LFD109x

GIT for Distributed Development

Version 2022-03-22

The Linux Foundation | Training & Certification

Version 2022-03-22

Published by:

the **Linux Foundation**
https://www.linuxfoundation.org

No representations or warranties are made with respect to the contents or use of this material, and any express or implied warranties of merchantability or fitness for any particular purpose or specifically disclaimed.

Although third-party application software packages may be referenced herein, this is for demonstration purposes only and shall not constitute an endorsement of any of these software applications.

Linux is a registered trademark of Linus Torvalds. Other trademarks within this course material are the property of their respective owners.

If there are any questions about proper and fair use of the material herein, please go to https://trainingsupport.linuxfoundation.org.

Nondisclosure of Confidential Information

“Confidential Information” shall not include any of the following, even if marked confidential or proprietary: (a) information that relates to the code base of any open source or open standards project (collectively, “Open Project”), including any existing or future contribution thereto; (b) information generally relating or pertaining to the formation or operation of any Open Project; or (c) information relating to general business matters involving any Open Project.

This course does not include confidential information, nor should any confidential information be divulged in class.

Contents

Chapter 1

Introduction to GIT

Exercise 1.1: cgit Example at git.kernel.org

- To see a comprehensive implementation of the **cgit** repository browser, go to https://git.kernel.org which serves as the host for many projects growing out of the **Linux** kernel community, including **git** itself.
- Look at a particular repository by scrolling down to the project and clicking on `summary`. See if you can find the main **Linux** kernel repository. Look at the history and patches etc.
- This is not the same as looking at https://kernel.org which is also powered by **cgit**.

Chapter 2

Git Installation

Exercise 2.1: Getting the Latest Git with Git, and Compiling

- If you already have **Git** installed, a nifty way to get the latest version is to use **Git** itself. If your network connection is good enough, download from the public repository by doing:

  ```
  $ git clone -v https://github.com/git/git.git
  ```

  if you do not have **git** already installed, you can download a zipped archive from the same site, and then unpack it and the rest of the instructions for compiling and installing will be the same.

- The full **Git** repo will take up about 145 MB after being downloaded. Most of this space is taken up by the `.git` directory, which contains the whole revision history. The archive itself is much smaller.

- If you have **git** installed you can see what version of you are running with:

  ```
  $ git --version
  git version 2.27.0
  ```

  To see what the latest version of **git** is in the downloaded repository, go to the `git` directory created during the cloning operation:

  ```
  $ cd git
  ```

  There are a number of methods, including:

  ```
  $ git tag
  ```

  ```
  .....
  v2.29.3
  ....
  ```

  where we have located the highest version number that is tagged. However, that might not be accurate as use of tags is voluntary. For example, doing:

  ```
  $ git log
  ```

  ```
  commit 7e391989789db82983665667013a46eabc6fc570 (HEAD -> main, origin/main, origin/HEAD)
  Author: Junio C Hamano <gitster@pobox.com>
  Date:   Fri Apr 30 13:38:07 2021 +0900
  ```

```
    The thirteenth batch

    Signed-off-by: Junio C Hamano <gitster@pobox.com>
....
```

may indicate a later version, which we can check once we prep, compile, and install using the steps we mentioned earlier:

```
$ cd git
$ make
$ sudo make install
```

- With this particular version, after compilation one finds:

  ```
  $ ./git --version
  ```

  ```
  git version 2.31.1.442.g7e391
  ```

- By default if you compile and then install everything will be placed under your `$HOME` directory, with the executables all in the `bin` directory therein. However, you can specify an alternative when you compile as in:

  ```
  $ make prefix=/usr/local
  ```

  or

  ```
  $ make prefix=/opt
  ```

  etc. If you then place the new directory in your path, you will be using the newer version of **git** rather than the installed ones, as in

  ```
  $ export PATH=/opt/bin:$PATH
  ```

- When you compile with the **make** command, it is possible you may have to install some additional software (usually for header files). For example, on **Red Hat Enterprise Linux** distributions and their close kin you might need to do:

  ```
  $ dnf install curl-devel expat-devel openssl-devel
  ```

  or on **deb**-based systems:

  ```
  $ apt-get install libcurl4-gnutls-dev libexpat1-dev libssl-dev
  ```

  As an alternative that loses some functionality, you can do:

  ```
  $ make prefix=/opt NO_CURL=1 NO_EXPAT=1 NO_SSL=1
  ```

  Reading the `Makefile` will show you other such options.

- Once you have compiled, you can do

  ```
  $ make prefix=/opt install
  ```

  and you have the latest version installed, which you can always use by putting it earlier in your path than the system version.

- To learn further about all of this, read the `README`, `INSTALL`, and `Makefile` files in the main **Git** source directory.

Chapter 3

Git and Revision Control Systems

Exercise 3.1: Converting a Subversion repository to Git

- Make sure you have the appropriate **subversion** software installed to do the following steps. In addition to the basic **subversion** package one needs **subversion-perl**. There may be other useful packages which are obtainable from the **EPEL** repository on **RHEL**-based systems.

- The easiest way to get a copy, or clone, of a **Subversion** project is to use **git svn**.

- To obtain a copy of part of the **Subversion** repository itself one can do:

```
$ git svn clone  https://svn.apache.org/repos/asf/subversion/trunk/doc my_svn_repo
```

  where we have chosen only the **doc** module of **Subversion** in order to keep things small.

- However, this can take a long time as it gathers the entire history of the project and sometimes hangs. It is easier for learning purposes to just get the most recent version (which for **git** we would call a **shallow clone**). There is no easy way to just say give me the last version with **Subversion**; we need an actual release number which can get with:

```
$ svn log  https://svn.apache.org/repos/asf/subversion/trunk/doc | head
```

```
------------------------------------------------------------------------
r1663949 | brane | 2015-03-04 05:55:13 -0600 (Wed, 04 Mar 2015) | 1 line

* doc/svn-square.jpg: Copy the logo used by Doxygen from the site tree.
------------------------------------------------------------------------
r1663948 | brane | 2015-03-04 05:53:09 -0600 (Wed, 04 Mar 2015) | 1 line
...
```

- We can then pick just the latest release for the shallow clone:

```
$ git svn clone -r1663949  https://svn.apache.org/repos/asf/subversion/trunk/doc my_svn_repo
```

```
Initialized empty Git repository in /tmp/SVN/my_svn_repo/.git/
        A       doxygen.conf
        A       svn-square.jpg
        A       programmer/gtest-guide.txt
        A       programmer/WritingChangeLogs.txt
        A       README
        A       user/svn-best-practices.html
        A       user/lj_article.txt
        A       user/cvs-crossover-guide.html
r1663949 = f2f1312fe65123c1be0935421d05cc862d0d008e (refs/remotes/git-svn)
Checked out HEAD:
  https://svn.apache.org/repos/asf/subversion/trunk/doc r1663949
```

- This creates a **git** repository under the `my_svn_repo` directory, which you can examine. Notice all the **git** information goes in the `.git` subdirectory.
- If you want to compare with original **Subversion** repository, you can bring that down with:

  ```
  $ svn checkout https://svn.apache.org/repos/asf/subversion/trunk/doc doc
  ```

  ```
  ....
  ```

  where the repository information will go under `.svn` directories.
- Comparing:

  ```
  $ diff -qr my_svn_repo/ doc/
  ```

  ```
  Only in my_svn_repo/: .git
  Only in doc/: .svn
  ```

Chapter 4

Using Git: an Example

Exercise 4.1: Setting up a repository, and making changes and commits

Using just the simple subset of commands we have already given you, we are going to set up a simple repository.

1. First initialize the repository with `git init`. Then add author and email information using `git config`.
2. Create a couple of simple text files and add them to the repository and commit them with `git add` and `git commit`.
3. Now modify one of the files, and run `git diff` to see the differences between your working project files and what is in the repository.
4. Add the changed file to the repository again and then run `git diff` again.
5. Finally commit again, and then using `git log` examine your history.

At various stages of doing all this, examine the contents of the `.git` directory and see what files are being changed, what their content is, etc. Learn as much as you can.

Solution 4.1

1.
```
  # Set up the directory we are going to work in

  rm -rf git-test ; mkdir git-test ; cd git-test

  # initialize the repository and put our name and email in the .config file

  echo -e "\n\n*********   CREATING THE REPOSITORY AND CONFIGURING IT\n\n"

  git init
  git config user.name "A Smart Guy"
  git config user.email "asmartguy@linux.com"
```

2.
```
  echo -e "\n\n*********   CREATING A COUPLE OF FILES AND ADDING THEM TO THE PROJECT AND COMMITTING\n
```

```
# create a couple of files and add them to the project
# we'll do this as two commits, although we could do it as one

echo file1 > file1
git add file1
git commit file1 -s -m "This is the first commit"

echo file2 > file2
git add file2
git commit . -s -m "This is the second commit"
```

3.

```
# modify one of the files and then see the difference with the repository

echo -e "\n\n**************   MODIFYING ONE OF THE FILES AND THEN DIFFING\n\n"

echo This is another line for file2 >> file2
git diff
```

4.

```
# now stage it and diff again
git add file2
git diff
```

5.

```
echo -e "\n\n**************   ADDING THE MODIFIED FILE AND THEN DIFFING AGAIN\n\n"

echo -e "\n\n**************   RECOMITTING FOR THE THIRD TIME\n\n"

# now get it all in with another commit

git commit . -s -m "This is the third commit"

echo -e" \n\n**************   LOOKING AT THE HISTORY OF THE PROJECT\n\n"

# look at the history

git log
```

You can download a script with the above steps from

`s_04/lab_gitexample.sh`

in your solutions file.

Please see SOLUTIONS/s_04/lab_gitexample.sh

Chapter 5

Git Concepts and Architecture

Exercise 5.1: Distinguishing Features of Git

What are some features of **Git** that make it different from most, if not all, other revision control systems?

Solution 5.1

The following list is of course incomplete and even debatable. Please think of additional aspects.

1. **Git** is not based on files as basic units. Instead the primary quantities stored are binary blobs. If the same file appears in more than once place in the project, only one blob is needed to store the contents.
2. Most (but not all) revision control systems have a central repository that is authoritative. With **Git** all repositories, no matter their location or ownership, are essentially the same. Leadership and authority is by social contract more than anything else.
3. **Git** was designed for distributed development from its outset, rather than evolved to work with it. Thus it has always emphasized minimizing transfer sizes and speeds and was built completely on the Internet.
4. Dealing with multiple **branches** and **merging**, **rebasing** and **forking** have always been organic features.
5. **Committing** (checking in changes) and **Publishing** (making them widely available) are quite distinct steps by design.
6. Other features, such as the use of **bisection** to locate and fix problems, are rather innovative and unique.

Chapter 6

Managing Files and the Index

Exercise 6.1: Getting practice with some basic file commands

1. First initialize the repository, configuring it with name and email etc. Then add a couple of files to the project and commit them.
2. Remove one of the files with `git rm` and with `git diff` see the difference with the repository.
3. Rename the remaining file with `git mv` and with `git diff` see the difference with the repository, once again.
4. Commit again and look at the history with `git log`. Then do `git ls-files` without any arguments
5. Add two new files, make one of them **ignored** by **git** and modify the original remaining file. Do `git ls-files` again.
6. Now try some different options to `git ls-files`, such as `-t` and `-o`. Do `man git ls-files` to see the various options available and try some others.
7. Now add the new files that are not ignored with `git add`, commit once again, and do `git ls-files` with some options to see the results. You may want to do `git log` again as well.

Solution 6.1

1.

```
# initialize the repository and put our name and email in the .config file

echo -e "\n\n*************   CREATING THE REPOSITORY AND CONFIGURING IT\n\n"

rm -rf git-test ; mkdir git-test ; cd git-test
git init
git config user.name "A Smart Guy"
git config user.email "asmartguy@linux.com"

echo -e "\n\n*********   CREATING A COUPLE OF FILES AND ADDING THEM TO THE PROJECT AND COMMITTING\n

# create a couple of files and add them to the project, and then commit

echo file1 > file1
echo file2 > file2
```

```
git add file1 file2
git commit . -s -m "This is our first commit"
```

2.

```
# remove one of the files and then see the difference with the repository

echo -e "\n\n*************   REMOVING ONE OF THE FILES AND THEN DIFFING\n\n"

git rm file2
git diff
```

3.

```
echo -e "\n\n*************   RENAMING ONE OF THE FILES AND THEN DIFFING\n\n"

# now rename a file and diff again

git mv file1 file1_renamed
git diff

echo -e "\n\n*************   RECOMITTING\n\n"
```

4.

```
# now get it all in with another commit

git commit . -s -m "This is the second commit"

echo -e "\n\n*************   LOOKING AT THE HISTORY OF THE PROJECT\n\n"

# look at the history

git log

echo -e "\n\n*************   DO git ls-files"

# do git ls-files

git ls-files
```

5.

```
echo -e "\n\n*************   ADD TWO NEW FILES, MAKE ONE IGNORED, AND MODIFY A FILE\n\n"

echo extra1 >> extra1
echo extra2 >> extra2
echo anotherline >> file2
echo extra1 >> .gitignore

echo -e "\n\n*************   DO git ls-files WITH NO ARGS\n\n"

git ls-files
```

6.
```
echo -e "\n\n*************   DO git ls-files WITH -t \n\n"

git ls-files -t

echo -e "\n\n*************   DO git ls-files WITH -t --others\n\n"

git ls-files -t -o
```

7.
```
echo -e "\n\n*************   RECOMMIT AND CHECK AGAIN\n\n"

git add  extra2
git commit -a -s -m  "third commit"

git ls-files -t -c -o -s

git log
```

You can download a script with the above steps from

`s_06/lab_gitbasics.sh`

in your solutions file.

Please see SOLUTIONS/s_06/lab_gitbasics.sh

Chapter 7

Commits

Exercise 7.1: Bisecting with git

1. First initialize a repository, configuring it with name and email address, etc.
2. Then make a significant number of commits, say 64. Each commit should add a file. In one of the commits, have the file include the string **BAD**. We will interpret this as the bug introduction.
3. Now start a `git bisect` procedure. Designating the last commit with

   ```
   $ git bisect bad
   ```

   and the first one with

   ```
   $ git bisect good
   ```

   See how many bisections it takes to find the one that introduced the bug, the file with **BAD** in it.
4. You can do this manually, or you can use the `git bisect run ...` procedure with a script to make it automated.
5. When done check the history of your bisection with `git bisect log`.

Solution 7.1

1.

```
# initialize the repository and put our name and email in the .config file

echo -e "\n\n*************   CREATING THE REPOSITORY AND CONFIGURING IT\n\n"

rm -rf git-test ; mkdir git-test ; cd git-test
git init
git config user.name "A Smart Guy"
git config user.email "asmartguy@linux.com"
```

2.

```
echo -e "\n\n*************   CREATING A NUMBER OF  FILES AND ADDING"
echo -e "                    THEM TO THE PROJECT AND COMMITTING\n\n"
```

```
n=0
while [ $n -lt 64 ] ; do
    n=$(($n+1))
    file=file$n
    echo file > $file
    if [ "$n" == "19" ] ; then
        echo BAD >> $file
    fi
    git add $file
    git commit $file -s -m"$file"
    git tag $file
    echo I added and committed $file
done

echo -e "\n\n************** I PUT THE BAD LINE IN file19\n\n"
```

3.
```
echo -e "\n\n************** STARTING THE BISECTION\n\n"

git bisect start
git bisect bad
git bisect good file1

echo -e "\n\n************** SEARCHING FOR THE BAD FILE\n\n"

echo -e "\n\n************** DOING TH BISECTION MANUALLY\n\n"

over=0
while [ "$over" == "0" ] ; do
    if [ "$(grep BAD file*)" == "" ] ; then
        git bisect good | tee gitout
    else
        git bisect bad  | tee  gitout
    fi

    if [ "$(grep 'revisions left' gitout)" == "" ] ; then
        over=1
        echo "****************** FOUND THE BUG!"
    fi
done
```

4.
```
echo -e "\n\n**************** SETTING UP A TESTING SCRIPT\n\n"

cat <<EOF > my_script.sh
#!/bin/bash

if [ "\$(grep BAD file*)" == "" ] ; then
    exit 0
fi
exit 1
EOF
chmod +x my_script.sh
```

```
# reset to original state

git reset

git bisect start
git bisect bad  file64
git bisect good file1

# do automated script

git bisect run ./my_script.sh
```

5.

```
# check log

git bisect log
```

You can download a script with the above steps from

`s_07/lab_bisect.sh`

in your solutions file.

`Please see SOLUTIONS/s_07/lab_bisect.sh`

Chapter 8

Branches

Exercise 8.1: Working with a Development Branch

1. First initialize the repository, configuring it with name and email, etc. Then add a couple of files to the project and commit them.
2. Create a new development branch and then check it out.
3. Modify a file and add another one, and then do a new commit. List the files that are present and also do `git ls-files`.
4. Now checkout the original **main** branch. Once again list the files that are present and also do `git ls-files`.

Solution 8.1

1.
```
  # initialize the repository and put our name and email in the .config file

  echo -e "\n\n********   CREATING THE REPOSITORY AND CONFIGURING IT\n\n"

  rm -rf git-test ; mkdir git-test ; cd git-test
  git init
  git config user.name "A Smart Guy"
  git config user.email "asmartguy@linux.com"

  echo -e "\n\n*********   CREATING A COUPLE OF FILES AND ADDING THEM TO THE PROJECT AND COMMITTING\n

  # create a couple of files and add them to the project, and then commit

  echo file1 > file1
  echo file2 > file2
  git add file1 file2
  git commit . -s -m "This is our first commit"
```

2.
```
  # create a new development branch
```

```
echo -e "\n\n**************** CREATING A NEW BRANCH, devel ***\n\n"

git branch devel

# checkout the new branch

echo -e "\n\n*************** CHECKOUT BRANCH devel ****\n\n"

git checkout devel
```

3.

```
# modify a file and add a new one

echo another line >> file1
echo file3 >> file3

git add file1 file3

echo -e "\n\n*************    DIFFING\n\n"
git diff

echo -e "\n\n*************   RECOMITTING\n\n"

# now get it all in with another commit

git commit . -s -m "This is a commit in the devel branch"

#  list files

echo -e "\n\nlist files, then git ls-files\n\n"

ls -l
git ls-files
```

4.

```
# now checkout the original branch

echo -e "\n\n************ LIST BRANCHES AND CHECKOUT main **\n\n"

git branch
git checkout main

#  list files

echo -e "\n\n***************** list files, then git ls-files\n\n"

ls -l
git ls-files

echo -e "\n\n***************** SHOW THE BRANCH\n\n"

git branch
```

You can download a script with the above steps from

`s_08/lab_branch.sh`

in your solutions file.

```
Please see SOLUTIONS/s_08/lab_branch.sh
```

Chapter 9

Diffs

Exercise 9.1: Exploring changes with git diff

- You can work with one of your repositories created earlier, but it will be richer to work with a full repository such as the one for the **git** project itself.
- Working in the main project directory first do:

  ```
  $ git tag
  ```

  to get a list of references. Then to get a full difference between two versions you can do something like:

  ```
  $ git diff v1.7.0 v1.7.0-rc2
  ```

- This is likely to be a long output; try with the `--stat` flag to see a short summary of changes.
- Now look at the changes to a particular directory, such as in:

  ```
  $ git diff v1.7.0 v1.7.0-rc2 Documentation
  ```

  or you can pick to look at one or more particular files.

Chapter 10

Merges

Exercise 10.1: Resolving Conflicts While Merging

1. Either begin with the **main** and **devel** repositories from the previous session on branches, or recreate them.
2. Make modifications in the **main** branch that conflict with those in the **devel** branch. Commit them, and then use **git merge** to merge the **devel** branch into the **main** branch.
3. Resolve the conflicts, either by doing a **git reset** and modifying either of the branches, or by modifying the conflicted files and then committing once again.

Solution 10.1

1. You can skip the following step if you use the results from the previous labs.

```
# initialize the repository and put our name and email in the .config file

echo -e "\n\n*********   CREATING THE REPOSITORY AND CONFIGURING IT\n\n"

rm -rf git-test ; mkdir git-test ; cd git-test
git init
git config user.name "A Smart Guy"
git config user.email "asmartguy@linux.com"

echo -e "\n\n*********   CREATING A COUPLE OF FILES AND ADDING THEM TO THE PROJECT AND COMMITTING\n

# create a couple of files and add them to the project, and then commit

echo file1 > file1
echo file2 > file2
git add file1 file2
git commit . -s -m "This is our first commit"

# create a new development branch

echo -e "\n\n**************** CREATING A NEW BRANCH, devel ***\n\n"
```

```
git branch devel

# checkout the new branch

echo -e "\n\n*************** CHECKOUT BRANCH devel ****\n\n"

git checkout devel

# modify a file and add a new one

echo another line >> file1
echo file3 >> file3

git add file1 file3

echo -e "\n\n*************    DIFFING\n\n"
git diff

echo -e "\n\n*************   RECOMITTING\n\n"

# now get it all in with another commit

git commit . -s -m "This is a commit in the devel branch"

#  list files

echo -e "\n\nlist files, then git ls-files\n\n"

ls -l
git ls-files

# now checkout the original branch

echo -e "\n\n************ LIST BRANCHES AND CHECKOUT main **\n\n"

git branch
git checkout main

#  list files

echo -e "\n\n***************** list files, then git ls-files\n\n"

ls -l
git ls-files

echo -e "\n\n***************** SHOW THE BRANCH\n\n"
```

2.

```
git branch

# modify some stuff in the original branch to conflict with devel
echo -e "\n\n*************** MODIFYING MAIN BRANCH AND COMMITTING\n\n"

echo different content >> file1
```

```
# add and recommit

git add file1
git commit -s -m"a re-modification"

git diff main devel

# try to do a merge

echo "\n\n************** TRYING TO DO A MERGE \n\n"

git merge devel

echo "\n\n************* THE MERGE FAILED ****\n\n"
echo "\n\n************* FIX file1 and then add and commit\n"
echo "\n\n************ IN another window, edit the file and type anything here\n\n"
read something
```

3.
```
git add file1
git commit -s -m"finally got it right"

git ls-files
git diff
git branch
```

You can download a script with the above steps from

`s_10/lab_conflict.sh`

in your solutions file.

`Please see SOLUTIONS/s_10/lab_conflict.sh`

Exercise 10.2: Rebasing

1. After you have produced a development branch make some changes to the main branch and recommit.
2. Do **git log** on the **devel** branch to show the history of commits.
3. Rebase the development branch, using **git rebase**, off the modified **main** branch.
4. Once more, do **git log** on the **devel** branch to show the history of commits, and note the difference.

Solution 10.2

1. You can skip this step if you use the results from the earlier labs, or repeat the first step from the previous lab.

2.
```
git branch

# diff the branches

echo -e "\n\n****************** DIFFING THE BRANCHES ****\n\n"
```

```
git diff main devel

# get a log of the devel branch
echo -e "\n\n****************** git log on devel branch *** \n\n"

git checkout devel
git log
```

3.

```
# make a new change to main branch
echo -e "\n\n****************** Make a new change to main branch and commit\n\n"
git checkout main
echo something >> file4
git add file4
git commit -s -m"a new commit to the main branch"

# do the rebase

echo -e "\n\n******************* REBASING **********\n\n"

git rebase main devel

git branch
git ls-files
# diff the branches

echo -e "\n\n****************** DIFFING THE BRANCHES ****\n\n"

git diff main devel
```

4.

```
# get a log of the devel branch
echo -e "\n\n****************** git log on devel branch *** \n\n"

git checkout devel
git log
```

You can download a script with the above steps from

`s_10/lab_rebase.sh`

in your solutions file.

`Please see SOLUTIONS/s_10/lab_rebase.sh`

Chapter 11

Managing Local and Remote Repositories

✎ Exercise 11.1: Accessing Your Repository Remotely with the git:// Protocol

- Create a repository and populate it; you can use a solution for a previous session to do this. To make a proper remote repository first clone it on your local machine using the `--bare` option, as in:

  ```
  $ git clone --bare <pathto>/git-test /tmp/my-remote-git-repo
  ```

- To make it accessible through the `git://` protocol you will have to create the `git-daemon-export-ok` file in the main project directory or in the `.git` subdirectory, and start the **git-daemon** process. If you do:

  ```
  $ git daemon
  ```

  someone can clone your repository by doing:

  ```
  $ git clone git://ipaddress/tmp/my-remote-git-repo
  ```

  substituting a correct value for `ipaddress` and giving a full path.

- More conveniently you can specify a root, or base, directory for the repositories by doing:

  ```
  $ git daemon --base-path=/tmp
  ```

  and then someone can clone your repository by doing:

  ```
  $ git clone git://ipaddress/my-remote-git-repo
  ```

> **Please Note**
>
> You do not have to be a superuser to run **git-daemon** and you will probably want to run it in background or figure out how to get it to run as a service on boot.

- You might try and see what happens if you either leave out the step of starting the daemon, or creating the `git-daemon-export-ok` file.

- If you happen to have a partner on another machine, or are running two machines, try to clone each other's repositories using this method.

Exercise 11.2: Accessing Your Repository Remotely using ssh

- Make a new clone of the repo using the **ssh** protocol, using both of the two following methods:

  ```
  $ git clone ssh://user@ipaddress/tmp/my-remote-git-repo
  $ git clone user@ipaddress:/tmp/my-remote-git-repo
  ```

  substituting a correct value for `user@ipaddress`.

- If you happen to have a partner on another machine, or are running two machines, try to clone each other's repositories using this method.

- In order to get this to work you may have to install an **ssh server**. On **RPM**-based machines this might involve:

  ```
  $ sudo dnf install openssh-server
  ```

  and on **deb**-based systems:

  ```
  $ sudo apt-get install openssh-server
  ```

Exercise 11.3: Accessing Your Repository Remotely using http

- Make a new clone of the repo using the **http** protocol, as in:

  ```
  $ git clone https://ipaddress/my-remote-git-repo
  ```

  substituting a correct value for `ipaddress`.

- If you happen to have a partner on another machine, or are running two machines, try to clone each other's repositories using this method.

- In order to get this to work you may have to install an **http server**. On **RPM**-based machines this might involve:

  ```
  $ sudo dnf install httpd
  ```

  and on **deb**-based systems:

  ```
  $ sudo apt-get install apache2
  ```

  and then start it up with

  ```
  $ sudo systemctl start httpd
  ```

- Do not forget to run

  ```
  $ git --bare update-server-info
  ```

  in the project directory before trying to access the repository through `https://`.

- For this to work the repository has to be available through your web server. For simplicity, you can put it under `/var/www/html` (or in `/var/www/git` on **deb**-based systems), or you can set up a link from there to the actual location, as in:

  ```
  $ sudo ln -s /tmp/my-remote-git-repo /var/www/html/my-remote-git-repo
  $ sudo ln -s /tmp/my-remote-git-repo /var/www/git/my-remote-git-repo
  ```

  Of course you can put in in other places, but we do not want to get into the details of web server configuration here.

Exercise 11.4: Pushing Changes into the Remote Repository

- Make some changes to your local repository which we will then **push** to the remote location. You might just add a file, or change one of them etc.

- First try using the **ssh** protocol. Do you have any problems?
- Now try with the **git:**// protocol. For this to work you will have to make sure to configure the daemon either with `--enable=receive-pack` for a global change for all repositories on your system (which is not smart) or by configuring the `config` file or your repository as noted previously.
- Can you push an update through the **https:**// protocol?

Chapter 12

Using Patches

Exercise 12.1: Synchronizing with Patches

1. First initialize a repository, configuring it with name and email address, etc. Then add a couple of files to the project and commit them.

2. Now make a clone of the repository with `git clone` and change to the new directory.

3. Change a file in the copy and create a new file. Use `git add` and `git commit` to bring the repository fully up to date.

4. Produce a patch using `git format-patch`. Use the `-s` option to produce a `signed-off` line.

5. Go back to the original repository and first try to apply the patch with `git apply --check`. If that succeeds apply the patch with `git am`.

For an extra exercise try to use `git send-email` to send the patch to yourself.

Solution 12.1

1.
```
# initialize the repository and put our name and email in the .config file

echo -e "\n\n********   CREATING THE REPOSITORY AND CONFIGURING IT\n\n"

rm -rf git-test ; mkdir git-test ; cd git-test
git init
git config user.name "A Smart Guy"
git config user.email "asmartguy@linux.com"

echo -e "\n\n********   CREATING A COUPLE OF FILES AND ADDING THEM TO THE PROJECT AND COMMITTING\n\

echo file1 > file1
echo file2 > file2
git add file1 file2
git commit . -s -m "This is our first commit"
```

2.

```
echo -e "\n\n************   MAKING A NEW CLONE\n\n"

cd ..
git clone git-test git-newer
```

3.

```
echo -e "\n\n************   MAKING CHANGES TO THE REPOSITORY*\n\n"

cd git-newer

echo another line >> file2
echo a third file > file3

echo -e "\n\n************   ADDING AND COMMITTING THE CHANGES\n\n"

git add file2 file3
git commit -s -m"modifications from the new clone"
```

4.

```
echo -e "\n\n************   PRODUCING THE PATCH*\n\n"

git format-patch -1 -s
mv 00* ..
```

5.

```
echo -e "\n\n************   SEEING IF THE PATCH WORKED\n\n"

cd ..
cd git-test
git apply --check ../00*

echo -e "\n\n************* NOW APPLY THE PATCH\n\n"

git am ../00*
```

You can download a script with the above steps from

`s_12/lab_patches.sh`

in your solutions file.

`Please see SOLUTIONS/s_12/lab_patches.sh`

Chapter 13

Advanced Git Interfaces: Gerrit

Exercise 13.1: A Gerrit Walk through

Since we do not have a **Gerrit** server set up, it is not easy to do an exercise demonstrating its capabilities.

The project, however, has a great walk through example at https://gerrit-review.googlesource.com/Documentation/intro-gerrit-walkthrou html which demonstrates step by step how to:

- Making a change
- Creating the review
- Reviewing the change
- Reworking the change
- Verifying the change
- Submitting the change.

Please read this example thoroughly.