## Lab 05: Software Version Control with Git

### Case Study

CodeNest, a software development firm delivering agile solutions for various industries, was struggling with project coordination among its distributed development teams. Developers working from different locations found it challenging to track changes, collaborate effectively, and manage versions of the same codebase. Often, developers would overwrite each other’s work or face delays due to unclear version histories. Additionally, releasing updates became a logistical nightmare as there was no standardized process to merge changes or manage conflicts. These issues led to slower development cycles, reduced code quality, and frequent rollbacks of faulty deployments.

After implementing Git and GitHub as their primary version control tools, the company experienced a major transformation. Git allowed developers to create isolated branches for features, commit changes locally, and collaborate without affecting the main codebase. Integrating with GitHub offered seamless cloud-based synchronization, code review workflows, and centralized repository management. As a result, development became faster, more reliable, and much easier to coordinate, regardless of team location.

### Business Challenge

CodeNest faced growing difficulties in managing software development across multiple developers and teams. Without a version control system, team members encountered frequent issues such as untracked changes, overwritten code, merge conflicts, and lack of accountability. These problems increased development time and introduced bugs in production, leading to dissatisfied clients and missed project deadlines.

To resolve these inefficiencies and adopt modern DevOps practices, CodeNest brought in a DevNet-certified engineer to design and implement a scalable version control workflow using Git and GitHub.

### Solution

In this lab, you will build foundational skills in Git and GitHub that are essential for collaborative software development. You will learn how to:

1. Initializing a Git repository
2. Staging and Committing a File in the Git Repository
3. Managing the File and Tracking Changes
4. Branches and Merging
5. Handling Merge Conflicts
6. Integrating Git with GitHub

|  |
| --- |
| **// Initializing Git Repository**  1. First, launch the DEVASC VM.  2. To open the terminal,double-click the **Terminal** Emulator icon on the desktop.  3. Use the **ls** command to display a listing of the current directory. Remember that commands are case-sensitive.  4. Next, configure user information to be used for this local repository. This will associate your information the work that you contribute to a local repository. Use your name in place of **“SampleUser”** for the name in quotes **“ ”.** Use **@example.com** for your email address. You can review these setting with the **git config --list** command at any time.    **Note:** These settings can be anything you want at this point. However, when you reset these global values in Part 7, you will use the username for your GitHub account. If you wish, you can use your GitHub username now.  5. Use the **cd** command to navigate to the **devnet-src folder.**    6. Make a directory git-intro using command **mkdir git-intro** and change directory into it using command **cd git-intro.**    7. Use the **git init** command to initialize the current directory (git-intro) as a Git repository. The message displayed indicates that you have created a local repository within your project contained in the hidden directory **.git**. This is where all of your change history is located. You can see it with the **ls -a** command.    8. As you work on your project, you will want to check to see which files have changed. This is helpful when you are committing files to the repo, and you don’t want to commit all of them. The **git status** command displays modified files in working directory that are staged for your next commit.  This message tells you:   * That you are on branch master. (Branches are discussed later in this lab) * The commit message is Initial commit. * There is nothing changed to commit.   You will see that the status of your repo will change once you add files and start making changes. |

|  |
| --- |
| **// Staging and Committing a File in the Repository**  1. The git-intro repository is created but empty. Using the **echo** command, create the file **DEVASC.txt** with the information contained in quotes. Use the **ls -la**command to verify the file, as well as the **.git** directory, are in the **git intro** directory. Then use **cat** to display the contents of **DEVASC.txt**.    2. Now, examine the repository status using **git status**. Notice that Git found the new file in the directory, and knows that it is not tracked.    3. Use the **git add** command to **“stage”** the **DEVASC.txt** file. Staging is an intermediate phase prior to committing a file to the repository with the git commit command. This command creates a snapshot of the contents of the file at the time this command is entered. Any changes to the file require another git add command prior to committing the file. Using the **git status** command again, notice the staged changes displayed as **“new file: DEVASC.txt”.**    4. Now, that you have staged your changes, you will need to commit them in order to let Git know you want to start tracking those changes. Commit your staged content as a new commit snapshot by using the **git commit** command. The **-m message** switch enables you to add a message explaining the changes you have made. Note the number and letter combination highlighted in the output. This is the commit ID. Every commit is identified by a unique SHA1 hash. The commit ID is the first 7 characters of the full commit hash. Your commit ID will be different than the one displayed.    5. Use the **git log** command to show all commits in the current branch’s history. By default, all commits are made to the master branch. (Branches will be discussed later.) The first line is the commit hash with the commit ID as first 7 characters. The file is committed to the master branch. This is followed by your name and email address, the date of the commit and the message you included with the commit. |

|  |
| --- |
| **// Modifying the File and Tracking the Changes**  1. Make a change to **DEVASC.txt** using the **echo** command. Be sure to use **“>>”** to append the existing file. The **“>”** will overwrite the existing file. Use the **cat** command to view the modified file. Use the **cat** **DEVASC.txt** command to view the modified file.    2. Verify the change in the repository using the **git status** command.    3. The modified file will need to be staged again before it can be committed using the **git add** command again.    4. Commit the staged file using the **git commit** command. Notice the new commit ID.    5. Use the **git log** command again to show all commits. Notice that the log contains the original commit entry along with the entry for the commit you just performed. The latest commit is shown first. The output highlights the commit ID (first 7 characters of the SHA1 hash), the date/time of the commit, and the message of the commit for each entry.    6.  When you have multiple entries in the log, you can compare the two commits using the **git diff** command adding original commit ID first and the latest commit second: **git diff <commit ID original> <commit ID latest>.** You will need to use your commit IDs. The “+” sign at the end, followed by the text indicates the content that was appended to the file. |
| **// Branches and Merging**  1. In this part, you will create a new branch, checkout the branch, make changes in the branch, stage and commit the branch, merge the branch changes to the master branch, and then delete the branch.  2. Create a new branch called **feature** using the **git branch <branch-name>** command. Use the **git branch** command without a branch-name to display all the branches for this repository. The **\*** next to the master branch indicates that this is the current branch (the branch that is currently “checked out”).    3. Use the **git checkout <branch-name>** command to switch to the feature branch.    4. Verify you have switched to the feature branch using the **git branch** command. Note the **“\*”** next to the **feature** branch. This is now the working branch.    5. Append a new line of text to the **DEVASC.txt file**, again using the **echo**command with the **“>>”** signs. Verify the line was appended to the file using the **cat** command.    6. Stage the updated file to the current**feature** branch. Use the **git status** command and notice the modified file **DEVASC.txt** is staged in the **feature**branch.    7. Commit the staged file using the **git commit** **-m "Added a third line in feature branch"** command. Notice the new commit ID and your message    8. Use the **git log** command to show all commits including the commit you just did to the feature branch. The prior commit was done within the master branch.    9. Now, switch to the master branch using the **git checkout master**command and verify the current working branch using the **git branch** command.    10. Branches are often used when implementing new features or fixes. They can be submitted for review by team members, and then once verified, can be pulled into the main codebase – the master branch.  Merge the contents (known as the history) from the feature branch into the master branch using the **git merge <branch-name>** command. The branch-name is the branch that histories are pulled from into the current branch. The output displays that one file was changed with one line inserted.    11. Verify the appended content to the **DEVASC.txt** file in the master branch using the **cat** command.    12. Verify the **feature** branch is still available using the **git branch** command.    13. Delete the **feature** branch using the **git branch -d <branch-name>** command. Verify the **feature** branch is no longer available using the **git branch** command. |
| **// Handling Merge Conflicts**  1. In this Part, you will create a test branch, modify its content, stage and commit the test branch, switch to the master branch, modify the content again, stage and commit the master branch, attempt to merge branches, locate and resolve the conflict, stage and commit the master branch again, and verify your commit.  2. Create a new branch **test** using command **git branch** **test**. Checkout (switch to) the branch **test** using command **git checkout test**. Verify the working branch is the **test** branch using command **git branch**.    3. Verify the current contents of the **DEVASC.txt** file. Notice the first line includes the word “Cisco”. Use the **sed** command to change the word **Cisco** to **NetAcad** in the **DEVASC.txt** file.    4. Verify the change to the **DEVASC.txt** file using **cat** command.    5. Stage and commit the file with a single **git commit -a** command. The **-a** option only affects files that have been modified and deleted. It does not affect new files. Checkout (switch to) the **master** branch. Verify that the master branch is your current working branch.    6. Use the **sed** command to change the word **Cisco** to **DevNet** in the DEVASC.txt file. Verify the change to the file using **cat** command.    7. Stage and commit the file using the **git commit -a** command.    8. Attempt to merge the test branch history into the master branch using the command **git merge test**. Use the **git log** command to view the commits. Notice that the **HEAD** version is the master branch. This will be helpful in the next step.    9. Use the **cat** command to view the contents of the DEVASC.txt file. The file now contains information to help you find the conflict. The HEAD version (master branch) containing the word **DevNet** is conflicting with the test branch version and the word **NetAcad.**    10. Use the **vim** command to edit the file. Use the up and down arrow to select the proper line of text. Press **dd** (delete) on the following lines that are highlighted. **dd** will delete the line the cursor is on. Save your changes in vim by pressing **ESC** (the escape key) and then typing : (colon) followed by **wq** and press enter. Verify you changes using the **cat**command.    11. Now, Stage and commit the master branch.    12. Use the **git log** command to verify the commit. If necessary, you can use **q** to quit out of the git log display. |
| **// Integrating Git with GitHub**  1. All the changes you have made to your file have been stored on your local machine. Git runs locally and does not require any central file server or cloud-based hosting service. Git allows a user to locally store and manage files.  Although Git is useful for a single user, integrating the local Git repository with a cloud-based server like GitHub is helpful when working within a team. Each team member keeps a copy on the repository on their local machine and updates the central cloud-based repository to share any changes.  2. Go to **github.com** and create a GitHub account. If you have a GitHub account, log into your GitHub account.  3. Select the “**New repository**” button or click on the **“+”** icon in the upper right corner and select “**New repository**“.  4. Now, create a new repository. In the Repository name write **devasc-study-team,** add aDescription **Working together to pass the DEVASC exam,**  and in Public/Private chose **Private.** Then, select **Create repository.**    5. Open terminal and change into **git-intro** directory. Make a new directory called devasc-study-team. The directory does not have to match the name as the repository. Use the **cd** command to change directories to **devasc-study-team**.    6. Use the **cp** command to copy the **DEVASC.txt** from **git-intro** parent directory to the **devasc-study-team** sub-directory. The two periods and a slash prior the file name indicates the parent directory. The space and period following the file name indicates to copy file in the current directory with the same file name.  Verify the file was copied with the **ls** command and the contents of the file with the **cat** command.    7. Use the **git init** command to initialize the current directory (devasc-study-team) as a Git repository. The message displayed indicates that you have created a local repository within your project contained in the hidden directory **.git**. This is where all of your change history is located.    8. Next, check your global git variables with the **git config --list** command.    9. If the **user.name** and **user.email** variables do not match your GitHub credentials, change them using the commands **git config --global user.name "GitHub username"** and **git config --global user.email GitHub-email-address.**    10. Use the **git remote add** origin command to add a Git URL as a remote alias. The value **origin** points to the newly created repository on GitHub. Use your GitHub username in the URL path for **github-username.** Verify the **remote** command is running on github.com using **git remote --verbose.**    11. View the **git log**. The error indicates that there are no commits. Use the **git add** command to stage the **DEVASC.txt** file. Then, use **git commit** command to commit the **DEVASC.txt** file.    12. Now, use the **git log** command again to verify the commit.    13. Use the **git status** command to view status information. The phrase “working tree clean” means that Git has compared your file listing to what you have told Git, and it is a clean slate with nothing new to report.    14. Now, use the **git push origin master** command to send (push) the file to your GitHub repository. You will be prompted for a username and password, which will be the one you used to create your GitHub account.    15. To verify, go to your GitHub account and under “Repositories” select **username/devasc-study-team**. You should see that the **DEVASC.txt** file has been added to this GitHub repository. You can click on the file to view the contents. |