

GIT!!

DON'T PANIC

BASIC GIT and comparison with SVN ...

Checking out

- SVN
 - You get a working copy of the specified branch (trunk, or another branch) on your machine
- GIT
 - You get a standalone repository of the specified branch (or default master), in addition to a working copy where you do your work. You also get a staging area (index). In total three trees – we will get back to it

Committing

- SVN
 - You commit your changes to the central repository online
 - You can not edit or improve your commit messages
- GIT
 - You add files you want to commit, then commit to your local repository (can be done in one step if you want)
 - To share your work, you push the HEAD of your local repo to the remote repo
 - You can work with GIT fully (committing, checking history etc etc) without contact with the remote repo, but to update your repo with work others have done, or make others aware of the work you have done, you must contact the remote repo (pull, push)
 - You can edit your commit messages, and also squash commits together – many small commits can become one big clearer one

... BASIC GIT and comparison with SVN

Branches

- SVN
 - Each branch is a full standalone directory of files belonging to that branch
 - Checking out several branches means you have separate directories for each branch. You have a full working copy of each branch.
- GIT
 - Each branch is just a pointer to a specific state of your file collection (pointer to a snapshot of your repo)
 - → branches are very lightweight and used much more flexibly and often than in svn

Merging

- SVN
 - Merging involves pushing to the remote repo, fixing mistakes makes the history very messy
 - Each commit requires a separate merge.
- GIT
 - You can sort out all mistakes, conflicts etc on your own fork and clear up history before you merge into the upstream repo
 - Merge is not performed until all conflicts are actually solved
 - You can squash your commits and merge a logical unit rather than a series of small related commits

GIT FOR SUBVERSION USERS

presented by TOWER – the best Git client for Mac and Windows



Creating a New Repository

With `git init`, an **empty** repository is created in the current folder of your local hard drive. The `git add` command then marks the current contents of your project directory for the next (and in this case: first) commit.

```
$ svnadmin create /path/to/repo
$ svn import /path/to/local/project http://
  example.com/svn/trunk -m "Initial import"
```

```
$ git init
$ git add .
$ git commit -m "Initial commit"
```

Cloning a Remote Repository

Getting a copy of the project from a remote server seems almost identical. However, after performing `git clone`, you have a **full-blown local repository** on your machine, not just a working copy.

```
$ svn checkout
  svn+ssh://svn@example.com/svn/trunk
```

```
$ git clone
  ssh://git@example.com/path/to/git-repo.git
```

Inspecting History

To inspect historic commits, both systems use the `log` command. Keep in mind, however, that `git log` doesn't need to ask the remote server for data: your project's history is already at hand, saved in your local repository.

```
$ svn log | less
```

```
$ git log
```

Committing Local Changes

Inspecting your current local changes is very similar in both systems.

```
$ svn status
$ svn diff | less
```

```
$ git status
$ git diff
```

In case you've created new files or deleted old ones, you should tell Git with the `git add` and `git rm` commands. You'll be pleased to hear that it's safe to inform Git **after** deleting or moving a file or even a folder. This means you should feel free to delete or move even complete directory structures in your favorite editor, IDE, or file browser and **later** confirm the action with the `add` and `rm` commands.

```
$ svn add <file>
$ svn rm <file>
```

```
$ git add <file>
$ git rm <file>
```

In its simplest form, committing can feel just like in Subversion. With the `-a` option, you tell Git to simply **add** all current local changes to the commit.

```
$ svn commit -m "message"
```

```
$ git commit -a -m "message"
```

Although short-circuiting Git's staging area like this can make sense, you'll quickly begin to love it once you understand how valuable it is:

You can add selected files to the staging area and even limit this to certain parts (or even lines) of a file by specifying the `-p` option. This allows you to craft your commits in a very **granular** way and only add changes that belong to the **same topic** in a single commit.

```
$ git add <file1> <file2>
$ git add -p <file3>
```

Branching & Tagging

In contrast to Subversion, Git doesn't use directories to manage branches. Instead, it uses a more powerful and lightweight approach. As you might have already noticed, the `git status` command also informs you about which branch you are currently working on. And in Git, you are **always** working on a branch!

```
$ svn copy http://example.com/svn/trunk/
  http://example.com/svn/branches/<new-branch>
```

```
$ git branch <new-branch>
```

To switch to a different branch and make it active (then also referred to as the **HEAD** branch), the `git checkout` command is used. Because switching can take some time in Subversion, it's not unusual to instead have multiple working copies on your disk. In Git, this would be extremely uncommon: since operations are very fast, you only keep a single local repository on your disk.

```
$ svn switch
  http://example.com/svn/branches/<branch>
```

```
$ git checkout <branch>
```

Listing all available local branches just requires the `git branch` command without further arguments.

```
$ svn list http://example.com/svn/branches/
```

```
$ git branch
```

Creating tags is just as quick & cheap as creating branches.

```
$ svn copy http://example.com/svn/trunk/
  http://example.com/svn/tags/<tag-name>
```

```
$ git tag -a <tag-name>
```

Merging Changes

Like in newer versions of SVN, you only need to provide the branch you want to integrate to the `git merge` command.

```
$ svn merge -r REV1:REV2
  http://example.com/svn/branches/<other-branch>
$ svn merge
  http://example.com/svn/branches/<other-branch>
  (or in newer SVN versions)
```

```
$ git merge <other-branch>
```



Everything else is taken care of for you: you can merge two branches as often as you like, don't have to specify any revisions and can expect the operation to be blazingly fast if you're merging two local branches.

If a merge conflict should occur, Git will already update the rest of the working copy to the new state. After resolving a conflicted file, you can mark it using the `git add` command.

```
$ svn resolved <file>
```

```
$ git add <file>
```

Sharing & Collaborating

To download & integrate new changes from a remote server, you use the `git pull` command.

```
$ svn update
```

```
$ git pull
```

If you only want to download & inspect remote changes (before integrating them), you can use `git fetch`. Later, you can integrate the downloaded changes via `git merge`.

```
$ git fetch
```

In Subversion, data is automatically uploaded to the central server when committing it. In Git, however, this is a separate step. This means you can decide for yourself **if and when** you want to share your work. Once you're ready, the `git push` command will upload the changes from your currently active branch to the remote branch you specify.

```
$ git push <remote> <branch>
```

Your teammates, too, will publish their work like this on a remote (with the `git push` command). If you want to start working on such a branch, you need to create your own local copy of it. You can use the `git checkout` command with the `--track` option to do just that: create a local version of the specified remote branch. You can later share the additional commits you've made at any time with the `git push` command, again.

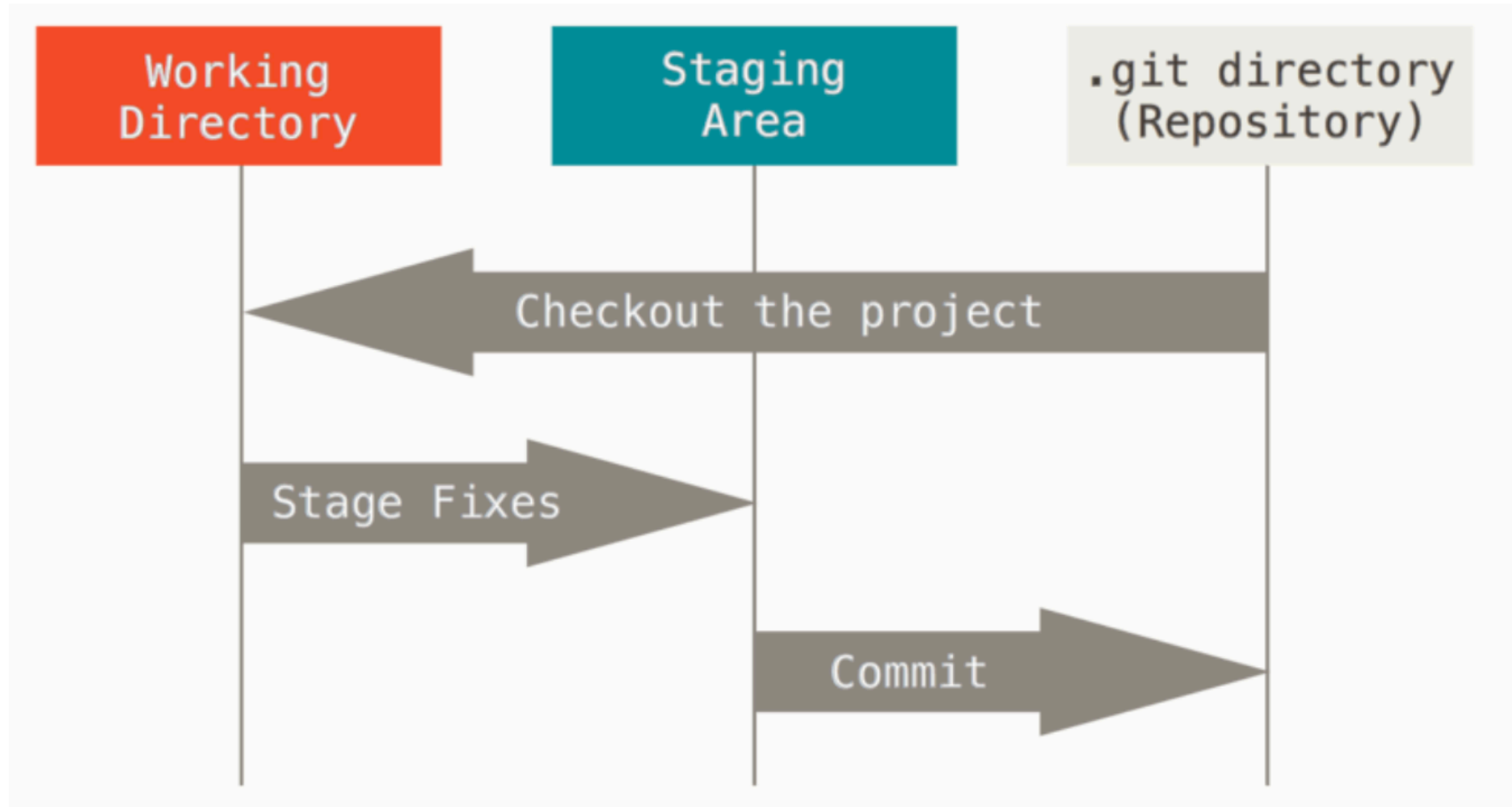
```
$ svn switch
  http://example.com/svn/branches/<branch>
```

```
$ git checkout --track <remote>/<branch>
```

The concepts

- A fork
 - Basically a full standalone copy of a chosen repository in a **different namespace** (usually your namespace)
 - A fork will also allow you to share your work with others, and lives on the Coderefinery GitLab instance
 - It allows you to do work without “muddling” up the central repo
- A local copy
 - `git clone` creates a local copy of the fork (or any other repo) for you to work on
- Working directory: holds your actual files for you to work on
- Index: staging area for changes you want to commit
 - You can select what changes you want to commit, or commit everything that has changed
- HEAD: points to your last commit of the branch you are currently on

Your local repository - the three “trees”



My proposal of our branch model

3 branches on the remote repo

- **master** (kind of trunk)
 - Releases will be tags on master
- **minor**
 - for development of features aimed for minor release
 - You create a development branch off the minor branch
 - You merge into minor branch once done
 - Currently made off of 5.4
- **major**
 - for development aimed for major release
 - You create a development branch off the major branch
 - You merge into major branch once done
 - Currently made off of 6.0

Bugfixes

- You create a development branch off the major branch
- You merge into master branch once done
- Bugfixes are automatically (hopefully) merged into minor and major branches

At time of release:

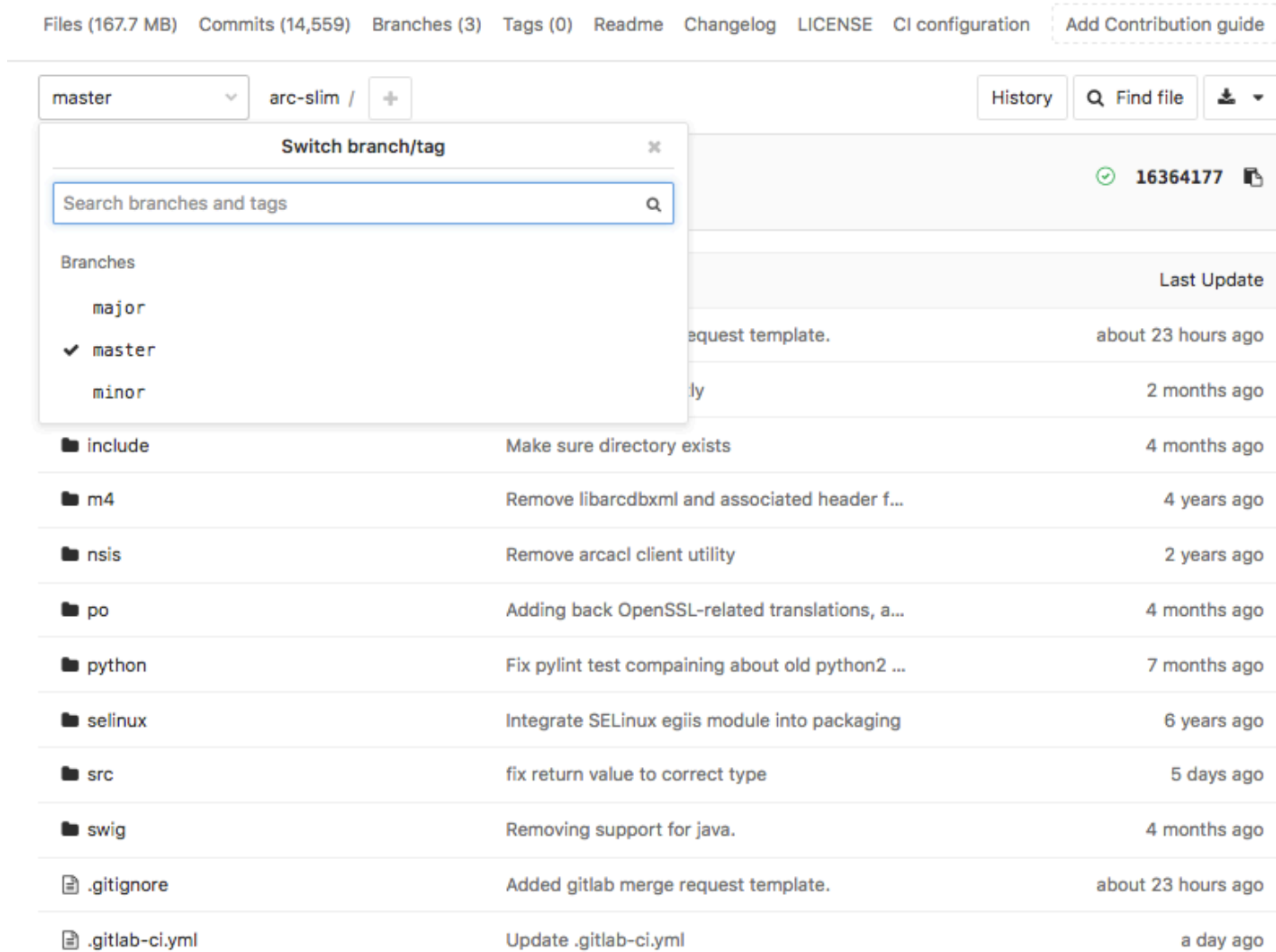
- if minor: minor branch will be merged into master
 - Tag master for release
- If major: major branch will be merged into master
 - Tag master for release

Training repo












 <https://source.coderefinery.org/nordugrid/arc-slim>

I have added some of you as members with role “developer”

<https://docs.gitlab.com/ee/user/permissions.html>



The screenshot shows the GitLab interface for the repository 'arc-slim' on the 'master' branch. At the top, there are navigation links: Files (167.7 MB), Commits (14,559), Branches (3), Tags (0), Readme, Changelog, LICENSE, CI configuration, and an 'Add Contribution guide' button. Below the navigation is a search bar with 'Find file' and a dropdown menu. A 'Switch branch/tag' dialog box is open, displaying a search bar and a list of branches: 'major', 'master' (selected), and 'minor'. The main content area shows a list of files and directories with their commit messages and last update times.

		Last Update
	16364177	
	request template.	about 23 hours ago
	ly	2 months ago
	include	Make sure directory exists 4 months ago
	m4	Remove libarcdbxml and associated header f... 4 years ago
	nsis	Remove arcaccl client utility 2 years ago
	po	Adding back OpenSSL-related translations, a... 4 months ago
	python	Fix pylint test complaining about old python2 ... 7 months ago
	selinux	Integrate SELinux egiis module into packaging 6 years ago
	src	fix return value to correct type 5 days ago
	swig	Removing support for java. 4 months ago
	.gitignore	Added gitlab merge request template. about 23 hours ago
	.gitlab-ci.yml	Update .gitlab-ci.yml a day ago

Overview of workflow

Forking, branching, merging

- Fork the central repo. Central repo will be called upstream.
- Create branch from minor (major) branch.
- Do some work.
- Keep your fork branch up to date with upstream
 - Drag down changes from the upstream repo (all branches)
 - Merge or rebase onto your fork. Branch by branch.
 - Bring you feature branch up to date with the now updated parent branch by merging.
- Before pushing the commits to your remote repo please consider if you can squash your commits.
- Push to the remote (fork) repo with some regularity.
- (This step might instead be just done directly onto the upstream branch) Once you are completely ready with the work, or ready with a sufficiently standalone part: merge the development branch into your parent branch (minor or major branch).
 - Make sure you have synchronized your minor/major branch with the upstream branches
 - Resolve any conflicts that may occur
 - Delete your development branch, both local copy and remotely on the fork
- Now you are ready to create a merge request of your forks minor(major) branch to the upstream minor (major) branch.
- Repeat procedure if you have a new feature, or some new development on the old feature.

Merge conflicts

- Big improvement in working model is that each developer will be responsible for keeping his development branch up to date with the parent branch
 - After all, the developer is the best one to solve conflicts related to his code
- At the time of a merge request, it means that conflicts are already solved on the development branch
 - Surely there will be some exceptions, but anyway a huge improvement
- Be aware that once you have opened a merge request, all commits that come after this will be included in the merge.
 - Important to not start working on some other stuff on the same branch, but just perform commits to solve merge conflicts
 - We should look at a development branch containing a smaller unit of development, and delete the branch after this is done. Then start a new one.

Fork the repo using the GitLab web interface.

Create your local repo of the fork and cd into it:

```
git clone git@source.coderefinery.org:maikenp/arc.git arc-fork
cd arc-fork
```

Add references to your fork repo, and the upstream repo

```
git remote add origin git@source.coderefinery.org:maikenp/arc.git
git remote add upstream git@source.coderefinery.org:nordugrid/arc.git
```

Fork the repo using the GitLab web interface.

Create your local repo of the fork and cd into it:

```
git clone git@source.coderefinery.org:maikenp/arc.git arc-fork
cd arc-fork
```

Add references to your fork repo, and the upstream repo

```
git remote add origin git@source.coderefinery.org:maikenp/arc.git
git remote add upstream git@source.coderefinery.org:nordugrid/arc.git
```

All of the branches from the remote repo will not be visible to you. You set them up by demand. In this example you will need the minor branch

```
git checkout -b minor --track origin/minor #create local branch minor which is a copy of the origin/minor branch, and switch into that branch
```

Create a new development branch based on the existing minor branch and call it something useful, e.g. restplugin. This command will at the same time check you out of any other branch you were in, and check you into your new branch restplugin:

```
git checkout -b restplugin minor
```

Do your work.

Fork the repo using the GitLab web interface.

Create your local repo of the fork and cd into it:

```
git clone git@source.coderefinery.org:maikenp/arc.git arc-fork
cd arc-fork
```

Add references to your fork repo, and the upstream repo

```
git remote add origin git@source.coderefinery.org:maikenp/arc.git
git remote add upstream git@source.coderefinery.org:nordugrid/arc.git
```

All of the branches from the remote repo will not be visible to you. You set them up by demand. In this example you will need the minor branch

```
git checkout -b minor --track origin/minor #create local branch minor which is a copy of the origin/minor branch, and switch into that branch
```

Create a new development branch based on the existing minor branch and call it something useful, e.g. restplugin. This command will at the same time check you out of any other branch you were in, and check you into your new branch restplugin:

```
git checkout -b restplugin minor
```

Do your work.

Add the files you worked on, commit them to your local branch, squash, then push them to the fork repo. Note the different git add options: `git add --all` : stages All files, in addition to the removing files, `git add .` : stages new and modified, but does not delete, `git add -u` : stages modified and deleted, but not new, `git add myfile.cpp` : will only add that file).

```
git add . #stage the changes to your index
git commit " #make a nice commit message, and commit the changes to your local branch
```

Repeat.

Let's say you had 10 commits related to the same change.

```
git reset --soft HEAD~10 #go back 10 commits
git commit -m "New message for the combined commit" #apply all your changes all in one here
git push -u origin restplugin #push the updates to the remote repo
```

Fork the repo using the GitLab web interface.

Create your local repo of the fork and cd into it:

```
git clone git@source.coderefinery.org:maikenp/arc.git arc-fork
cd arc-fork
```

Add references to your fork repo, and the upstream repo

```
git remote add origin git@source.coderefinery.org:maikenp/arc.git
git remote add upstream git@source.coderefinery.org:nordugrid/arc.git
```

All of the branches from the remote repo will not be visible to you. You set them up by demand. In this example you will need the minor branch

```
git checkout -b minor --track origin/minor #create local branch minor which is a copy of the origin/minor branch, and switch into that branch
```

Create a new development branch based on the existing minor branch and call it something useful, e.g. restplugin. This command will at the same time check you out of any other branch you were in, and check you into your new branch restplugin:

```
git checkout -b restplugin minor
```

Do your work.

Add the files you worked on, commit them to your local branch, squash, then push them to the fork repo. Note the different git add options: `git add --all` : stages All files, in addition to the removing files, `git add .` : stages new and modified, but does not delete, `git add -u` : stages modified and deleted, but not new, `git add myfile.cpp` : will only add that file).

```
git add . #stage the changes to your index
git commit " #make a nice commit message, and commit the changes to your local branch
Repeat.
```

Let's say you had 10 commits related to the same change.

```
git reset --soft HEAD~10 #go back 10 commits
git commit -m "New message for the combined commit" #apply all your changes all in one here
git push -u origin restplugin #push the updates to the remote repo
```

Make sure to frequently update your fork and development branch to take into account important fixes upstream. Assume some bufixes have been merged to the parent minor branch you are working from.

```
git fetch upstream #update your local copy of the upstream repo
git checkout minor #move into your forks minor branch working copy
git merge upstream/minor minor #merge the upstream changes in minor branch into your forks minor branch working copy
git commit -a #add changed files and commit
git checkout restplugin
#switch to the restplugin branch working copy git merge minor restplugin #merge the updates from the parent minor branch into your restplugin development branch
```

Fork the repo using the GitLab web interface.

Create your local repo of the fork and cd into it:

```
git clone git@source.coderefinery.org:maikenp/arc.git arc-fork
cd arc-fork
```

Add references to your fork repo, and the upstream repo

```
git remote add origin git@source.coderefinery.org:maikenp/arc.git
git remote add upstream git@source.coderefinery.org:nordugrid/arc.git
```

All of the branches from the remote repo will not be visible to you. You set them up by demand. In this example you will need the minor branch

```
git checkout -b minor --track origin/minor #create local branch minor which is a copy of the origin/minor branch, and switch into that branch
```

Create a new development branch based on the existing minor branch and call it something useful, e.g. restplugin. This command will at the same time check you out of any other branch you were in, and check you into your new branch restplugin:

```
git checkout -b restplugin minor
```

Do your work.

Add the files you worked on, commit them to your local branch, squash, then push them to the fork repo. Note the different git add options: `git add --all` : stages All files, in addition to the removing files, `git add .` : stages new and modified, but does not delete, `git add -u` : stages modified and deleted, but not new, `git add myfile.cpp` : will only add that file).

```
git add . #stage the changes to your index
git commit " #make a nice commit message, and commit the changes to your local branch
Repeat.
```

Let's say you had 10 commits related to the same change.

```
git reset --soft HEAD~10 #go back 10 commits
git commit -m "New message for the combined commit" #apply all your changes all in one here
git push -u origin restplugin #push the updates to the remote repo
```

Make sure to frequently update your fork and development branch to take into account important fixes upstream. Assume some bufixes have been merged to the parent minor branch you are working from.

```
git fetch upstream #update your local copy of the upstream repo
git checkout minor #move into your forks minor branch working copy
git merge upstream/minor minor #merge the upstream changes in minor branch into your forks minor branch working copy
git commit -a #add changed files and commit
git checkout restplugin
#switch to the restplugin branch working copy git merge minor restplugin #merge the updates from the parent minor branch into your restplugin development branch
```

Continue working, adding, committing. Keep your development and release branches up to date as above.

Once you are ready with a logical part of the work, or all the work, create a merge request to merge your branch into the upstream repo. Go to the upstream repos web interface and create a merge request of the restplugin branch onto the minor branch.

Clean up your development branch in your fork repo, both locally and on the remote fork repo.

```
git branch -d restplugin #remove local branch
git push origin --delete restplugin #remove remote branch
```


Merge request

- Select template for writing merge request
 - Help-text here to be sorted out
- Short description should be informative and in a state that can be used for release notes
- Detailed description should specifically answer what has changed, why, and what implications this has for end user.
- We will probably use tags
- Proposal: This will create a merge commit which will take the role of the changelog.
 - Will it? Can it? Is it ok?

Merge request

- Fork
- Create new branch for the work on my fork, do work, add (stage) changes, commit to branch.
- Push to remote fork repo
 - Git replies by suggesting that you do a merge request, and how to do it, with url provided

Maiken / arc-slim

This project Search

Project **Repository** Issues 0 Merge Requests 0 Pipelines Wiki Snippets Settings

Files Commits Branches Tags Contributors Graph Compare Charts

⚠ There will be maintenance on Thursday November 23rd between 10-12 AM EET. Expect short (<60 seconds) unavailability.

You pushed to **changelog** 2 minutes ago

Create merge request

changelog arc-slim / +

Find file

Name	Last commit > 82f72b6c 3 minutes ago - Added entry in changelog for second fix for negative v... History	Last Update
debian	Use space consistently	2 months ago
include	Make sure directory exists	4 months ago
m4	Remove libarcdbxml and associated header files since it is not needed by anything (Fixes #3...	4 years ago
nsis	Remove unused client utility	2 years ago

Maiken Pedersen - ARC F2F Ljubljana

To fork or not to fork

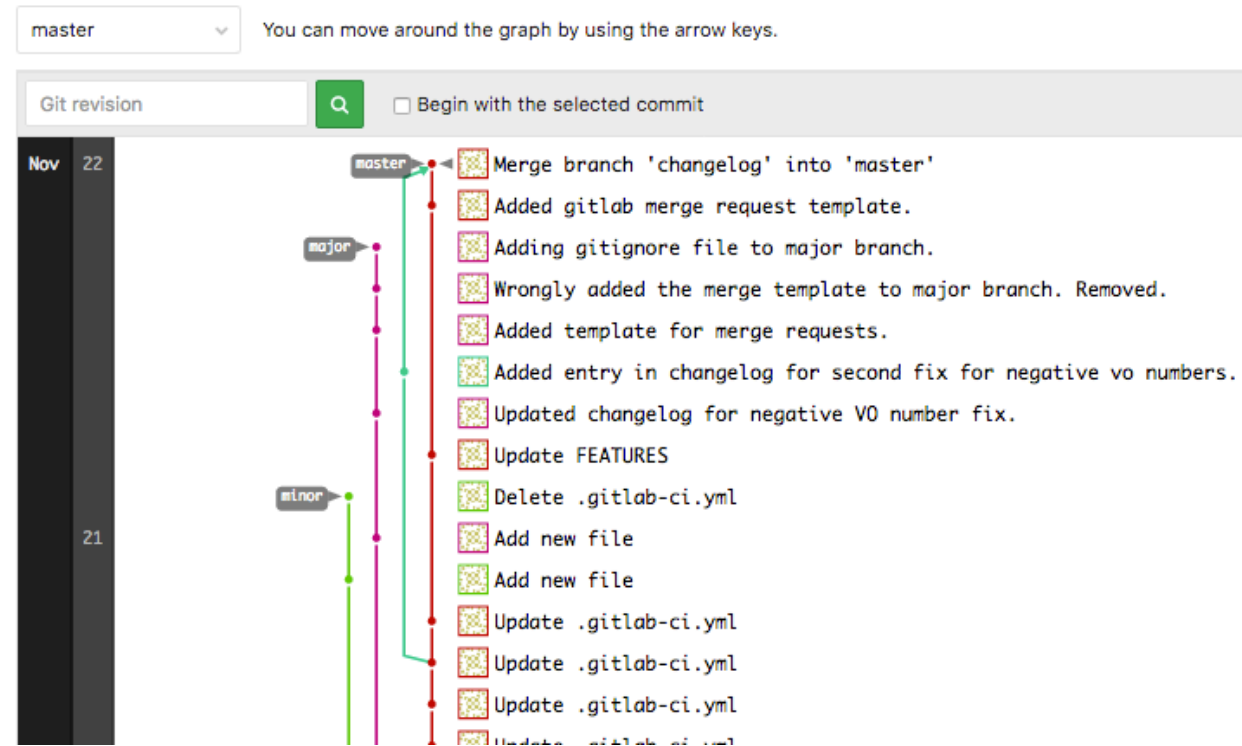
We could work without forking. Each developer then just makes a development branch right off of the central git repo.

Pros with forks

- Working with forks allows each developer to make as much mess as he wants without cluttering the remote official git repo

Cons with forks

- Well, the only con I can think of is that the public will not see all that is going on, but that is not necessarily a con!



GitLab functionality ...

- Continuous integration

- Currently for test using University cloud
 - Builds the branch in a docker container running on a predefined machine, git clones the repo on that machine, and performs build, live build on gitlab, report on success/fail
- Can be set up for all branches, or some branches
 - Depends on space on the build machine used

- Marek has set up CI and deployment using Jenkins from our svn
 - <http://autodeploy.grid.upjs.sk:8080/>
 - Will integrate this into our git repo, not started sorting out details here yet
 - He has also set up rpm production. Remains to set up for all supported platforms

The screenshot shows the GitLab Pipelines interface. At the top, there are navigation links for Project, Repository, Registry, Issues (0), Merge Requests (0), Pipelines, Wiki, Snippets, Members, and Settings. Below this, there are tabs for Pipelines, Jobs, Schedules, Environments, and Charts. The main content area shows a table of pipeline runs with columns for Status, Pipeline, Commit, and Stages. The table includes a 'Run Pipeline' button and a 'CI Lint' button. The pipeline runs are as follows:

Status	Pipeline	Commit	Stages
passed	#93 by latest	Y master -> 16364177 Merge branch 'changelog' into 'mast...	00:18:18 about 21 hours ago
failed	#92 by latest	Y master -> 9fbf798a Added gitlab merge request template.	00:04:45 about 22 hours ago
failed	#91 by latest	Y major -> ac7002d1 Adding gitignore file to major branch.	00:09:34 about 22 hours ago
passed	#90 by latest	Y major -> 416527d3 Wrongly added the merge template t...	00:19:27 about 22 hours ago
passed	#89 by latest	Y major -> 035232d7 Added template for merge requests.	00:17:34 about 22 hours ago

The screenshot shows a GitLab commit page for the 'master' branch in the 'arc-slim' repository. The commit message is 'Update .gitlab-ci.yml' by Maiken, committed a day ago. A message indicates that the GitLab CI configuration is valid. Below this, the content of the '.gitlab-ci.yml' file is displayed:

```
1 centos-build-arc:
2   image: maikenp/centosarc2
3   stage: build
4   script:
5     - ./autogen.sh
6     - ./configure
7     - make
8     - make install
9   only:
10    - master
11    - minor
12    - major
```


... GitLab functionality ...

Issues

- I expect we continue using bugzilla for reporting bugs/features?
- Issues in GitLab can be used by developers only.
 - Must decide how actively we use issues.
 - One issue per merge? Point to bugzilla record?
 - Use issue to discuss and request feature?

Project Repository Registry **Issues 0** Merge Requests 0 Pipelines Wiki Snippets Members Settings

List Board **Labels** Milestones

Labels can be applied to issues and merge requests. Star a label to make it a priority label. Order the prioritized labels to change their relative priority, by dragging. [New label](#)

Prioritized Labels

Star labels to start sorting by priority

Other Labels

☆ Doing	Project Label	view merge requests	view open issues	Subscribe	↑	✎	🗑️
☆ To Do	Project Label	view merge requests	view open issues	Subscribe	↑	✎	🗑️
☆ bugfix	Project Label	view merge requests	view open issues	Subscribe	↑	✎	🗑️
☆ critical	Project Label	view merge requests	view open issues	Subscribe	↑	✎	🗑️
☆ discussion	Project Label	view merge requests	view open issues	Subscribe	↑	✎	🗑️
☆ documentation	Project Label	view merge requests	view open issues	Subscribe	↑	✎	🗑️
☆ enhancement	Project Label	view merge requests	view open issues	Subscribe	↑	✎	🗑️
☆ major	Project Label	view merge requests	view open issues	Subscribe	↑	✎	🗑️
☆ minor	Project Label	view merge requests	view open issues	Subscribe	↑	✎	🗑️
☆ suggestion	Project Label	view merge requests	view open issues	Subscribe	↑	✎	🗑️

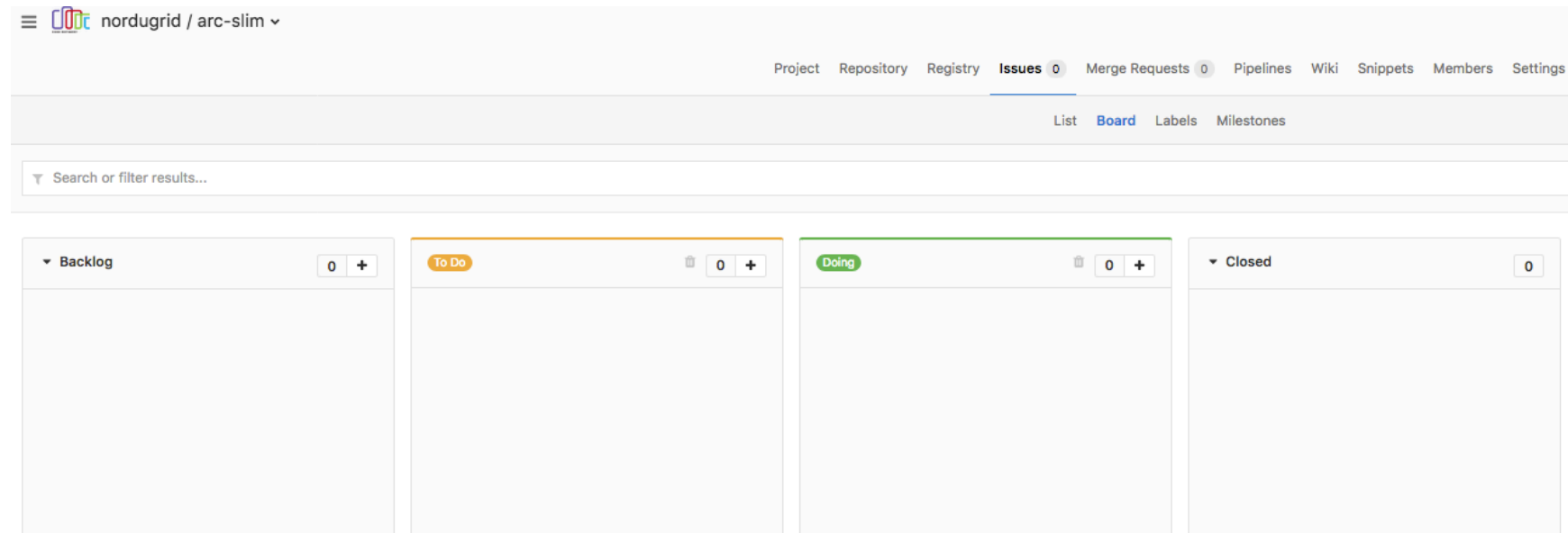
... GitLab functionality

Many possibilities available

- Milestones
 - Organizes issues and merge requests into a cohesive group.
 - Can be used keeping track of an upcoming software version.
- Work flow boards
 - Can be used to track issues

- Release notes

- Plan to set up automatic draft of release notes based on commit messages/merge commits/issues



How to do releases, and how to use various GitLab functionality

- How can current release procedure be simplified/automated/improved ?
 - Release notes should be pushed to web automatically once the rpms are available
 - Move release notes from doc repo to arc1 docs
 - Will at last have release note tag same as release
 - Other suggestions?
- Release notes on GitLab?
- Documentation on GitLab?
- Webhooks – read-the-docs?
- Webhooks – bugzilla?

Content of release notes:

- Each merge request should be linked to an issue? That means one must create an issue in GitLab that e.g. could link to a bugzilla bug/feature request.
 - Not clear to me how it will look if we issues, or just merge request comments for release notes. Needs some testing once set up.

Automatic merging

- Want to set up automatic merging of bugfixes
 - When bugfixes are merged onto master branch, port (merge) these to minor and major branch
 - Requires use of tags and possibly use of issues for merging
- To be sorted out
- <https://www.atlassian.com/blog/git/git-automatic-merges-with-server-side-hooks-for-the-win>

Useful things

Git log with graph option

```
1x-193-157-237-153@arc-slim (git:master)$ git log --pretty=format:"%h %s" --graph
* 163641772 Merge branch 'changelog' into 'master'
| \
| * 82f72b6c7 Added entry in changelog for second fix for negative vo numbers.
* | 9fbf798ae Added gitlab merge request template.
* | 9d80215cf Update FEATURES
* | 086874ae9 Update .gitlab-ci.yml
| /
* 9dd5b446a Update .gitlab-ci.yml
* 9d66efec5 Update .gitlab-ci.yml
* a168c8a06 Update .gitlab-ci.yml
* 5d... Update .gitlab-ci.yml
```

Useful tip: to always see which branch you are on:

In `.bashrc` do:

```
parse_git_branch() {  
    git branch 2> /dev/null | sed -e '/^[^*]/d' -e 's/* \(.*\)/ (git:\1)/'  
}  
export PS1="\h@\W\$(parse_git_branch)\[\033[00m\]$"
```

Committed, but want to change the commit message?

`git commit –amend`

- If you just committed, and have not made any changes, this will just open the editor with your last commit message, and you can edit it
- If you did make changes, add those, and with `git commit –ammend`, the changes will be committed, together with the (edited) commit message
- The second commit replaces the first one, so you will only have a single commit

Getting help

- <https://git-scm.com/book/en/v2/Getting-Started-About-Version-Control>
 - Chap 1-3 are very useful
- <https://learngitbranching.js.org/>
- <https://try.github.io/levels/1/challenges/1>

Git Cheat Sheet



Git Basics

<code>git init <directory></code>	Create empty Git repo in specified directory. Run with no arguments to initialize the current directory as a git repository.
<code>git clone <repo></code>	Clone repo located at <repo> onto local machine. Original repo can be located on the local filesystem or on a remote machine via HTTP or SSH.
<code>git config user.name <name></code>	Define author name to be used for all commits in current repo. Devs commonly use <code>--global</code> flag to set config options for current user.
<code>git add <directory></code>	Stage all changes in <directory> for the next commit. Replace <directory> with a <file> to change a specific file.
<code>git commit -m "<message>"</code>	Commit the staged snapshot, but instead of launching a text editor, use <message> as the commit message.
<code>git status</code>	List which files are staged, unstaged, and untracked.
<code>git log</code>	Display the entire commit history using the default format. For customization see additional options.
<code>git diff</code>	Show unstaged changes between your index and working directory.

Undoing Changes

<code>git revert <commit></code>	Create new commit that undoes all of the changes made in <commit>, then apply it to the current branch.
<code>git reset <file></code>	Remove <file> from the staging area, but leave the working directory unchanged. This unstages a file without overwriting any changes.
<code>git clean -n</code>	Shows which files would be removed from working directory. Use the <code>-f</code> flag in place of the <code>-n</code> flag to execute the clean.

Rewriting Git History

<code>git commit --amend</code>	Replace the last commit with the staged changes and last commit combined. Use with nothing staged to edit the last commit's message.
<code>git rebase <base></code>	Rebase the current branch onto <base>. <base> can be a commit ID, a branch name, a tag, or a relative reference to HEAD.
<code>git reflog</code>	Show a log of changes to the local repository's HEAD. Add <code>--relative-date</code> flag to show date info or <code>--all</code> to show all refs.

Git Branches

<code>git branch</code>	List all of the branches in your repo. Add a <branch> argument to create a new branch with the name <branch>.
<code>git checkout -b <branch></code>	Create and check out a new branch named <branch>. Drop the <code>-b</code> flag to checkout an existing branch.
<code>git merge <branch></code>	Merge <branch> into the current branch.

Remote Repositories

<code>git remote add <name> <url></code>	Create a new connection to a remote repo. After adding a remote, you can use <name> as a shortcut for <url> in other commands.
<code>git fetch <remote> <branch></code>	Fetches a specific <branch>, from the repo. Leave off <branch> to fetch all remote refs.
<code>git pull <remote></code>	Fetch the specified remote's copy of current branch and immediately merge it into the local copy.
<code>git push <remote> <branch></code>	Push the branch to <remote>, along with necessary commits and objects. Creates named branch in the remote repo if it doesn't exist.

Additional Options +

git config	
<code>git config --global user.name <name></code>	Define the author name to be used for all commits by the current user.
<code>git config --global user.email <email></code>	Define the author email to be used for all commits by the current user.
<code>git config --global alias.<alias-name> <git-command></code>	Create shortcut for a Git command. E.g. <code>alias.glog log --graph --oneline</code> will set <code>git glog</code> equivalent to <code>git log --graph --oneline</code> .
<code>git config --system core.editor <editor></code>	Set text editor used by commands for all users on the machine. <code><editor></code> arg should be the command that launches the desired editor (e.g., vi).
<code>git config --global --edit</code>	Open the global configuration file in a text editor for manual editing.

git log	
<code>git log --<limit></code>	Limit number of commits by <code><limit></code> . E.g. <code>git log -5</code> will limit to 5 commits.
<code>git log --oneline</code>	Condense each commit to a single line.
<code>git log -p</code>	Display the full diff of each commit.
<code>git log --stat</code>	Include which files were altered and the relative number of lines that were added or deleted from each of them.
<code>git log --author="<pattern>"</code>	Search for commits by a particular author.
<code>git log --grep="<pattern>"</code>	Search for commits with a commit message that matches <code><pattern></code> .
<code>git log <since>..<until></code>	Show commits that occur between <code><since></code> and <code><until></code> . Args can be a commit ID, branch name, HEAD, or any other kind of revision reference.
<code>git log -- <file></code>	Only display commits that have the specified file.
<code>git log --graph --decorate</code>	<code>--graph</code> flag draws a text based graph of commits on left side of commit msgs. <code>--decorate</code> adds names of branches or tags of commits shown.

git diff	
<code>git diff HEAD</code>	Show difference between working directory and last commit.
<code>git diff --cached</code>	Show difference between staged changes and last commit

git reset	
<code>git reset</code>	Reset staging area to match most recent commit, but leave the working directory unchanged.
<code>git reset --hard</code>	Reset staging area and working directory to match most recent commit and overwrites all changes in the working directory.
<code>git reset <commit></code>	Move the current branch tip backward to <code><commit></code> , reset the staging area to match, but leave the working directory alone.
<code>git reset --hard <commit></code>	Same as previous, but resets both the staging area & working directory to match. Deletes uncommitted changes, and all commits after <commit> .

git rebase	
<code>git rebase -i <base></code>	Interactively rebase current branch onto <code><base></code> . Launches editor to enter commands for how each commit will be transferred to the new base.

git pull	
<code>git pull --rebase <remote></code>	Fetch the remote's copy of current branch and rebases it into the local copy. Uses git rebase instead of merge to integrate the branches.

git push	
<code>git push <remote> --force</code>	Forces the <code>git push</code> even if it results in a non-fast-forward merge. Do not use the <code>--force</code> flag unless you're absolutely sure you know what you're doing.
<code>git push <remote> --all</code>	Push all of your local branches to the specified remote.
<code>git push <remote> --tags</code>	Tags aren't automatically pushed when you push a branch or use the <code>--all</code> flag. The <code>--tags</code> flag sends all of your local tags to the remote repo.

Hands-on training

Plan hands on

- We practice the workflow
- I will do some bugfix merges into the master branch
- You will have to keep your feature branch up to date with the upstream repo
- When ready you push to your remote fork branch, then do a merge request
- We try various variations of this
- Using tags and issues for each development for instance