SDA Sp. z o.o.
Al. Zwycięstwa 96/98
81-451 Gdynia, Poland

+48 732 489 313
info@sdacademy.pl
sdacademy.pl

software
development
academy

Table of contents:

1. GIT introduction

Hello, welcome to the Git course. My name is Marcin Chadkowski and I hope that the series of few videos contained in this course will bring you closer to the subject of GIT which is extremely important, especially when working in a team regardless of the programming language you use so let's get started.

2. Command line

a. Command line - installation

To download the git, we need to go to the git-scm.com website.
In the lower right part of the website, you will see a button that will allow us to download the git installer. At the time of recording this video, the latest released version is 2.23.0. After clicking this button, the download will start and we will land on a page that tells us what other versions of the git we can download. When the download is complete, click the file showing at the bottom. If you are asked for administrator rights, agree to grant them this application during installation. After carefully studying the license agreement, click next. Then select the directory where you want to install git. In my case, such a folder already exists, so yes, I agree that it will be replaced. If you want, you can configure additional installation options here. Then choose what git name should be available as a shortcut from the Start menu. The next step is to choose the default text editor. In our case, we will use vim as the default editor for the git, however, if you have another editor, such as notepad ++, you can click on the drop-down list and select it from those available. If you don't have this editor, you can abort the installation, install it and then start the git installation process again, at which point you will be able to choose the text editor notepad ++. This program is an extended version of the notebook that you can know from windows. We do not change any other installation options, going to the next stages with the next button. On the last screen of the installer, select the install button. Now we can finish the installation process with the finish button. After installation, you should be able to launch an application called git bash from the start menu. This is the tool you will use in this tutorial (and probably also often outside of it) to work with git. After running git bash, with the git - version command, we can check if and what version of git has been installed. We see that the installed version corresponds to the one we downloaded.

b. Command line

Hi, In this video, I will tell you about the command line - How to communicate with a computer. From the very beginning of the existence of computers. One of the most important needs during their use was the need to communicate

with them. How would we give orders to computers? How to tell them what we need from them? How to say what we expect from them? And also how to understand what they want to tell us? How to read the results they provide? At the very beginning of the computer age, this was done by setting the appropriate states in the computer and then reading the results via the complex lamp system. One of the next steps in the way of communication with the computer was printing results, and the data was given to him, for example in the form of punched cards. However, this communication was extremely ineffective, fortunately, it improved with the invention of the monitor, on which the computer could inform us about the progress of its work in text format, and on which commands entered by the user using the keyboard could be displayed. Later more and more developed operating systems began to appear, such as the screenshot from ms-dos presented here

Over time, communication with the computer took the graphic form as we know it now. Where all exchange of information takes place via the graphic interface, windows, buttons, and so on.

The Graphical User Interface appeared in the Windows 95 version and all its subsequent editions However, communication with the operating system using text commands is still an important form of information exchange with a computer. An example of such communication is the command line in Windows. To work with git we will also need a console, reminiscent of some Unix systems, and because many of us have Windows, we will download a tool that will help us in this. You will find the git installer under the link on the screen. After downloading the appropriate version and running it, an installer window will open that will guide you through the installation process. After completing it, you should be able to launch an application called git-bash from the start menu. It is this tool that we will use in this course (and probably also often outside it) to work with git.

Here are the basic command line commands. The pwd command will allow us to Print the directory in which we are currently located, in which all further commands entered by us will be executed. The LS command will allow us to list the contents of the directory we are currently in. The mkdir command will create a directory with the given name, which is subordinate to the directory in which we are currently located. The Touch command will create an empty file with the given name if it is not already in the folder in which we are. The mv command will move the file at the specified path to the specified destination. The rm command is used to delete the specified file or directory. The cp command allows you to copy an existing file or folder to the location of your choice. These commands are only a fraction of all the commands available to use. However, I list them here because they are the basis for working with the console. And now we will see the application of the above commands in practice.

c. Command line - demo

Start the git bash console. With the pwd command we can print the directory path to where we are now. ls command will write us the contents of the directory. ls command run with the flag minus l will write us the contents of the directory in separate lines. And running with the -a flag it allows you to list all content, including hidden files and folders. Used flags can be combined. The ls command with the simultaneous use of the flags -l and -a will allow us to list the entire contents of the directory, including hidden files and directories. In separate lines the flags used can be combined by putting the appropriate arguments next to each other, after a single minus sign. If we want to check, what are the possible options to use in a given command, let's use the --help option, which will write us documentation for this command. The cd - change directory command - allows us to change our locations, e.g. to another folder where we want to do work. Two dots refer to the parent directory of ours and a single dot to the directory in which we are currently located, i.e. the cd command .. will move us from the c / users / user directory to c / users, the folder we are currently in is also written in yellow font at the end of the line that appears after each command issued. If we want to go to some parent directory, e.g. User , we need to use the cd command and enter the name of the destination folder, in our case User. As we can see, after moving to the user directory, at the end of the next line instead of the current path to the directory in which we are, the tilde is displayed - it means that we are in the user's home directory which can be confirmed by viewing the contents of this directory Let's now go to the user's desktop directory, i.e. to the desktop Using the mkdir command we create a directory with the given name. Let's create a folder called a repository. Let's check if it is listed. It was actually created. Let's go to him now and see what's inside. It's empty for now. Let's go to the directory above and with the touch command create a file.txt and then check if it appears in the current folder it is listed, which means it was actually created with the mv - move - command. Let's move our file.txt to the repository directory. In the first place after the command, you have to say what we want to move, and in the second where we want to move it. Check the contents of the current directory. There is actually no file.txt. We go to the repository directory and there we check whether the file is present to delete a file, use the rm command and then give the file name to be removed. Check if it disappeared from our directory. Yes, it is not there anymore. Go to the parent directory and try to delete the entire repository folder now. The command did not work on the directory. Let's check the documentation of the rm command. We can see that to delete a folder you need to use the minus r flag. So let's use it to delete the repository directory. Let's see if he disappeared. Yes, it is not in our catalog now. Let's create the repository directory again. Let's also create the file.txt back with the cp command - that is copy - we can copy the file.txt to the repository directory We see that despite copying, the.txt file remains in our

current directory. We check if the file.txt has appeared in the repository directory. Yes, It has been successfully copied.

3. Version control system

Hi In this video, I will tell you about version control systems and the problem they solve, which is the problem of versioning files.
Imagine working on a certain file in which you will make changes repeatedly. For example: You decide to create a file to put the text "Ala". However, after a while you find that it would be better to add that Ala has animals - you have to change it. Soon after you notice a mistake - Ala really only has a cat but no dog - you need to change it in the file. What if at the very end we find that it would be best to return to the first version of the file? We can only do this if we have that version of the file saved somewhere. And such problems are solved for us by version control systems. The version control system is a software used to track changes in files over time, it also allows restoring older versions of files and viewing the history of changes. Version control systems allow restoring the entire project from earlier stages of work, e.g. from several weeks or months ago. In addition, you can compare the changes made. Thanks to this, in the event of any errors or data loss, they can be restored and repaired. Thus, the main functions of version control systems are: - tracking the history of changes, along with information about who and when introduced them. The ability to restore any version of a file or project.

The function of synchronizing changes introduced by various authors in a distributed environment - the basis for team work using remote repositories. There are two different approaches to code versioning: The first is a centralized version control system. Server and client exist in centralized version control systems. The server is the main repository that contains all versions of the code. To work on any project, the user or client must first obtain the code from the main repository or server. Thus, the client communicates with the server and downloads all code or the current version of the code from the server to the local computer. In other words, we can say that you need to download the update from the main repository and then get a local copy of the code on your system. So when you get the latest version of the code, you start making your own changes to the code, and then you just need to commit them directly in the main repository. Approving the change simply means merging your own code in the main repository or creating a new version from the source code. In this model, everything is centralized, which can sometimes be problematic, because data can be lost in the event of a central server failure. However, this is a risk that always exists if the entire history of the project is in one place.
The second approach is distributed version control systems. In distributed version control systems, most mechanisms work the same as in centralized ones. The only significant difference you will find here is that instead of one

repository, which is the server, here every programmer or client will have a copy of the version of the code and all its branches on their computer. Basically, any customer or user can work locally and without a connection, which is more convenient than centralized source control and is therefore called distributed. You don't have to rely on a central server, you can clone the entire history or copy of the code to your hard disk. When you start working on a project, you clone the code from the remote repository to your own hard disk, and after making changes to it, you commit the changes and at this point your local repository will have a "set of changes", but will still be in a different state from the remote repository, so that is "align", you push the local repository code into the remote repository. Downloading changes from the repository is called "pulling" and merging local "changesets" into the remote repository is called "pushing".
The advantages of distributed version control systems include:
- the ability to make changes without connecting to a remote server - you can be offline
- speed of work - unlike centralized systems, you don't have to communicate with a remote server with every command - everything happens locally, with us
- data security - each of the developers has a local copy of the repositories, so even if the central server fails, it is possible to recover data.

4. What is GIT?

Hi In this video I will tell you what git is and what it allows us to do. GIT is an example of a distributed version control system. It was created by programmers working on the Linux kernel in the year two thousand and five.
It allows us to manage the source code by offering us:
- speed
- efficient work with large projects
- and full dispersion

In addition, git is used by many people around the world who create a huge community - which enables a valuable exchange of experiences, good practices and easy help in case of problems. Equally important, there are many hosting sites on the network for git repositories - for example: Github, BitBucket or GitLab - you've probably heard about them somewhere, or read something about them. In the next videos we will start learning how to work with it.

5. Creating local repository

Hi In this video I will tell you about what repositories are and how to create them on your computer. The repository is the source code and all information

about changes made to it. The repository created on your computer will be called a local repository. To create such a local repository, we need to use the git init command in git bash, let's see how it looks in practice.

6. Files lifecycle

Hi, in this video I will tell you about the possible states in which files may be found. We can divide our local repository into three parts:
Working directory - that is the place where our project files are located
staging area - is a store sometimes also called an index - there are files waiting for the changes to be committed
and *git repository* - it's the git folder in which all information about our project is contained.
In this arrangement, all the changes we make happen in the working directory. To add them to the staging area we must use the git add command. Then, if we want to save these changes to our repository, we must use the git commit command.Then we can repeat the whole process. We'll talk about the git checkout command shown here later in the course. Now let's discuss the possible states of files in our repository. Modified state - in this state is any file in which we make any changes, provided that it is already tracked by git then with the help of the git add command we add this file to the staging area and its status changes to staged - i.e. prepared for being committed with the next commit finally, using the git commit command, we commit the changes, saving them to the repository. What about when we add a new file to our project that has never been tracked by a git before? Then it displays in a git with the status untracked. Here, too, we use the git add command to add this new, freshly added file to the staging area. Let's check how it all looks in practice. Then let's go to the Desktop directory, using tab helps us to navigate in the console. A tab is a tool for suggesting phrases that we can use at a given time. If I write: cd repo, then press tab, the console will automatically suggest the rest of the directory name. If more than one option will fit, after the first press tab, the autocomplete won't work. Then we can press Tab twice quickly, and we will be given a list of possible choices to pick from. We'll see more examples of this in further videos. Let's see the status of the repository with the git status command. We can see that at the moment there were no commits, and there is nothing that could be committed. Here as well we can use the -help parameter to find out how we can use the git status command. It opens its documentation, which is saved on our disk. Let's create a file.txt, we see that it was created, let's check the status of the repository. We can now see that in the Untracked files section our newly created file has appeared - this means that at that moment git won't track changes occuring in it. In addition, under the name of the Untracked section, git showed us a hint on what to do. To add files to the tracked files. Let's use this hint. Using the git add command, we'll start to track changes occurring in this file- it will have a staged status. Let's

see the status of the repository. Your file is now in the "changes to be committed" section - that is, it will be committed with the next commit command. Let's commit the changes, using the git commit command. In addition, we will use the -m flag to be able to give title to our commit, enclosed in quotation marks. Yet we are missing something, so that we can confirm the changes we lack the name and the email address, which will be used to sign all the changes that we make - these are very important settings git immediately tells us how this todo -let's use his tips with the git config command, we can assign appropriate values to various Git settings. Git config --list will print our current git configuration. We can now see user.email and user.name entries set to proper values if you do not have such an entry as core.editor and a value assigned to it. You don't have to worry about it, I'll talk about this later. We're back to our repository. Let's try to commit again. So do we have to re-enter the git commit command? You can do it, but you can also use arrow keys to navigate through the history of given commands, then click enter to execute the command selected from the history. I found the right command in history, that is git commit, And I used it let's see the status of the repository after executing the git commit command. As you can see our working directory is empty again, and the whole process can be started over. Now we make changes to the file file.txt we will use the vim command for this, which opens the default text editor for git bash, named vim. It's a kind of Microsoft word, only in the console. vim file.txt command will open the file.txt in the editor. Vim does not have a graphical interface as we know it, e.g. from word instead, we work there using text commands. At first, after opening any file, we are in watching mode, meaning we can not write anything. To begin to write, we need to press the 'i' key and then enter the text. To get out of vim and save your changes, we must press the keys one after the other. Escape button exits editing mode, and everything we write after clicking it, will appear at the bottom of the console window, take a good look. Colon, 'w' and 'q', allows us to exit vim 'w' stands for write - changes are saved and 'q' stands for quit - exit from the program confirming such a string characters with enter, will close vim. I will run vim again for a moment to see if the changes are saved. Yes, there are our changes, so we press escape, colon, 'q' and then enter and exit vim. Okay, we did some changes in the file tracked by git. Let's check the status of the repository. We can see that our file.txt has the modified status according to the hints of git, if we want to add it to the staging area, then we have to use the git add command. Git add command can be used with the name of the file you want to add, however, if we want to add more than one file, e.g.file all of the current catalog, we can use the dot here. Let's see if the file has been added. It's great, now we can commit changes with the git commit command. Remember as I said before, to use the git commit command with -m flag to immediately provide a comment to commit? Let's see what happens if I run the git commit command itself, without any parameter. A text editor has been opened, in my case it's notepad ++, because that's how I have

configured the git- that's what the entry core.editor in the settings is for. In your case, a plain console text editor might have been opened, similar to vim. No matter what text editor has been opened, it happened because we did not specify the message for our commit. Until we fill the commit message, the commit is stopped by the Git. Now we have to complete the commit message - i.e. a short description of the changes. Now we need to save the changes then close the program. Commit has been completed, the changes have been approved. Let's see the state of the repository. It is empty, which confirms that the commit actually has been performed. Lets add file newFile.txt. Lets check the status of the repository. Let's add some content to this file. We save the changes and leave. Lets see the status of the repository again. Let's add the file to the staging area.
Let's see the state of the repository. We will introduce changes in the file. We can add something to the file. Or delete everything and replace it with something else. Now let's see the status of the repository. Initially it may seem that this file is in a state of staged and modified at once. The truth is that the file version from earlier is staged - the one with two lines of text. And in the modified state is the latest version of the file in which we replaced these two lines with one new one. Let's add the latest, changed version of the file, to the staging area. We can see that now this file is listed only once, in the files that will be committed with the next commit. At the end, we will commit our changes and check the status of the repository.

7. Undoing a file_s state

Hi, in this video I will tell you what to do in a situation where we made wrong decisions and now we want to cancel them. Imagine a situation in which you accidentally added a file from a working directory to a staging area, and in the middle of that file, you forgot to delete ugly comments. Do you have to commit this file so that your colleagues can see what you wrote? Luckily no, the git reset command can help us here, used on the file it will work the opposite of the git add command, it will withdraw our files from the staging area to the working directory. Let's see how it looks in practice.
Let's open git bash and go to the directory with our repository. Currently there are no changes in it, let's open the file.txt in vim and make some changes in it, save the changes and leave, this file now has the status of modified, let's use the command git add, we can see that the file is now in the staging area. If we change our minds and want to undo this file from the staging area to the working directory, we must use the git reset command, what git suggests. This command will work just the opposite to the git add command**.** Now imagine a different situation. While working on the project, we create experimental functions, not knowing if it will ever work or whether it will ever be useful to us. After a few hours, however, we find that our efforts do not lead to anything, and what we have changed so far in the file should be discarded, and we should

return to the proper working version. In this situation, the git checkout command will help us. Used on a file, it will allow us to restore the state of the file to the form saved by the git. Let's see how it looks in practice. Let's launch git bash and go to our repository. Let's change the content of the file.txt. Let's see the state of the repository, now we add the file to the staging area, let's commit the changes, let's change the content of this file again , what to do in a situation when we find that the changes just introduced are incorrect and we would like to go back to the state of the file that was confirmed at the last commit, then again we can use the hints of git, this time using the git checkout command on the file to discard the changes made to it and restore its state to the last saved by git, after checking the state of the repository we can see that the changes have been removed, there is nothing to commit, and the file again has the content we recently committed, what we see in vim. Another situation that will meet us in our daily work is the need to remove the file from our project. This is done with the git rm command, let's see how it looks in practice. Let's open git bash and go to our repository, there are no changes in it for now, with the git rm command we will remove the newFile.txt. We see that it actually disappeared from the directory and git status shows us information about the fact that we actually deleted this newFile.txt. What if we change our mind and want to keep this file? Then, as prompted by git, we must use the git reset HEAD command with the name of the file to be restored. We can see, however, that this file has not returned to our directory yet. We have just moved the change back from the staging area to the working directory to actually restore the file itself, we must use the git checkout command with the name of the file on which this command should run only then our changes- in this case, deletion- will be completely withdrawn and the file will return to our directory.

8. Change story

Hi, in this video I will tell you how to review the history of changes and undo unwanted commits. First, let's specify what commit is, because until now we have already used the git commit command, which just creates them, but we never really said what they are. Commit is a snapshot of changes from the staging area. The git commit command remembers a project snapshot, along with additional information, including the number of changed files, statistics of added and removed code lines, as well as the SHA-1 checksum of changes. And it is by using the checksum git is able to identify and distinguish individual commits. Let's imagine that we are making changes to the project and we make three commits, first, second and then third. Then their story could look like the picture. However, as I mentioned earlier, git identifies commits not by their order or by comments, but by their checksum, so an illustrative drawing of the commit history could look like this. You may be wondering why only the first few characters of the checksum are visible in the drawing, not the whole. In general, the checksums calculated by the git are usually much longer and at

the same time so diverse that the identification of changes is possible knowing only the first few characters of the sum. After creating several commits or after downloading the repository, we may need to review the history of existing commits to understand what is going on in the project. The most basic and also very useful is the git log command. It prints the list of commits made in the repository we are currently in, from newest to oldest. Additionally, information about a given commit is attached, such as: checksum, author, data and commit message. Optional arguments with which we can use the git log command are:
--patch - displays changes made in every commit.
--stat - displays some statistics, e.g. the number of lines changed
--oneline - displays each commit in one separate line.
Open the git bash and go to the directory with our repository
With the git log command, we can see the history of commits in our repository, in order from the latest on the top, to the oldest below
What we can see here is:
-the checksum of each commit - which is used by the git to distinguish commits
- the author of commit - this is where the previously configured user.name and user.email show up
- date- that is when the commit was made and the commit message that we have to provide for each commit- i.e. a brief description of changes. Running the git log command with the minus minus oneline parameter will allow us to print the history of our repository in a short, single-line form. We will see the beginning of the checksum of each commit and its message. Running the git log command with the minus minus patch parameter will additionally show us the changes introduced in each commit. If the information is too much to be displayed on the screen, we can navigate through them using arrows on the keyboard, or by using the page up and page down keys.To quit browsing information, press the 'q' key. Running the git log command with the minus minus stat parameter will show us the commit history along with some statistics regarding the changes, e.g. the number of lines changed in the file.
The git add command can also be used with the minus minus author parameter, to see commits made only by the given author. We can use more than one parameter, just as we could do in other command line commands. Let's see, for example, only commits made by the author user, but we want to receive information in a short, one-line form, we can add a minus minus patch parameter to see the changes made in these commits - still in the short form but now instead of the changes itself, let's see some statistics on changes. If we want to limit the number of commits displayed, we can do it by appending the minus and then the number commits to be shown to the command. here: 3. Another useful parameter of the git log command is the minus minus graph parameter, which allows us to draw the commit history in graph form. Here, however, we do not see the full potential of this option, because the commit

history is linear at the moment. We will see the usefulness of this command in further videos. Here we can also use two parameters, in this case combining the graph parameter with the oneline parameter to get a clearer graph. If we want to narrow down the results only to the commits that have affected a particular file, we need to add minus minus at the end of the command, and then enter the name of the file to which the commits are to refer, after the space, of course, we can still use other parameters of the git log command such as e.g. patch or online there is also a parameter minus minus format that will allow us to freely configure the way the information will be displayed, for more information, see the git documentation - there is a sea of possibilities here. Will now see how to use the git checkout command to undo the repository to the selected commit. Start the git bash console and let's go to the directory with our repository. Let's see the contents of the file.txt. Let's see the commit stories. Let's assume that we want to go back with the state of our repository, i.e. our files, to the version it was in after the second commit, the one with the checksum starting with characters f,1,9,2, 7 in this case, let's copy the beginning of this commit's checksum and use it in the git checkout command, pasting it at the end. We also see information that we are now in a detached HEAD state, it means that the changes we will introduce here will not belong to any branch and will be lost if we switch to another commit or branch, unless we create a new branch from this place. Let's check the commit history after we have used this command. We see only two commits, which proves that the git checkout command moved us back to the state of files from commit starting with f 1 9 2 7, when the next, newer commits did not yet exist. Let's look at the contents of the file.txt - we can see that there is a very old version of content with the git checkout master command, we can go back to the 'present' - that is, to the latest commit we have made. The git log command now shows many more commits, more precisely all done on the master. And, in file.txt we can see its current state, which was influenced by all commits from the one starting with the characters e d a b, up to the newest, which starts with 9 8 f b.
During everyday work, it may happen that you need to undo changes introduced by one or more commits - for example, it may turn out that the functionality introduced in the past contains security flaws and you need to withdraw it. In this situation, we'll need the git revert command. It allows you to reverse changes from the selected commit. The choice of commit that we want to reverse is made by giving the first few characters of the checksum of this commit.Then running the git revert command will create a new commit that will contain the changes inverse to the removed commit.

Let's see how it looks in practice.
Start the git bash console and let's switch to the directory with our repository, let's see the status of the repository, let's change the file.txt, wanting to add some new, experimental functionality, let's add a file to the staging area and

let's commit the changes, let's see the commits history. What if after some time we discover that the introduction of this experimental functionality caused huge problems in our project, which we did not notice at once? But we don't have time to fix them, because in 5 minutes we perform in front of the audience that we want to present our project to? Then we'll need the git revert command, which together with the commit identifier allows us to reverse the changes that have occurred in this particular commit. This command in its operation creates a new commit, which will contain changes reversing the effects of the commit, which we want to get rid of - that's why the text editor has opened, so that we could give a message for this new commit, which will reverse the changes. Will use the message suggested by the git, save the changes and close the text editor. Let's see the history of our commits. Seeing the contents of the file.txt finally confirms that the revert command allowed us to remove the experimental changes that we wanted to get rid of.

9. Creating remote repository

Hi, in this video I will show you how to create a remote repository. To be able to collaborate with your teammates you must be able to push your changes outside and also download updates created by your colleagues. This is what the remote repository is for - available over the internet or another network - it enables code sharing.
The most popular git repository hosting services are
GitHub
GitLab
BitBucket

Remote in a git is something similar to a pointer, pointing to another repository from which you may want to download the code or to which you may want to send the code. Command: git remote - print the short names of all specified servers for this repository and the git remote command used with the -v flag also prints the URL assigned to these names. Let's see how it looks in practice.

Go to github.com, if you don't have an account, create one to continue. I already have an account, so I will go to the sign-in option - where the logging in is. We log in with the same data we provided during registration, after logging in we land on our home page. On the left side we usually can see a section with the links leading to our repositories, but since it is our first visit, we have a bit of a different layout. Let's click the "create repository" button. In the repository name, we give the name we want to give our remote repository. If we want, we can give a brief description of our project in the description field. Then we have to choose whether our repository will be public - that is, which everyone can see or private, to which only we and people of our choice will have access to. The initialization with a readme option is good when you

create a remote repository first and then you want to create a local one. Then github will show you further instructions that you need to follow to get the job started. However, we already have an existing repository, so I leave that option unchecked. When you've set everything up, click the create repository button. Here we see the help from github, if we would like to create a new repository from the command line and connect it with the newly created github repository. There is also a hint section for people like us who want to send an existing local repository to github. Let's copy the selected fragment. Open git bash and go to the directory with our local repository. Then paste the git remote command copied from github, which will allow us to set the remote called 'origin' so that it leads to our remote repository on github - which in my case has the address as shown on the screen, then let's return to github and copy the second, last command. Pasting them into the console and approving them with the enter button will send our local changes to the server. That's the way the git push command works - it takes the branch - in our case the master - from the local repository and updates its counterpart in the repository which origin points to. If in the remote repository the counterpart of the local branch is not present, it will be created there. Before this happens, you will be asked to login to github with the same credentials as on the website after successful login, the git push command will be executed, and our changes should be visible on github after refreshing the page. Each of the files found here also has information on what was the commit message of the last commit that made changes to it, and also when it was last modified each of the files in our repository can be accessed from the github level by clicking on its name. We can return to the directory above by clicking the appropriate part of the file path. We also see the number of commits we have made so far if we click this button, their history will be shown, along with their messages, authors and creation date, clicking the first icon will copy the checksum of this commit, clicking the second one will open a preview of the changes introduced by this commit and clicking the third will open the repository directory in the state it was at that moment in the content preview we also have the option of editing it by clicking on the button with the pencil drawing, if we clicked the button with trash bin drawing, we would delete the file being viewed. Let's get back to git bash and see how the git remote command works.

10. .gitgnore file

Hi, in this video I will tell you how to prevent some files from being added to the repository. Imagine adding an existing project to be tracked through the git.
Git will be collecting everything from our working directory, which will cause a lot of problems. The mechanism for ignoring certain files by Git comes in handy. If we create a gitignore text file, then we fill each one line of this file with the correct pattern, git will ignore every file in the working directory that has a path matching any of the patterns. Patterns matching files will be ignored for

all git commands, just as if they didn't exist. If the gitignore file is placed in the main project directory, it will be referenced to the entire repository, while files in internal folders refer to only to their content.

Let's see how it looks in practice.
Let's open git bash and go to the directory with our local repository. We see that we have files in the repository a.trash, b.xml, c.config and out directory, in the out directory there are files with the extension .class. If we now use the "git add ." command, its effect would be adding all these files to the staging area. We don't want to do this as we don't want these files to be in our remote repository. How do you make git ignore these files, so that changes inside of them won't be tracked? For this, you need a .gitignore file, which we will create with the touch command. Let's open this file in vim so that we can change its content. In each line of this file we can add the filename or path pattern to the file that git should ignore. So, after typing a.trash, the file with such name and extension will be ignored by the git, after saving changes and leaving vim let's see the status of the repository. We can see that the a.trash file has completely disappeared from the list. Let's add all files with the .iml extension to the ignored ones. To do this, you need to write a pattern that will match all such files, in this case * . iml. An asterisk replaces any number of any characters here. All files with the iml extension will be ignored, regardless of their name. Let's save the changes, close vim and see if our repository has changed. The result of the git status command is the same as last time, but now, in case of creating a file with the .iml extension, from the beginning of its existence it will be ignored by git. Let's also ignore files with the .xml extension. We can see that the b.xml file is no longer tracked by the git. Now we also get rid of files with the .config extension. In the end , let's also make the whole out directory to be ignored too, if we want, we can also add .gitignore file to the ignored ones, so it will not appear on the list of files possible to add into a commit.

11. Creating branches

Hi, in this video I will tell you how to create branches and work on them. Branch - this is a ramification (branch?) in our project. With each commit, the branch automatically moves forward, i.e. after the next commit, the master will point to the latest commit. And it will happen every time we commit some changes. If we want to create a new branch, we must use the git branch command along with the name of the branch we want to create. Let's assume that while working on the repository from previous slides, we want to create a branch called testing, in which we will test new functions that we want to include in our project. We do this because in this way we will always be able to come back to branch master in case our changes in branch testing turn out to be broken. To create a new branch called testing, in git bash we have to enter the command: git branch testing and our branch structure after this command will be as

follows. But how does git know which branch you're currently working on? It has a special HEAD pointer. This is an indicator of the local branch you are on. In our case, we're still on the master branch because the git branch command created a new branch but didn't switch us to it. If we want to switch to another branch, we must use the git checkout command along with the target branch name. In our case, after switching to branch testing, our repository structure will look like this. What does this give us? Let's see what our repository will look like if I commit some new changes at this point. It is interesting because now the testing branch has moved forward, but the master branch still indicates the same set of changes as when using git checkout to change the active branch. Let's switch back to the master branch with the git checkout master command. The command did two things. It moved the HEAD pointer back to the master branch and has restored files in the working directory to the state from commit pointed to by master. It also means that the changes you make from now on will fork out from the older version of the project. It actually undoes the work you did on the testing branch temporarily, so you can go in a different direction with further changes. Let's check what our change history will look like if at this point we would decide to add some changes while still at the master branch. Now the project history has been split. We created and switched to the branch, did the work on it, and then we returned to the main branch and did another work. Both sets of changes are isolated from each other in separate branches: you can switch between them and merge them together when you are ready. And all this with the help of two simple commands: git branch and git checkout. Let's show how they look in practice.
Open git bash and go to the directory with our repository.
Let's check its contents. The git log minus Minus oneline command will allow us to see the history of our repository. Let's create a new branch with the git branch testing command. This command will create a new branch called testing. Let's see that the status of our repository has not changed. After running the git log minus minus oneline command once more, we can see that branch testing is in the same place as branch Master. And our branch's Graf is not much different yet. Let's switch now to our newly created 'testing' branch. Run from here The git log Minus minus one line command looks identical to running on the master. Let's create the fileC Dot txt. Let's preview the status of our repository. And now let's add some content to the fileC dot txt. Let's save the changes and close vim. Next, let's add and commit these changes. Now let's see what will git log minus Minus oneline command print on the screen. We see that our newly created branch testing is in a different location from the Master branch. It is all because we were in the testing branch and created a new commit. Let's switch to the Master branch now. This will be done by the 'git checkout Master' command. If we now run the git log Minus minus one line command from the Master branch, we will not see changes made to the testing branch here. Let's create fileD dot txt. Then add some content to it, save the changes and close the editor. Let's also make some changes to the fileA

dot txt. Let's see the status of our repository. Let's add both files to be committed with the next commit. And then let's commit these changes. If we now run the git log Minus minus oneline command, we will see that Master is now pointing to a different commit than it was a moment ago. He points to the newly created commit with the message "added functionality D and changed functionality a" these changes will not be available to us when we will switch to branch testing, What we can check with the GIT log command. Both branch master and branch testing have a few common commits, however, at some point, the work has forked, as you can tell by looking at the commits they are currently pointing to. Let's see what files we currently have in our directory at the time we are testing. Let's check the contents of the fileA dot txt with the Cat command. Cat command allows you to print the contents of the selected file in the console window. Let's switch to the Master branch, and let's look at the fileA dot txt again. We can see how this file differs from the version stored in branch 'testing'.

12. A - Merging branches

Hi, in this video I will tell you about merging branches and conflicts when merging them. Continuing work on our previous project, we can work independently on master and testing branches. It is worth noting here that the work done on the testing branch is not included in the files on the master branch, and vice versa. What if we find that work on the testing branch's functions has been completed? Then we want to transfer the changes from branch testing to branch master. To do this, we must first of all be on the branch to which we want to merge the changes, in our case to the master. In our case, we're already there, but if we weren't, we'd have to use the git checkout master command. Then we need to use the git merge command.This command appends the changes from the branch that we listed at the end of the command, to the branch we are currently in. Let's see what will be the result of the git merge testing command - used in our project. Git has automatically created a new commit that is the result of merging two branches, testing, and master. This commit is called a merge commit - it is unique because it has more than one parent, as seen in the picture. The newly created commit has two arrows coming out of it, to its previous commitments, i.e. to parents.The whole branches joining process is commonly called merging.When changes have been merged, we no longer need branch testing. To remove the selected branch, we must use the git branch command with the minus d argument and the name of the branch that we want to delete. Let's see how it all looks in practice. And now let's see how the merge goes on in a conflict-free situation. Open git bash and let's switch to the directory with our repository. With the git branch command, we can list existing branches. Let's check the contents of our catalog while we're on the Master branch. Here we

see the fileA dot txt. FileB dot txt and FileD dot txt. Note that the file C dot txt is not here. Now let's switch to the testing branch.
If you don't know which branches you can switch to using the git checkout command, Double-clicking the tab will help us here again. Thanks to this, we will see options we can use on the screen. In our case, we'll use the hint with the testing branch. The git branch command will list the branch list again and notice that the asterisk placed on branch testing suggests that we are on it. Let's check the contents of our repository, being in the testing branch.
We see that there is a fileA dot txt, FileB dot txt, and File C dot txt. There is no fileD dot txt here. The differences in comparison with the Master branch are visible at first glance: the presence of the fileC dot txt and the lack of the fileD dot txt. With the git status command, we check that there is nothing to commit. Let's return now to the Master branch. While on the Master branch, let's try to merge the testing branch into it. After entering the git merge testing command, the text editor opens. This happens so that we can complete the message of the commit that will be created as a result of merging the testing into the master. I will use the message proposed by git, save the changes and exit the editor. In the console window, we can see now that the merge went right. Its result is the addition of a C file dot txt file. We see that in our branch Master there is nothing to commit. And after seeing the contents of our directory, we see that file C dot txt appeared here. Let's check the commit history with the git log command. We can see that our Master branch currently indicates a branch starting with 4 7 0 1 9 and with a "merge branch testing" message. The testing branch, on the other hand, indicates a commit starting with 9 9 1 7 and with "added functionality C" message. This proves that the Master branch has been updated and now points to a commit that was created by merging two branches. If we use the git log command with the minus minus Graph parameter, we can see in the graphical form how our commit history is arranged. In this way, it is clear that starting from the commit starting with 1 0 0 6, work on our project has forked.In our graph, the asterisk symbolizes a commit. So we can see that the commit starting with e 6 c 0 "added functionality D and changed functionality a" has been created on the Master branch. However, a commit starting with 9 9 1 7, "added functionality C" has been created on the testing branch. However, at the top we see a commit starting with 4 7 0 1, it is a commit with "merge branch testing" message. This is a commit that was created as a result of merging these two branches and it is a commit that master branch currently points to. After merging the changes from branch testing into branch master, branch testing isn't needed any longer. We can delete it with the branch minus D command. We can see that only branch Master has survived in our repository. After seeing the history of our commits in the form of a graph again, we can see that history has not changed, but the indicator of a branch testing has disappeared because the branch has been removed.

12. B - Merge conflicts

Hi, in this video I will tell you about merging branches and conflicts when merging them. Let's go back to the point where our branches were not merged. It is in such situations that so-called merge conflicts occur. This means that the version indicated by HEAD (in our case, branch master) is in the upper part of the block, starting from less-than signs to equal signs,
and the version pointed to by branch testing is below the equal signs, up to greater-than signs. To resolve the conflict, we must choose one version of the change, or the second version of the changes, or manually combine both versions of the changes.
Open git bash and go to the directory with our repository. Let's check the status of our repository and its contents. With the cat command, let's write the contents of the fileA dot txt. The Cat command lets you write the contents of any file to the console screen. We see that in the fileA dot txt there is a text: "Alex has a cat and fish ". Let's see what branches exist in our repository. Let's switch to branch testing now. Let's see the contents of our catalog. And let's look at the content of file A dot txt with cat command. It has the text "alex has a dog". Let's return to the Master branch. Let's see the contents of the fileA dot txt again. We see that when we were in the testing branch, which was indicated by a blue word at the end of the line saying "testing", the fileA dot txt contained the text "Alex has a dog". However, when we switched to the Master branch, which is indicated by the blue word Master appearing at the end of the line, we see that the fileA dot txt now has the content "Alex has a cat and fish ". At this point, merging the testing branch to the Master branch should not go as smoothly as in the previous case. And in fact, after issuing the git merge testing command, being on the Master branch, we can see that git has shown us information that there is a conflict in the fileA dot txt. It is noteworthy that now in parentheses at the end of the line in blue is Master, Vertical line, merging. We often call this vertical line a pipe. So, Master pipe Mering, suggests that we are in the process of merging some branches into master. Let's check the status of the repository. We see an additional section here, which is called unmerged paths.This means that the files that appear there have not been merged due to conflicts. In our case, the fileA dot txt has been modified by both branches. To resolve this conflict, we must manually make changes to the fileA dot txt. For this, we will use vim. We now see three sections: less-than signs and the word 'Head', which tells us that the changes under this line were made by the branch to which we are trying to add other changes, in our case: these changes are present in the Master branch.
Equal signs mean that what is under them already belongs to the branch we are trying to add to the master. Below we see the text "Alex has a dog", which belongs to the testing branch, what is indicated by the greater-than signs and the word testing, which is the name of the branch we are merging. Our task now is to edit the content of this file to leave only valid changes. We must

remove all less-than, greater-than and equality signs as well as the word 'HEAD' and 'testing'. We can do this by leaving changes made on the Master branch or leaving changes made on the testing branch. We choose the third option: we will try to combine these two sets of changes into one. We save changes and exit vim. Let's check the repository status again. If we delve into what the git wrote to us here, we'll see other useful hints here. If we use git add and the file name, we will mark this file as the file in which the conflicts were resolved. If we use the git merge minus minus abort command, we will stop the merging process. And to continue the merging process, we must use the git commit command. That's what we will do. Let's add a modified file. And then let's commit the changes. After issuing the git commit command, the text editor has reopened, but this time to complete the Message for this commit. I will use the proposition made by git, save the changes and close the text editor. Let's check the status of the repository. We see that there is nothing there that could be committed. Now let's look at the contents of our directory and the contents of the fileA dot txt. We see that the fileA dot txt actually contains the changes we saved during conflict resolution (and also that I forgot to remove less-than signs and the HEAD word). Let's see the commit history now. The graph presented here is a bit more complex because it also contains the commits I made earlier. However, we can see that the branch testing currently points to commit 2 6 e 1, and the Master branch points to commit 3 d 5 2, which is a merge commit. Now we can do nothing but remove the unnecessary branch.

13. Fast forwarding

Hi, in this video I will tell you what it is and how fast forwarding works. Let's imagine the following situation: together with a colleague we are working on a joint project. At some point in the work, we decided to split up the remaining tasks to be done. We'll take care of functionality number one and the colleague will take care of functionality number two. We have completed our task as the first, and now we want to add our changes to the branch master. we do this by switching to the master branch, and then executing the git merge feature #1 command. The output of this command will contain the phrase fast-forward. What does it mean? Because the change set pointed to by the merged branch was a direct parent of the current change set, Git moves the pointer forward. In other words, if you try to merge a set of changes with another that you can reach by following the history of the first, Git simplifies everything by moving the pointer forward, because there are no forks to merge along the way - hence the name "fast forward". Fast forwarding is a process during which no additional merge commit is created. Our changes are now part of the snapshot of the changeset pointed to by the branch master. After merging, the branch master indicates the same place as branch feature # 1, which is no longer needed. Removing it can be done with the git branch -d feature # 1

command. Fast forwarding is the default merging strategy, which means that unless we say otherwise, git will try to use fast forwarding wherever possible.

Let's see how it looks in practice.
Open git bash and go to the directory with our repository. Let's check the status of our repository and the contents of our catalog. Let's print the contents of the fileA dot txt file and the fileB dot txt onto the screen. While being on the Master branch let's create a new feature one branch.Then switch to it by the git checkout feature one command. We can see that it is no different from the Master branch. Let's create a fileC dot txt and add some content to it. Then let's add it to the staging area and then commit it. In the commit history, we see that our feature one branch is one commit ahead of the Master branch. Let's create a fileD dot txt and add some content to it. Let's save the changes and close vim. Let's add it to the staging area and then commit it. In the commit history, we see that our branch feature one is now ahead of branch Master by two commits. Let's edit the contents of the fileA dot txt By changing the text inside. Then add this file to the staging area And then commit the changes. Let's save the message of this commit and leave the text editor. If we check the history of our commits now, we will see that the graph is very linear. It happened because our feature one branch came from the master branch and then made only a few changes. If we have now switched to the Master branch, and look at the history of changes, we will see that there are only two commits on the master. If at this point we use the git merge command, along with the branch name, in our case 'feature one', this merge will run without problems. However, as a result of this command, git will display some information on the screen. Among this information, we will see the phrase Fast Forward. This means that the merge was made in Fast Forwarding mode, that is by moving the Master branch pointer forward. This can easily be seen again by reviewing the history of our commits. We now see that both branch Master and branch feature one point to the same commit. It actually happened by switching the master pointer to a commit, which branch feature one currently points to. This process is called Fast Forwarding. There is now nothing else to do but remove the unnecessary branch feature one. If we review the history of our commits again in the form of a graph, we will notice that the feature one indicator has disappeared from there because this branch has been removed.

14. Rebase vs. Merge

Hi, in this video I will tell you how merge and rebase are different. Let's imagine that we have a master branch, which looks like this. Then we create a branch to implement some new functionality in it and we make some changes there. At that time, someone else modified the master branch. Using merge, we create an additional commit, the so-called merge commit, which will be visible in the commit history, and the history itself will be bifurcated, as shown in the

figure below. However, if we want to keep a linear history of changes, there is an alternative for us. The rebase command allows us to change the base of a branch. We can say that it integrates changes from our branch to the target branch. This is done by rewriting the commits from our branch to the target branch, after its latest commit. Let's see how it will look in the commits history. Returning to the situation from before merge, if we switch to branch feature # 1, and then use the git rebase master command , we then change the parent of the first commit in our branch. Our history will look like this.
It may seem that the commits that make up our branch # 1 have changed their checksums. However, this is not true - they are in fact completely new, created by git commits, but containing the same changes that we introduced earlier. And it is these new commits that are attached to the branch we gave when issuing the git rebase command. In our case, new commits have been attached to the branch master. To update the master, we need to merge branch feature #1 to it. So first go to the master using git checkout master command, and then issue the git merge feature # 1 command. Then our commit history will look like this. You can also run the git rebase command with the minus i argument, which will cause it to be run in interactive mode. This mode allows you to make changes to commits while making the rebase. We can then change the commit message of individual commits, or join them together, if we think they are too small to be separate. The rebase interactive mode is a very useful tool for programmers and is often used in everyday work. Rebasing, like any other solution, has its pros and cons.

Its advantages are:
Simplifying the potentially complex story.
Manipulating a single commit is easier.
Preventing merge commits from appearing in the repository.

In turn, its disadvantages are:
Collecting several commitments into one can distort the context of work
Performing a rebase in a public repository can be dangerous if you work as a team A little more work is required: You need to use rebase to keep branch updated

And now let's see how rebasing looks in practice.
Open git bash and go to the directory with our repository. If we now check the history of our commits, we will see that the Master branch points to a different commit than the test branch. However, they have a common ancestor, commit starting with 8 6 2 f "Changing content of file A". The Master branch has a commit starting with 6 d 7 7 while the test branch has a pair of commits including such as "adding file E on a test branch", "changing content of file C on a test branch" and "changing content of a file A". Let's now move onto the test branch. If we now execute the Git rebase master command, that is if we want

to change the base of our test branch so that it derives from the Master branch, the effect of this command will be a minor error. One of our files has a conflict, namely the file C dot txt. Let's preview the status of our repository. We can see that the file file C dot txt has been modified in two places, which generates a conflict. We have to solve it like last time. Open it in a text editor, select the content to be saved, then save the changes and close vim. Let's see the status of our repository again. let's follow the git hint, and use the git add command to mark fileC as a file where the conflicts have been resolved. If we use the git status command again, git will tell us that to continue the rebase process, we must use the git rebase minus minus continue command. Let's use this command. We see that the commits with messages "changing content of file C on a test branch" and "Changing content of a file A" are applied on the Master branch. Let's look at the history of our commits. We can see that the Master branch currently indicates commit 6 d 7 7 "Changing content of file C on branch master". Please note that all commits that once belonged to branch test, i.e. commits A E A E "changing content of a file A" and commit 3 1 1 b "changing content of file c on a test branch" and commit 8 A F 1 "adding file e on test branch" were added after the latest commit on the branch master. On our graph, there are commits 6 D 7 7. It is worth noting, however, that despite having the same commit messages, they have different checksums. This is because they are actually three completely new commits, which, however, contain the same changes from when they were on the test branch. We can see that our commit history is now linear. This is exactly the way we wanted to achieve by executing the git rebase command. What else we may want to do right now might be to merge the test branch to the master branch, and then remove the Test branch that would then no longer be needed. Let's open git bash and go to our repository. Let's see the commit history in our repository. We can see that there are two branches: test branch and master branch. They have a common ancestor, a D 7 C 0 commit. However, on each of these branches, both on Test and on Master, some commit was made. Let's switch to the test branch now, and then let's execute the git rebase Master command. This time, however, let's use the minus i parameter, which will allow us to perform rebase in interactive mode. At this point, our default text editor will open so that we can make changes to the commits that are to be rebased. It is a very powerful tool often used in everyday work by programmers. Here we see all three commits that have been added on the branch test. Next to each of them, apart from the beginning of their checksum and their message commit, there is also the word pick. This is one of the commands that are shown below. These commands allow you to decide what will happen to each of these commits at the time of rebasing. The pick command will simply transfer this commit and apply it to our new target branch. The reword command will also transfer this commit but will allow us to edit the commit message. The edit command will also transfer our commit but will stop the rebasing process for a while so that we could modify the changes made in this

commit. The squash command, especially often used by programmers, will transfer the changes made in a given commit but will append its changes to the previously made commit. The FixUp command works similar to squash but lets you immediately reject the commit message. The Drop command will cause the commit at which this command stands not to be included in the rebase. Let's use the DROP command on the middle commit, as we can see that it was an accidental and unwanted one. This way it will not be moved while performing rebase. It will be skipped. On the commit starting with 4 8 0 B let's use the squash command, which will cause changes made in it to be merged with the changes made in the previous commit. If we have finished managing the commits to be affected by the rebase command, we can save the changes to this file and then close the editor. At this point, the text editor opened for me a second time. This happened as a result of the squash command used on one of the commits. This command, as I said, causes changes from one commit to be merged with the changes made in an earlier commit. However, in this case, we have to decide whether to use commit Message from the first commit or commit Message from the second commit. We can, of course, combine both messages. Let's save the changes and close the text editor. Now let's look at the commit history in our repository. We see that the test branch contains one commit more than the Master branch. And this is exactly the commit message we have just set. It is the one, which was created as a result of merging two other commits with the squash command, and exactly this one commit has been rebased onto the Master branch. Now let's switch to the Master branch, then merge the test branch into it, and then remove the test branch you no longer need. In the end let's see the commit history again.

15. Stash

Hi, in this video I will tell you what stash is and how to use it. Often, when you are working on some part of your project and there are changes made to it, you would like to switch to another branch to work on another functionality. The problem is you don't want to commit changes that are only partially implemented, so that you can come back to them later. The solution to this problem is the git stash command. Stash - a clipboard - it is a place where you can postpone changes that we do not want to commit yet, and to which we may want to come back later.

Commands:
git stash - command to 'put introduced changes aside, without having to commit them
git stash list - allows you to see a list of saved changes in the stash
git stash apply - allows you to reapply changes recently stashed. This option only integrates changes, they still will be listed in the stash list

git stash pop - applies last stashed changes, then removes them from the stash list

Let's see how to use stash in practice.
Let's open git bash and go to our repository. Let's see the created branches. And then switch to branch test. Let's imagine that we want to work here on some functionality in the fileX dot txt. We make changes to this file, save them and close vim. We can see that this file is not yet in the staging area.
Imagine the situation, however, that at this moment we get an urgent phone call because something is not working on the Master branch. We need to go to the master branch quickly and fix things that don't work so that others can continue their work. At this point, we must therefore temporarily abandon the development of functionality in the fileX on the test branch to move to the Master branch and quickly introduce changes there. However, being already on the Master branch, after entering the git status command, we can see that we still see a fileX dot txt as a file that can be added to the staging area. In order not to mix up the changes introduced on different branches and avoid clutter in the working directory, we can add this file to the stash. Stash is a place where we can hide our changes for a moment so that they are not visible during work, and then restore them when we want to resume work on them. The GIT stash command will add all changes not yet posted to the staging area, to the stash. So in our case the fileX dot txt will be added to the stash. Now we can easily repair the fileA dot txt on the Master branch. We save the changes, then close vim. Then we add this file to the staging area and commit it afterward. After these fixes are made, we can go back to our branch test. And get back to work on the fileX. To see what has been stored in stash, you can use the git stash list command. A list of changes stored in the stash will be displayed on the screen along with information about the last commit made before storing these changes in the stash. We see that at the moment there is only one set of changes in our stash. The git stash apply command allows us to pull out the changes from the stash and apply them to our working directory. So in our working directory the fileX dot txt will appear again in the modified state. Let's see our stash once more with the git stash list command. We see that the changes we just applied are still present in it. This is because the git stash apply command applies changes stored in stash into the working directory, however, it does not delete them from the stash. The git stash pop command does this. Let's add changes made on the test branch to the staging area and then let's commit them. Let's move to the master branch and then create a new Experimental branch from there. The git checkout command with the minus B parameter will allow us to create a branch and switch to it at the same time. With the git stash list command we check if there are any changes in stash. Let's now use the git stash pop commands, to apply the changes from the clipboard onto the working directory. As you can see, the fileX dot txt appeared in our working directory in the modified status. If we now

display a list of changes stored in the stash, we will see that stash is empty. It happened because the git stash pop command, unlike git stash apply, removes changes from the stash, in addition to applying them into the working directory.