

Lab #2: Version Control

2.1. Objective

- To study the ideology and application of version control tools. -
- Master the skills of working with git.

2.2. Version control systems. General concepts

Version control systems (VCS) are used when several people work on the same project. Typically, the main project tree is stored in a local or remote repository that is configured to be accessible to project members. When changes are made to the content of the project, the version control system allows them to be fixed, to combine changes made by different project participants, to roll back to any earlier version of the project, if necessary.

Classic version control systems use a centralized model that assumes a single repository for storing files. Most of the version control functions are performed by a dedicated server. A project participant (user) before starting work, by means of certain commands, receives the version of files he needs. After making changes, the user places a new version in the repository. At the same time, previous versions are not deleted from the central repository and you can return to them at any time. The server may not save the full version of the changed files, but produce the so-called delta compression - save only changes between successive versions, which reduces the amount of stored data. Version control systems support the ability to track and resolve conflicts that may arise when multiple people work on the same file. You can merge (merge) changes made by different contributors

(automatically or manually), manually select the desired version, undo changes altogether, or lock files for modification. Depending on the settings, the lock does not allow other users to get a working copy or prevents the working copy of the file from being modified by means of the OS file system, thus providing privileged access to only one user working with the file.

Version control systems can also provide additional, more flexible functionality. For example, they can support working with multiple versions of the same file, maintaining a common change history up to the version branch point and each branch's own change history. In addition, information is usually available about which of the participants, when and what changes were made. Typically, this kind of information is stored in a change log, access to which can be restricted. Unlike classical ones, in distributed version control systems, the central repository is optional.

Among the classic VCS, CVS, Subversion are the most famous, and among the distributed ones, Git, Bazaar, Mercurial. The principles of their work are similar, they differ mainly in the syntax of the commands used in the work.

2.3. Instructions for laboratory work

The Git version control system is a collection of command line programs. They can be accessed from the terminal by typing the git command with different options.

Due to the fact that Git is a distributed version control system, the backup a copy of the local storage can be made by simple copying or archiving.

2.3.1. Basic git commands

The most commonly used git commands are:

– creating the main repository tree:

- `git init`

– receiving updates (changes) of the current tree from the central repository:

- `git-pull`

- sending all the changes made to the local tree to the central repository
thorium:

- `git push`

– view the list of changed files in the current directory:

- `git status`

– view current changes:

- `git diff`

– saving current changes:

– add all modified and/or created files and/or directories:

- `git add .`

– add specific modified and/or created files and/or directories:

- `git add filenames`

– delete a file and/or directory from the repository index (in this case, the file and/or directory remains in the local directory):

- `git rm filenames`

– saving the added changes:

– save all added changes and all modified files:

- `git commit -am 'Commit description'`

- save the added changes with a comment through the built-in editor:

```
- git commit
```

– creating a new branch based on the current one:

```
- git checkout -b branchname
```

- switch to some branch:

```
- git checkout branch_name
```

(when you switch to a branch that is not yet in the local repository, it will be created and linked to the remote)

– pushing changes of a specific branch to the central repository:

```
- git push origin branchname
```

– merge the branch with the current tree:

```
- git merge --no-ff branchname
```

- deleting a branch:

– deleting a local branch already merged with the main tree:

```
- git branch -d branchname
```

- forced deletion of the local branch:

```
- git branch -D branchname
```

– deleting a branch from the central repository:

```
- git push origin :branchname
```

2.3.2. Standard operating procedures in the presence of a central repository

The work of the user with his branch begins with checking and receiving changes from the central repository (at the same time, to the local tree before starting this procedure should not be changed):

```
- git checkout master
2 git pull
3 git checkout -b branchname
```

You can then make changes in the local tree and/or branch.

After the completion of making some change to the files and / or directories of the project you need to place them in the central repository. To do this, you need to check, which files have changed so far:

```
git status
```

and, if necessary, remove unnecessary files that we do not want to send to the central repository.

It is then useful to review the text of the changes for compliance with the rules of the code.
clean commits:

```
git diff
```

If some files should not be included in the commit, then we mark only those files that whose changes you want to save. To do this, use the commands add and/or deletion with the necessary options:

```
git add...  
git rm ...
```

If you want to save all changes in the current directory, then use:

```
git add .
```

Then we save the changes, explaining what was done:

```
git commit -am "Some commit message"
```

and send it to the central repository:

```
git push origin branchname
```

or

```
git push
```

2.3.3. Working with a local repository

Let's create a local repository.

First, let's do a preliminary configuration by specifying the name and email of the owner of the repository:

```
git config --global user.name "FirstName LastName"  
git config --global user.email "work@mail"
```

and setting utf-8 in git output:

```
git config --global quotepath false
```

To initialize a local repository located, for example, in a directory ~/tutorial must be typed on the command line:

```

1 cd
2 mkdir tutorial
3 cd tutorial
4 git init

```

After that, a `.git` directory will appear in the tutorial directory, in which it will be stored change history.

Let's create a test text file `hello.txt` and add it to the local repository:

```

- echo 'hello world' > hello.txt
2 git add hello.txt
3 git commit -am 'New file'

```

Let's use the `status` command to view the changes in the working directory, data since last revision:

```

- git status

```

While working on a project, one way or another, files can be created that are not needs to be added to the repository later. For example, temporary files created by editors or object files created by compilers. Can prescribe templates of file types ignored when adding to the repository in the file `.gitignore` with services. To do this, you first need to get a list of available templates:

```
curl -L -s https://www.gitignore.io/api/list
```

Then download the template, for example, for C and C++

```

- curl -L -s https://www.gitignore.io/api/c >> .gitignore
curl -L -s https://www.gitignore.io/api/c++ >> .gitignore

```

2.3.4. Working with the repository server


For subsequent user identification on the repository server, it is necessary generate a key pair (private and public):

```
1 ssh-keygen -C "First Name Last Name <work@mail>"
```

The keys are stored in the `~/.ssh/` directory.

There are several repository servers available with the option of free data placement. For example, <https://github.com/>.

To work with it, you must first create on the website <https://github.com/> accounting record. Then you need to upload the public key we generated earlier. For this go to <https://github.com/> under your account and go to the GitHub setting menu. After that, select SSH keys in the GitHub setting side menu and click

Add key . By copying the key from the local console to the clipboard

```
1 cat ~/.ssh/id_rsa.pub | xclip -sel clip
```

paste the key into the field that appears on the

site. After that, you can create a repository on the site by selecting Repositories Create

Repository in the menu. To upload the repository from the local directory to the server,

do the following:

commands:

```
- git remote add origin
2  ssh://git@github.com:<username>/<reponame>.git git push -u
3  origin master
```

You can then follow the standard procedures for working with git on your local computer if you have a central repository.

2.4. Gitflow workflow

Gitflow workflow. We will describe it using the git package flow.

2.4.1. general information

- Gitflow Workflow published and popularized by Vincent Driessen of Gitflow vie.
- Gitflow Workflow involves building a strict branching model, taking into account project release.
- This model is great for organizing a relay-based workflow call.
- Working on the Gitflow model includes creating a separate branch for bug fixes in the working environment. – The sequence of actions when working according to the Gitflow model: – A develop branch is created from the master branch. – A release branch is created from the develop branch. – Feature branches are created from the develop branch. – When work on the feature branch is completed, it is merged into the develop branch. – When work on the release branch is completed, it is merged into the develop and master branches.
- If a problem is found in master, a hotfix branch is created from master. – When the hotfix branch is completed, it is merged into the develop and master branches.

2.4.2. Software installation

- For Windows, the Chocolatey package manager is used. Git-flow is included git package.

```
1 choco install git
```

– For MacOS, use the Homebrew package manager.

```
1 brew install git-flow
```

– Linux

– Gentoo

```
1 emerge dev-vcs/git-flow
```

– Ubuntu

```
1 apt-get install git-flow
```

2.4.3. Workflow with Gitflow

2.4.3.1. Main branches (master) and development branches (develop)

To commit the history of the project within this process, two branches are used instead of one master branch. The master branch holds the official release history, while the develop branch is for merging all the features. In addition, for convenience, it is recommended to assign a version number to all commits in the master branch.

When using the git-flow extension library, you need to initialize the structure in an existing repository:

```
git flow init
```

For github, the Version tag prefix should be set to v. After that, check which branch you are on:

```
git branch
```

2.4.3.2. Feature branches

Each new feature should have its own branch that can be pushed to a central repository for backup or team collaboration. Feature branches are created not on the basis of master, but on the basis of develop. When work on a feature is completed, the corresponding branch is merged back into the develop branch. Features should not be pushed directly to the master branch. Typically, feature branches are created from the latest develop branch.

Creating a feature branch Let's create a new feature branch:

```
git flow feature start feature_branch
```

Further we work as usual.

End of work on a feature branch Upon completion of work on a feature merge feature_branch with develop:

```
git flow feature finish feature_branch
```

2.4.3.3. Release branches

When the develop branch has enough features to release, the develop branch a release branch is created . Creating this branch starts the next release cycle, and from that moment on, new functions can no longer be added - only debugging is allowed, creating documentation and solving other problems. When the release preparation is completed, the release branch is merged into master and given a version number. Then you need to merge with the develop branch, in which, since the creation of the release branch, changes occur.

Due to the fact that a special branch is used for preparing releases, one a team can refine the current release while another team continues work on features for the next one.

You can create a new release branch with the following command:

```
git flow release start 1.0.0
```

The following commands are used to complete work on the release branch :

```
git flow release finish 1.0.0
```

2.4.3.4. Hotfix branches

Support branches or hotfix branches are used for quick fixes to working releases. They are created from the master branch . This is the only thread that must be created directly from master. Once the fix is complete, the branch should be merged into master and develop. The master branch must be tagged with updated version number.

Having a dedicated bug fix branch allows the team to resolve issues without interrupting the rest of the workflow or waiting for the next cycle.

release.

A hotfix branch can be created with the following commands:

```
git flow hotfix start hotfix_branch
```

When finished, the hotfix branch is merged into master and develop:

```
git flow hotfix finish hotfix_branch
```


2.5. Exercise

– Create a basic configuration for working with git. – Create an *SSH key*. – Create a *PGP key*. – Set up git signatures. – Register on *Github*. – Create a local directory for completing assignments in the subject.

2.6. Work sequence

2.6.1. github setup

1. Create an account on <https://github.com>. 2. Fill in the basic details on <https://github.com>.

2.6.2. Software installation

2.6.2.1. Installing git-flow on Fedora Linux

– This software has been removed from the repository. – You must install it manually:

```
1 cd /tmp
2 wget --no-check-certificate -q https://raw.githubusercontent.com/petervanderdoes/gitflow/develop/contrib/
  gitflow-installer.sh 3 chmod +x gitflow-installer.sh 4 sudo ./gitflow-installer.sh install stable
```

2.6.2.2. Installing gh on Fedora Linux

```
sudo dnf install gh
```

2.6.3. Basic git setup

– Set the name and email of the repository owner:

```
- git config --global user.name "Name Surname" git config --
2 global user.email "work@mail"
```

– Set up utf-8 in git output:

```
- git config --global core.quotePath false
```

- Set up verification and signing of git commits. – Set the name of the initial branch (we will call it master):

```
git config --global init.defaultBranch master
```

– Parameter autocrlf:

```
git config --global core.autocrlf input
```

– safecrlf parameter :

```
git config --global core.safecrlf warn
```

2.6.4. Create *ssh* keys

– using the *rsa* algorithm with a 4096-bit key:

```
1 ssh-keygen -t rsa -b 4096
```

- according to the *ed25519* algorithm :

```
1 ssh-keygen -t ed25519
```

2.6.5. Create *pgp* keys

- Generating a key

```
1 gpg --full-generate-key
```

- Choose from the following options:

– type *RSA and RSA*;

- size 4096; - select

the expiration date; default value is 0 (does not expire never).

– GPG will ask for personal information, which will be stored in the key: –

Name (at least 5 characters). – Email address. – When entering an email, make sure it matches the address used on

GitHub.

- Comment. You can type anything or press the enter key to leave this field blank.

2.6.6. Adding a PGP Key to GitHub

– Display the list of keys and copy the fingerprint of the private key:

```
1 gpg --list-secret-keys --keyid-format LONG
```

– A key fingerprint is a sequence of bytes used to identify longer than the key fingerprint itself.

- String format:
sec Algorithm/Key_Fingerprint Creation_Date [Flags] [Expiry_To]
key_id
- Copy your generated PGP key to clipboard:

```
gpg --armor --export <PGP Fingerprint> | xclip -sel clip
```

- Go to GitHub settings (<https://github.com/settings/keys>), click on the *New GPG key* button and paste the received key into the input field.

2.6.7. Set up automatic git commit signatures

- Using the email you entered, tell Git to use it when signing commits:

```
git config --global user.signingkey <PGP Fingerprint> git config --global  
commit.gpgsign true 3 git config --global gpg.program $(which gpg2)
```

2.6.8. Setting gh

- First you need to log in

```
gh auth login
```

- The utility will ask some leading questions. – You can log in through a browser.

2.6.9. Workspace Template

- Repository: <https://github.com/yamadharma/course-directory-student-template>.

2.6.9.1. Course Repository Consciousness Based on Template

- You need to create a workspace template. – For example, for the 2021-2022 academic year and the subject "Operating Systems" (code object os-intro) the creation of the repository will take the following form:

```
1 mkdir -p ~/work/study/2021-2022/"Operating Systems" 2 cd ~/work/study/  
2021-2022/"Operating Systems" 3 gh repo create study_2021-2022_os-intro -y --  
template=yamadharma/ course-directory-student-template --public 4 git clone --  
recursive y git@github.com:<owner>/study_2021-2022_os-intro.git os-intro
```

2.6.9.2. Setting up the course directory

- Navigate to the course directory:

```
1 cd ~/work/study/2021-2022/"Operating Systems"/os-intro
```

- Delete extra files:

```
1 rm package.json
```

– Create the necessary directories:

```
1 make COURSE=os-intro
```

– Send files to the server:

```
- git add . git
2 commit -am 'feat(main): make course structure' 3 git push
```

2.7. Report content

1. Title page indicating the number of laboratory work and the name of the student. 2. Formulation of the purpose of the work. 3. Description of the results of the task: - screenshots (screen shots) that record the performance of laboratory work; – listings (source code) of programs (if any); – results of program execution (text or screenshot, depending on

tasks). 4.

Conclusions agreed with the purpose of the work. 5. Answers to control questions.

2.8. test questions

1. What are version control systems (VCS) and what tasks they are designed to solve hope?
2. Explain the following VCS concepts and their relationships: repository, commit, history, working copy.
3. What are centralized and decentralized VCS? Give examples of each type of VCS.
4. Describe the actions with the VCS when working with the repository alone. 5. Describe how to work with VCS shared storage. 6. What are the main tasks performed by the git tool ? 7. Name and briefly describe git commands.
8. Give examples of use when working with local and remote repositories
- riami.
9. What are branches and why might they be needed? 10. How and why can some files be ignored during commit?