NETWAYS®

GitLab

Version Control System & Continuous Integration

Version: 2.2.1

Training Hints

- Ask questions! Dumb questions do not exist.
- Add your own notes! Having learnt something is reflected best with your own additions.
- Join the conversation! Many of these exercises require real world scenarios and your feedback and questions.
- Work together! Be part of the training team and find solutions together.

1 Git Introduction

NETWAYS TRAINING

Version Control

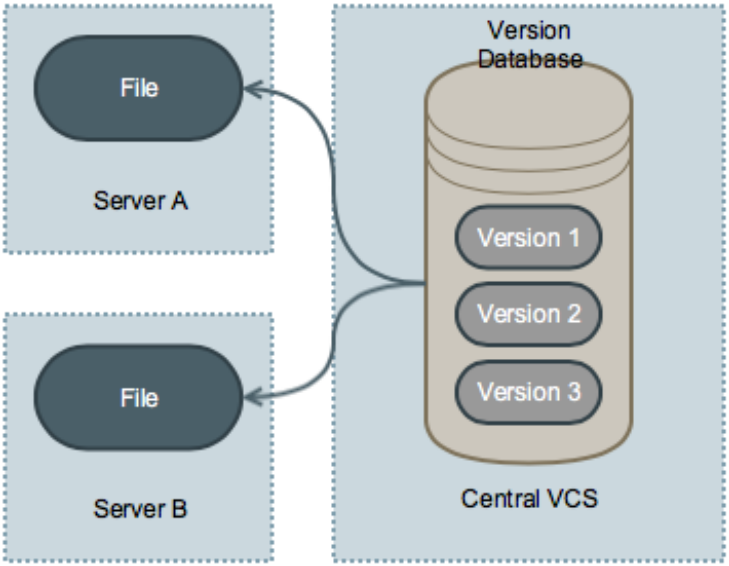
- Version Control System (VCS)
- Record changes of file(s)
- Revert changes
- Compare changes over time
- Who, when, what

The most simple version control system is to copy directories and add a suffix like the last date it worked. Though this might cause trouble when you're in the wrong directory. Afterall it pollutes your filesystem structure over time.

Long time ago developers invented version control systems which store the file revisions in a database.

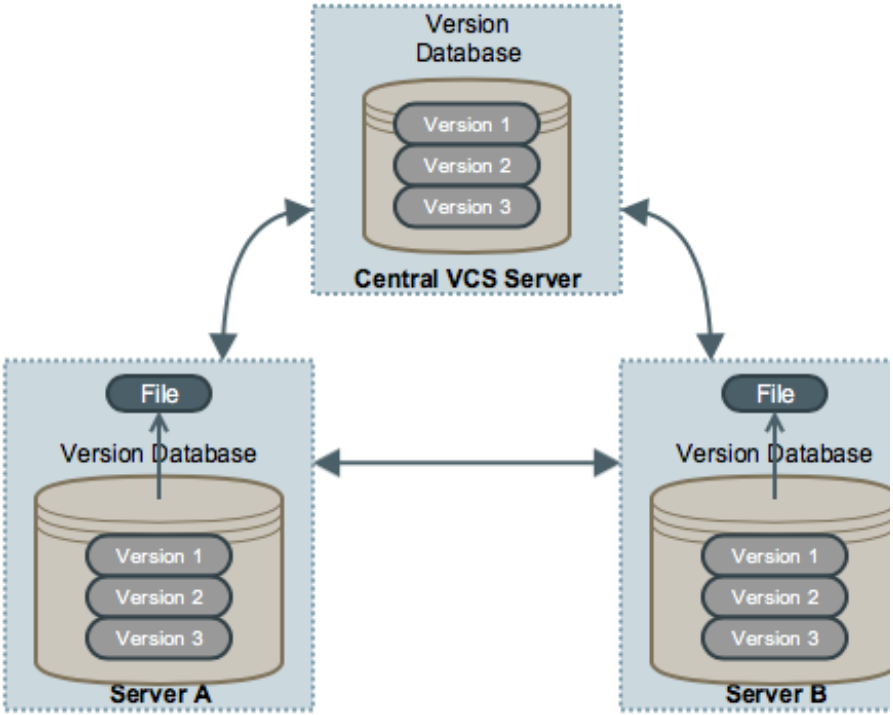
Centralized VCS

- Multiple computers required file revisions
- CVS, Subversion
- What happens if the server is down?



Distributed VCS

- Clients mirror the repository
- Git, Mercurial, Bazaar, etc.
- Server dies, client continues



History of Git

- Linux kernel development
  - Patches and archives
  - Proprietary software `BitKeeper`

- Controversy in 2005, no more free to use

- Kernel developers invented `Git`

NETWAYS TRAINING

Design Goals for Git

- Speed
- Simple design
- Non-linear development (many branches)
- Fully distributed
- Handle large projects in speed and size

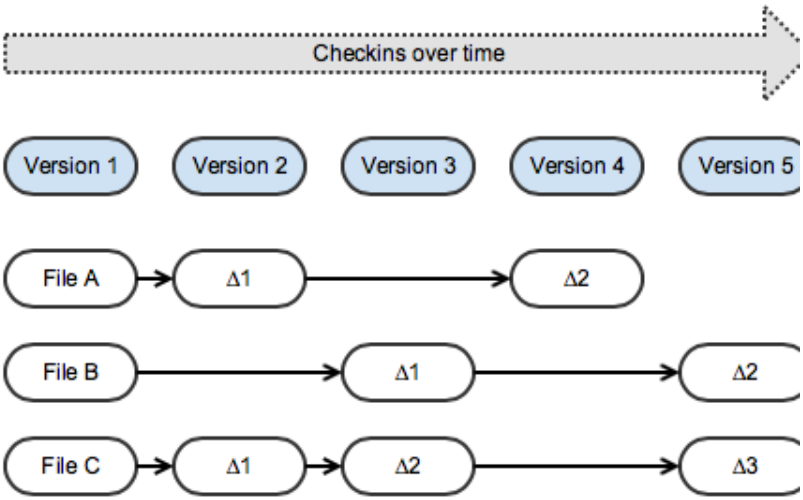
NETWAYS TRAINING

Git Basics

- Learn Git, forget about other VCS systems (SVN, etc.)
- Git interface is similar to existing VCS
- ... but Git behaves differently

Snapshots and Differences

- File changes and deltas over time
  - Git Commit, take snapshot, store reference to that snapshot
  - Set of snapshots of a mini-filesystem
  - No change - link to the previous identical stored file



NETWAYS TRAINING

Work locally

- No network latency involved as with other VCS systems
- Local repository clone, fast operations
  - Browse the history
  - Show differences between specific branches
- Work offline and push changes later

Integrity

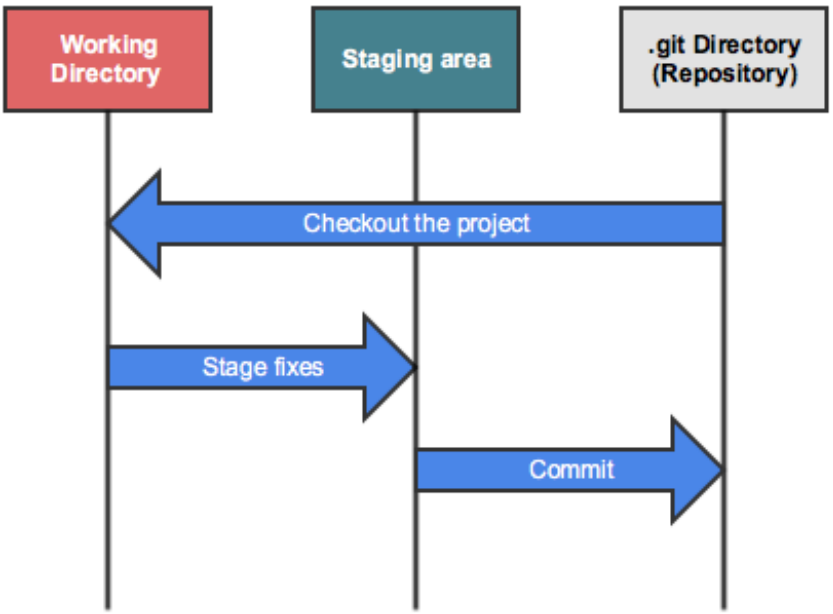
- Everything has a check sum (SHA-1)
- No changes possible without Git knowing about them
- Checksums are literally used everywhere
- Revert changes and even restore deleted files

Example:

```
7f0b824ba55e1fd4ffc5c461df0a0f48a94195cc
```

The three states

- Working directory ("modified")
- Staging area ("staged")
- Git directory ("committed")



`Modified` means that you have changed the file but have not committed it to your Git database yet.

`Staged` means that you have marked a modified or added file in its current version to go into your next commit snapshot.

`Committed` means that the data is safely stored in your local database.

The `working directory` is a single checkout of one version of the project. These files are pulled out of the compressed database in the Git directory and placed on disk for you to use or modify.

The `staging area` is a file, generally located in your Git directory, that stores information about what will go into your next commit. It is sometimes referred to as the "index", but it’s also common to refer to it as the staging area.

The `Git directory` is where Git stores the metadata and object database for your project. This is the most important part of Git, and it is what is copied when you clone a repository from another computer.

NETWAYS TRAINING

Basic Git Workflow

- Modify files in `working directory`
- Stage the files which add snapshots to the `staging area` ("git add")
- Commit ("git commit")
  - Takes files in `staging area`
  - Stores snapshot permanently in `.git directory`

- If a particular version of a file is in the `.git directory`, it's considered `committed`.
- `Staged` means that the file has been modified and it was added to the staging area
- `Modified` means that the file was changed since it was checked out but has not been staged yet.

Git CLI

- Work on the CLI
- GUIs implement partial feature sets of the CLI tool
- Shell sub commands
- Bash-completion

NETWAYS TRAINING

Git Installation

- Available as package
- macOS, Windows installers
- Bash Integration
  - Completion
  - Show status and branch in the terminal

```
michi@mbmif ~ $ cd coding/icinga/icinga2
michi@mbmif ~/coding/icinga/icinga2 (feature/api-prett
```

Lab 1.1: Install Git

- Objective:
  - Install the `git` package

- Steps:
  - Use the package manager to install the git package

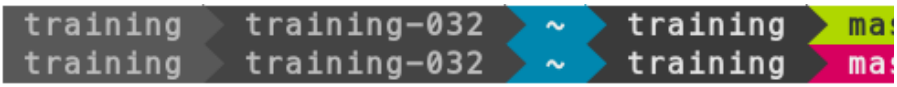
Lab 1.2: Install Git Bash Completion

- Objective:
  - Install the `bash-completion` package
  - Modify your prompt to highlight the git state

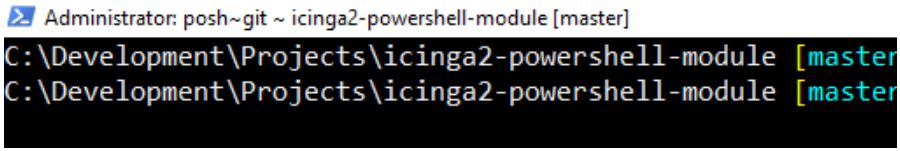
- Steps:
  - Use the package manager to install the `bash-completion` package
  - Fetch the `git-prompt.sh` script from https://github.com/git/git - `contrib/completion/git-prompt.sh`
  - Customize your prompt in your `$HOME/.bashrc` file

More Git Shell Integrations

- Powerline Shell



- Windows Powershell: git-posh



Powerline Shell Integration

The Powerline shell integration can be found here: https://github.com/b-ryan/powerline-shell

```
# yum -y install epel-release
# yum -y install python-pip
# pip install powerline-shell

@@@ Sh
$ vim $HOME/.bashrc
```

```
function _update_ps1() {
    PS1=$(powerline-shell $?)
}

if [[ $TERM != linux && ! $PROMPT_COMMAND =~ _update_p
s1 ]]; then
    PROMPT_COMMAND="_update_ps1; $PROMPT_COMMAND"
fi
```

Powerline needs an additional font which can be downloaded from here: https://github.com/powerline/fonts/blob/master/Meslo%20Slashed/Meslo%20LG%20M%20Regular%20for%20Powerline.ttf

Choose `Raw` and install it into your system.

Modify your terminal profile settings and choose `Meslo LG` as font.

Windows Powershell Integration

The Windows Powershell integration requires NuGet. Open a new Powershell prompt and enter:

```
Install-Module git-posh
Add-PoshGitToProfile
```

2 Git Configuration

Configuration Overview

- Global configuration for the user
  - `$HOME/.gitconfig`
- Local to the repository
  - `training/.git/config`

Configuration

- CLI command support
- $HOME/.gitconfig

Example:

```
$ git config --global color.ui auto

$ cat $HOME/.gitconfig
[color]
ui = auto
```

More information can be found in the documentation at https://git-scm.com/book/tr/v2/Customizing-Git-Git-Configuration

NETWAYS TRAINING

Configuration Sections

- Commit author (`user`)
- Aliases (`alias`)
- Colors for diff and verbose commit (`color`)
- Core functionality (`core`)

Example:

```
[user]
        email = michael.friedrich@netways.de
        name = Michael Friedrich
```

Lab 2.1: Configure your username and email address

- Objective:
  - Configure your username and email address using Git CLI commands

- Steps:
  - Use `git config --global user.name "Your Name"`
  - Use `git config --global user.email "name@domain.com"`

- Next steps:
  - Verify the changes with `git config --global --list`

3 Git Basics

Git Command Overview

- Start a working area (clone, init)
- Work on current changes (add, reset)
- Examine the history and state (status, log)
- Grow, mark and tweak the history (branch, checkout, commit, merge, rebase)
- Collaborate (fetch, pull, push)

Example from Git CLI command:

```
start a working area (see also: git help tutorial)
   clone      Clone a repository into a new directory
   init       Create an empty Git repository or reinitialize an exis
ting one

work on the current change (see also: git help everyday)
   add        Add file contents to the index
   mv         Move or rename a file, a directory, or a symlink
   reset      Reset current HEAD to the specified state
   rm         Remove files from the working tree and from the in
dex

examine the history and state (see also: git help revisions)
```

```
   bisect     Use binary search to find the commit that introduc
ed a bug
   grep       Print lines matching a pattern
   log        Show commit logs
   show       Show various types of objects
   status     Show the working tree status

grow, mark and tweak your common history
   branch     List, create, or delete branches
   checkout   Switch branches or restore working tree files
   commit     Record changes to the repository
   diff       Show changes between commits, commit and worki
ng tree, etc
   merge      Join two or more development histories together
   rebase     Forward-port local commits to the updated upstre
am head
   tag        Create, list, delete or verify a tag object signed with
 GPG

collaborate (see also: git help workflows)
   fetch      Download objects and refs from another repository
   pull       Fetch from and integrate with another repository or
a local branch
   push       Update remote refs along with associated objects
```

Start a project

- `git clone`
  - Clone a copy of existing remote repository.
  - Can be newly created in GitLab/GitHub and empty.

- `git init`
  - Initialize a local empty Git repository.

`git clone` clones a copy of an existing remote Git repository.

`git init` initializes an empty directory as local Git repository.

When you clone a repository, Git automatically adds a shortcut called `origin` that points back to the "parent" repository, under the assumption that you'll want to interact with it further on down the road. This is called an `origin`.

NETWAYS TRAINING

Lab 3.1: Clone an existing Git repository

- Objective:
  - Clone an existing Git repository

- Steps:
  - Navigate to https://github.com/icinga/icinga2
  - Copy the clone URL
  - Navigate into your home directory
  - Use `git clone` to clone the remote Git repository

Lab 3.2: Initialize a local Git repository

- Objective:
  - Initialize git repository

- Steps:
  - Create a new directory called `training` in your home directory
  - Change into it
  - Run `git init`

We will be working inside the `training` directory throughout the training unless noted otherwise.

NETWAYS TRAINING

Add current changes

- `git add`
  - Add (modified) file(s) from the working directory into the staging index.
- `git mv`
  - Rename file(s) tracked by Git.

`git add` will add the file(s) and their content to the staging index waiting for the commit.

`git mv` renames an existing file tracking the change for the commit. If you manually move the file, you will need to rm and add it again.

Remove changes

- `git reset`
  - Reset files added to the staging index.
  - `--soft` keeps the changes (default), `--hard` removes them indefinitely.

- `git rm`
  - Remove the file(s) from working tree and Git repository.
  - Note that file(s) will be visible in Git history, and can be restored from it.

`git reset` resets files added to the staging index.

`git rm` removes the file from the working tree and also from the git index.

Lab 3.3: Add a new README.md file

- Objective:
  - Add a new README.md file

- Steps:
  - Change into `$HOME/training`
  - Create README.md and add `# Git Training Notes` as first line
  - Use `git add` to add README.md to the current change index

- Next steps:
  - Verify the change with `git status`

Best practice is to have a README.md file written in Markdown in every project. This gets rendered by GitHub/GitLab in readable HTML.

During this training we will learn many new things. Keep notes in the `README.md` file.

Lab 3.4: Reset File from Staging Index

- Objective:
  - Reset file from staging index

- Steps:
  - Change into `$HOME/training`
  - Remove the previously added `README.md` file from the staging index with `git reset --soft README.md`
  - Verify it with `git status` and explain what happened.
  - Re-add the `README.md` and examine again with `git status`.

NETWAYS TRAINING

Examine the current state

- `git status`
  - Show current working tree status.
  - `Untracked` files and added to staging area.
  - Modifications
- `git diff`
  - Compare changes between working tree and latest commit.

Later we will learn how to compare specific commits and branches too.

`git status` shows the current working tree status. Untracked files and changes (not) staged for commit.

`git diff` shows changes between the current working tree and the last commit. You can also compare specific commits.

Lab 3.5: Examine current changes

- Objective:
  - Examine current changes

- Steps:
  - Change into `$HOME/training`
  - Edit README.md and add notes
  - Use `git status` to see unstaged changes
  - Add the changed files to the staging area
  - Use `git status` again

Lab 3.6: Use Git Diff

- Objective:
  - Use `git diff`

- Steps:
  - Change into `$HOME/training`
  - Edit README.md
  - Use `git diff` to compare unstaged changes
  - Add the changed file to the staging area
  - Use `git diff` again
  - Compare the staging area with the latest commit in `.git repository`

Exclude files with .gitignore

- Build directories from source code compilation (e.g. `debug`, `release`)
- Files generated at runtime (e.g. test results or stats)
- User specific IDE settings
- Local Vagrant boxes and other temporary files

You can use wildcard pattern matches for files. If you'll end the string with a '/' git will only ignore directories instead of both directories and files matching the pattern.

If you want to ignore a file globally in your repository you can preceed the path with `**/` which means `any directory`.

Example for ignoring a debug file anywhere in your repository:

```
**/debug
```

If you prefer to keep `.gitignore` files in specific directories and not a global file, this will also

work.

Lab 3.7: Add .gitignore file and exclude files/directories

- Objective:
  - Add .gitignore file and exclude files/directories

- Steps:
  - Change into `$HOME/training`
  - Create a file `generated.tmp`
  - Create a directory `debug` with the file `.timestamp`
  - Examine the state with `git status`
  - Exclude them in a .gitignore file
  - Examine the state with `git status`

4 Git Commits

NETWAYS TRAINING

Git Commits

- View as history log on production changes
- Keep the subject short, simple and explaining
- Make it easier for others understanding the changes
- Add detailed explanations on larger changes
- Add references to existing commits and/or ticket ids for further details

Work on Git History

- `git commit`
  - Selected changes from the staging index
  - All changes (`-a`) not necessarily added to the staging index

- Verbose mode shows changes compared to latest commit (`-v`)

- Uses the configured editor (vim, nano, etc.)
  - `-m` allows for a short commit message without editor view

Commits use the `user.name` and `user.email` settings to qualify you as the author of this change. In addition to that the date and time is stored.

`git commit` collects and records all changes stages for commit. It uses the configured user name and email address as commit author. This command opens the configured editor requiring you to add a commit message.

Lab 4.1: Commit Changes

- Objective:
  - Modify files and commit your changes

- Steps:
  - Change into `$HOME/training`
  - Modify the `README.md` file and add more docs
  - Add the change to the changing index
  - Commit the change to your Git history with `git commit -v README.md`

- Next steps:
  - Use `git log` to verify the history

Good Commits

- Selectively add changes for commit (do not commit everything)
  - `git add` for all changes in a file
  - `git add -p` for interactive selection of changes in a file

- `git commit <file1> <file2>`

- Enable the verbose mode `-v` to show the differences below the editor
  - Write a short summary based on the visible changes

Commit Message Overview

- Pick a short telling subject (max. 80-120 characters)
- Add a new line
- Add a body text explaining the issue (max. 80-120 characters in a line)
- Optional: Add external reference markers, e.g. for ticket systems

Example:

```
A short subject for the commit line
<newline>
Some body text explaining the issue.
80-120 characters max width.
<newline>
refs #<ticketid>
```

Examine the Git history

- `git log`
  - Show commit history of the current branch.
  - Supports `-10` notation to show a limited number of commits.

`git log` shows the commit history of the current branch.

NETWAYS TRAINING

Examine the Git history

- `git show`
  - Print commit details.
  - If the commit id is omitted, the last commit is printed.
  - Supports `-10` notation to show a limited number of commits.

- `git diff`
  - Show changes between working tree and last commit.
  - Supports source and target parameters.
  - Can be used to compare 2 commit ids, branches, etc.

`git show` will print the last commit details. If you want to print a specific commit id, add it afterwards.

`git diff` shows changes between the current working tree and the last commit. You can also compare specific commits.

Lab 4.2: Examine the Commit History

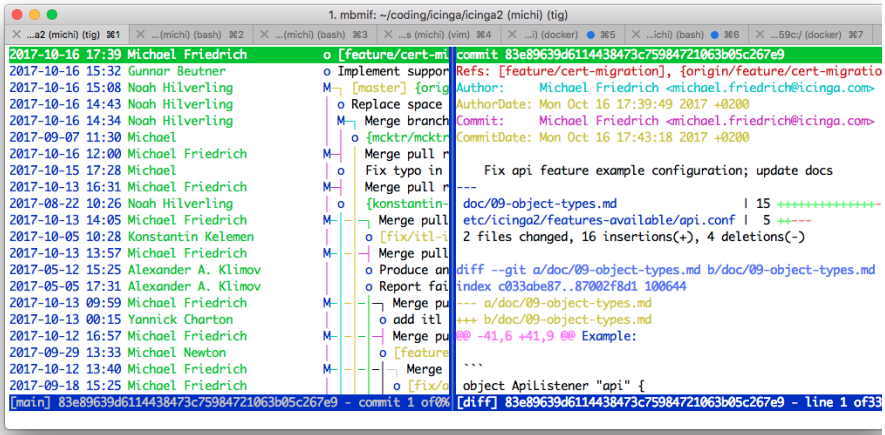
- Objective:
  - Examine the commit history

- Steps:
  - Change into `$HOME/training`
  - Add and commit remaining changes e.g. `.gitignore`
  - Use `git log` to print the current history
  - Use `git show` to show specific commits (defaults to the latest)
  - Use `git diff` to compare changes between specific revisions

NETWAYS TRAINING

Advanced history with tig

- `tig` helps visualize history and branches
- Inspect specific commits while scrolling



Lab 4.3: Learn more about tig

- Objective:
  - Install and use tig

- Steps:
  - Install the `tig` package
  - Run tig in `$HOME/training`
  - Clone a different repository and run tig there e.g. `$HOME/icinga2`

- Next steps:
  - Select a line and press `Enter`
  - `q` quits the detail view and the application

Amend changes to commits

- Change the commit message, e.g. typos or missing changes broke the build
- Amend changes from staging
- Helps if new files were added but not committed
- `git commit --amend` changes the latest commit
  - Amending commits in Git history is possible, explained later with `git rebase`.

If you amend changes to a specific commit, a new unique commit id is generated. This changes the Git history and we will learn later how to resolve possible problems in collaboration with others.

Lab 4.4: Amend changes to commits

- Objective:
  - Use git amend

- Steps:
  - Change into `$HOME/training`
  - Modify `README.md` and add docs about amend
  - Add `README.md` to the staging index and commit the change
  - Edit `README.md` again and add it to staging
  - Use `git commit --amend README.md` and explain what happens

- Bonus:
  - Adopt the commit message using `git commit --amend`

5 Git Branching

Working with Git branches

A git branch creates a new history line starting from the current git commit.

Branches are useful to develop features/fixes in their isolated environment.

- Master branch
- Develop a new feature in a dedicated branch
- Put fixes into the master branch (production)
- Continue to work on the feature

Git Branch CLI commands

- `git branch`
  - List, create and delete branches.
- `git checkout`
  - Switch branches.
  - Restore the working tree.

`git branch` allows you to list, create and delete branches.

`git checkout` will switch between branches, or restore your current working tree.

Lab 5.1: Show the current branch

- Objective:
  - Show the current branch

- Steps:
  - Change into `$HOME/training`
  - Use `git branch` to highlight the current branch

Lab 5.2: Create and checkout a new branch

- Objective:
  - Create and checkout a new branch

- Steps:
  - Change into `$HOME/training`
  - Create a new branch `feature/docs` based off `master` with `git branch feature/docs master`
  - List the branches with `git branch`
  - Checkout the new branch with `git checkout feature/docs`

- Bonus:
  - Explain `git checkout -b feature/docs2`

Lab 5.3: Delete the branch

- Objective:
  - Delete the previously created branch

- Steps:
  - Change into `$HOME/training`
  - Switch to the master branch
  - Use `git branch -d` to delete the selected branch

- Bonus:
  - Try to delete the branch you are currently on

HEAD and "smart pointers"

A git commit is unique and identified by its SHA1 checksum.

`HEAD` is a pointer to the latest commit in the current branch.

`HEAD^` identifies the second latest commit.

`HEAD~9` points to the tenth latest commit. Counting starts at zero.

This can be used for `git show`. More advanced techniques will be discussed later.

Lab 5.4: Show the second commit

- Objective:
  - Use `HEAD` and only show the second latest commit.

- Steps:
  - Change into `$HOME/training`
  - Combine `git show` with `HEAD^` or `HEAD~1`

Branches and "smart pointers"

`feature/docs` as branch name also points to the latest commit.

You don't need to change branches to

- show different branch histories
- show specific commits where you don't know the commit id
- compare branches and committed code

Example:

```
$ git show feature/docs
commit b825ff86e4022a8fbcf52cb5a1d9a1984bd2a310 (feature/doc
```

Lab 5.5: Show history of different branch

- Objective:
  - Use `git log` from the master branch on another branch
- Steps:
  - Create a new branch aside from master, if not existing: `git checkout -b feature/docs`
  - Switch to the master branch
  - Use `git log feature/docs`
- Bonus:
  - Modify and commit changes
  - Diff current HEAD against `feature/docs` branch

6 Git Server

Introduction

- Central storage for repositories
- Collaboration between teams
- User based access control
- Trigger events (e.g. for CI)

There is a variety of Git server tools, web interfaces and addons out there.

- GitLab
- gitosis
- gitolite

In case you don't want to host your own Git server, there are open source and enterprise hosting options available.

NETWAYS also provides GitLab hosting services:

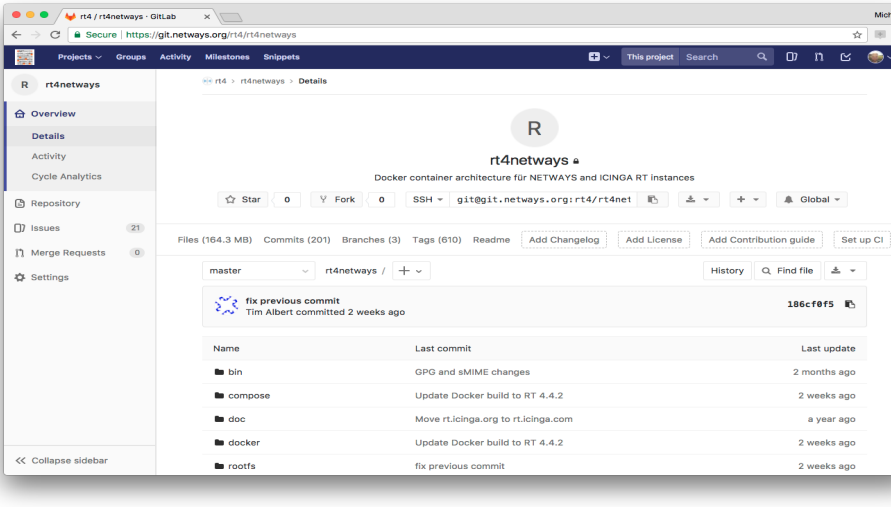
- https://www.netways.de/managed_hosting/gitlab_ce_hostir
- https://nws.netways.de/products/gitlab-ce

Git Server Overview

- Git server daemon
- Web interfaces
- Entire collaboration suites
  - GitHub
  - GitLab
  - Bitbucket

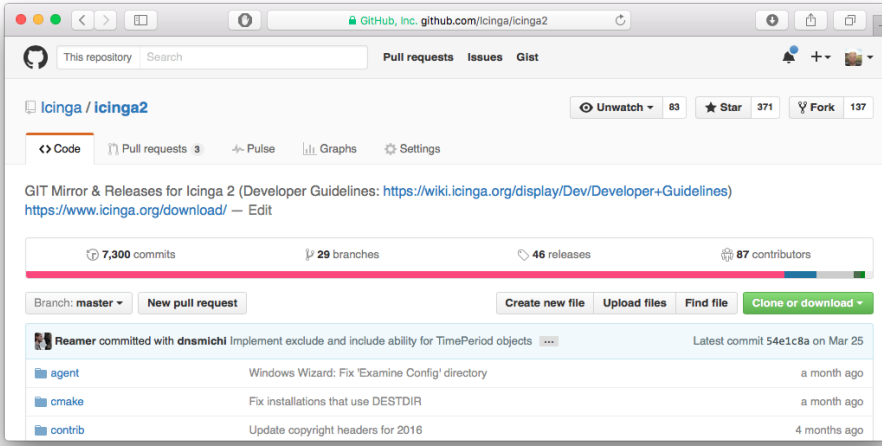
NETWAYS TRAINING

GitLab



NETWAYS TRAINING

GitHub



Git Server Protocol

- Read/write access via SSH
  - `git@github.com:username/repo.git`
- HTTPS protocol (write access via oauth tokens)
  - `https://my-gitlab.nws.netways.de/username/repo.git`
- Git protocol
  - `git://domain.com/repo.git`
- Local protocol
  - `file:///opt/git/repo.git`

GitLab Introduction

GitLab is available as self-hosted or cloud based repository management system.

- Git repositories
- User and group management and fine granular permissions
- Issue tracking and project management (dashboards, etc.)
- Merge, review, report
- Continuous integration/deployment (CI/CD)

Hosted:
https://www.netways.de/managed_hosting/gitlab_ce_h

NWS: https://nws.netways.de/products/gitlab-ce

GitLab Editions

- Community Edition (CE), part of this training
- Enterprise Edition (EE), additional features
  - Multiple LDAP servers
  - LDAP group import
  - Git Hooks (not web hooks)
  - Search backend with Elasticsearch

Overview: https://about.gitlab.com/features/
Comparison: https://about.gitlab.com/images/feature_page/gitlab-features.pdf

The source code is publicly available for both editions. You'll need a valid license for running EE in production.

GitLab Components

- Ruby on Rails application (unicorn, sidekiq)
- Nginx webserver
- PostgreSQL database backend
- Redis cache
- NodeJS for Javascript rendering
- Golang for background daemons

It is recommended to use the Omnibus installation package or use a managed cloud hosting service.

Omnibus packages:
https://about.gitlab.com/installation/?version=ce

More details on the manual installation instructions can be found in the official documentation:
https://docs.gitlab.com/ce/install/installation.ht

ml

Git Server Installation

This training focuses on a pre-installed GitLab instance.

Each user gets access to NWS - https://nws.netways.de and a GitLab CE app.

GitLab supports working with

- SSH
- HTTPS

Manual installation instructions are provided in the handout.

Lab 6.1: Create GitLab app in NWS

- Objective:
  - Create a new GitLab app in NWS

- Steps:
  - Navigate to https://nws.netways.de and register a trial account if not existing
  - Choose Apps > GitLab CE > Basic
  - Deploy the app
  - Choose Live View and set a secure password for the `root` user.
  - Login

Connect Local Repository to Remote Server

- Local standalone repository
- Connect to remote server
- Clone, Pull, Fetch, Push via SSH/HTTPS

You can also start fresh without any local repository and clone that from remote.

For training purposes we've started to work offline in `$HOME/training`. Now we want to publish the local commits to a newly created Git repository in GitLab.

Requirements

- SSH or HTTPS auth
  - NWS apps come pre-defined with HTTPS clone/fetch only.
- New GitLab repository for this user
- Configure local repository for the remote server

SSH Keys

Generate a new SSH key pair on your client.

```
ssh-keygen
```

Copy the public key into your GitLab settings.

```
cat $HOME/.ssh/id_rsa.pub
```

User > Settings > SSH Keys

Lab 6.2: Create GitLab Project

- Objective:
  - Create a new GitLab project for the current user

- Steps:
  - Click the `+` icon next to the search field
  - Choose `New Project`
  - Add the name `training`
  - Leave it as `Private`
  - Create the project

- Note:
  - Learn about the project view and the HTTPS clone URL

Lab 6.3: Configure Client Credentials Helper

- Objective:
  - Configure client credentials helper

- Steps:
  - Run `git config --global --get credential.helper`
  - Verify that it is set to `/usr/libexec/git-core/git-credential-gnome-keyring` on CentOS 7

More details can be found in your NWS GitLab app in the FAQ section on top.

Lab 6.4: Add the repository as remote origin

- Objective:
  - Add the GitLab project as remote origin

- Steps
  - Open the project in GitLab and extract the `HTTPS` clone URL
  - Navigate into your local repository in `$HOME/training`
  - Use `git remote add origin <remoteurl>`
  - Push your local history with `--set-upstream` (short: `-u`)

- Bonus
  - Set default push method to `simple`

Lab 6.5: Explore Project History

- Objective:
  - Learn more about GitLab and the project's history
  - Compare the local history to the remote project's history

- Steps:
  - Click on `History` in the project view and examine the Git commits
  - Run `git log` or `tig` on your shell and compare them to GitLab

7 Collaboration with others

Collaboration

- Work locally
- Push to remote repository
- Share your work with others
- Review, discuss, collaborate
- Change, adopt, release

Collaboration with others

- `git fetch`
  - Update the remote branch reference pointers to the latest commit and cache it locally.
  - Does not pull in any remote commit history.

- `git pull`
  - Fetch and update the local history from remote repository (implicit fetch).
  - This pulls in source code changes and commits.

`git fetch` downloads objects and references from another remote repository.

`git pull` invokes a fetch and updates the local history with commits from the remote repository.

Collaboration with others

- `git push`
  - Update remote references and push local history to remote repository.
  - This pushes source code changes and commits.
  - Halts if the remote history diverged from your local history.

- `git remote`
  - Configure/list remote repository URLs (default `origin`).

`git push` updates remote references and pushes your local commit history to the remote repository.

`git remote` allows you to configure and list the remote repository. By default this is called `origin`.

Lab 7.1: Learn more about git push

- Objective:
  - Learn more about git push

- Steps:
  - Change into `$HOME/training`
  - Edit `README.md` and add a note on `git push`
  - Add and commit the changes
  - Push the changes

Lab 7.2: Learn more about git fetch and git pull

- Objective:
  - Learn more about git fetch and git pull
- Steps:
  - Reset your local commit history by one with `git reset --hard HEAD^`
  - Fetch and pull changes from remote
  - Explain the difference
- Bonus:
  - Repeat push and pull multiple times

Git Tags

- Tag specific points in history
- Add, list, delete
- Push tags to remote repository
- Checkout branches based on tags
- Release software versions based on tags (e.g. v2.9.0)

Example for checking out a tag into a new branch:

```
$ git checkout -b version01 v0.1
$ git branch
```

Lab 7.3: Add Git Tag

- Objective:
  - Add git tag

- Steps:
  - Use `git tag` and add the `v0.1` tag
  - Verify the added tag with `git tag -l`

- Bonus:
  - Add a tag description with `-m`
  - Push tags to remote origin with `git push --tags`

Advanced Git Commands

- `git stash`
  - Put current changes on a temporary stack.
- `git cherry-pick`
  - Collect specific commit into your working tree.

`git cherry-pick` collects a specific commit into your working tree.

`git stash` allows you put your current changes on a temporary stack (`stash`). This comes in handy when you want to change branches with a different history where your uncommitted changes will not apply. Use `git stash pop` to fetch the changes again. You can stash multiple uncommitted stages, `git stash list` will list them.

Lab 7.4: Learn more about git stash

- Objective:
  - Learn more about git stash

- Steps:
  - Change into `$HOME/training`
  - Edit `README.md`
  - Examine the status with git status
  - Stash your current changes to the working directory
  - Run git status again
  - Examine the stash with `git stash list` and `git stash show -p`
  - Fetch the previously stashed changes with `git stash pop`

Reset Git Commits

- `git reset`
  - Remove the current commit(s).
  - `--soft` adds changes to the staging index.
  - `--hard` drops them indefinitely.

Try it out with the trainer.

```
$ git reset --soft HEAD^

$ git reset --hard HEAD^
```

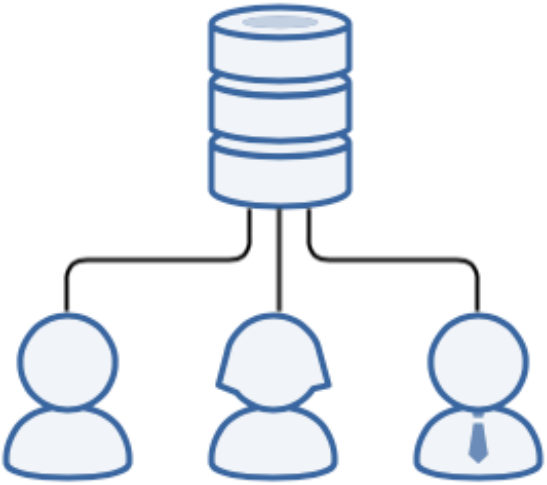
8 Git Workflows

Git Workflows

- Single user environments
- Multiple users working together
- Branching models
- Patch and pull request workflow integration

Centralized Workflow

- Multiple users
- Each user has a local copy
- Central `master` branch



Compared to other VCS systems, Git provides the advantage of also storing a local copy of the repository and branches.

That way a developer can work isolated on changes on its own while other developers can do the same. These isolated environments ensure that each developers works independantly from other changes in the project.

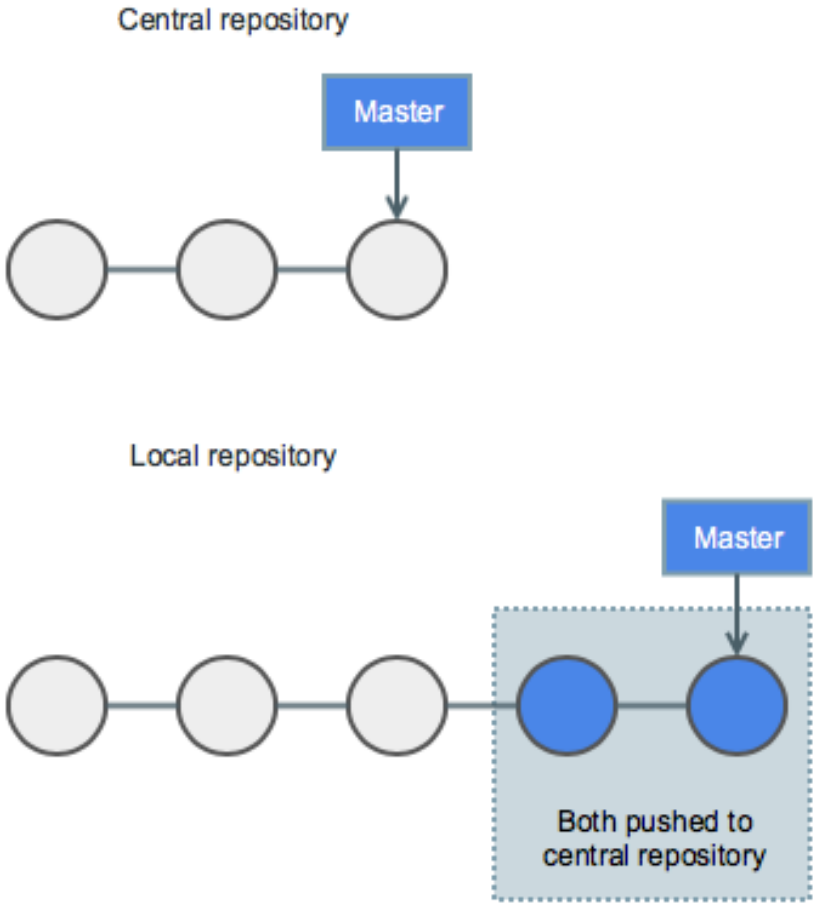
In addition to that the Git branching model provides a fail-safe mechanism for integrating and sharing code changes between repositories.

Centralized Workflow - Developers

- Developers clone the central repository
- Work in local copies
- New commits are stored locally
- Importing the remote repository's changes is optional

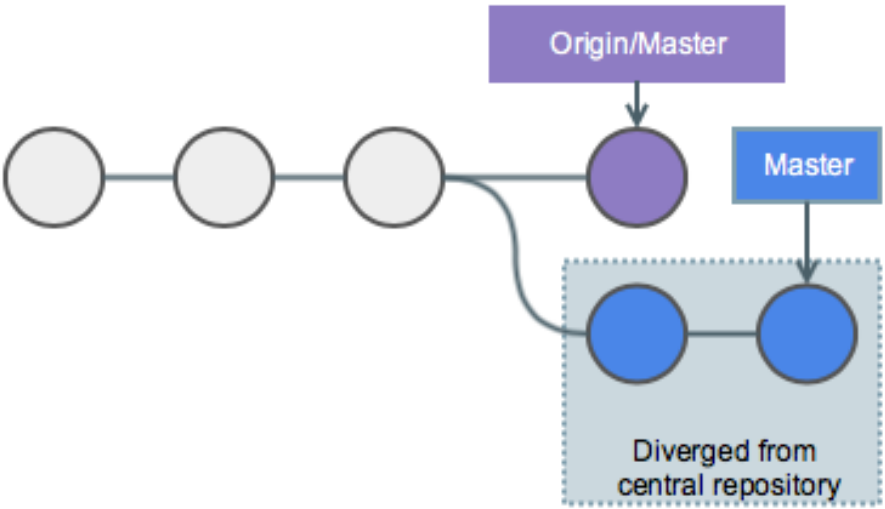
Centralized Workflow - Publish Changes

- Developers push their local master branch
- Stored in central repository
- Adds all local commits that are not in the central master branch


Central repository
Master
Local repository
Master
Both pushed to
central repository

Centralized Workflow - Managing Conflicts

- Central repository's commit history is important
- If local commit history diverges, pushing changes is denied



Lab 8.1: Collaborate in a central repository

- Objective:
  - Clone the training repository twice and add diverging commits

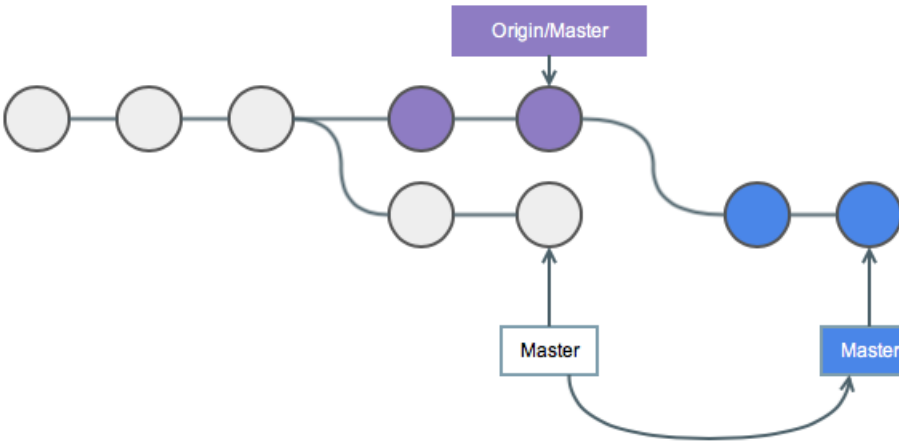
- Steps:
  - Clone the `training.git` repository into `$HOME/training1` and `$HOME/training2`

- Steps for both:
  - Change into each directory
  - Add/modify a file and commit the change
  - Push your change to the remote repository

- Explain the error message after the second push

Centralized Workflow - Managing Conflicts Solution

- Fetch the remote history
- Rebase local changes on top of it
- Linear history



Lab 8.2: Resolve conflicts in a central repository

- Objective:
  - Rebase your local history with the remote repository

- Steps:
  - Reset the local history by 2 commits with `git reset --hard HEAD~2`
  - Update and commit README.md
  - Fetch remote and compare with `git diff origin/master`
  - Rebase with `git rebase origin/master`
  - Resolve possible merge conflicts, add them and continue with `git rebase --continue`, push rebased history

Hint: `>>>` marks conflicts and show the differences on merge.

Instead of `git fetch` and `git rebase` you can also use the `git pull` command with the additional `--rebase` flag. This helps if you are familiar with

SVN and `svn update`.

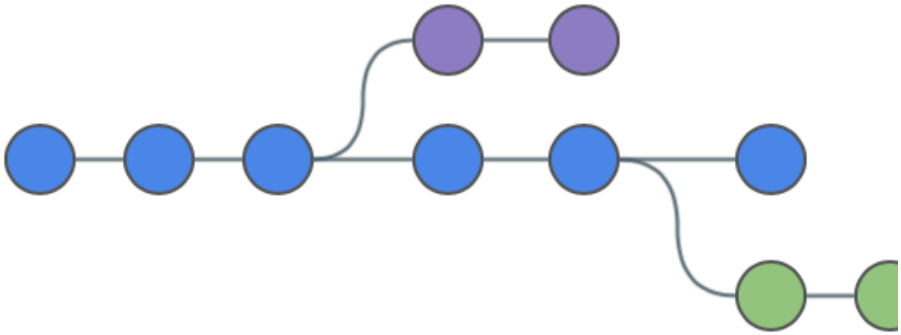
If you forget the `--rebase` flag it will still work but generate merge commits. This will merge your commits in historical order but not rebase them on top of the existing history.

For using a centralized workflow it is better to use `rebase` instead of generating a merge commit.

NETWAYS TRAINING

Feature Branch Workflow

- Feature development happens in named branches
- Does not interfere with the main codebase
- Master branch does not contain broken code
- Feature branches can be rebased against stable master branch on demand



Feature Branch Workflow - How it works

- Central repository
- Create a new branch for each feature
- Descriptive branch names, e.g. `feature/docs-workflows`
- Changes in a feature branch similar to centralized workflow
- Push feature branches to central repository for collaboration with other developers

Lab 8.3: Use Feature Branches

- Objective:
  - Create a new feature branch `feature/docs-workflows`

- Steps:
  - Change into `$HOME/training`
  - Use `git checkout -b feature/docs-workflows` to create a new feature branch based on the master
  - Add and commit changes
  - Push the branch to your central repository

Lab 8.4: Merge Feature Branches

- Objective:
  - Update `master` branch and merge feature branch `feature/docs-workflows`

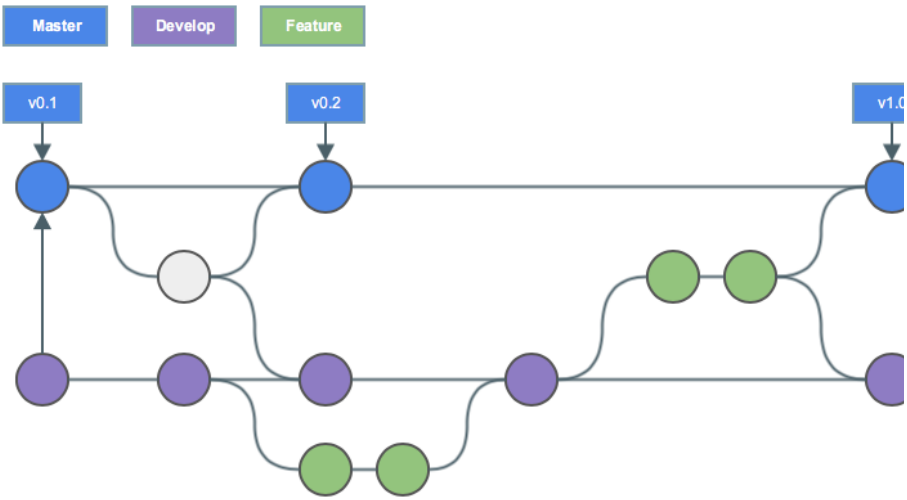
- Steps:
  - Checkout the feature branch `feature/docs-workflows`
  - Edit `README.md`, add and commit the changes
  - Diff the feature branch to the current master with `git diff master`
  - Checkout the `master` branch, merge the feature branch as non-fast-forward with `--no-ff`
  - Show the history tree with `tig` or inside GitLab and explain why the forced merge commit with `--no-ff` is important

Gitflow Workflow

- Strict branching model for project releases
- Based on Feature Branch Workflow
- Assigns roles to different branches
- Defines interaction between branches for releases
  - Prepare
  - Maintain
  - Record

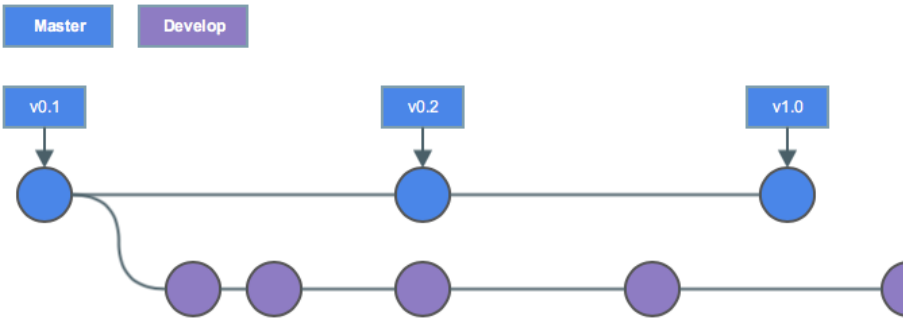
NETWAYS TRAINING

Gitflow Workflow



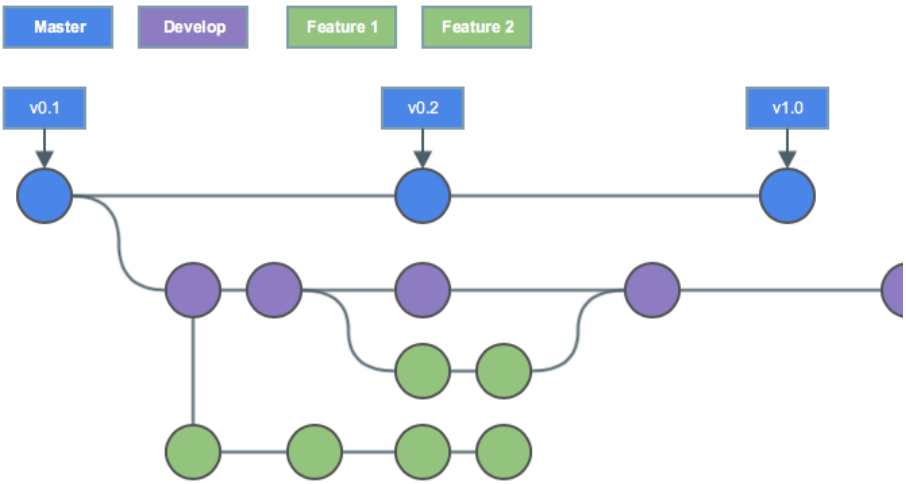
Gitflow Workflow - Historical Branches

- Master branch for release history (including version tags)
- Master branch is always `stable` and in production
- Develop branch for feature integration



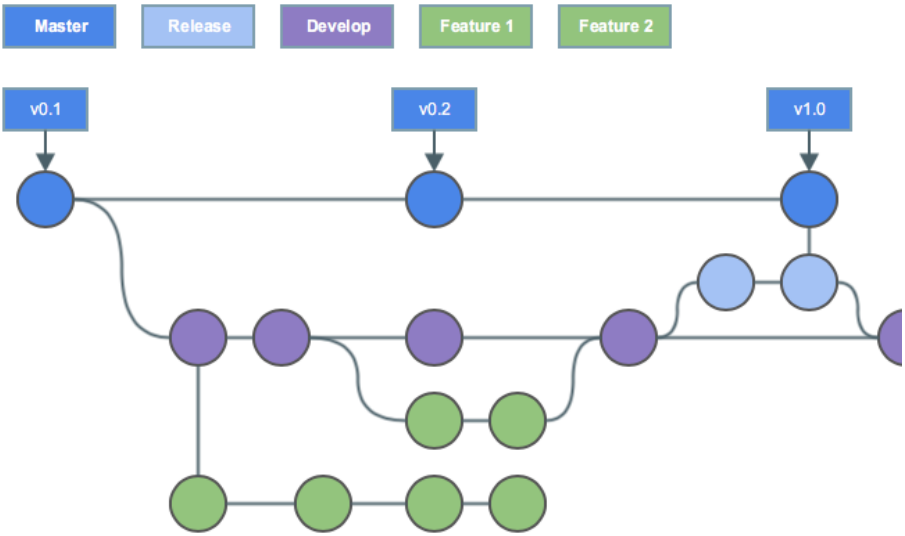
Gitflow Workflow - Feature Branches

- New features in their own branches
- Feature branches use `develop` as their parent branch
- Once completed, merged back to `develop`



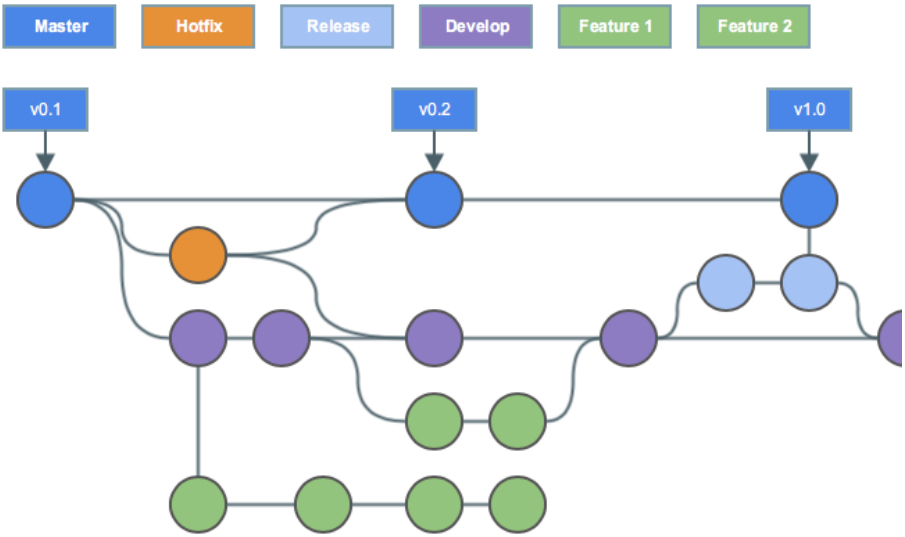
Gitflow Workflow - Release Branches

- Enough features in `develop` : `release` branch based on `develop`
- Ready to ship: merged to `master` and tagged with version



Gitflow Workflow - Maintenance Branches

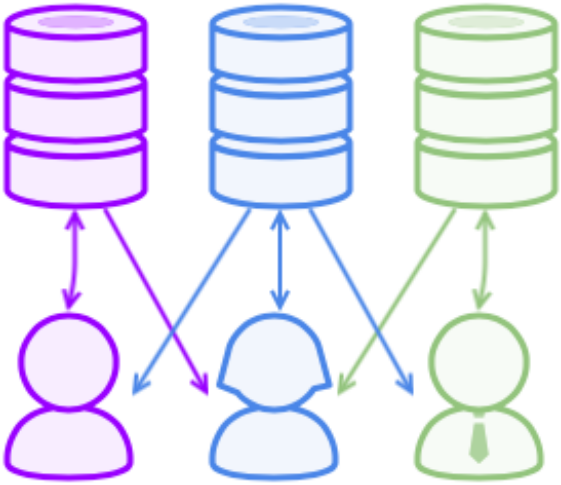
- Fixes based on `master`
- Merged to `master`, tagged and merged to `develop`



NETWAYS TRAINING

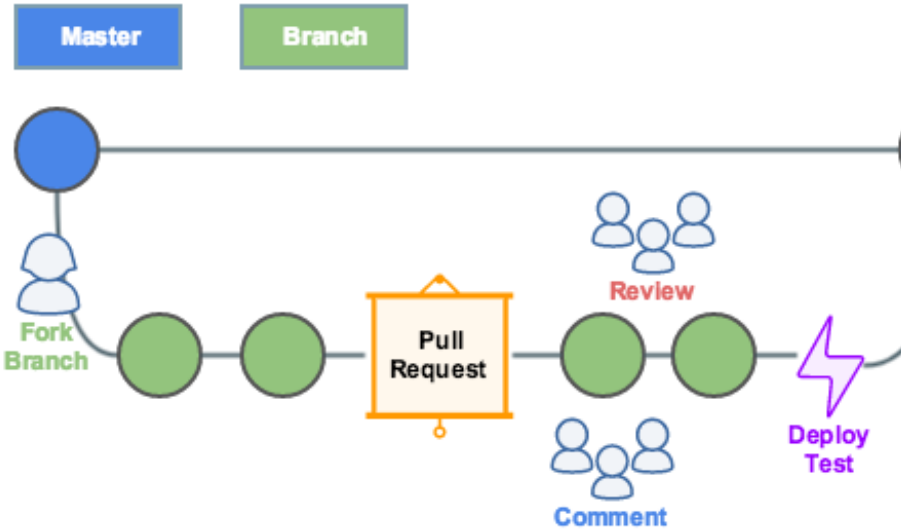
Forking Workflow with GitHub/GitLab

- Every developer has
  - a server-side repository
  - a private repository as copy of the server-side repo
- Developers push their own server-side repository
- Project maintainer pushes to official repository



Forking Workflow - How it works

- Developers fork, commit and push into their own repository
- Developers create a `Pull Request` for the official repository, CI triggers automated tests for the PR
- Developers/Maintainers review and merge PR, CI triggers deployment task



References:

GitHub:
https://guides.github.com/introduction/flow/
GitLab:
https://docs.gitlab.com/ce/workflow/forking_workflow.html

Forking Workflow - Multiple Remote Repositories

- **Multiple** remote repositories
  - `origin` is yours, e.g. git@github.com:dnsmichi/icinga2.git
  - `upstream` is official with releases, e.g. https://github.com/icinga/icinga2.git

- Merge branches from `origin/<branch>` to `upstream/<branch>` and vice versa

Example:

```
$ git remote -v
origin  git@github.com:dnsmichi/icinga2.git (fetch)
origin  git@github.com:dnsmichi/icinga2.git (push)
upstream    https://github.com/icinga/icinga2.git (fetch)
upstream    https://github.com/icinga/icinga2.git (push)
```

GitHub/GitLab Workflow - Keep in Sync

- Pull changes from `upstream` HEAD to own repository master branch to sync development
- Remote `HEAD` is the default branch for this repository, usually `master`
- Remember, HEAD is just a smart pointer. You can use a branch name too.

Example:

```
$ git checkout master
$ git fetch upstream
$ git pull upstream HEAD
$ git push origin master
```

Lab 8.5: Create Merge Request

- Objective:
  - Create merge request from feature branch

- Steps:
  - Change into `$HOME/training`
  - Create the branch `feature/docs-merge-request`
  - Edit `README.md`, add, commit and push the changes
  - Open the proposed GitLab URL in your browser
  - Fill in the merge request and submit it
  - Simulate a review and merge it
  - Pull changes to local master branch and use `tig`

Practical Examples for Git Workflows

If you are using GitHub/GitLab repositories, use forking and pull/merge requests.

Smart adoptions of the Gitflow workflow could include the following:

- development branch (`master`)
- release branches (`support/<version>`)
- Fix and feature branches are based off the `master` branch
- Bugfixes are cherry-picked into the release branches

More Hints: Rebase and Squash

The `git rebase` command can also be used to perform certain actions on a specific commit history.

If you are contributing to open source projects developers might ask you to either rebase your history or even `squash` all commits into one commit.

```
commit1  => commit
commit2
commit3
```

Ask the trainer to draw an image for better illustration and discussion.

Lab 8.6: Rebase and squash commits

- Objective:
  - Rebase and squash commits

- Steps:
  - Add 3 commits to your history
  - Use `git rebase -i HEAD~3` to start the interactive mode. `HEAD~3` takes the last 3 commits compared to current HEAD.
  - Use `pick` for the top commit
  - Replace `pick` with `squash` for the other commits
  - Save and edit the final commit message
  - Use `git log` to verify the history

- Bonus:
  - Push the changed commit history using `git push -f` and explain what happens

---

`git rebase` can also be used to `edit` the commits in your history. This is helpful if you want to

amend changes to the test commit. Or if they are missing issue references, or just contain wrong information.

This is a common scenario for code reviews before merging the actual history into the development branches.

More Hints: Delete Remote Branches

You have learned that you can create remote (feature) branches. But what if you want to delete such branches?

`git push origin <branch>` is short for `git push origin <localbranch>:<remotebranch>`.

Pushing `NULL` into a remote branch will delete it.

```
git push origin :<remotebranch>
```

Hint: You can delete branches in GitLab/GitHub too.

Lab 8.7: Delete remote branch

- Objective:
  - Delete remote branch

- Steps:
  - Change into `$HOME/training`
  - Create or identify a remote branch `feature/docs-wrong-name`
  - Delete the remote branch

9 Git Integrations

Git Integrations

- Ticket system integration
- Continuous integration (Jenkins, Travis)
- Puppet environments
- Git Hooks
- Development workflow

Git Hooks

- Client or server-side scripts
- Conditional execution (update, post-receive, etc.)
- Integration with external tools
- Web hooks available in GitLab, GitHub, etc.

Client Hooks

- pre-commit
- pre-rebase
- post-checkout
- post-merge
- Useful to
  - Preserve file modes
  - Check code style/syntax

Server-side Hooks

- pre-receive
  - Deny non-fast-forward pushes
- post-receive
  - After everything is updated
  - Notify external tools (Mail, IRC, Jenkins, etc.)
- update
  - Check branch permissions

Server-side hooks receive the arguments through STDIN in the following format:

```
oldref newref refname
```

Example:

```
aa453216d1b3e49e7f6f98441fa56946ddcd6a20 68f7abf4e6f
922807889f52bc043ecd31b79f814 refs/heads/master
```

Example for a simple post-receive demo hook:

``` $ vim post-receive
```

!/bin/bash

oldref newref refname

read line set -- $line

print the commits between oldref and newref and count the lines

num=$(git log --pretty=oneline ${1}..${2}| wc -l)

echo "New ref name '${3}' created. Pushed ${num} commits. Old ref '${1}' to new ref '${2}'." exit 0

$ chmod +x post-receive ```

Test on the client:

``` $ whoami $ cd $HOME/training.git $ echo "1" > hooktest $ git add hooktest $ git commit -av -m "1. hooktest" $ echo "2" >> hooktest $ git commit -av -m "2. hooktest"
```

$ git log $ git push ... remote: New ref name 'refs/heads/master' created. Pushed 2 commits. Old ref ... to new ref ... . ```

Web Hooks

- HTTP Callback
- Limit on event, e.g. push, merge, comment, etc.
- Integrate test and deployment services
  - Travis CI
  - Jenkins
- Chat bot integration
  - Rocket.Chat
  - Slack

Reference example (German):
https://blog.netways.de/2016/06/29/gitlab-webhooks/

Travis CI does not integrate well with GitLab, they prefer GitHub.

Jenkins

Jenkins builds with web books:
https://github.com/jenkinsci/gitlab-plugin/wiki/Setup-Example

Rocket.Chat

Rocket.Chat web hook integration:
https://rocket.chat/docs/administrator-guides/integrations/gitlab/

Slack

Official GitLab Slack integration:
https://gitlab.com/gitlab-org/gitlab-ce/blob/master/doc/user/project/integrations/slack.md

Try it out

Both Rocket.Chat and GitLab are available at NWS:

- https://nws.netways.de/products/gitlab-ce
- https://nws.netways.de/products/rocket-chat

Puppet Environments

- Multiple Puppet environments (prod, stage, ...)
- Git branches for each environment
- Different module versions and configuration for each environment
- Automated deployment with r10k

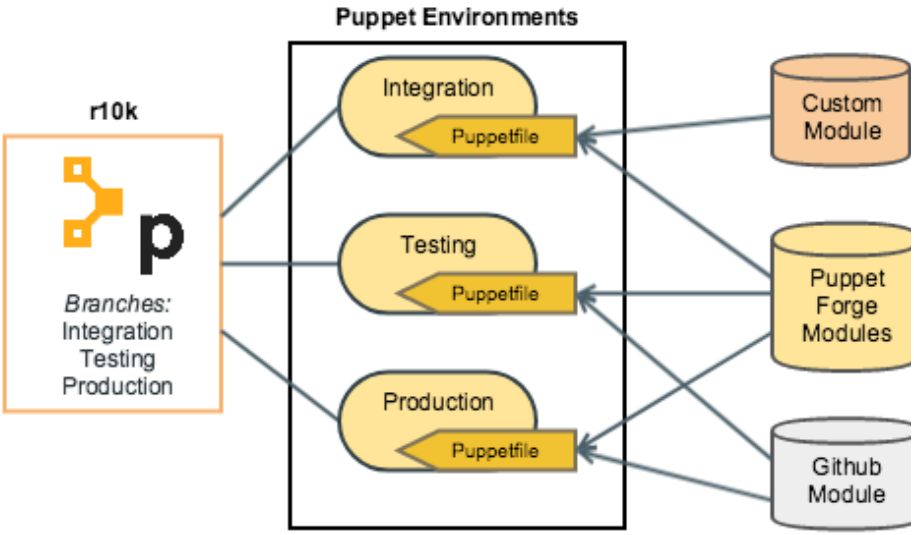
r10k is a toolset provided by Puppet for deploying environments and modules.

Reference blog post (German):
https://blog.netways.de/2014/11/07/git-workflow-bei-puppet-mit-r10k/

Puppet environments and r10k are part of the "Scaling Puppet" training:
https://www.netways.de/en/events_trainings/puppet_trainings/scaling_puppet/

NETWAYS TRAINING

Puppet Environments: r10k




```
mod "puppetlabs/powershell", "2.0.2"

mod "icinga2",
    :git => 'https://github.com/Icinga/puppet-icinga2.git',
    :ref => 'v1.3.0'
```

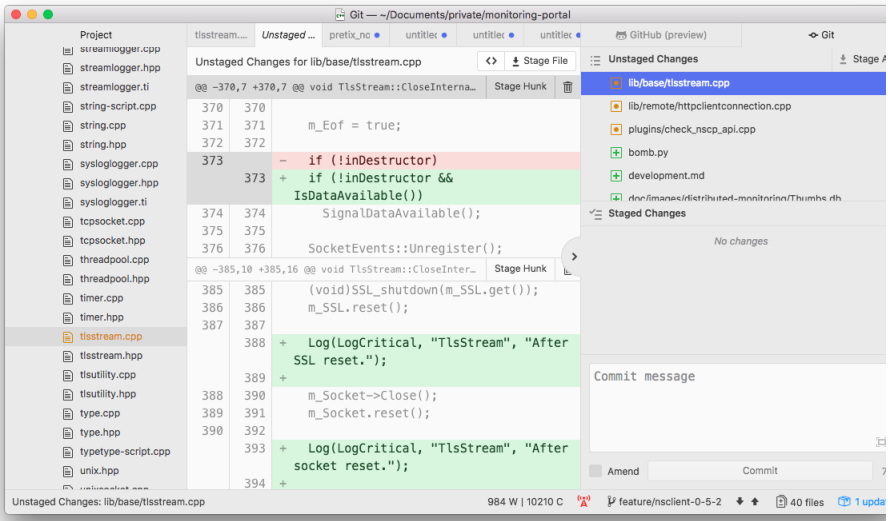
IDEs

- Atom
- JetBrains PHPStorm, ...
- Visual Studio
- Eclipse
- vim

- Eclipse egit: http://www.eclipse.org/egit/
- vim-fugitive: https://github.com/tpope/vim-fugitive

Atom

- Line diffs, tree view, etc.
- Default editor for command line commits
- GitHub integration with Pull Requests

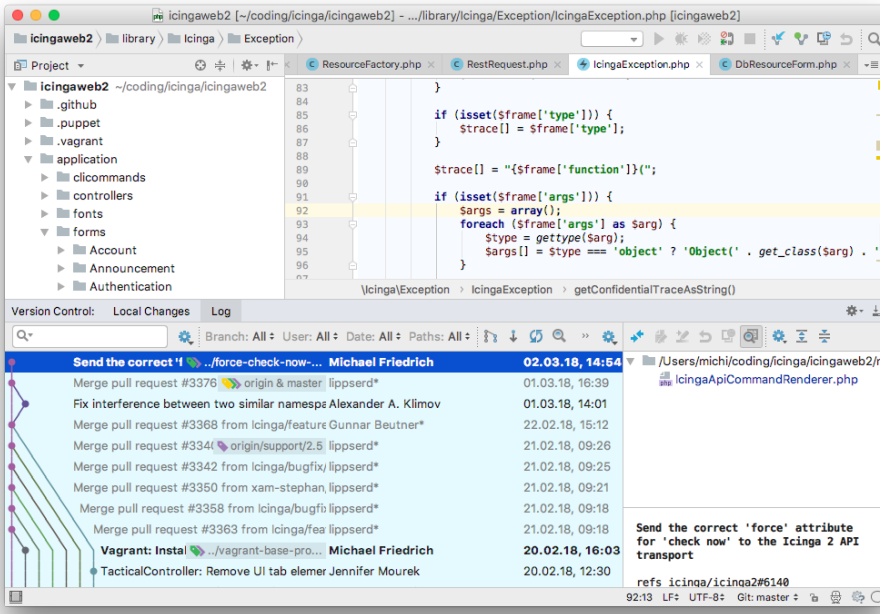


References:

http://blog.atom.io/2014/03/13/git-integration.html
http://blog.atom.io/2017/05/16/git-and-github-integration-comes-to-atom.html

JetBrains

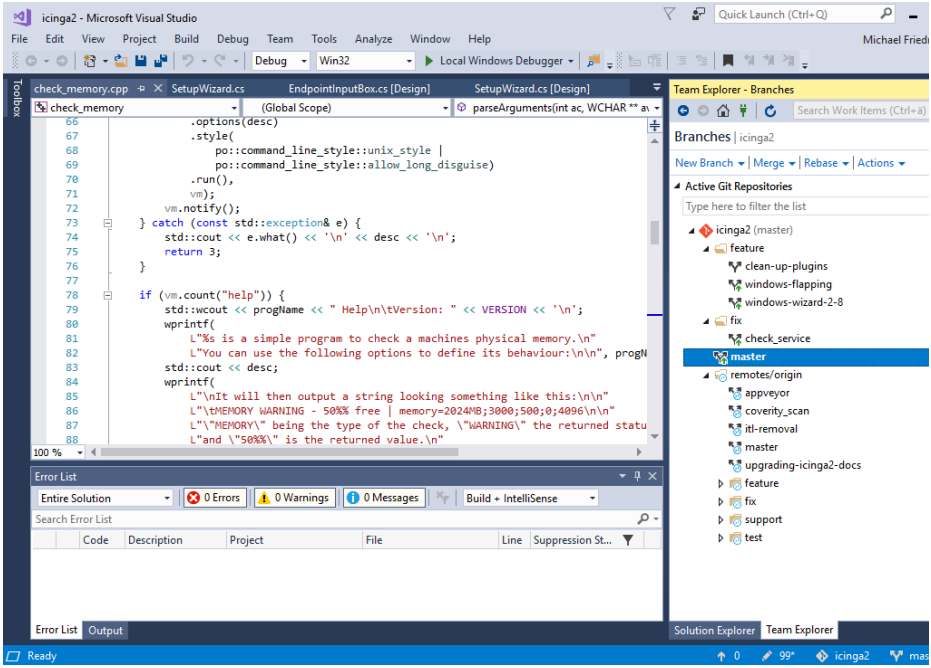
- Status and change tracking
- Add, commit, fetch, push, pull, etc.
- GitHub integration



Reference:
https://www.jetbrains.com/help/idea/using-git-integration.html

Visual Studio

- Git for Windows bundled in 2015+
- Native GUI integration



References:

- https://docs.microsoft.com/en-us/vsts/git/gitquickstart?

tabs=visual-studio
- https://visualstudio.github.com/

NETWAYS TRAINING

10 Continuous Integration

Continuous Integration

- Immediately test changes in feature branches
- Code quality checks
- Deploy branches into environments (dev, staging, production)
- Visualize failures and notify users
- Reporting

Continuous Integration: Overview

- Travis CI
- Jenkins
- GitLab CI

Continuous Integration: Travis CI

- Cloud based CI system
- Supports Ubuntu 14.04 LTS and macOS
- Code and unit tests
- Test multiple PHP, Perl, etc. versions at once
- Easy to integrate into GitHub projects
- De-facto standard for Open Source projects hosted on GitHub

Travis CI Example

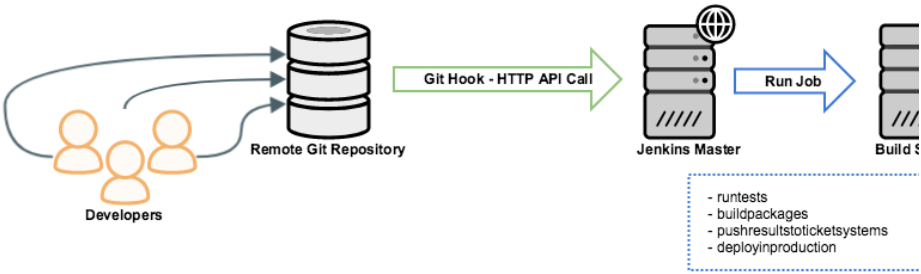
```
$ cat .travis.yml

language: php

php:
  - '7.1'

branches:
  only:
  - master

script:
  - php phpunit.phar -c modules/test/phpunit.xml --verbose
```

Continuous Integration: Jenkins

- Self-hosted application
- Jobs
- Push trigger or time-based cronjobs
- Build pipelines
- Build agents for Linux/Unix, Windows, etc.
- Written in Java
- Many plugins available

Jenkins Workflow

- Trigger build jobs from git commits
- Start build jobs from specific branches (e.g. daily snapshots)
- Run tests which update your ticket system
- Build packages and deploy them into your repository



Adding Jenkins to your Git workflow requires additional toolsets.

A Git hook configured on the git server is able to trigger a Jenkins job. It passes the git

commit id to the Jenkins job which runs a local git checkout and further job actions afterwards, for example tests and/or package builds.

Vice versa a Jenkins build job hook needs to update the ticket system. This will notify the user of an unsuccessful run immediatly allowing further actions.

The issue id can be for example passed from the git hook into the Jenkins job. This requires strict rules on git commits - a git commit without any referenced issue id will destroy this workflow.

Define and document a git commit format specification and enforce this to your users. In addition to that install a git hook on the server which rejects a git push when the commit is missing an issue id.

Continuous Integration: GitLab CI

- Build Pipelines
- Natively integrated into GitLab
- Invokes GitLab Runners
- Local container registry for runners

Documentation:
https://docs.gitlab.com/ce/ci/README.html

Reference example (German):
https://blog.netways.de/2017/05/03/gitlab-ce-continuous-integration-jobs-and-runners/

GitLab CI: Introduction

- `.gitlab-ci.yml` configuration file in Git repository
- Runner is triggered on specific events, e.g. `git push`
- Jobs can be run on-demand
- Built-in and external runners
- Container registry enabled for the project (optional)

Documentation references:

https://docs.gitlab.com/ce/user/project/container_registry.html

GitLab Runners

- Written in Go
- Linux/Unix, macOS, Windows, Docker support
- Run multiple jobs in parallel
- Run jobs locally, in Docker containers, remote via SSH
- Can run Bash, Windows Batch/Powershell

Documentation reference:
https://docs.gitlab.com/runner/

GitLab Runners: Installation and Configuration

- Separate server
- Installation via package repository
- `gitlab-runner register`

Note: This is not needed for the built-in `docker executor` runner.

Documentation References:

https://docs.gitlab.com/runner/install/linux-repository.html
https://docs.gitlab.com/runner/register/index.html

GitLab CI: Container Registry

- Enable the Container Registry (administration server setting)
- Enable the Container Registry for the project
- Advanced usage only

```

## vim /etc/gitlab/gitlab.rb

registry_external_url 'https://gitlab.example.com:5000'

## gitlab-ctl reconfigure
```

```

Documentation References:

https://docs.gitlab.com/ce/user/project/container_registry.html
https://docs.gitlab.com/ce/administration/container_registry.html
```

NETWAYS TRAINING

GitLab CI: Docker, Containers - how to use it

- Run an application in an isolated environment
- Layered images providing additional libraries and tools, e.g. base linux, mysql, apache, ruby
- Start container, run tests, return results
- Light-weight and fast, can run on each Git push
- Reliable same run each time, container is destroyed on stop

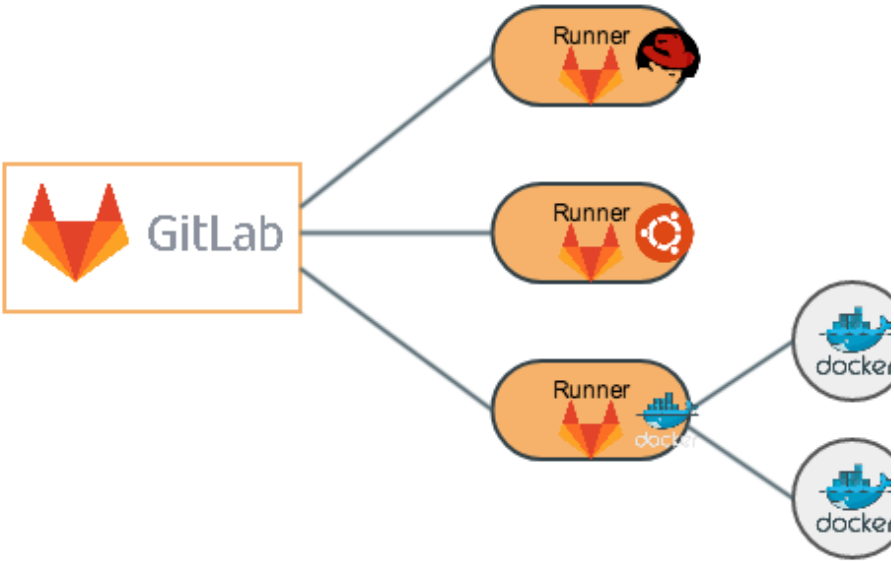
---

Documentation References:

https://docs.docker.com

NETWAYS TRAINING

GitLab CI: Docker and CI Runners



Documentation References:

https://docs.docker.com
https://docs.gitlab.com/runner/install/docker.html

Lab 10.1: Inspect CI Runner settings

- Objective:
  - Inspect CI Runner Settings

- Steps:
  - Navigate to Admin > Overview > Runners
  - Inspect the token
  - Check existing runners

Reference: https://gitlab.com/gitlab-org/gitlab-runner/blob/master/docs/install/linux-repository.md

GitLab CI: Configuration in .gitlab-ci.yml

- `image` as container base image
- `services` which should be running
- `all_tests` as job name

Example:

```
image: alpine:latest

all_tests:
  script:
    - exit 1
```

Documentation References:

https://about.gitlab.com/2016/03/01/gitlab-runner-with-docker/

Lab 10.2: Create .gitlab-ci.yml

- Objective:
  - Create CI configuration for the training project

- Steps:
  - Create the `.gitlab-ci.yml` file in the `training` directory (vim, nano, etc.)
  - Add `image: alpine/latest` to specify base image
  - Add job `all_tests` with `script` as array element, which itself runs `exit 1`

Lab 10.3: Push to GitLab

- Objective:
  - Add .gitlab-ci.yml to Git and push to GitLab
- Steps:
  - Use `git add .gitlab-ci.yml` and commit the change
  - Push the commit into the remote repository
  - Navigate to the project into `CI / CD` and verify the running job
- Bonus
  - Modify the exit code to `0`, add, commit, push and verify again

GitLab CI: .gitlab-ci.yml Templates

- Repository: https://gitlab.com/gitlab-org/gitlab-ci-yml
  - PHP: https://gitlab.com/gitlab-org/gitlab-ci-yml/blob/master/PHP.gitlab-ci.yml
  - Python: https://gitlab.com/gitlab-org/gitlab-ci-yml/blob/master/Python.gitlab-ci.yml
  - C++: https://gitlab.com/gitlab-org/gitlab-ci-yml/blob/master/C++.gitlab-ci.yml
  - ...

Documentation References:

https://docs.gitlab.com/ce/ci/README.html#examples

https://gitlab.com/gitlab-org/gitlab-ci-yml

Lab 10.4: Practical Example for CI Runners: Preparations

- Objectives:
  - Prepare container to convert Markdown to HTML
- Steps:
  - Modify `.gitlab-ci.yml` and add a `before_script` section after the `image` section
  - Update `apk` package manager and install `python` and `py-pip` packages
  - Use `pip` to install the `markdown` and `Pygments` libraries
  - Commit and push the changes

Example:

```
before_script:
  - apk update && apk add python py-pip
  - pip install markdown Pygments
```

The base image uses Alpine Linux which has a very small installation size and footprint.

In contrast to the "typical" Linux distribution containers, this allows for faster tests and deployment scripts.

If your build pipeline involves specific distributions, choose the best and reliable container distribution you prefer.

Alpine uses its own package manager. This exercise installs

- Python and its package manager `pip`
- Markdown conversion packages for Python

Reference: https://wiki.alpinelinux.org/wiki/Alpine_Linux_package_management

Lab 10.5: Practical Example for CI Runners: Create Docs

- Objective:
  - Create HTML docs from Markdown

- Steps:
  - Add a new `markdown` and use `script` to generate `README.html`
  - Add `archifacts` with `paths` pointing to `README.html`. Expires in `1 week`
  - Commit and push the changes, then download and view the `README.html` file in your browser

Example:

```
markdown:
  script:
    - python -m markdown README.md > README.html
  artifacts:
    paths:
    - README.html
    expire_in: 1 week
```

Lab 10.6: Practical Example for CI Runners: Update Docs

- Objective:
  - Add what you have learned so far into README.md and generate docs

- Steps:
  - Edit `README.md`
  - Commit and push changes
  - Download and view the `README.html` file in your browser

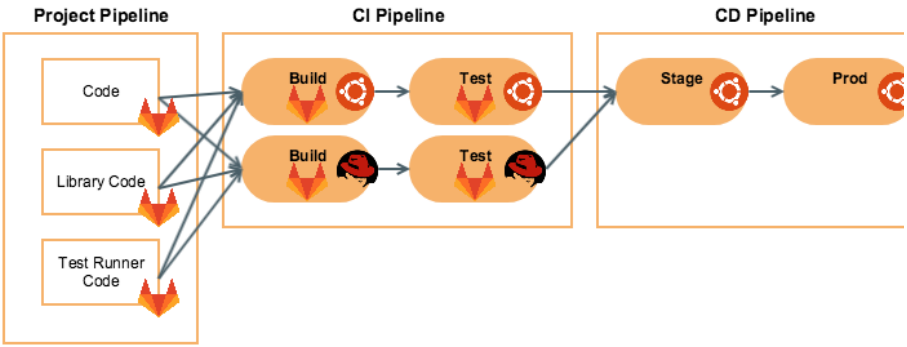
GitLab Pipelines

Code => Build => Test => Deploy

- Encapsulate QA steps into pipelines
- Visible which pipeline is affected
- Halt if one step is failing
- Direct feedback to developers
- CI and CD
  - Continuous Integration
  - Continuous Delivery

Reference:
https://docs.gitlab.com/ce/ci/pipelines.html

GitLab Pipelines: Example



NETWAYS TRAINING

11 Git Hints

Best Practices

- Ensure to set user configuration (name, email)
- Always keep good commits in mind
- Do not commit all changes at once
- Work with branches if applicable
- Avoid merge commits, fetch and rebase your history

NETWAYS TRAINING

Hints for details and tests

- verbose mode for cli commands: `-v`
- test-drive but don't change anything: `--dry-run`

NETWAYS TRAINING

Debugging

- `git bisect`
  - Binary search for commits causing a bug
- `git blame`
  - Open editor with commit/author/date addition
- `git grep`
  - Search for pattern in commit history

`git bisect` can be used to invoke a binary search to find the commit that introduced a bug.

`git blame` prints a file and each line changed by the latest commit and author.

`git grep` prints all lines from the commit history matching a pattern.

Lab 11.1: Use Git Blame

- Objective:
  - Use `git blame`

- Steps:
  - Pick a file from your local git repository
  - Use `git blame filename`
  - Explain the line prefix and its meaning

Restore deleted file

This is where `git rev-list` comes in handy.

Find the last commit which affected the given path.

`git rev-list -n 1 HEAD -- <filepath>` .

Then checkout the version at the commit before using the caret `^` symbol.

`git checkout <deletingcommit>^ -- <filepath>`

Apply and Create Patches

Manually apply and create Git patches.

- `git apply` : apply git patch
- `git am` : read patches from STDIN and apply them
- `git format-patch` : create git patch

`git apply` applies a git patch file to your working tree.

`git am` reads git patches from your mailbox by default. It also can be used to read patches from the shell's STDIN (e.g. curl downloading a patch from the web). The `-s` option allows you to sign off the patch, for example if you have reviewed the patch from an external committer.

`git format-patch` requires either a negative number of commits from current HEAD or a git commit id. It will then create numbered patch files for all required commits.

Git Aliases

- Shortcuts
- Custom commands
- "Simulate" SVN commands

Lab 11.2: Add an alias for git diff

- Objective:
  - Add an alias for git diff

- Steps:
  - Edit the `$HOME/.gitconfig` file
  - Add a new `[alias]` section if not existing
  - Add `d` as an alias for `diff`

Git Attributes

- Control the file line ending mode on different OS
- Ignore specific files on git archive

Example:

```
$ vim .gitattributes

modules/** eol=lf
```

Git Submodules

- Pointer to a specific commit in remote repository
- Independent sub directory with own `.git` repository

Example:

```
$ git submodule add https://github.com/Icinga/puppet-icinga2
$ git status
  new file: .gitmodules
  new file: puppet-icinga2

$ git submodule
  ad5e309... puppet-icinga2 (heads/master)
```

Fresh clone:

```
$ git clone --recursive ...

$ git clone ...
$ git submodule init && git submodule update
```

Note: Git submodule removal is not trivial.

https://davidwalsh.name/git-remove-submodule

Git Subtree

- Merge an external git repository history
- Directory in your main repository (`--prefix`)
- Squash the remote history into one commit
- No external dependency (e.g. git submodule repository not available anymore)
- No local configuration - document the external reference

Example for imported Puppet modules:

```
$ git subtree add --prefix modules/icinga2 \
https://github.com/Icinga/puppet-icinga2 HEAD --squash

$ git subtree pull  --prefix modules/icinga2 \
https://github.com/Icinga/puppet-icinga2 HEAD --squash
```

Adding a git subtree requires

- Existing directory tree except for the one created
- Git clone URL
- The branch which should be added (use `master` as

default)
- Optionally `--squash` all commits into one

Example:

```
git subtree add --prefix clippy.js https://github.com/smore-inc/clippy.js.git master --squash
```

12 GitLab Hints

Issues, Merge Requests, Charts

- Issues applied to task workflow
- Merge requests with branches "ready to merge"
- Overview charts and boards for better organization
- Your "own" development and operations workflow

Administration

- Default Settings
- Stats, Versions, Users
- Permission Management
- Monitoring, System Hooks
- Appearance and Customizing

Explore this with the trainer and discuss specific topics.

Best Practices

- HTTPS only
- Performance tuning
- Monitoring
- Troubleshooting

Secure HTTPS environment

- Use Let's Encrypt or own certificates
- Enable HTTPS in GitLab/Nginx
- Forward all HTTP requests to HTTPS

Reference:
https://docs.gitlab.com/ce/install/installation.html#using-https

Performance Tuning

- 4+ GB RAM make GitLab happy
- Move PostgreSQL 9.2+ to a dedicated host

Monitoring

- HTTP(S)
- PostgreSQL
- Redis
- Background daemons
- Logs

GitLab API

- Automation
  - Users, Groups, Projects, etc.
- Handle Merge Requests
- Jobs, Pipelines, CI
- ...

You can manage nearly everything.

Reference:
https://docs.gitlab.com/ce/api/README.html

API Clients:
https://about.gitlab.com/applications/#api-clients

GitLab Applications

GitLab provides an extensive list of external applications:

https://about.gitlab.com/applications/

- CLI clients
- API clients
- GUIs
- ...

Reference:
https://docs.gitlab.com/ce/api/README.html

API Clients:
https://about.gitlab.com/applications/#api-clients