

LAB MANUAL



Comprehensive Git

Version 1.0

Table of Contents

Lab 1: Create a Repository	5
Lab 2: Create a Repository in GitHub	11
Lab 3: Branching in GitHub	15
Lab 4: Workflow	19
Lab 5: Merging with Conflicts in GitHub.....	23
Lab 6: Git Internals	31
Lab 7: Git Configuration.....	39
Lab 8: Rebase.....	47
Lab 9: Git Sharing.....	51

Lab 1: Create a Repository

Overview

In this lab, you will create an instance of a Git repository. The purpose of this lab is to acquaint you with the use of the Git Gui to create and manage repositories. You will initialize a repository and track changes to a file in that repository. You will get familiar with staging and committing from the Git Gui. We recommend that you use both the GUI approach and the command line approach. Some situations will develop that can only be resolved with command line tools, so familiarity with the command line is a must for Git users.

High-level Objectives (for those wanting a less-guided approach)

You will perform these general tasks either with Git Gui and/or the command line:

- Create a folder for the repository somewhere on your file system and name it: **/git-repos/my-project**
- Initialize the folder with a repository (not a bare repository) using the **init** command
- Create a **README.md** file in the folder, stage it and commit the change
- Change the file, stage the file and then commit the change
- Remove the file, stage the change and then commit the change
- We recommend doing this first with the GUI and then repeating using the command line

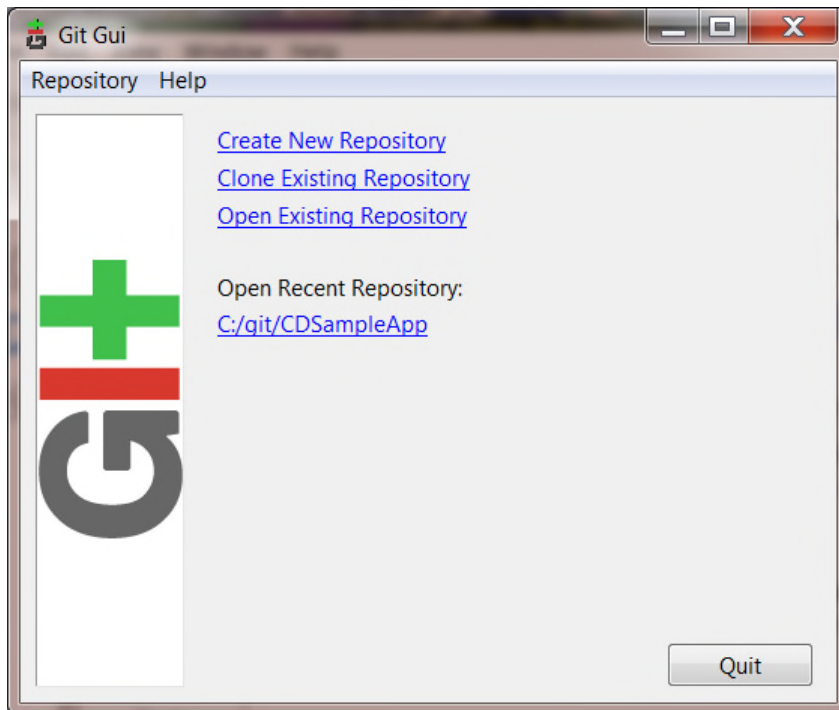
Detailed Instructions

We will need a playground to practice Git. The first step is to create a folder to put the repository in. Next, it will be populated with a repository. Then, a file will be added. Finally, it will be changed, staged and committed a couple of times to practice the process.

***Note:** The command line instructions are at the very end of the exercise.*

1. Open Windows Explorer (Windows) or Finder (Mac) to create a folder and subfolder in the root of the file system named **"/git-repos/my-project"**. Navigate to that new folder.
 - To get to the root folder on Mac press **shift + ⌘ + G** then type **/** and click **Go**
2. Open the **Git Gui** Client. In the file manager, you can right-click to view a context menu with **Git Gui Here**.

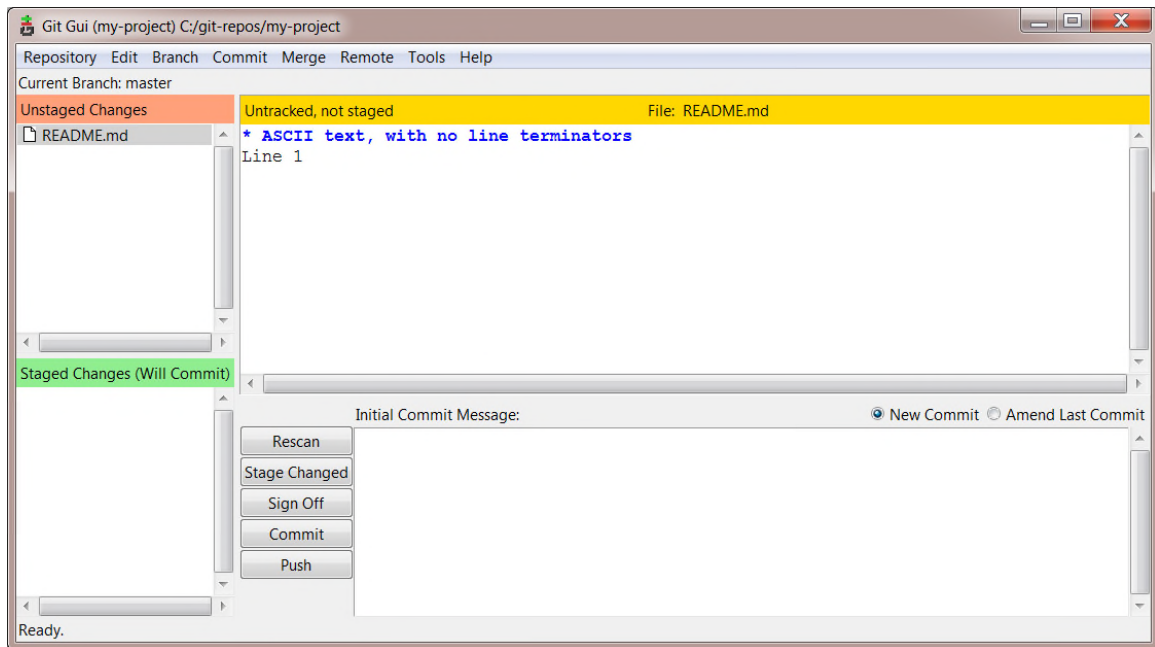
- For Windows: **All Programs → Git → Git Gui**
- For Mac: from the terminal type **Git Gui**



3. Click the **Create New Repository** link, which will open a dialog box. Click the **Browse** button and navigate to the folder you just created. (C:/git-repos/my-project) Click the **Select Folder** or **choose** (on Mac) button.
4. Click the **Create** button and the screen will refresh with the contents of the new repository. You will also notice that in the directory **/git-repos/my-project**, there is now a **.git** folder which has several files and folders under it for the Git repository.
 - The **.git** folder is a hidden folder so it might not be visible depending on your system settings.

You now have a Git Repository. It has the folder structure for Git to retain its content. There is nothing in the repository at this time.

5. Using the editor of your choice, in the folder **/git-repos/ my-project** create a file named **README.md**. Add one line to the file with the content **Line 1**. Save the file.
6. In the Git Gui click the **Rescan** button and the screen will look similar to the one below. If the **README.md** file is not displayed in the File window on the right, select it from the Unstaged Changes window on the left.



- Click the **Stage Changed** button. A dialog box will pop up confirming that you want to Stage the changed file. Click **Yes**. The **README.md** file moves from the top list to the bottom list on the left under Staged Changes (Will Commit). If you select the **README.md** file, you will see the changes as a new file with a mode setting and +Line 1 and no new line at the end.

This step creates a reference to the README.md in the index file in the root of the **.git** folder. You can't really read the index file with any editor. To see what is in the index you can do a **git status** from the command line.

- Click the **Commit** button. Since we haven't entered a comment in the comment box, a dialog warning us about that will pop up. It is a good idea to explain why there is a change so that others will know the purpose of the change.

Note: If you are using a new installation of GIT and have not committed anything yet, Git will tell you it needs to know who you are. There are 2 ways to fix this:

- In the Git Gui, click **Edit → Options** and fill out a username and email address for **This (my-project repository)** and **Global (All Repositories)**.
- On the command prompt or terminal type:

```
- git config --global user.email "you@example.com"
- git config --global user.name "Your Name"
```

9. Enter a commit comment "**Initial Commit**" in the dialog box on the right. Click the **Commit** button. You will notice that there are no files being displayed. That is because there are no files with changes. Until a change is made to the file again, there is nothing to display.
10. Edit the file again and add a second line **Line 2**. Click **Rescan**. Click **Stage Changed**. Confirm by clicking the **Yes** button. Enter a commit comment "**Add Line**". Click the **Commit** button.
11. Now we will see how to track file removals. Go to the file system using the menu option under **Repository** → **Explore Working Copy**. Delete the **README.md** file. Click the **Rescan** button. Click the **Stage Changes** button. Then add a comment **Remove the README** and click the **Commit** button.
12. Congratulations!! You are a Git user! That's all there is to it. We initialized the repository, added a file, change the file and committed it a couple of times and then removed it. That is a very quick overview of what working with Git involves.

Now, try the same tasks using the command line. Delete the folder **/git-repos/my-project** before executing the steps below.

Walk-through using the Command line

- *Lines that are in black italics are the commands to type*
- The output in **blue** below will look similar to the output you will get.
- Mac commands are in the left column; Windows commands are in the right.

Mac Commands:

```
sudo mkdir -p /git-repos/my-project  
sudo chown -Rv "username" git-repos
```

```
cd /git-repos/my-project
```

```
gituser@githost MINGW64 /c/git-repos/my-project
```

```
git init
```

```
Initialized empty Git repository in C:/git-repos/my-project/.git/
```

```
gituser@githost MINGW64 /c/git-repos/my-project (master)
```

```
touch README.md  
echo "Line 1" >README.md
```

```
gituser@githost MINGW64 /c/git-repos/my-project (master)
```

Windows Commands:

```
mkdir C:\git-repos\my-project
```

```
cd C:\git-repos\my-project
```

```
git init
```

```
echo. 2> README.md  
echo "Line 1" >README.md
```



```
git add README.md
```

```
git add README.md
```

```
gituser@githost MINGW64 /c/git-repos/my-project (master)
```

```
git status
```

```
git status
```

```
On branch master
Initial commit
Changes to be committed:
  (use "git rm --cached <file>..." to unstage)
        new file:   README.md
gituser@githost MINGW64 /c/git-repos/my-project (master)
```

```
git commit -m 'Initial commit'
```

```
git commit -m "Initial commit"
```

```
[master (root-commit) fdc3834] Initial commit
1 file changed, 1 insertion(+)
 create mode 100644 README
gituser@githost MINGW64 /c/git-repos/my-project (master)
```

```
git status
```

```
git status
```

```
On branch master
nothing to commit, working directory clean
gituser@githost MINGW64 /c/git-repos/my-project (master)
```

```
vi README.md
```

```
notepad README.md
```

add "Line 2" to the README.md file

```
gituser@githost MINGW64 /c/git-repos/my-project (master)
```

```
git status
```

```
git status
```

```
On branch master
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in
working directory)
        modified:   README.md
no changes added to commit (use "git add" and/or "git commit -
a")
gituser@githost MINGW64 /c/git-repos/my-project (master)
```

```
git add README.md
```

```
git add README.md
```

```
gituser@githost MINGW64 /c/git-repos/my-project (master)
```

```
git status
```

```
git status
```

```
On branch master
```

```
Changes to be committed:
```

```
(use "git reset HEAD <file>..." to unstage)
```

```
    modified:   README.md
```

```
gituser@githost MINGW64 /c/git-repos/my-project (master)
```

```
git commit -m 'Add Line'
```

```
git commit -m "Add Line"
```

```
[master 93e4c60] Add Line
```

```
1 file changed, 1 insertion(+)
```

```
rm README.md
```

```
del README.md
```

```
git add README.md
```

```
git add README.md
```

```
git commit -m 'Remove README.md'
```

```
git commit -m "Remove README.md"
```

```
[master ce241c9] Remove README.md
```

```
1 file changed, 1 deletion(-)
```

```
delete mode 100644 README.md
```

```
git log
```

```
git log
```

```
commit ce241c974a3d264593d87699de2b19c7d36eede8
```

```
Author: Git User <gituser@gmail.com>
```

```
Date: Sat Mar 19 20:16:25 2016 -0500
```

```
Remove README.md
```

```
commit 93e4c60fcced7d6932ca8c4d97395b9d502ae5b3
```

```
Author: User<email>
```

```
Date: Sat Mar 19 18:34:44 2016 -0500
```

```
Add Line
```

```
commit fdc383416d589a92482fd9f6eb9ecff20543cecc
```

```
Author: User<email>
```

```
Date: Sat Mar 19 18:32:19 2016 -0500
```

```
Initial commit
```

Lab 2: Create a Repository in GitHub

Overview

In this lab, you will create an instance of a Git repository as you did in the first exercise, but this time with GitHub. The purpose of this lab is to introduce you to GitHub for the creation and management of repositories. You will initialize a repository and track changes to a file in that repository using the GitHub web interface. You will get familiar with staging and committing from the GitHub context. You will also use GitHub Desktop.

You will need to sign-up for a GitHub account if you don't have one already. If you don't want to keep the account, you can close it after the class.

High-level Objectives

You will perform these general tasks, with either GitHub or GitHub Desktop:

- Sign up for a **GitHub** account at <https://github.com>
- Create a new public repository **example-project**
- Initialize the folder with a repository with a **README.md**
- Change the file by adding a line and commit the change
- Install the **GitHub Desktop** application from GitHub
- Clone the project to the desktop
- Edit the file locally and commit, then sync the changes back to GitHub

Detailed Instructions

1. Sign up for a GitHub account at <https://github.com/> and pick a username, specify a valid email address and a password. These are personal choices and you will need to select a username that is not already in use. You may be able to use your existing username from another system and append some digits to make it unique. Your email can be your work email or a personal email you may have. The password rule is under the box: at least one letter, one number and at least 7 characters in length.
2. After creating your account, you will be on the main page with a **Read the Guide** button. You may want to use the button and do the **Hello World** Tutorial for some practice using GitHub. We will provide some steps that will follow the pattern from the first exercise.

3. We will first create a repository to track the changes in our project. You will see on the right side of the screen a green button with the label **New repository**. Click the button.
 - If you using a new GitHub account, it will ask you to verify the account before you can make a new repository. There will be an email in your inbox with a link for verification.
4. You will create a Public repository, since this is a free account which only has the ability to create Public repositories. We will create a repository using the name **example-project**. We will keep the checkbox checked to initialize with a README as we want to populate the README as we did with our first project. At this point you could choose to add a **.gitignore** and a license file if you want. Neither are required, so we will not add either in this lab. Add a description to your project with the content **This will be the first GitHub project for this account**. This is also not required, but is useful for documentation. Click the **Create Repository** button.
5. Notice that the **README.md** file was already created and added to the repository. Also, the first commit was done with an initial comment of **Initial commit**. That covers both of the first tasks in our first exercise, and we haven't done much of anything. GitHub makes a lot of things about Git really easy!
6. In order to monitor changes, we will need to make some changes. Let's edit the **README.md** to add some content, to see how easy it is to edit content in GitHub. Click the **README.md** link, which will open a browser on the file. You will see **three lines (2 sloc) 73 bytes**. To the right of that, there is a pencil icon. Click the pencil icon to edit the file. Add a line **Add another line to the README file** to the **README.md** with a commit comment **Update the README file** and click the **Commit Changes** button.
7. The screen updates and sets the line count to 4, and the source lines of code (sloc) is now 3.
8. These changes have all been done on the GitHub site. But to be useful we want them to be pulled to our local machine. We are going to make that happen using GitHub Desktop by cloning the project.
9. If you have not yet downloaded GitHub Desktop for your platform, you can download it now from **<https://desktop.github.com>**. (Download the Official version, not the Beta edition!) It should automatically install.

After installing GitHub Desktop, open GitHub Desktop. On the Welcome screen, begin by logging in using the email address and the password for the account that was just created. Click **Continue** on the configuration. Click on the **dashboard** link.

Skip all the popups. Click on the **+** and then the **Clone** button. Select the **example-project** and click the blue checkmark to **Clone example-project**.

- A. Use the default location for the repository:
 <homeFolder>\Repository for Mac
 <homeFolder>\Documents\GitHub for Windows
 - B. If you don't see a Welcome screen, go to **Preferences** to log in.
 - C. For Mac: **GitHub Desktop** → **Preferences...** → **Accounts**
 - D. For Windows: Click on the gear on the top-right corner of the screen → **Options** → **Add Account**
10. After the clone completes, you will be able either to view it locally on the system using the cloned repository, or using GitHub Desktop to view the remote project. You may also use GitHub to view the original project at GitHub. You will be able to push and pull between the repositories.
 11. With GitHub Desktop in the foreground, right-click on the project **example-project** and select **Open in Explorer** (or **Finder** for Mac), which will launch the file system explorer. You will see the **README.md** file and the **.git** folder. Open the **README.md** in the editor of your choice and add a line **Add a line locally to push to GitHub** to the **README.md** file. Save the file and go back to GitHub desktop. Click on the **Changes** tab, which will be highlighted with a dot or small circle. Fill in the summary with **Adding a line** and put the same in the description. Click the **Commit to Master** button.
 12. Sync the changes using the **Sync** button. This will push the content to the repository on GitHub. If there were changes on GitHub it would also pull them down to the local repository.
 13. Open the project in GitHub and notice that the changes made on the local machine are now synchronized with the GitHub repository. You can see that there are three commits. If you select the **README.md**, you will see the changes that were made.

Lab 3: Branching in GitHub

Overview

In this lab, you will practice branching in a Git repository using the GitHub Desktop. The purpose of this lab is to acquaint you with the use of the GitHub Desktop to create and manage branches. You will initialize a repository and track changes to a group of files in two separate branches using the GitHub web interface. You will see how easy it is to create branches and merge changes between branches.

This will simulate two separate development paths. Imagine a developer working on a new feature is interrupted by the need to push out a hotfix to production. We will create two branches: one for the hotfix and one for the feature. We will then merge both back into the main branch to finish.

We are going to practice only on the local file system, not using remote branching. That will be covered in a later exercise.

High-level Objectives


You will perform these general tasks with GitHub Desktop:

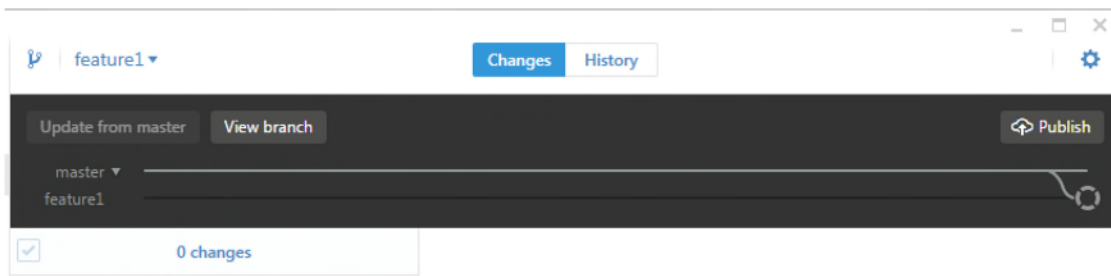
- Create a new repository using GitHub Desktop called **simple-web-site**
- Create a file named **index.html** with a simple HTML content and commit the change. (Step 2 has the syntax for a simple Hello World HTML file.)
- Create a branch **feature1**
- Edit the file and commit
- Switch back to the **master** branch
- Create a branch **hotfix**
- Edit the file by adding new content at the end of the file and commit
- Switch back to the **master** branch and update from **hotfix**
- Delete the **hotfix** branch
- Switch to branch **feature1** and verify the change
- Switch back to the **master** branch and update from **feature1**
- Delete the **feature1** branch
- Verify the code has the content from both **feature1** and **hotfix**

Detailed Instructions

1. Open GitHub Desktop and create a new repository. Call it **simple-web-site**. You will get a new repository in the GitHub directory of your user document folder. To see what got created in the folder, open using the Explorer by right-clicking the repository and selecting **Open in Explorer** (or **Open in Finder** for Mac).
2. To have a file that we can see the changes to the file system when we switch back and forth between branches, we will create a web page **index.html** file. Create a new file named **index.html** with the content below. Commit the change using a comment **Add the web page**.

```
<html><body>
Hello World
</body></html>
```

3. Create a new branch using the  icon to the left of the master branch. Click the branch icon to bring up a dialog box. Enter the name **feature1**. Click **Create new branch**. Notice that the branch shows up under the master branch line. You also will see a circle on the far right indicating which is the current branch in the working folder.



4. Make a small change to the **index.html**. Change **Hello** to **Goodbye** in **index.html**. Save and commit the change with the comment **Change Hello to Goodbye**.
5. Switch back to the **master** branch, by selecting it from the drop-down branch list on the top of the GitHub Desktop. In the Explorer window, open the **index.html** and notice there is only **Hello World** in **index.html**.
6. Switch back to the **feature1** branch by selecting it from the drop-down list and notice the content of **index.html** has **Goodbye World**.
7. Now is when the production fix is identified. Switch back to the master branch. Create a new branch named **hotfix**. Open the **index.html** file and add a line after **Hello World** and before **</body>**.

```
Welcome to our web site.
```


8. Save the file and commit the changes with the comment **Welcome Users**.
9. Now, to remove the hotfix development branch, the changes will need to be merged back into the master branch. Select the **History** view. Select **master** from the drop-down. Click **Compare** and select the **hotfix** branch. The button **Update from hotfix** will be above the hotfix and master branches listed in the history map. Click **Update from hotfix**. The changes will be merged with a comment referencing the branch merge content and committed automatically.
10. We no longer need the hotfix branch since the code has been merged. Select the **hotfix** branch. Click the **gear icon** on the right. In the menu that pops up, select **Delete hotfix...** to delete the branch.
 - If you're on a Mac you will not see a gear icon. Instead click:
Branch → Delete "Hotfix".
11. Finally, we will merge in the feature so we have both changes in our main branch. Select the **feature1** branch and verify that the change we made is what we expect. It should be the text **Goodbye World** with begin and end tags for html and body.

Select the master branch again. Click **Compare** and select **feature1**. Click the **Update from feature1**. You will receive an error that there are conflicts. **Continue** to resolve the conflicts. Open the **index.html** and change the content to match the content below.

```
<html><body>
Hello and Goodbye World
<br>Welcome to our web site.
</body></html>
```

12. Commit the changes and delete the **feature1** branch.

Lab 4: Workflow

Overview

In this lab, you will practice remote branching in a Git repository using the GitHub Desktop, the Git Gui and Git Shell. The purpose of this lab is to demonstrate how to manage workflow with multiple remote branches. You will initialize a repository and create two separate clones of the repository, and treat one as a repository for one team and the other clone will be used by the other team. This is slightly more complicated than it would normally be since we are doing it all on the same file system and not using a Git Server.

This will simulate using two separate development repositories. Imagine one team working from one repository and a developer working from a different repository. The team will work from a repository **work-project** which gets cloned to **work-project.git** as a shared repository. The developer will be working from a different repository cloned to **dev-work-project** from the **work-project.git**. The team will work in the branch **feature1** and the developer will work in a different branch named **hotfix**. The team will make several updates, commit and push the changes to the master branch of the central repository. The developer will make several updates, commit and push the changes to the master branch of the central repository.

We are going to practice only on the local file system and not using a GitHub remote repository.

High-level Objectives

You will perform these general tasks:

- Create a new repository using **GitHub Desktop** called **work-project**
- Add a **README.md** file to the project and commit
- Clone the project to **work-project.git** as a **bare** repository
- Add a remote link as origin to the bare repository on the work-project
- Clone the repository into a local repository **dev-work-project**
- Add a remote link as origin to the bare repository on the work-project
- Create the **feature1** branch in work-project
- Create a file named **index.html** with simple HTML content and commit the change. (Step 8 has the syntax.)
- Create the **hotfix** branch in dev-work-project
- Create a file named **firstpage.html** with simple HTML content and commit the change. (Step 11 has the syntax.)
- Merge the branches to the respective master branches and delete the branches

- In the dev-work-project, push the changes to the remote repository
- In the work-project folder, pull the changes from the remote repository and then push them back
- Verify the work-project and dev-work-project have the same SHA master reference

Detailed Instructions

1. Create a new repository using GitHub Desktop in **user's home folder/Repository/work-project** folder as work-project.
2. Open Windows Explorer (Windows) or Finder (Mac) to navigate to **user's home folder/Repository/**.
3. Right-click on the folder and, from the context menu, select **Git Bash Here**. For Mac you will use **Terminal**.
 - For Mac only:
In the work-project folder, type the following:


```
git commit --allow-empty -m "initial commit"
```


This creates the master branch and lets us create the branches we need for this lab.
4. Using the Git command line, clone the repository **work-project.git** as a remote repository (use the **--bare**) option. You will first navigate back to the repository directory.


```
cd ..  
git clone --bare work-project work-project.git
```
5. Using Windows Explorer or Finder, navigate to the newly created folder and notice that there is no **.git** folder. The reason is this is a *remote* repository not intended to be a development workspace, but only for storing Git objects and attributes (not source content).
6. Using Git Gui, clone the repository from the location created in step one to the **user's home folder/Repository/dev-work-project**.
 - Open Git Gui and click the blue link **clone existing repository**. On the next screen set:
Source: **user's home folder/Repository/work-project.git**
Target: **user's home folder/Repository/dev-work-project**
7. In the **dev-work-project** repository, create a branch **hotfix**.

8. Create a web page **index.html**. Add the content below to the file. Set the commit comment to **Add the home page**.

```
<html><body>
Hello World
</body></html>
```

9. Commit the changes; (rescan, stage, add a comment and then commit).
10. In the **work-project**, create a branch **feature1**.
11. Create a new file **FirstPage.html** and add the content below.

```
<html><title>First Page</title><body>
First Page
</body></html>
```

12. Commit the changes using the comment **Add the first page**.
13. First, merge the changes in **dev-work-project** to the master branch. To do this, first switch to the **master** branch and then select **merge**.
14. Second, merge the change in **work-project** branch **feature1** to the master branch.
15. At this point, we now have two separate repositories for product development. One is for the main development effort. And the other is for a separate development effort. Each of these repositories can have different branches. At some point the two repositories will need to get synched up. Both repositories have a change to the master. They need to be merged together. Here's how to do that. Open a Git Bash, Git Cmd or Git Shell in the **dev-work-project** repository, (for Mac use the Terminal).
16. When ready to sync the **dev-work-project** to the **work-project**, execute the Git command:

```
git push origin master
```

This pushes the entire content, including history, of the **dev-work-project** repository into the repository of **work-project.git**. It also has updated the reference of the head of the master to match the reference that was pointed to in **dev-work-project**. You can verify this by running the command **git log** in the **work-project.git** folder and also by displaying the content of **refs/head/master**, which will have the same SHA as the first Git log entry.

17. Before we can pull the changes from the remote repository into our working tree, we need to set up a link between **work-project** and **work-project.git** as a remote. It should be done by the **clone remote**, but that doesn't always do it. To determine if the remote has been added, execute the command in the **work-project** folder:

```
git pull origin master
```

18. If the command fails because **origin** is not defined, then execute the following command in the **work-project** folder:

```
git remote add origin ../work-project.git  
git pull origin master
```

19. It is now possible to merge the changes from **work-project** into **work-project.git** using the Git command:

```
git push origin master
```

20. To pull the most recent changes from the **work-project** back to **dev-work-project**, the changes need to be synced using the Git command in the folder dev-work-project:

```
git pull origin master
```

- On Mac this will open the **vi** editor so you can add a message. To exit, hit the Esc-key and type **:q! → return**.
21. Using the tool **Git Gui**, visualize the branch history for both projects **work-project** and **dev-work-project** and they will have the same final version even though the history of each project is different.

You have now seen how to use Git with multiple remote repositories to manage workflow.

Lab 5: Merging with Conflicts in GitHub

Overview

In this lab, you will practice merging in a Git repository using the GitHub Desktop. You will initialize a repository and track changes to a group of files in two separate branches using the GitHub web interface. You will get see how easy it is to create branches and merge changes between branches.

This will simulate using three separate development paths. Imagine a developer working on a new feature while another developer is working on a second feature and a third developer is working on an architectural change that will impact both features. We will create three branches and change content on all three. We will then merge all three back into the main branch and delete the branches to finish.

We are going to practice only on the local file system, not using remote branching. That will be covered in a later exercise.

High-level Objectives

You will perform these general tasks using GitHub Desktop, Git Gui and command line:

- Create a new repository using GitHub Desktop called **merge-web-site**
- Create a file named **index.html** with simple HTML content and commit the change. (Step 2 has the syntax. Also, you can copy and paste all lengthy source content used in this lab from the **snippets.txt** file in the setup folder.)
- Create a branch **feature1**
- Create a **contact.html** page to display contact information and commit. (See step 4 for HTML.)
- Edit the **index.html** file to link to the page and commit. (See step 5 for HTML.)
- Switch back to the **master** branch and create a branch **feature2**
- Create a **tablepage.html** with an HTML table in it and commit. (See step 8 for HTML.)
- Edit the **index.html** to link to the page and commit. (See step 9 for HTML.)
- Switch back to the **master** branch and create a branch **ui-change**
- Add a file **styles.css** and update **index.html** to reference it and commit. (See Steps 11 and 12 for content.)
- Merge all three branches to **master**
- Explore the Git Gui Visualizer then delete the branches

Detailed Instructions

1. Open GitHub Desktop and create a new repository. Call it **merge-web-site**. You will get a new repository in the GitHub directory of your user document folder. To see what got created in the folder, open using the Explorer by right-clicking the repository and selecting **Open in Explorer** (or **Open in Finder** for Mac).
2. We will create a web page **index.html** so we can see the changes to it on the file system when we switch back and forth between branches.

Note: Source content can be found in the **snippets.txt** file in the setup folder. You can copy and paste from **snippets.txt**. Do not copy and paste from this lab manual as special characters like curly quotes and long dashes can cause errors.

- Create a new file named **index.html** with the content below.
- Commit the change using the comment **Add the web page**.

You can test the results of the page by opening it in a browser. In the **Explorer** (or **Finder** on Mac), a double-click will launch the page in a browser.

```
<!DOCTYPE html>
<html><head><title>The Company Home Page</title></head>
<body>
The Company
<br>
Home Page
</body></html>
```

3. Create a new branch named **feature1**.
4. Make a new file named **contact.html** with the content below; (the content can be found in the snippets.txt of the setup folder). Save and commit the change with the comment **Add the contact page**.

```
<!DOCTYPE html>
<html lang="en">
<head><title>Contact Page</title>
<link href="styles.css" rel="stylesheet" />
</head>
<body>
<div class="company"><strong>The Company</strong></div>
<div>
<ul>
<li><a href="index.html">Home</a></li>
</ul></div>
<hr/>
<div>
Contact:
```



```
<br>The Company
<br>1234 Any Street
<br>Anytown, TX, 77777
<hr>
Phone:      555-555-1245
<hr>
Email: someemail@somecompany.com
</div>
</body></html>
```

5. An additional change needs to be made to **index.html** to reference the contact page. Open the **index.html** and add the menu section below the line **Home Page**. Save and commit the changes as **Add the contact page link**. You can test the changes by opening **index.html** in the browser. You can navigate from the home page to the contact page and back to the home page.

```
<div>
<ul>
<li><a href="contact.html">Contact</a></li>
</ul></div>
```

6. Switch back to the **master** branch. In the Explorer window notice, there is only **index.html**. We made the change only in the **feature1** branch so there are no changes in the **master** branch yet. We will deal with that later.
7. Create a new branch named **feature2**.
8. Make a new file named **tablepage.html** with the content below; (the content can be found in snippets.txt). Save and commit the change with the comment **Add the table page**.

```
<!DOCTYPE html>
<html lang="en">
<title>Table Page</title>
<head>
  <link href="styles.css" rel="stylesheet" />
</head>
<body>
<div class="company"><strong>
The Company</strong>
</div>
<div>
<ul>
<li><a href="index.html">Home</a></li>
</ul></div>
<div>
<p>See the table below for details</p>
<hr/>
```

```

<table>
<tr><th>Name</th><th>Description</th><th>Price</th></tr>
<tr><td>Car</td><td>Toy car</td><td>$3.00</td></tr>
<tr><td>Boat</td><td>Toy boat</td><td>$5.00</td></tr>
<tr><td>Truck</td><td>Toy truck</td><td>$1MILLION</td></tr>
</table></div>
</body></html>

```

9. An additional change needs to be made to **index.html** to reference the table page. Open the **index.html** and add the menu section below the line **Home Page**. Save and commit the changes as **Add table page menu**. Test the page changes in the browser. You can navigate from the home page to the table page and back to the home page.

```

<div>
<ul>
<li><a href="tablepage.html">Table Page</a></li>
</ul></div>

```

10. Switch back to the **master** branch. In the Explorer window notice again, there is only **index.html**. We made the change only in the **feature2** branch so there are no changes in the **master** branch yet. We will also deal with this later.
11. It was decided by architecture that there is a major UI change needed. It was decided to incorporate use of stylesheets for UI consistency. This is a new development effort and needs its own branch. Create a new branch named **ui-change**. Add a file named **styles.css**.

```

table {
  border-style: solid;
  border-color: red;
}
th {
  background: lightblue;
}
.company {
  font-size: 24px;
  display: block;
  background-color: pink;
}
.home{
  font-size: 12px;
  background-color: grey;
}

```

12. Also, change **index.html** to look as shown below. Then, save and commit the changes with the comment **Change to stylesheets**.

```
<!DOCTYPE html>
<html lang="en">
<head><title>The Company Home Page</title>
      <link href="styles.css" rel="stylesheet" />
</head>
<body>
<div class="company"><strong>The Company</strong></div>
<div></div>
<hr/>
<div>Home Page Content</div>
</body></html>
```

13. It is now time to merge the changes from both features and the UI change into the master branch.
14. Select the **History** view and change to the master branch. A **compare** button is now visible. Select it and, from the drop-down, select **ui-change**. The button will change to **Update from ui-change**. (It will be above the ui-change and master branches listed in the history map.) Click **Update from ui-change**. The changes will be merged, with a comment referencing the branch merge content, and committed automatically.
15. To view the results of the merge, double-click **index.html** in **Explorer** (or **Finder** on Mac). "The Company" should now be on a pink background.
16. We will now start the merge of **feature1**. First, we will examine the changes for **feature1**. Select the branch **feature1** and then select the **History** button. Examine each change in the History by clicking on the change. Notice that the first in the list is the same as the first in the master branch. Then, you see a change to add the Contact page and the reference in the index.html as a link. The first change will not cause any merge issues, but the second will cause a conflict.
17. Switch back to the master branch. The **compare** button will have an arrow on the right side that will create a drop-down, from which you can select **feature1**. Selecting that will change the button to **Update from feature1**. We won't select that just yet. First, we will examine the changes that will be applied.
18. On the right side of the screen there are two files listed and beside each file is an angle bracket indicating that they can be expanded; (on Mac you won't see an angle bracket and the files will already be expanded). Expand the file **contact.html**. Notice the entire file is new content. This will merge without an issue. Expand the file **index.html**. Notice several lines that show both deletions "-" and insertions "+". That will cause a merge conflict that will need to be resolved. Click the **Update from feature1** button and you will be presented with the opportunity to resolve the conflicts.

19. **For Windows:** Click **View conflicts**. The file **contact.html** has no conflicts. The file with conflicts has not been checked in, so it will not be committed if the **commit** button is clicked. Select **index.html** so that you can identify the conflicts.

To resolve the conflicts, the file needs to be edited. It can't be done directly in GitHub Desktop. Click the gear icon at the top left of the screen. Click **Open in Explorer**. Open the file **index.html** in an editor. Change the code to read as below and click **Commit to master**.

For Mac: Both files will be selected, but you will see 5 green blocks next to **contact.html** and 5 yellow blocks next to **index.html**. When you click **index.html** it will show a message that there are conflicting changes to the file. Click **show in finder** and edit the file to read as below and click **Commit to master**.

```
<!DOCTYPE html>
<html lang="en">
<head><title>The Company Home Page</title>
      <link href="styles.css" rel="stylesheet" />
</head>
<body>
<div class="company"><strong>The Company</strong></div>
<div></div>
<hr/>
<div>Home Page Content</div>
<div>
  <ul>
    <li><a href="contact.html">Contact</a></li>
  </ul></div>
</body></html>
```

Note: If the merge step caused merge errors in GitHub Desktop and the last line fails to merge, open **Git Shell** using the **gear** icon and select **Git Shell**. Perform a **git status**. Assure that **tablepage.html** and **index.html** have been added. If not, add them. If the results are that the merge is not yet complete, then perform a **git commit**. The commit will use the comment assigned by the merge.

20. Perform the same merge for **feature2**. Change **index.html** to read as below. Save the file. If there are issues performing the merge, GitHub Desktop will not allow the commit to complete. If the last line does not merge correctly, see the note below.

```
<!DOCTYPE html>
<html lang="en">
<head><title>The Company Home Page</title>
      <link href="styles.css" rel="stylesheet" />
</head>
<body>
<div class="company"><strong>The Company</strong></div>
```

```
<div></div>
<hr/>
<div>Home Page Content</div>
<div>
  <ul>
    <li><a href="contact.html">Contact</a></li>
    <li><a href="tablepage.html">Table Page</a></li>
  </ul></div>
</body></html>
```

21. Using Git Gui, use the **Visualizer** and view all branch history. Notice all branches lead back to the **master** branch. There are no branches now with changes that have not been applied to **master**.
22. Now, to remove the development branch **ui-change**:

For Windows: In GitHub Desktop, select the branch from the drop-down next to the **branch** icon. From the **gear** drop-down select **Delete ui-change**.

For Mac: In GitHub Desktop, select the branch from the drop-down next to the branch icon. On the menu, click **Branch → Delete...**

It will take a moment to delete the branch. Do the same for **feature1** and **feature2**.

Lab 6: Git Internals

Overview

In this lab, you will gain familiarity with Git internals. Git is essentially a Version Control System wrapped in a file system. In this exercise, we will perform several tasks that will illuminate the directory structure and the file contents. We will also use some utilities to explore the content of the files.

The basic flow will be:

- Create a repository
- Add content to the repository using Git utility commands
- View content added to the repository using utility commands
- Use file system commands to view space used by the repository

High-level Objectives

You will perform these general tasks either with Git Gui or command line:

- Create a new repository called **test** using **/git-repos** as the parent folder
- Examine the subfolders created for the repository
- Create an object in the repository using **hash-object**
- Using the **SHA1** for the object just created, use **cat-file** to display content
- Create a new file in the workspace name **README.md**
- Using the Git Gui, add the file to the index and commit it
- Using the Git Bash window, execute a Git command to retrieve the commit SHA and display the content using **cat-file**. Keep the SHA!
- Follow the chain of SHAs to the **README.md** file and display its contents as well. There are three consecutive SHAs in the chain.
- Copy the **content** from **Setup/test** to the current folder and add the **POM** and **src** folder to the repository using either Git Gui or the command line, then commit it
- Using the command line, perform a **cat-file** of the master tree
- Using the command line, stage the **test.txt** file and add a tree with a reference to the staged file using **write-tree**
- Using **read-tree**, name the tree bak. Keep the SHA!
- Update the tree as a history using **write-tree**. Keep the SHA!
- Check the staging area for content using **status**
- Commit the staged changes using **commit-tree**, using the SHA from the **read-tree** and the original SHA from the original commit as the parent. Keep the

SHA and perform a second commit-tree using the SHA from the **write-tree** above and the SHA just returned as the parent

- Display the file space used for the objects using **du**. Run the garbage collector and then display the file space used again. Observe the space savings.

Detailed Instructions

Note: For several steps, hash values will be generated and reused later in the exercise. For convenience sake, copy the hash values to a simple text editor like Notepad (Windows) or TextEdit (Mac). Each time a hash value is to be reused, we will let you know so you can make note of it. Also, your 40-digit hash values will likely not be the same as in the example below.

1. Open the **Windows Explorer** (Windows) or **Finder** (Mac) and create a folder and subfolder in the root of the file system named **/git-repos/test**. Navigate to that new folder. If it already exists, delete the contents of the folder.
2. Open a **Git** window.
 - A. On Windows use the **Git Bash**.
 - From the context menu, click **Git Bash Here**
 - Or, open it via the **Start** button → **All Programs** → **Git** → **Git Bash**
 - B. On Mac, you can open the **Terminal**.

Change the directory to the folder just created. Create a new repository using the **git init** command:

```
git init
```

3. Change to the **.git** folder (this may be a hidden folder, depending on your system settings). Execute a **dir** or **ls** command on the **.git** folder. There will be several new folders created as a result of the **git init** that was done by the **Git Bash**.

New folders:

hooks - contains several sample hooks that can be enabled

info

objects - objects that have been committed

refs - references to objects by hash for both heads and tags

There will also be three new files:

HEAD

config

description

4. Change back to the root of the **test** repository. Execute the **find** command to see that no objects have yet been created in the repository.

```
find .git/objects -type f
```

There are no objects yet, so there is nothing to display.

5. Create some content in the database, not a file yet, but some content.

```
echo 'this is a test' | git hash-object -w --stdin
```

The result is a hash value representing the content we just put in the database.

```
90bfc510602aa11ae53a42dcec18ea39fbd8cec
```



Note: Make a note of the hash value as you will use it in steps 7 and 17.

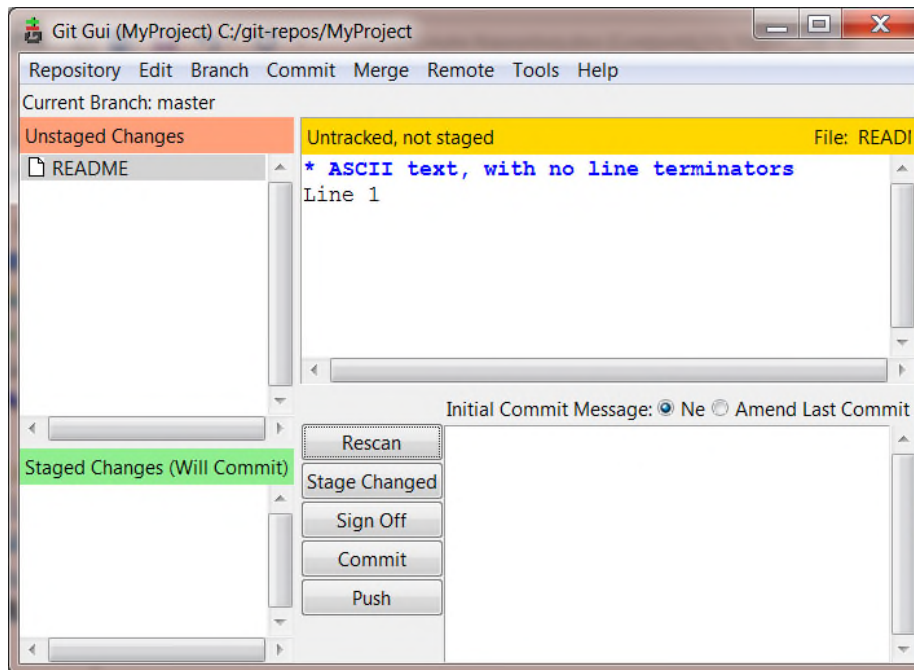
6. Execute the **find** command again and now you have an object in the repository.
7. To print the content of the file back out, use the Git command **cat-file** and print the contents of the object.

```
git cat-file -p 90bfc510602aa11ae53a42dcec18ea39fbd8cec
```

The result printed will be the text we originally put into the document.

Note: Use the hash value that was returned to you in step 5.

8. Using the editor of your choice, in the folder **/git-repos/test**, create a file named **README**. Add one line to the file with the content **Line 1**. Save the file.
9. In the Git Gui, click the **Rescan** button and the screen will look like below. If the **README** file is not displayed in the **File** window, select it.



10. Click the **Stage Changed** button. A dialog box will pop up confirming that you want to Stage the changed file. Click **Yes**. The **README** file moves from the top box to the bottom box on the left under **Staged Changes (Will commit)**. If you select the **README** file, you will see the changes as a new file with a mode setting and **+Line 1** with no new line at the end.
11. Enter a commit comment **Initial Commit** in the dialog box on the right. Click the **Commit** button. You will notice that there are no files being displayed. That is because there are no files with changes. Until a change is made to the file again, there is nothing to display.
12. In the Git Bash window, we will again execute the **find** command. There will be several objects now in the **objects** folders. The **cat-file** command can be used on any of the files just added to determine what is in them. To look at the contents of the committed readme file, use the **git log** command to find out the most recent commit.

```
git log
```

returns:

```
commit 96bc63072658446520f7bac9c24c52f08c388416
Author: ...
Date: ...
Initial Commit
```

Note: This hash value will be used in step 21 and in the next part of this step. In addition, each part of this step uses the hash value previously returned.

The value following the commit is an ID that will allow **cat-file** to reference the object which represents the commit information.

```
git cat-file -p 96bc63072658446520f7bac9c24c52f08c388416
```

returns:

```
tree 07c82b558b4984838a495c6d11df2d1d0a9b6ae8
author ...
committer ...
```

This will give the ID of the object that was committed. Using the **cat-file** on that object ID will give us:

```
git cat-file -p 07c82b558b4984838a495c6d11df2d1d0a9b6ae8
```

returns:

```
100644 blob 3be9c81fd3d8fcf01e59701e5ee48041bcdf8631    README
```

The output of that command is an object ID that will represent the content of the file:

```
git cat-file -p 3be9c81fd3d8fcf01e59701e5ee48041bcdf8631
```

returns:

```
Line 1
```

13. Copy the folder content from the **Setup/test** folder to the current folder. The folder should now have a folder **src** and two files, **pom.xml** and **README**.

Only on Macs, another file **.DS_Store** is created. It is used by OS X to track file attributes. This file is not relevant for Git so you can use the following command for Git to ignore it. You can ignore any **.DS_Store** related differences in your output throughout the exercise.

```
echo .DS_Store >> $HOME/.config/git/ignore
```

14. Go to the **Git Gui**, rescan the folder and stage the changes. Commit the code so that we have a set of changes for the tree structure. Use the comment 'Make this a maven project with a prototype Java application.' There will now be a more normal repository structure, as most repositories don't have just one file.
15. One way to view the tree is by using the **cat-file** command. Go back to the **Git Bash** command line and enter the **cat-file** command, but rather than providing an ID, provide the branch reference and specify to include the tree.

```
git cat-file -p master^{tree}
```

returns:

```
100644 blob 3be9c81fd3d8fcf01e59701e5ee48041bcd8631 README
100644 blob cc247eee6579cc0c86a5f328e92aad3bceeb6fdc pom.xml
040000 tree 7481ffdd7a28a38e2f08a07e05334222f368d6e0 src
```

The results above show that there are two file objects and one directory tree.

16. Using Git to manually create a staged change, go back to the Git command line and execute the **update-index** command (all on one line):

```
git update-index --add --cacheinfo 100644
90bfc510602aa11ae53ag42dcec18ea39fbd8cec test.txt
```

Note: Use the hash value that was returned to you in step 5.

17. To write the content to a tree for later commit, execute a **git write-tree**. The returned id will be used when the tree is committed later.

```
git write-tree
```

returns:

```
37d5f7b435bb0b9f80aaed23f68bcc61434ab703
```

Note: This hash value will be used in steps 18 and 21.

18. To give the tree a name in the file hierarchy, use the **read-tree** and add a prefix. The command is:

```
git read-tree --prefix=bak
37d5f7b435bb0b9f80aaed23f68bcc61434ab703
```

19. This change needs to be associated with a history so perform another **write-tree**.

```
git write-tree
```

returns:

```
e247d699af2b6bde23524116bff8993d11bb5f95
```

Note: This hash value will be used in step 22.

20. It is now time to commit the changes we have made. Before the commits, display the **status** of the changes.

```
git status
```

returns:

On branch master

Changes to be committed:

(use "git reset HEAD <file>..." to unstage)

```
    new file:   bak/README
    new file:   bak/pom.xml
    new file:   bak/src/main/java/com/example/app/test/App.java
    new file:   bak/src/test/java/com/example/app/test/AppTest.java
    new file:   bak/test.txt
    new file:   test.txt
```

Changes not staged for commit:

(use "git add/rm <file>..." to update what will be committed)

(use "git checkout -- <file>..." to discard changes in working directory)

```
    deleted:    bak/README
    deleted:    bak/pom.xml
    deleted:    bak/src/main/java/com/example/app/test/App.java
    deleted:    bak/src/test/java/com/example/app/test/AppTest.java
    deleted:    bak/test.txt
    deleted:    test.txt
```

21. Perform a commit using the utility method, rather than the regular commit since this is a tree. The command is **commit-tree** specifying the tree id and the previous commit id.

```
echo 'commit comments.txt' | git commit-tree
37d5f7b435bb0b9f80aaed23f68bcc61434ab703 -p
96bc63072658446520f7bac9c24c52f08c388416
```

returns:

```
e60f0ef70978f61708d801f878856bb01b15803f
```

Note: For the first hash value, use the one returned in step 17 and; for the second, use the one returned in step 12. This returned hash value will be used in step 22.

22. To commit the second change of adding the **comments.txt** to the **bak** folder, there needs to be a **second commit**.

```
echo 'second commit' | git commit-tree  
e247d699af2b6bde23524116bff8993d11bb5f95 -p  
e60f0ef70978f61708d801f878856bb01b15803f
```

returns:

```
7aea98306c837becf0994a40bb48146e5c53f81d
```

Note: For the first hash value, use the one returned in step 19; and for the second, use the one returned in step 21.

Use the returned hash value returned to update the HEAD reference so that the commit is not a dangling commit which will prevent it and any other commits after this from showing when the git log is executed.

```
git update-ref HEAD 7aea98306c837becf0994a40bb48146e5c53f81d
```

Instructor Note: During the lab review, the instructor will show you how the workspace would be modified to reflect the changes in the repository. By doing a hard reset, the folder bak and its content will show up in the folder.

23. Run garbage collection.

```
du -sh
```

returns: 108k (number may be different)

```
git gc
```

returns on objects counts: Total 24 (delta 3), reused 0 (delta 0)

```
du -sh
```

returns: 79k (number may be different)

24. In this lab, you performed all the steps in a normal change, stage and commit in utility mode. No staging area was created and no files were created.

Lab 7: Git Configuration

Overview

In this lab, you will gain familiarity with Git configuration settings. There are many configuration settings that can be used to make Git simple to use. We will perform several tasks to customize how Git operates, including creating aliases.

The basic flow will be:

- Set user configurations
- Set global configurations
- Enable the ID trigger

High-level Objectives

You will perform these general tasks either with the command line or Eclipse:

- Clone a repository from **Setup/lab2**
- Display the **global** configuration and the **local** configuration
- Configure the **user name** and **email**
- Create a **README.md**, edit it, stage it and commit, and display the committer
- Change the configured user name and email
- Edit the **README.md**, stage the changes, commit and display the committer
- If you have installed **Eclipse**, open it, view the Git configuration and edit the user name and email (works the same in NetBeans)
- Repeat the edit, commit and display committer to demonstrate the Eclipse updated the correct file
- Update the **.gitattributes** file to always add the **ID** to Java files and treat **README.md** as a text file
 - Edit and commit the **/src/main/java/com/example/lab2/App.java** file
 - Clone the repository into a working repository, display the content of **App.java** and notice that an ID has been added on checkout
- Enable the hook for **prepare-commit-msg**. Update the **App.java** file and commit it
- Display the commit. Notice that **Signed Off** was added by the hook

Detailed Instructions

1. Open the Windows Explorer (Windows) or Finder (Mac) and create a folder and subfolder in the root of the file system named **/git-repos/**. Navigate to that new folder. Remove any **lab2** folder and its contents.
2. Open a Git window. On Windows use the **Git Bash**. It will be in the main list of programs when you click the **Start** button. On Mac you can open the Terminal. Change the directory to the folder just created. Create a new repository using the **git clone** command:

For Windows: **git clone file:///c:/setup/lab2**

For Mac: **git clone file:///Users/{yourUserFolder}/setup/lab2**

3. To display the current Git configuration setting, enter the command:

```
git config --global --list
```

Results will be similar to:

```
core.symlinks=false  
core.autocrlf=false  
color.diff=auto  
color.status=auto  
color.branch=auto  
color.interactive=true  
help.format=html  
http.sslcainfo=C:/Program Files/Git/mingw64/ssl/certs/ca-  
bundle.crt  
diff.astextplain.textconv=astextplain  
rebase.autosquash=true  
credential.helper=manager  
user.name=Git User  
user.email=gituser@example.com  
...  
push.default=simple
```

4. To display the local configuration specific to the current repository, execute the same command but change **--global** to **--local**. The results will be a much shorter list of configuration settings.

```
core.repositoryformatversion=0  
core.filemode=false  
core.bare=false  
core.logallrefupdates=true  
core.symlinks=false  
core.ignorecase=true  
core.hidedotfiles=dotGitOnly
```


5. The first thing to work with is the email and user name information. Rather than setting all the repositories to use this new email, by using the **--local** option, it is possible to have each repository commit with a different email address. Set the email to your email address using the configuration command:

```
git config --global user.name "Git User"
git config --global user.email gituser@domain.com
```

This will set the global user name and email provided to commit changes to the repository. To do the same thing for the current repository use the **--local** parameter instead.

```
git config --local user.name "Different User"
git config --local user.email gituser2@domain.com
```

You can also leave **--local** off entirely. Without a **--local** or **--global** parameter Git assumes to you mean **--local** so the next 2 lines do the same thing:

```
git config --local user.name "Different User"
git config user.name "Different User"
```

6. Open the working directory using your file explorer. Edit the file **README.md** and add a line:

Line 2

Save the file. In the Git window, add the changed file and commit:

```
git add README.md
git commit -m "2nd Commit"
git log
```

The log command will show that the change was committed by the user Git User.

7. A different way to set those options would be to open the configuration file in an editor and set the settings. Open the **.git/config** file from the current working folder using the editor of your choice. Notice the last three lines were added to the configuration file for this repository.
 - Edit the user name to be **Git User3**.
 - The email to be **gituser3@domain.com** and save the file.

Run the **configuration list** command and notice the name changed to **Git User3**. Notice that the results reflect a change the same as when you use the Git command to change the settings.

```
git config --list
```

To query a specific setting:

```
git config user.name
```

8. Open the working directory using your file explorer again. Edit the file named **README.md** and add a line:

Line 3

Save the file. In the Git window, add the changed file and commit:

```
git add README.md  
git commit -m "3rd Commit"  
git log
```

The log command will show that the change was committed by the user **Git User3**.

9. Setting configurations is much easier using Eclipse. Open Eclipse using the desktop shortcut created during installation.
 - For Windows: Open **Window** → **Preferences**.
 - For Mac: Open **Eclipse** → **Preferences**.
 - Navigate to **Team** → **Git** → **Configuration**.

Notice that there are three tabs for configuration settings for Git:

- **User Settings**
- **System Settings**
- **Repository Settings**

The User and System settings apply across projects. The Repository Settings tab contains a drop-down list to edit settings per repository.

- Select the **User Settings** tab, change the **User Name** to something different and apply the settings.
- On the command line, view the configuration as was done in the previous step.

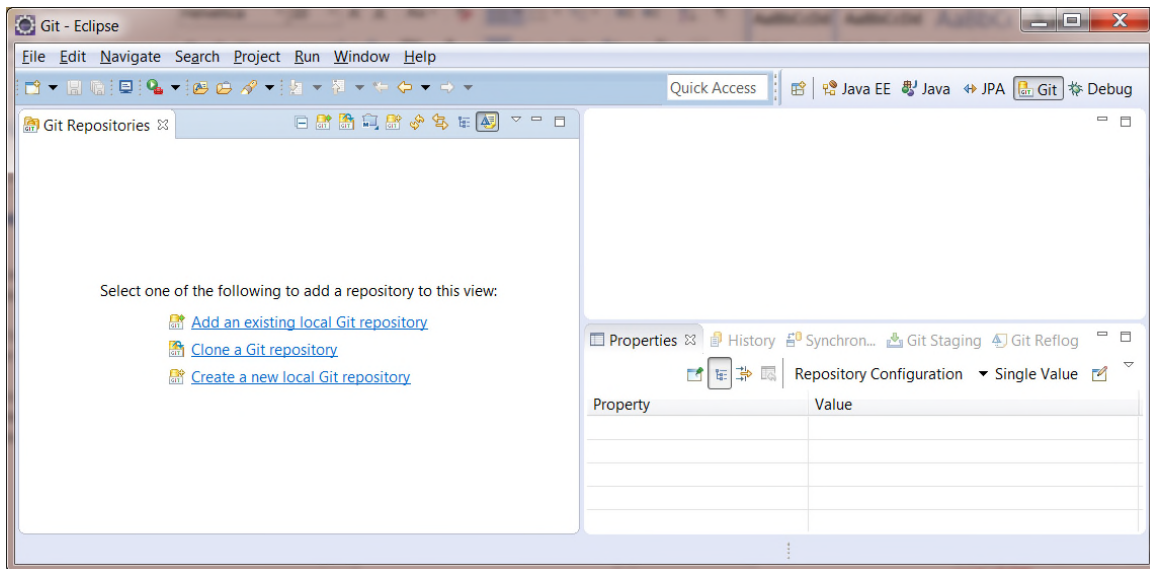
```
git config --global user.name
```

Note that the **--global** parameter is used here because the **User Settings** are global. Notice, too, that the configuration settings applied in Eclipse also apply to the Git command line because the user configuration file is being edited in Eclipse.

- In Eclipse set the user name back to the correct name.

10. In this step, you will import the **lab2** repository into Eclipse and configure its properties.
 - A. In Eclipse, select the **Git** tab. See picture below.
 - B. Select **Add an existing local Git repository**. Either enter the directory or **Browse** to and select the location used for the repository **lab2** created above.
 - C. Check the checkbox beside the folder shown in the bottom window and click **Finish**.

- D. Right-click on the **Working Tree** and select **Import projects...**
- E. In the dialog select **Import existing projects** radio button.
- F. Click **Next** and click **Finish**.
- G. Right-click on **lab2** and select **properties**. Set the user name to a different user name so that commits to **lab2** will use a different name than commits to other projects. This can be helpful when committing changes for someone else while they are absent. Apply the changes.



From the project folder on the command line, verify that the settings are applied. This can be done by making additional changes to the file then saving, committing and logging the history of commits.

11. Attributes cannot be set using Eclipse. EGit and JGit do not support the **.gitattributes** files at all. If there are **.gitattribute** settings that need to be applied, they must be activated using the command line.

Create a **.gitattributes** file in the root of the project. Add the following content to the file.

```
* text=auto
```

12. There are other settings in **.gitattributes** that may be useful.
 - Add the following to the content of **.gitattributes** to preserve the content type of the **readme.md** file.

```
readme.md text
```
 - To add the ID to the Java source when committing the changes to Git, add the following:

```
*.java ident
```

13. To see the output of the change:

- Edit **src/main/java/com/example/lab2/App.java**.
- Add a new comment line with the content * **\$Id\$** to the already existing comment section.

```
git add src/main/java/com/example/lab2/App.java
git commit -m "add IDs to the java"
```

Fetching the content from the repository in the future will return the Java source in App.java with an ID reference.

To see that, clone the repository to a dummy folder using the command:


For Windows: **git clone file:///c:/git-repos/lab2 test3**

For Mac: **git clone file:///git-repos/lab2 test3**

To display the file in the console, use the command:

```
cat test3/src/main/java/com/example/lab2/App.java
```

14. In this step, you will enable one of the hooks to demonstrate how hooks get applied when committing changes to Git.

- In the **git bash** window or terminal copy the hook **.git/hooks/prepare-commit-msg.sample** to **.git/hooks/prepare-commit-msg** in order to enable triggering the hook when Git commits content.
- In Eclipse, open the **Git perspective** and select the project **lab2**. Click the refresh icon, which is two curved arrows, at the top of the tab. 
- Expand the project, the **Working Tree**, the **.git** folder and the **hooks** folder and then open the **prepare-commit-msg** file. Remove the **#** from the beginning of the bottom two lines. Save the file.
- Change to the Java perspective and find the **Package Explorer** window on the left and find the project and expand. In the package **com.example.lab2**, open the **App.java** file and make a change to the file.
- The hook will only get used if the commit is done using Git from the command line. So, go to the command line, add the changed file and commit it.

```
git add src/main/java/com/example/lab2/App.java
git commit -m "a different change"
```

Results from the command line are:

```
[master 66c6e05] a different change Signed-off-by: Git User
<gituser@example.com>
```

1 file changed, 1 insertion(+)

If you follow the same steps in Eclipse on Windows and commit from within Eclipse, the hook will not be executed (under most circumstances).

15. This lab demonstrated the following:

- Configuration of the Git environment
- Making some changes to the configuration and seeing the results
- Adding hooks to Git and executing them

Lab 8: Rebase

Overview

In this lab, you will practice rebasing a branch in a Git repository. You will use the GitHub Desktop to create and manage branches. You will initialize a repository and track changes to a group of files in two separate branches. You will get see how easy it is to create branches and merge changes between branches.

This will simulate using two separate development paths. Imagine a developer working on a new feature is interrupted by the need to push out a hotfix to production. We will create a branch for the hotfix and do the main development on the master. We will then rebase the feature onto the main branch. Finally, we will merge the hotfix onto the main branch.

High-level Objectives

You will perform these general tasks using GitHub Desktop and the command line:

- Create a new repository using GitHub Desktop called **another-web-project**
- Create a file named **index.html** with simple HTML content and commit the change. (Step 3 has the syntax.)
- **Create a branch hotfix**
- Create a file named **contact.html** with simple HTML content and commit the change. (Step 5 has the syntax.)
- Switch back to the **master** branch
- Edit the **README.md** file
- Create a file named **firstpage.html** and commit the change. (Step 7 has the syntax.)
- From the Git command line, checkout **hotfix** and perform the **rebase** to the **master**
- View the history using the logs and Git Gui
- Make a change to the **contact.html**
- Checkout the **master** and **merge** branch **hotfix** and commit
- Verify the code has the content from **hotfix** plus all the changes to the **master**
- Verify the history log shows the **master** fixes were applied before the **hotfix** fixes

Detailed Instructions

1. Open GitHub Desktop. Create a new repository and call it **another-web-project**. You will get a new repository in the GitHub directory of your user document folder.
2. Create a new file named **README.md**. Set the content to "**This is another web project which we will use to illustrate a rebase**". Add the file and commit the changes using the comment **Add the README.md**. Use **git** to display the log.

– Hint: In **GitHub Desktop**, click **Open in Git Shell** (**Open in Terminal** on Mac) and type **git log**.

3. We will create a web page named **index.html** and track changes to it. Create a new file named **index.html** and add the content below to the file. Commit the change using a comment **Add the web page**.

```
<html><body>
Hello World
</body></html>
```

4. Create a new branch named **hotfix** and switch to it.
5. Create a new file named **contact.html** and the content below to the file. Commit the change using a comment **Add the contact page**.

```
<html><body>
Contact:
The Company
123 Any Street
Anytown, TX, 77777
<hr>
Phone:      555-555-1212
<hr>
Email: somebody@somecompany.com
</body></html>
```

6. Switch back to the **master** branch. Edit the **README.md** so we have a different stream of changes between our master branch and the hotfix branch.
7. Add a new HTML file named **firstpage.html** and add the below content to it.

```
<html><title>First Page</title><body>
First Page
</body></html>
```


8. Save the file and commit the changes using the commit comment **Add the First Page**. We are still in the **master** branch.
9. In the **Explorer**, launch the **Git GUI** on the repository; (for Mac, execute **git gui** in the **Terminal**). Click the **Rescan** button and there will be no changes. From the **Repository** drop-down, select **Visualize All Branch History**. Verify that the **hotfix** branch changes are between the top two changes on the **master** branch.
10. Now to perform the rebase and merge, we will use a command line environment. If you have closed the command line environment, open it.
 - For Windows: In GitHub Desktop, right-click on your repository and select **Open in Git Shell**. (This is would be similar to opening it in **Git Bash** or **Git CMD**.)
 - For Mac: In GitHub Desktop, two-finger-click on your repository and select **Open in Terminal**.
11. To make sure the workspace is pointing at the hotfix branch, execute **git checkout hotfix**. This makes the **changeset** of the branch **hotfix** the current branch in the file system. This can be verified by looking at the file system in the file browser and checking the content of the files. There should be a file named **contact.html**. There should not be a file named **firstpage.html**.
12. Switch back to the **master** branch. Notice there is now a file named **firstpage.html** and there is not a file name **contact.html**.
13. Switch back to the **hotfix** branch.
14. Rebasing applies the **changesets** in the current repository branch against the branch specified as the target. It applies the changes into temporary files and resets the current branch to the same commit as the target. To apply the hotfix changes to the master branch, execute **git rebase master**. You will get a prompt:

Applying: Add the contact page

15. Notice that there is now a **firstpage.html** and a **contact.html**. The **hotfix** branch now has been attached to the tail of the **master** branch. Since the last change to the **master** branch was to add the **firstpage.html** the file is now visible.
16. Using the **Git GUI** again, **Visualize All Branch History**. Notice that **hotfix** is now at the top of the chain of changes.

Because we did the rebase, when we next merge changes from **hotfix** into **master**, they will be merged as a fast-forward, since the baseline for the **hotfix** is the tip of the **master** branch.

17. Checkout **hotfix** branch. Add the following line to the **contact.html** after the **email** line. Commit the change as **Add the thank you message**.

```
<strong>Thanks for your interest.</strong>
```

18. To finish the **hotfix**, we need to apply the **hotfix** branch back to the **master**. Change to the **master** branch and then finish the operation with a merge. Execute the following:

```
git checkout master
git merge hotfix
```

The following will show in the console.

```
Updating 8205473..73e6021
Fast-forward
 index.html | 3 +++
 1 file changed, 3 insertions(+)
 create mode 100644 index.html
```

Notice the merge was a fast-forward merge.

19. Verify that the rebase and merge completed. To verify, look at the file system and see that all four files are there. There are **contact.html**, **firstpage.html**, **index.html** and **README.md**.
20. Using the **Git GUI** again, **Visualize All Branch History**. Notice that the **hotfix** and **master** are now at the top of the chain of changes.
21. Now it is safe to remove the hotfix branch as there are no more unmerged changes. Execute the following:

```
git branch -d hotfix
```

Lab 9: Git Sharing

Overview

In this lab, you will gain familiarity with Git collaboration techniques. These include remote repository creation, archiving, patching and cherry-picking techniques.

The basic flow will be:

- Creating the shared repository
- Setting a connection to a remote repository
- Splitting** a large repository into smaller repositories
- Creating an **archive** for a specific version
- Creating a **patch** that covers a range of versions
- Perform a **rebase** from the changes in the workspace
- Incorporate a change by **cherry-picking** the changes to use

High-level Objectives

You will perform these general tasks:

- Create a bare repository named **sharelab.git**
- Clone the repository as **lab3**
- Copy content from **Setup/sharelab**
- Update **lab3** from the new content
- Verify everything is ok by checking out **feature1**, **feature2**, **v1.0.1** and **master**
- Using Eclipse or the IDE of your choice, view the contents of each **changeset**
- Rebase **v1.0.1** to **feature2**. Manually correct the **.classpath** errors, if any
- Cherry-pick **170daa** to **master**. Correct the **.classpath** errors manually. Keep the second definition. Commit the changed files. (Should be **.classpath** and **CourseManager.java**.)
- Using **Setup/splitlab.git** as the content, create a **bare** repository and clone to **splitlab** as the working repository
- Clone from **splitlab.git** to **courseWeb**. Using **filter-branch**, remove all but the subdirectory **courseWeb** from the repository, creating a new repository for each project. Do the same for **course-jsf** and **course-jpa**
- Using the Git **zip** command, create an archive of the latest commit on the **master** branch with a **zip prefix** of **test** and the six-digit ID of the commit
- Create a new repository and named **patchtest** and initialize, add a file and edit it. Stage the changes and commit them. Create a patch set. Create a blank repository named **patchtest2** and apply the patch created from **patchtest** to

patchtest2. Compare the two repositories. The working content should be identical.

Detailed Instructions

Part I. Setting Up the Project

1. Open the **Windows Explorer** (Windows) or **Finder** (Mac) to the subfolder in the root of the file system named **/git-repos**. Assure that there is no repository at **/git-repos/sharelab.git**. If there is a folder there, remove it.
2. Create the shared repository. Open the **Git Bash** shell and change to **/c/git-repos** on Windows and **/git-repos** on Mac. Execute the command below:

```
git init --bare sharelab.git
```

3. Clone the repository to the **lab3** folder as a local working tree. In the bash shell the repository can be cloned using the command:

For Windows: **git clone file:///c:/git-repos/sharelab.git lab3**

For Mac: **git clone file:///git-repos/sharelab.git lab3**

Move to the **lab3** directory for the next step.

4. Perform a **git branch** to see what the content of the repository is:

```
git branch
```

Results will be empty as there are no branches yet.

5. To create the initial branch, there needs to be some content and a commit. Create a **README.md** file and add a line to the file to describe the purpose of the project. Use 'This project will demonstrate the process of merging content from multiple stages of development':

```
git add .  
git commit -m "Initial Release"
```

Results will be similar to:

```
[master (root-commit) 588c8c5] Initial Release  
1 file changed, 1 insertion(+)  
create mode 100644 README.md
```

Confirm that change with **git log** and there will be one change made by the current user.

6. To illustrate that there has been a lot of development, copy the repository from the **Setup/sharelab.git** folder to replace the one in **git-repos**. This repository has several changes to illustrate a project being modified over time with multiple branches.
7. It is good practice to do a **git fetch** after any changes to the remote repository:

```
git fetch
```

8. In order to view the contents of the branches we will checkout each branch. In the **lab3** folder do the following, which will switch the branch to the respective branch:

```
git reset --hard origin/master
git checkout feature1
git checkout feature2
git checkout v1.0.1
git checkout master
```

This makes the branches available locally.

9. Most developers use an IDE. To demonstrate accessing the Git content in the IDE the repository needs to be linked in the IDE. Open Eclipse, in the Git perspective, right-click the **lab3** project and switch to master branch if not already there.

Since **lab3** is not yet added to Eclipse, add it by clicking:

- A. **File → Import...**
- B. **Git → Projects from Git → Existing Local Repository**
- C. **Add... → Browse... → (Find git-repos/Lab3 in the root of your drive) → Finish**
- D. **Next → Import existing Eclipse projects**
- E. **Next**
- F. Deselect all but **course-parent/course-jpa**, **course-parent/course-web**, **course-parent/courses-jsf** There are several other files within the directory structure that Eclipse thinks are projects, but they really aren't. We don't want to register them as projects in Eclipse so we deselect them now.
- G. **Finish**

10. Change the perspective to the Java perspective. (You can change perspectives by clicking the Window menu at the top of Eclipse, selecting Open Perspective, finding Java perspective and double-clicking it.)

Part II. Rebasing

1. All of the branches have different changes in the **CourseManager** class in **course-jpa**. They each add a different method to find a course.
 - A. Click on the **course-jpa** project, expand the **src/main/java** folder.
 - B. Expand the **com.example.course.entity.controller.CourseManager.java** class.

The changes are all at the end of this class.

- C. On the Git tab, right-click the **lab3** and select:

Team → Switch to → feature1

This will switch the branch to the **feature1** branch.

- D. Switch to the **master** branch and then switch to the **v1.0.1** branch.

2. Now you will perform the rebase.
 - A. Change the branch in **lab3** to the **feature2** branch.
 - B. Right-click the project **course-jpa** and select **Team → Rebase...**
 - C. Select the project branch **v1.0.1**. Click **OK**.
 - D. A popup will indicate what is about to be rebased. Click **OK**.

This will apply the changes in the class that were made as part of the **feature2** branch to the changes made in branch **v1.0.1**.

3. The rebase will need to be resolved.
 - A. On the Git tab right-click **lab3**. Navigate to the **.classpath** files in both **course-jpa** and **course-web**. Right-click and **Add to Index**.
 - B. In the bottom, right window, click the link to **Open Staging View** and click the **Continue** button.

The changes of **branch v1.0.1** are now part of **feature2**.

Note: Alternative solution: Reset the rebase by switching to the command line and doing a manual merge of the **.classpath**. Then still in the command line, execute:
`git rebase --continue`

Part III. Cherry-Picking

1. The next operation is going to illustrate using cherry-picking to update the **master branch**. This has to be performed from the command line, as Eclipse doesn't support cherry-picking.
2. Before a commit can be cherry-picked it must be brought from the remote repository to the local repository. That is accomplished with a pull.

In the command window, perform the **git cherry-pick** of the change that needs to be applied. This will be merged and committed automatically into **HEAD**. Follow that with a **push** to merge to the shared repository.

```
git pull
git checkout master
git cherry-pick 170daa
```

3. This might not work depending on the success with the previous steps, potentially, giving you the following error:

```
Error: could not apply 170daa2... Feature 1 finished - Course by
description
```

If it has an error and you do **git status**, it will show that **course-parent.course-jpa/.classpath** is the file that caused the error. The merge conflict needs to be resolved. To resolve conflicts the file needs to have the conflicting lines resolved in some way and then the files added to the staging area so the change can be committed.

```
Both modified: course-parent/course-jpa/.classpath
```

4. Edit the files and remove the second classpath, but keep the first.
5. Then continue as follows:

```
git add course-parent/course-jpa/.classpath
git add course-parent/course-
    jpa/src/main/java/com/example/course/entity/controller/CourseM
    anager.java
git commit -m "commit cherry-pick" --allow-empty
git push
```

Part IV. Filter-Branch and Archiving

The purpose of this part of the exercise is to split each of the projects into its own repository. A straight clone will not work as each of the repositories will have all the other projects in them. To remove the projects that are not intended to be part of the new repository the **filter-branch** command will be used. The filter will be applied to the contents of the subdirectory and only return the content from the subdirectory **course-web**.

1. Copy **/Setup/splitlab.git** to **/git-repos/splitlab.git** to prepare for the split.
2. Clone the repository **/git-repos/splitlab.git** to **/git-repos/course-web**.
3. Open the cloned repository in Git Bash. Split the **webapp** from the **content** to create a second repository.

```
git filter-branch --prune-empty --subdirectory-filter course-web
master
```

Results:

```
Rewrite 123 (1/1)
Ref 'refs/heads/master' was rewritten
```

This could now be shared as a new project **course-web**, which has the content from that folder.

4. Do the same for each of the projects in the repository: **course-jsf**, **course-jpa**.
 - Clone to **git-repos/course-jsf** etc.
5. Now, you will create an archive. Change the directory to **/git-repos/course-jsf**. Create an archive of the branch using the ID of the most recent commit.
 - To find your commit ID do **git log** and use the first 6 digits of the 40-digit commit ID. Use the commit ID instead of **abcdef0** in the command below.

```
git archive --format=zip abcdef0 -o ../testabcdef0.zip
```

Note: You can also use the branch name instead of the commit ID.

View the content of the archive using the archive manager tool for the OS. There will be a number of files in the archive.

Part V. Patching

In this part, we will create a patch. It is a set of differences between two commit points often used when exchanging deltas with email rather than sharing the repository directly. We get two commit points and create a patch between the two.

1. Create a new empty repository and name it **PatchTest**.

```
git init PatchTest
```

2. Add a file named **README.md** and add one line to the file **Hello**.
 - Commit the change with **Hello** as the commit message.
3. Add another line to the file **Hello Again!**
 - Commit the change with **Hello2** as the commit message.
4. Do a **git log** and you will see two commits, each with their own commit ID.
5. Now create a patch:

```
git format-patch -k -o patch commitID1 commitID2
```

results:

```
patch/0001-Hello.patch  
patch/0002-Hello2.patch
```

6. Create a new empty repository and name it **PatchTest2**. Move to the **PatchTest2** folder.
7. Apply the patch:

```
For Windows: git apply /c/git-repos/PatchTest/patch/*  
For Mac: git apply //git-repos/PatchTest/patch/*
```

8. Now when you check the contents of the folder you will see the file **README.md** created in the **PatchTest** repository and it will have the same content.
9. Create a new empty repository and name it **PatchTest3**. Move to the **PatchTest3** folder.
10. You can also apply a single patch file separately.

```
For Windows: git apply /c/git-repos/PatchTest/patch/0001-  
Hello.patch  
For Mac: git apply //git-repos/PatchTest/patch/0001-Hello.patch
```

11. Now when you check the **README.md**, the content will be like it was after the first commit.

This lab illustrated the following:

- Several ways to manage repositories and changes between repositories.
- Using remote repositories allows for collaborative sharing of content.
- How to split repositories when they become large and less manageable.
- How to export a snapshot of development for use outside of Git. For instance, when exchanging sets of changes with individuals that don't have access to the repository.
 - This can be done using archives.
 - Patches can also be used to exchange those proposed changes.
 - The team can pull in those changes and decide what to keep and what not to.

The lab also demonstrated two techniques for applying different sets of changes.

- Rebasing, which applies current changes in the local context to a different baseline.
- Cherry-picking, which allows the merge of specific change sets with the current local context.