with Gitlab and with each other.
7. Make sure you push all your updates to Gitlab when you’re finished and send out another email to let the other writes know the coast is clear.
8. Push your updated repository to Gitlab immediately after you do a commit (next step). This reduces the possibility of fast-forwarding situations.
9. Make your commit comments helpful to other writers and Engineering, as described in [Good Commit Messages](#).

## Good Commit Messages

**GOOD PRACTICE**—A good commit message has the following format:

1. A summary line (50—70 characters) describing the overall changes
2. A blank line separating the summary and the itemized list (REQUIRED)
3. A “dash space” bulleted list of details of what was changed, focusing on why the change was made

This is [standard git convention](#), beyond Riverbed. For more information read [here](#) and [here](#).

Commit messages have multiple audiences:

- For your team, to announce why you made a change
- For a future you, who may forget the context of why you made a change.
- For QA and other teams to know what and why things changed, without context.
- What bug was fixed?
- In all cases?
- What caveats exist?
- QA and other teams shouldn't be expected to see, to divine what changed.

You can see the commits and messages that have been pushed to Gitlab in the Gitlab web UI > Project > Commits page. (BTW: Since the changed file was binary, git can't tell what's changed in the file—only that the file itself has changed. Hence git logs “0 additions” and “0 deletions” for the file. This is another reason why good commit messages are important.)

### Bug-336699: Update MIfG doc with info about capture jobs

Updated `techpubs_src/alloy_server_doc/__monconfig/mifg.fm`:

- New use case added to 'Use cases for MIfGs'
- New example and screenshot
- New note: Refer to interfaces by parent MIfG, not ifx name

Showing **1** changed file with **0** additions and **0** deletions

techpubs\_src/alloy\_server\_doc/\_\_monconfig/mifg.fm

VIEW FILE @06c4f1d

INLINE SIDE-BY-SIDE

**GOOD PRACTICE**—A good commit message is helpful to other writers and to Engineering. Your commit message should include the following as applicable:

- Bug number
- Whether the commit includes updates to general source files or templates (`techpubs_src`), your own private folders (`techpubs_private`), a new help build (HTML), ePublisher stationeries/projects (`ww_production`), or `help.json` (repository root).
- Changes of special interest to other developers, QA, and/or Tech Pubs. Examples include:
  - Changes to `help.json` and fixes to broken help buttons
  - Updates to a topic that might affect a writer working on a related topic
  - Logos, text insets, import files, ePublisher projects

While there is *some* room for difference, here is the “NPM standard” we have been using:

High-level summary of what was done

← blank line separating high-level and detail →

Detail of why the change was done hard wrapped

to 70 characters with continuation indented with two spaces to line up with the dash space.

Another detail about why the change was made.

TODO: Add complete fix in M5.

Yet another item and remember to hard wrap and  
indent the continuation lines.

<- blank line before fields ->

ReviewedBy: hbae, jkennelly

ReviewLink: <https://review.lab.nbttech.com/r/123456/>

ReviewID: 123456

BugID: 654321 <optional if applicable>

## Fast-Forwarding: Why it's Bad and How to Avoid It

Fast-forwarding is when

1. Someone updates File A and uploads it to Gitlab;
2. You update your local copy of File A and try to upload it, and then
3. git generates an error because the Gitlab copy has changes that aren't in your local copy, and your copy has changes that aren't in the Gitlab copy.

```
error: failed to push some refs to 'https://gitlab.lab.nbttech.com/dev-npm-  
doc/alloy_help.git'
```

```
hint: Updates were rejected because the remote contains work that you do  
hint: not have locally. This is usually caused by another repository pushing
```

```
hint: to the same ref. You may want to first integrate the remote changes
```

```
hint: (e.g., 'git pull ...') before pushing again.
```

```
hint: See the 'Note about fast-forwards' in 'git push --help' for details.
```

If we were working with text files, merging the changes would be fairly easy, but we're not. If you get into this situation, you need to create a copy of your local file (*filename.LOCAL.fm*), overwrite your local file with the original, compare the Gitlab (*filename.fm*) and local (*filename.LOCAL.fm*) copies, merge the changes in manually, then upload your merged file to Gitlab. And if you have multiple files in this state it's even more of a hassle.

**GOOD PRACTICE**—To avoid this situation, the following practices are strongly recommended:

1. "Rebase" your local repository regularly—at least once a day. In other words, download any updated files from Gitlab to ensure that your local repository and Gitlab are in sync.
2. Do your "in-progress" work in your `techpubs_private/` folder. Do not update the files in `techpubs_src/` until your contents has been tech- and peer-review and is ready for publication.
3. When you are ready to update `techpubs_src/`, rebase your local repository and check with other writers working on the project to let them know you're about to update the Gitlab repository.
4. Make sure you push all your updates to Gitlab when you're finished and send out another email to let the other writers know the coast is clear.

## Gitlab Setup (*Advanced*): Creating Groups and Projects

These steps are primarily for the "Git Administrator" for SC-NPM-TP (SteelCentral NPM Tech Pubs).

### Create a group in Gitlab

1. Go to Gitlab (<https://git.nbttech.com/>)
2. Click Groups (left menu—you might need to expand the browser window to see the labels)
3. Click "+New Group"
4. In the New Group page, add the group name after the path
5. Click Create Group
6. You can now start creating a repository. To edit group, click Group Settings.

### Create a new Git project

1. Go to your group page, then click + (toolbar in top-right corner)
2. Enter project name in the Project Path field (all lowercase, underscores not dashes)
3. Select group owner in the Namespace field
4. Enter description
5. Set Visibility Level to public (this must be public b/c the automated build needs to be able to check this out anonymously)
6. Create project

**New Project**

Project path

Namespace

Import project from

Description (optional)

Visibility Level (?) ☒ **Private**  
Project access must be granted explicitly for each user.

☐ **Internal**  
The project can be cloned by any logged in user.

☒ **Public**  
The project can be cloned without any authentication.

**Note**—the Gitlab web UI has a lot of context-sensitive advice and will often provide required steps based on your current state.