GIT

Working with GIT Tags & Branches

Lesson Objectives

- **Working with GIT Branches**

Tags

- Git has the option to tag a commit in the repository history so that you find them more easily at a later point in time. Most commonly, this is used to tag a certain version which has been released.

Working with tags

- **Creating tags**
  - You can create a new tag via the git tag command. Via the -m parameter, you specify the description of this tag. The following command tags the current active HEAD.
  - git tag version1.6 -m 'version 1.6' You can also create tags for a certain commit id.
  - git tag version1.5 -m 'version 1.5' [commit id]
- **If you want to use the code associated with the tag, use:**
  - git checkout <tag_name>
- **Sharing tags**
  - By default the git push command does not transfer tags to remote repositories. You explicitly have to push the tag with the following command.
  - git push origin [tagname]
- **You can list the available tags via the following command:**
  - git tag

Branches

- Git allows you to create *branches*, i.e. copies of the files from a certain commit. These branches can be changed independently from each other. The default branch is called *master*.
- Git allows you to create branches very fast and cheaply in terms of resource consumption. Developers are encouraged to use branches frequently.
- If you decide to work on a branch, you *checkout* this branch. This means that Git moves the *HEAD* pointer to the latest commit of the branch and populates the *working tree* with the content of this commit.
- Untracked files remain unchanged and are available in the new branch. This allows you to create a branch for unstaged and uncommited changes at any point in time.

Working with Branches

- **Create new branch**
  - You can create a new branch via the git branch [newname] command. This command allows optionally to specify the commit id, if not specified the currently checked out commit will be used to create the branch.
  - git branch testing
  - git checkout testing
  - echo "Cool new feature in this branch" > test01
  - git commit -a -m "new feature"
  - git checkout master
  - cat test01
- **To create a branch and to switch to it at the same time you can use the git checkout command with the *-b* parameter.**
  - git checkout -b bugreport12
  - git checkout -b mybranch master~1

Working with Branches

- **Rename a branch**
  - Renaming a branch can be done with the following command
  - git branch -m [old_name] [new_name]
- **Delete a branch**
  - To delete a branch which is not needed anymore, you can use the following command.
  - git branch -d testing
  - git branch
- **Push a branch to remote repository**
  - By default Git will only push matching branches to a remote repository. That means that you have to manually push a new branch once. Afterwards "git push" will also push the new branch.
  - git push origin testing
  - git checkout testing
  - echo "News for you" > test01
  - git commit -a -m "new feature in branch"
  - git push

Working with Branches

- **Difference between branches**
  - To see the difference between two branches you can use the following command.
  - git diff master your_branch
- **Remote tracking branches**
  - Your local Git repository contains references to the state of the branches on the remote repositories to which it is connected. These local references are called *remote tracking branches*.
  - You can see your *remote tracking branches* with the following command
  - git branch -r
  - To see all branches or only the local branches you can use the following commands
  - *# list all local branches*
  - git branch
  - *# list local and remote braches*
  - git branch -a
- **Delete a remote branch**
  - It is also save to delete a remote branch in your local Git repository. You can use the following command for that.
  - git branch -d -r origin/<remote branch>

Working with Branches

- You can create new *tracking branches* by specifying the *remote branch* during the creation of a branch. The following example demonstrates that.
- # setup a tracking branch called newbrach# which tracks origin/newbranch
- git checkout -b newbranch origin/newbranch

Working with Branches

- **Updating your remote branches with git fetch**
  - You can update your remote branches with the git fetch command.
  - The git fetch command updates your *remote branches*. The fetch command only updates the *remote branches* and none of the local branches and it does not change the working tree of the Git repository. Therefore you can run the git fetch command at any point in time.
  - After reviewing the changes in the remote tracking branch you can merge or rebase these changes onto your local branches. Alternatively you can also use the git cherry-pick "sha" command to take over only selected commits
- **Compare remote tracking branch with local branch**
  - The following code shows a few options how you can compare your branches
  - *# show the long entries between the last local commit and the # remote branch*
  - git log HEAD..origin
  - *# show the diff for each patch*
  - git log -p HEAD..origin *# show a single*
  - *diff* git diff HEAD...origin

Working with Branches

Rebase your local branch based on the remote tracking branch

- You can apply the changes of the *remote branches* on your current local branch for example with the following command
- *# assume you want to rebase master based on the latest fetch # therefore check it out*
  git checkout master
- *# update your remote tracking branch*
  git fetch
- *# rebase your master onto origin/master*
- git rebase origin/master

Working with Branches

- **Fetch vs. pull**
  - The git pull command performs a git fetch and git merge (or git rebase based on your Git settings). The git fetch does not perform any operations on your local branches. I can always run the fetch command and review the incoming changes.

Working with Branches

Merging branches

- Git allows to combine the changes of two branches. This process is called *merging*.
- The git merge command performs a merge. If the commits which are merged are direct successors of the HEAD pointer of the currentbranch, Git simplifies things by performing a so-called *fast forward merge*. This *fast forward merge* simply moves the *HEAD* pointer of the current branch to the last commit which is being merged.
- This process is depicted in the following graphics. Assume you want to merge the changes of branch into your master branch. Each commits points to its successor.
- You can merge changes from one branch to the current active one via the following command.
- *# merges into your current selected branch*
- git merge testing

Working with Branches

- **Merge conflicts**
  - A merge conflicts occurs, if two people have modified the same content and Git cannot automatically determine how both changes should be applied.
  - If a merge conflict occurs Git will mark the conflict in the file and the programmer has to resolve the conflict manually. After resolving it, he can add the file to the staging index and commit the change.
- **Example process for solving a merge conflict**
  - In the following example you first create a merge conflict and afterwards you resolve the conflict and apply the change to the Git repository.
  - The following code creates a merge conflict

```
# Switch to the first directory
cd ~/repo01
# Make changes
echo "Change in the first repository" > mergeconflict.txt
# Stage and commit
git add . && git commit -a -m "Will create merge conflict 1"

# Switch to the second directory
cd ~/repo02
# Make changes
touch mergeconflict.txt
echo "Change in the second repository" > mergeconflict.txt
# Stage and commit
git add . && git commit -a -m "Will create merge conflict 2"
# Push to the master repository
git push

# Now try to push from the first directory
# Switch to the first directory
cd ~/repo01
# Try to push --> you will get an error message
git push


# Get the changes via a pull
# this creates the merge conflict in your
# local repository
git pull origin master
```

- Git marks the conflict in the affected file. This file looks like the following.
- <<<<<<< HEAD
- Change in the first repository
- =======
- Change in the second repository
- >>>>>>> b29196692f5ebfd10d8a9ca1911c8b08127c85f8

  - The above is the part from your repository and the below one from the remote repository. You can edit the file manually and afterwards commit the changes.
  - Alternatively, you could use the git mergetool command. git mergetool starts a configurable merge tool that displays the changes in a split screen. git mergetool is not always available but it is save to edit the file with merge conflicts by hand.
  - # Either edit the file manually or use
  - git mergetool
  - # You will be prompted to select which merge tool you want to use
  - # For example on Ubuntu you can use the tool "meld"
  - # After merging the changes manually, commit them
  - git commit -m "merged changes"

Rebase

You can use Git to rebase one branches on another one. As described the merge command combines the changes of two branches. If you rebase a branch called A onto another, the git rebase command takes the changes introduced by the commits of branch A and applies them based on the HEAD of the other branch. This way the changes in the other branch are also available in branch A.

- # create new branch
- git checkout -b rebasetest
- # To some changes
- touch rebase1.txt
- git add . && git commit -m "work in branch“
- # do changes in master
- git checkout master# make some changes and commit into testing
- echo "This will be rebased to rebasetest" > rebasefile.txtgit add rebasefile.txt
- git commit -m "New file created“
- # rebase the rebasetest onto master
- git checkout rebasetestgit rebase master
- # now you can fast forward your branch onto master
- git checkout master
- git merge rebasetest

Summary

- **Discussed how to work with branches**