GIT/GITHUB

WORK SHOP

A. Vishnu Sai Chaudary

BY

Mr.NATARAJ Sir

(SENIOR JAVA CONSULTANT)

NIT NARESHi technologies

An ISO 9001 : 2008 Certified Company

Introduction

Git is a distributed revision control and source code management system with an emphasis on speed. Git was initially designed and developed by Linus Torvalds for Linux kernel development. Git is a free software This Document explains how to use Git for project version control in a distributed environment while working on web-based and non-web-based applications development.

About the Document

This Document will help beginners learn the basic functionality of Git version control system. After completing this document, you will find yourself at a moderate level of expertise in using Git version control system from where you can take yourself to the next levels.

Prerequisites

We assume that you are going to use Git to handle all levels of Java and non-Java projects. So, it will be good if you have some amount of exposure to software development life cycle and working knowledge of developing web based and non-web-based applications.

GIT VS SVN

1. **GIT is distributed, SVN is not:**

   This is by far the ***core*** difference between GIT and other non-distributed version control systems like SVN, CVS etc. If you can catch this concept well, then you have crossed half the bridge. To add a disclaimer, GIT is not the first or only distributed VCS (version control system) currently available. There are other tools like Bitkeeper, Mercurial etc. which also work on distributed mode. But, GIT does it better and comes with much more powerful features.

   GIT like SVN do have centralized repository or server. But, GIT is more intended to be used in distributed mode which means, every developer checking out code from central repository/server will have their own cloned repository installed on their machine. Let's say if you are stuck somewhere where you don't have network connectivity, like inside the flight, basement, elevator etc. :), you will still be able to commit files, look at revision history, create branches etc. This may sound trivial for lot of people but, it is a big deal when you often bump into no-network scenario.

   And also, the distributed mode of operation is a biggest blessing for open-source software development community. Instead of creating patches & sending it through emails, you can create a branch & send a pull request to the project team. It will help the code stay streamlined without getting lost in transport. GitHub.com is an awesome working example of that.There were some rumors that the future version of subversion will be working on distributed mode. But, it's still an unknown at this point.

2. **GIT stores content as metadata, SVN stores just files:**

   Every source control systems stores the metadata of files in hidden folders like .svn, .cvs etc., whereas GIT stores entire content inside the .git folder. If you compare the size of .git folder with .svn, you will notice a big difference. So, the .git folder is the cloned repository in your

machine, it has everything that the central repository has like tags, branches, version histories etc.

3. **GIT branches are not the same as SVN branches:**

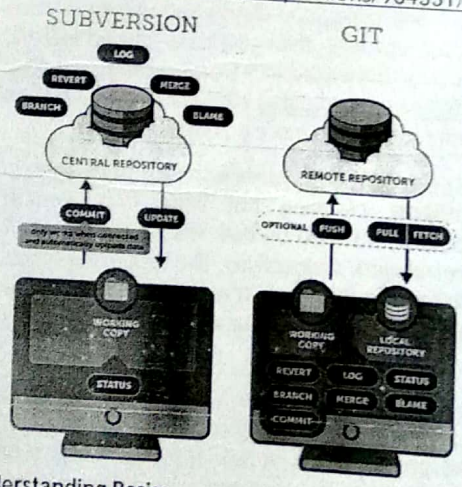
   Branches in SVN are nothing but just another folder in the repository. If you need to know if you had merged a branch, you need to explicitly run commands to verify if it was merged or notSo, the chance of adding up orphan branches is pretty big.Whereas, working with GIT branches is much easier & fun. You can quickly switch between branches from the same working directory. It helps finding un-merged branches and also help merging files fairly easily & quickly.

4. **GIT does not have a global revision no. like SVN do:**

   This is one of the biggest feature I miss in GIT from SVN so far. As you may know already SVN's revision no. is a snapshot of source code at any given time. I consider that as a biggest breakthrough moving from CVS to SVN.

5. **GIT's content integrity is better than SVN's:**

   GIT contents are cryptographically hashed using SHA-1 hash algorithm. This will ensure the robustness of code contents by making it less prone to repository corruption due to disk failures, network issues etc. Here is an interesting discussion on GIT's content integrity – http://stackoverflow.com/questions/964331/git-file-integrity



Understanding Basics

Version Control System

Version Control System (VCS) is software that helps software developers to work together and maintain a complete history of their work.

Listed below are the functions of a VCS:

- Allows developers to work simultaneously.
- Does not allow overwriting each other's changes.
- Maintains a history of every version.

Following are the types of VCS:

- **Centralized version control system (CVCS).** Ex:- Tortoise
- **Distributed/Decentralized version control system (DVCS).** Ex:-

In this chapter, we will concentrate only on distributed version control system and especially on Git. Git falls under distributed version control system.

Distributed Version Control System

Centralized version control system (CVCS) uses a central server to store all files and enables team collaboration. But the major drawback of CVCS is its single point of failure, i.e., failure of the central server. Unfortunately, if the central server goes down for an hour, then during that hour, no one can collaborate at all. And even in a worst case, if the disk of the central server gets corrupted and proper backup has not been taken, then you will lose the entire history of the project. Here, distributed version control system (DVCS) comes into picture.

DVCS clients not only check out the latest snapshot of the directory but they also fully mirror the repository. If the sever goes down, then the repository from any client can be copied back to the server to restore it. Every checkout is a full backup of the repository. Git does not rely on the central server and that is why you can perform many operations when you are offline. You can commit changes, create branches, view logs, and perform other operations when you are offline. You require network connection only to publish your changes and take the latest changes.

1. Advantages of Git

1.1 Free and open source

Git is released under GPL's open source license. It is available freely over the internet. You can use Git to manage propriety projects without paying a single penny. As it is an open source, you can download its source code and also perform changes according to your requirements.

1.2 Fast and small

As most of the operations are performed locally, it gives a huge benefit in terms of speed. Git does not rely on the central server; that is why, there is no need to interact with the remote server for every operation performed. The core part of Git is written in C, which avoids runtime overheads associated with other high level languages. Though Git mirrors entire repository, the size of the data on the client side is small. This illustrates the efficiency of Git at compressing and storing data on the client side.

1.3 Implicit backup

The chances of losing data are very rare when there are multiple copies of it. Data present on any client side mirrors the repository, hence it can be used in the event of a crash or disk corruption.

1.4 Security

Git uses a common cryptographic hash function called secure hash function (SHA1), to name and identify objects within its database. Every file and commit is check-summed and retrieved by its checksum at the time of checkout. It implies that it is impossible to change file, date, and commit message and any other data from the Git database without knowing Git.

1.5 No need of powerful hardware

In case of CVCS, the central server needs to be powerful enough to serve requests of the entire team. For smaller teams, it is not an issue, but as the team size grows, the hardware limitations of the server can be a performance bottleneck. In case of DVCS, developers don't interact with the server unless they need to push or pull changes. All the heavy lifting happens on the client side, so the server hardware can be very simple indeed.

1.6 Easier branching

CVCS uses cheap copy mechanism. If we create a new branch, it will copy all the codes to the new branch, so it is time-consuming and not efficient. Also, deletion and merging of branches in CVCS is complicated and time-consuming. But branch management with Git is very simple. It takes only a few seconds to create, delete, and merge branches.

DVCS Terminologies

1.7 Local Repository

Every VCS tool provides a private workplace as a working copy. Developers make changes in their private workplace and after commit, these changes become a part of the repository. Git takes it one step further by providing them a private copy of the whole repository. Users can perform many operations with this repository such as add file, remove file, rename file, move file, commit changes, and many more.

1.8 Working Directory and Staging Area or Index

The working directory is the place where files are checked out. In other CVCS, developers generally make modifications and commit their changes directly to the repository. But Git uses a different strategy. Git doesn't track each and every modified file. Whenever you do commit an operation, Git looks for the files present in the staging area. Only those files present in the staging area are considered for commit and not all the modified files.

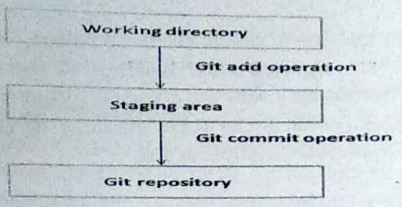
Let us see the basic workflow of Git.

Step 1: You modify a file from the working directory.

Step 2: You add these files to the staging area.

Step 3: You perform commit operation that moves the files from the staging area.

After push operation, it stores the changes permanently to the Git repository.



Suppose you modified two files, namely "sort.c" and "search.c" and you want two different commits for each operation. You can add one file in the staging area and do commit. After the first commit, repeat the same procedure for another file.

```
# First commit
[bash]$ git add sort.c

# adds file to the staging area
[bash]$ git commit –m "Added sort operation"
# Second commit
[bash]$ git add search.c

# adds file to the staging area
[bash]$ git commit –m "Added search operation"
```

1.7 Blobs

Blob stands for **B**inary **L**arge **O**bject. Each version of a file is represented by blob. A blob holds the file data but doesn't contain any metadata about the file. It is a binary file and in Git database, it is named as SHA1 hash of that file. In Git, files are not addressed by names. Everything is content-addressed.

1.8 Trees

Tree is an object, which represents a directory. It holds blobs as well as other sub-directories. A tree is a binary file that stores references to blobs and trees which are also named as**SHA1**hash of the tree object.

1.9 Commits

Commit holds the current state of the repository. A commit is also named by **SHA1** hash. You can consider a commit object as a node of the linked list. Every commit object has a pointer to the parent commit object. From a given commit, you can traverse back by looking at the parent

pointer to view the history of the commit. If a commit has multiple parent commits, then that particular commit has been created by merging two branches.

1.10 Branches

Branches are used to create another line of development. By default, Git has a master branch, which is same as trunk in Subversion. Usually, a branch is created to work on a new feature. Once the feature is completed, it is merged back with the master branch and we delete the branch. Every branch is referenced by HEAD, which points to the latest commit in the branch. Whenever you make a commit, HEAD is updated with the latest commit.

1.11 Tags

Tag assigns a meaningful name with a specific version in the repository. Tags are very similar to branches, but the difference is that tags are immutable. It means, tag is a branch, which nobody intends to modify. Once a tag is created for a particular commit, even if you create a new commit, it will not be updated. Usually, developers create tags for product releases.

1.12 Clone

Clone operation creates the instance of the repository. Clone operation not only checks out the working copy, but it also mirrors the complete repository. Users can perform many operations with this local repository. The only time networking gets involved is when the repository instances are being synchronized.

1.13 Pull

Pull operation copies the changes from a remote repository instance to a local one. The pull operation is used for synchronization between two repository instances. This is same as the update operation in Subversion.

1.14 Push

Push operation copies changes from a local repository instance to a remote one. This is used to store the changes permanently into the Git repository. This is same as the commit operation in Subversion.

1.15 HEAD

HEAD is a pointer, which always points to the latest commit in the branch. Whenever you make a commit, HEAD is updated with the latest commit. The heads of the branches are stored in **.git/refs/heads/** directory.

```
[CentOS]$ ls -1 .git/refs/heads/ master

[CentOS]$ cat .git/refs/heads/master
570837e7d58fa4bccd86cb575d884502188b0c49
```

1.16 Revision

Revision represents the version of the source code. Revisions in Git are represented by commits. These commits are identified by **SHA1**secure hashes.

2 Git Overview

As you develop software and make changes, add features, fix bugs, etc. it is often useful to have a mechanism to keep track of changes and to ensure that your code base and artifacts are well-protected by being stored on a reliable server (or multiple servers). This allows you access to historic versions of your application's code in case something breaks or to "roll-back" to a previous version if a critical bug is found.

The solution is to use a *revision control system* that allows you to "check-in" changes to a code base. It keeps track of all changes and allows you to "branch" a code base into a separate copy so that you can develop features or enhancements in isolation of the main code base (often called the "trunk" in keeping with the tree metaphor). Once a branch is completed (and well-tested and reviewed), it can then be *merged* back into the main trunk and it becomes part of the project.

You may already be familiar with similar online (or "cloud") storage systems such as Google Drive or Dropbox that allow you to share and even collaborate on documents and other files. However, a version control system is a lot more. It essentially keeps track of all changes made to a project and allows users to work in large teams on very complex projects while minimizing the conflicts between changes. These systems are not only used for organizational and backup purposes, but are absolutely essential when developing software as part of a team. Each team member can have their own working copy of the project code without interfering with other developer's copies or the main trunk. Only when separate branches have to be merged into the trunk do conflicting changes have to be addressed. Otherwise, such a system allows multiple developers to work on a very complex project in an organized manner.

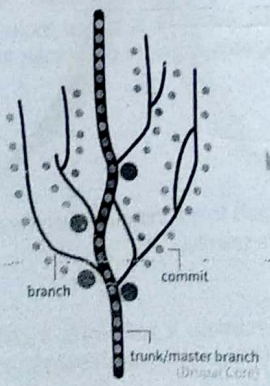


Figure 1: Trunk, branches, and merging visualization of the Drupal project

There are several widely used revision control systems including CVS (Concurrent Versions System), SVN (Apache Subversion), and Git. CVS is mostly legacy and not as widely used anymore.

SVN is a *centralized* system: there is a single server that acts as the main code repository. Individual developers can check out copies and branch copies (which are also stored in the main repository). They also check all changes into the main repository.

Git, however, is a *decentralized* system: multiple servers can act as repositories, but each copy on each developer's own machine is *also* a complete revision copy. Code commits are committed to the local repository. Merging a branch into another requires a push/pull request. Decentralizing the system means that anyone's machine can act as a code repository and can lead to wider collaboration and independence since different parties are no longer dependent on one master repository.

Git itself is a version control system that can be installed on any server (UNL has a Git repository setup at https://git.unl.edu). However, we'll primarily focus on Github (https://github.com), the main website used by thousands of developers across the globe.

The rest of this Document will describe how to use Github for use in your courses and to manage and share your code among your Friends for group assignments and projects.To work with GIT we need to arrange GIT Server and Client Softwares.

Popular GIT servers used by Industry are:

Assembla:

Assembla has a long history in the business with its Subversion hosting. It includes a Workspaces platform with issue management, time tracking, and collaboration tools like wikis and its own messaging system.

Beanstalk:

Being a very lean and reliable service, Beanstalk is a great choice for businesses. It offers its own deployment infrastructure and integrates well with other tools like Zendesk, FogBugz, Basecamp, or Lighthouse.

Bitbucket:

Coming from Atlassian, one of the giants in the industry. Therefore, integrations to other Atlassian products like the JIRA issue tracker are seamless.

CloudForge:

Services go well beyond just code hosting: it calls itself a platform for developers, offering development tools (e.g. Trac or Bugzilla) as well as integrations with JIRA, Basecamp, and others.

Codebase:

One of the few platforms that supports Git, Subversion, and Mercurial. Additionally, ticketing, project management and deployment infrastructure tools are included.

Fog Creek Kiln:

The makers of the popular issue tracking solution "FogBugz" offer a solid code hosting platform - and of course a tight integration with their bug tracking system.

GitHub:

The undisputed number one in the Open Source world with projects like Java, Ruby on Rails or jQuery being hosted at GitHub. Offerings include great code review and collaboration features.

GitLab:

With an OpenSource background, GitLab offers a fast-evolving platform for code collaboration. The free community edition and the enterprise edition can be installed on your own servers; a hosted offering that runs on GitLab.com is also available.
Note:Apart from above many more GIT service providers are there in market.

Popular GIT Client Softwares used by Industry are:

Git Clients for Windows

- Github for windows
- Sourcetree
- Aurees

Git Clients for Linux

- Gitg
- Giggle
- Qgit

Git ForceGit Clients for Mac

- Git Box
- Git-Xdev
- GitUp
- Fork

Cross-Platform Git Clients

- Smartgit
- Git Kraken
- Tower
- Git-Cola
- GitEye
- GitAhead

2.1 Registering

You can register for a GitHub account at https://github.com/. However, it is *strongly recommended* that you get a free "student" account. A normal, free GitHub account does not allow you to create "private" repositories. Any code you push to GitHub is automatically public and accessible by anyone. This is okay in general, however many of your courses will have Academic Integrity policies that will require you to *not* share code. A student account allows you up to 5 private repositories (normally $7/month as of this writing) so that you can comply with Academic Integrity policies.

To get a student account first register at GitHub using an email account that ends in .edu (to "prove" you're a student). Then go to https://education.github.com/pack and register for a "student pack." Sign up early as some have reported long wait times to receive their student pack. The student pack contains a lot of other free and reduced cost software packages, tools and services that may be of interest.

2.2 Installing Git on Your Machine

If you want to use Git on your own personal machine, then you may need to install a Git client. There are many options out there and you are encouraged to explore them, however the following suggestions are all free and open source.

- Git has released its own graphical user interface clients which are available for free for both Windows and Mac:
  - Windows: https://windows.github.com/
  - Mac: https://mac.github.com

  See section 2 for instructions on using the client.
- If you will be using the Eclipse IDE (http://www.eclipse.org/downloads/) for development, the most recent versions already come with a Git client. Eclipse will work on any system. See Section 4 for using Git with Eclipse.
- If you use Windows and prefer to use a command line interface, you can download and install TortoiseGit (https://code.google.com/p/tortoisegit/) a Windows Shell Interface to Git. See Section 3 for using Git via the command line interface.
- If you use Mac and want the command line version of Git, you can download and install here: http://www.git-scm.com/download/mac. Alternatively, you can install Git using a tool like MacPorts: http://iamphioxus.org/2013/04/20/installing-git-via-macports-on-mac-osx/. See Section 3 for using Git via the command line interface.

2.3 Creating a Repository on Github

You will eventually want to publish ("push") your project code to Github. To do this you'll first need to create a repository on Github's site:

1. Login to Github (https://github.com/) and click on the "repositories" tab.
2. Create a new repository (see Figure 2) with the name that will match your project folder (the names do not *have* to match, but it keeps things organized). Provide a short description and choose whether or not to make it public or private depending on whether or not you are allowed to share your code with your peers.

   You *may* choose to include a README file and/or establish a license (which creates a LICENSE file). However, for this tutorial we will assume that you start with an empty repo on Github. If you choose to create these files some extra steps may be necessary.

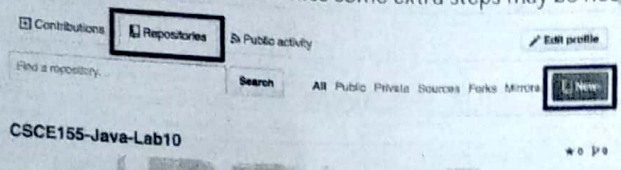


Figure 2: Creating a New Repository on GitHub

3 Using Git via Git's Clients

In this section we'll explore the basic uses of Git by using Git's client which provides a Graphical User Interface (GUI) to Git. A complete online help guide is available here: https://mac.github.com/help.htm!(Mac) and here: https://windows.github.com/help.html(Windows).Though the clients should be almost identical for Mac and Windows, there may be some slight differences;. .

Popular GIT Client Softwares used by Industry are:

Git Clients for Windows

- Github for windows
- Sourcetree
- Aurees

Git Clients for Linux

- Gitg
- Giggle
- Qgit
- Git Force

Git Clients for Mac

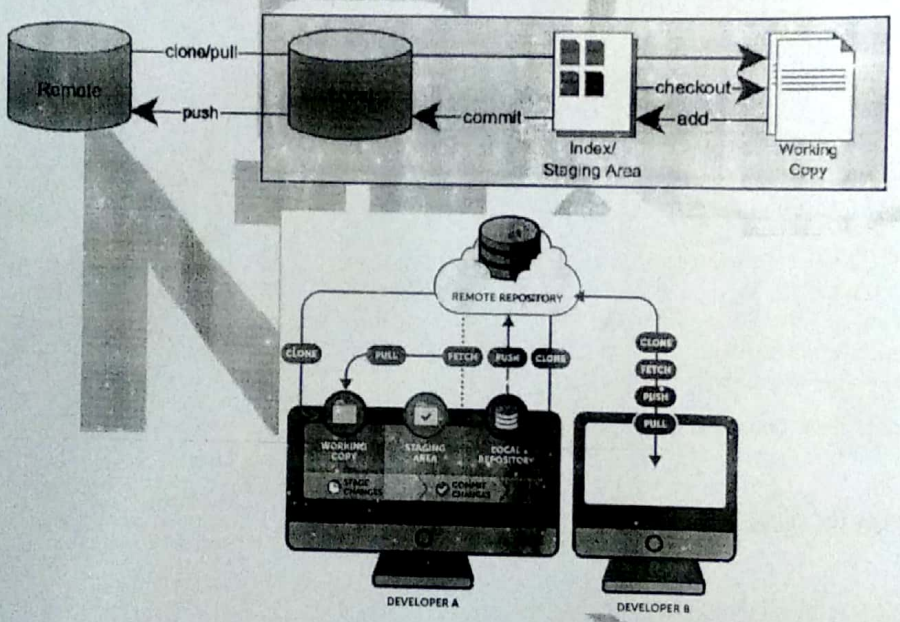
- Git Box
- Git-Xdev

- GitUp
- Fork

Cross-Platform Git Clients

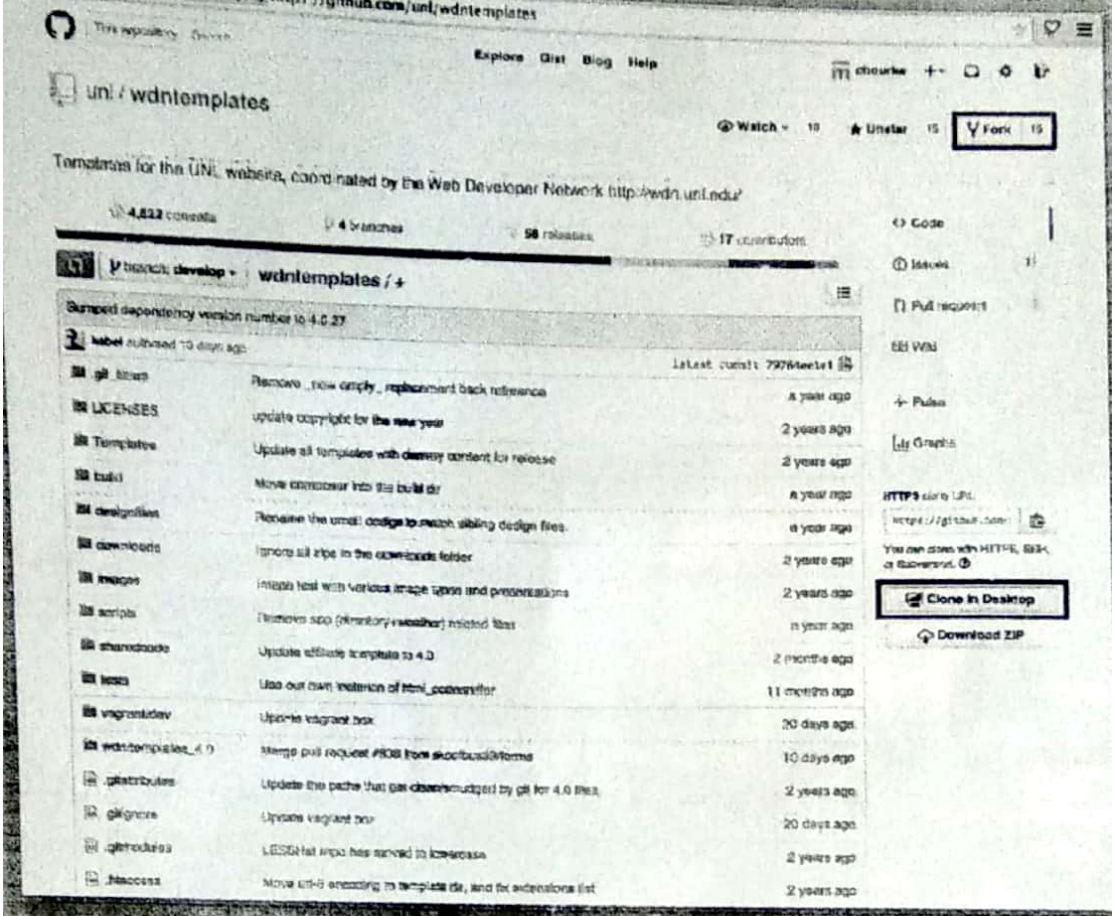
- Smartgit
- Git Kraken
- Tower
- Git-Cola
- GitEye
- GitAhead

GIT Lifecycle Diagram



Cloning an Existing Repository

To clone an existing repository hosted on GitHub, point your browser to its URL. On its page there will be several options to clone, fork or download the repository (see Figure 3).



Forking and/or Cloning on GitHub.com

ick the "Clone in Desktop" option, you'll be prompted to allow the GitHu
ne the repository to your local file system (you will be prompted to indica
setup a permanent clone path/directory). You will be able to make chang
t you will not be able to push changes to the original project unless you a
te permission. However, you can create a new repository in your GitHub a
ect back to your own repository.

essentially does this in reverse. If you choose this option, a new repositor
account and the project will be copied to this new repository. Then, in yo
e it as a local copy to work on by clicking the "Add a repository" button i
d in Figure 4.

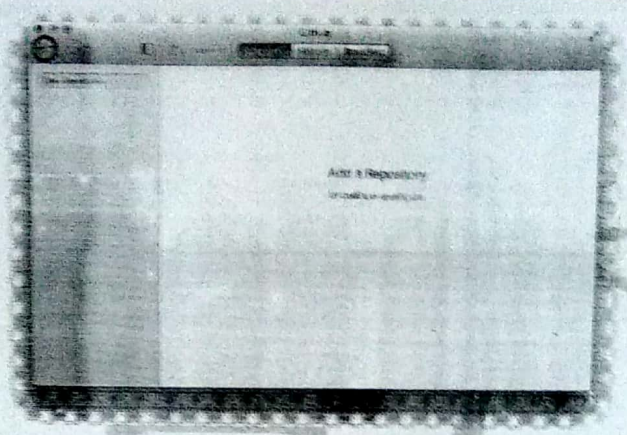


Figure 4: Cloning in the GitHub Client

3.1 Creating & Sharing Your Own Project

To share/publish a project to GitHub, you can start with an existing project or create a repository and then start working on your project.

1. Open your Git hub client and click the "Add a repository" button as in Figure 4.
2. Select the "Create" tab and select the directory of the project you wish to create a repository with as in Figure 5.

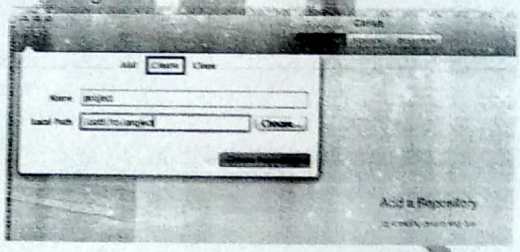


Figure 5: Creating a Repository in the GitHub Client

3. Upon success, the Git client should appear as in Figure 6; you can now make an initial commit by filling in the commit message and description and clicking "Commit to master"

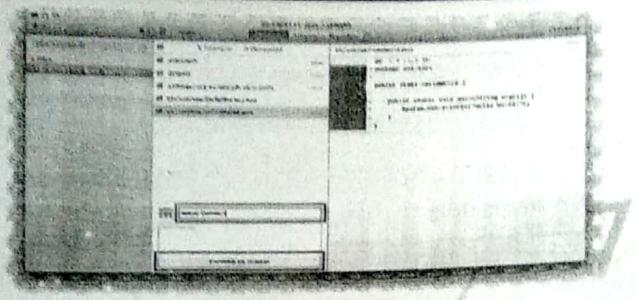
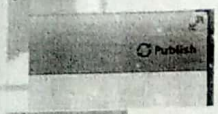
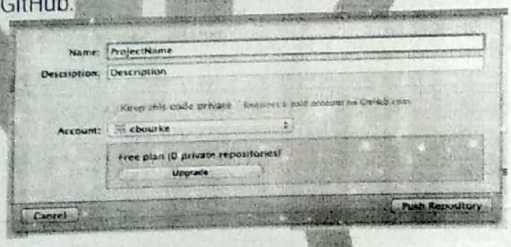


Figure 6: Committing in the GitHub Client

4. You can now "publish" your repository to GitHub by clicking the "Publish" icon in the top right of the Git client



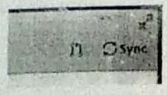
5. This opens a new dialog where you can specify the name and description of the project as it will appear on GitHub.



To finish up, click the "Push Repository" button and observe your new project on GitHub

3.2 Making, Committing & Pushing Changes

You can make changes to your local project and the changes will automatically be detected in the Git client. As in the previous step-by-step process, you can select a subset of changes to commit. Once committed, you can push the changes by clicking the "Sync" icon at the top right:



4 Using Git via the Eclipse

Eclipse is an industry-standard Integrated Development Environment (IDE) that integrates code editors (with markup) and build tools to streamline the development process. There are many plugins and utilities that can be used with Eclipse to interact with git. However, the latest version

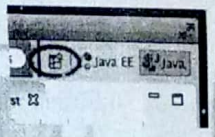
of Eclipse (Luna as of this writing) supports git natively. The process below describes how to use this functionality.

Advantage: using a single IDE/interface keeps things simple

Disadvantage: interface can be a bit clunky; it is more difficult to see differences

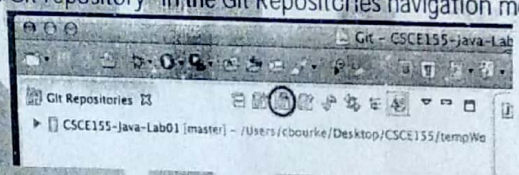
4.1 Cloning an Existing Repository

1. First we need a Git *perspective* (a context in the Eclipse User Interface that will allow us to work with Git). To open the Git perspective, click on the "Open Perspective" tab in the upper right:

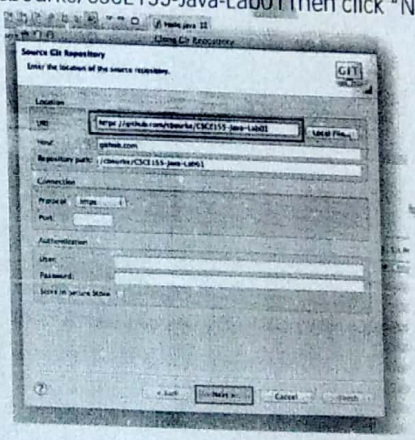


Select "Git" from the menu and click OK

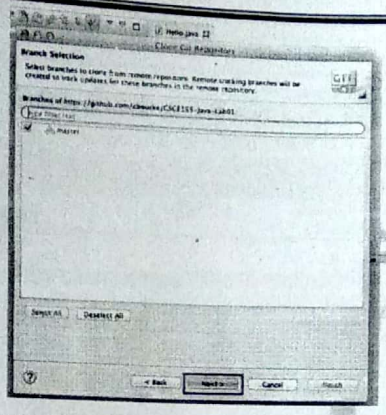
2. Click the "Clone a Git repository" in the Git Repositories navigation menu:



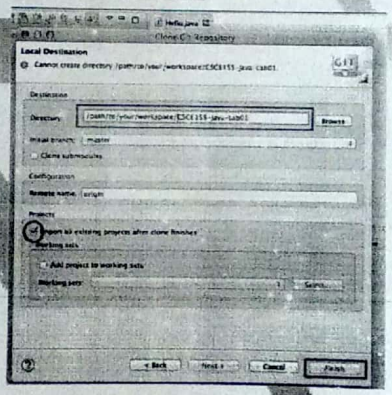
3. Copy/past or type into the URI field, the URL of the project that you want to clone. For example, if you wanted to clone Lab 01 for CSCE 155E/H, you would use the URL https://github.com/cbourke/CSCE155-Java-Lab01Then click "Next"



4. Once Eclipse has grabbed the project, the "master" branch should be selected (checkbox); click "Next" again.



5. Select the directory where you want your project to be saved. Caution: the default option may not correspond to your default workspace. You may want to change it to your workspace, but the choice is yours. Also mark the "Import all existing projects after clone finishes" checkbox option or you will need to manually import the cloned project into Eclipse.



6. Switch back to your Java or JavaEE perspective and you can see your cloned project.

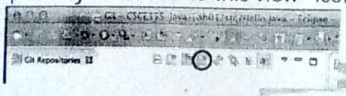
Note: this process assumes that the project you are cloning originated from an Eclipse project. Eclipse expects that files be organized in a particular way and that configuration files are present that describe how the project is setup. If the project was not an Eclipse project, you'll need to clone/setup the project in Eclipse manually.

If the owner of the repository that you just cloned ever makes changes, you can "pull" those changes from the repository by right-clicking the repo in the Git Perspective and selecting "Pull."

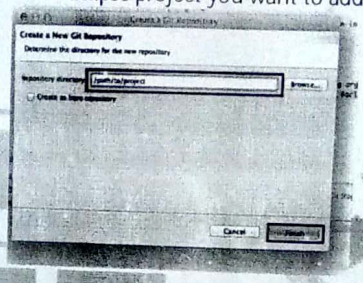
4.2 Creating & Sharing Your Own Project

Create and develop your own project in Eclipse and get it to the point where you want to make an initial commit and push to Github. Then do the following:

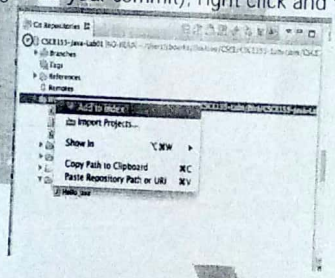
1. Before continuing you will need to create a repository on Github. To do this, refer to the steps in Section 1.3.
2. Open the Git Perspective in Eclipse.
3. Click the "Create a new repository and add it to this view" icon:



4. Select the project folder for the Eclipse project you want to add as a git repo

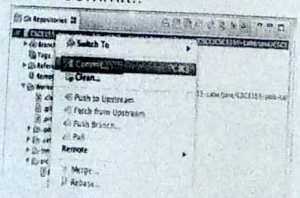


5. Expand the directory structures and select the file(s) you wish to add to the index (that is, the files you want to "stage" for your commit), right click and "add to index".

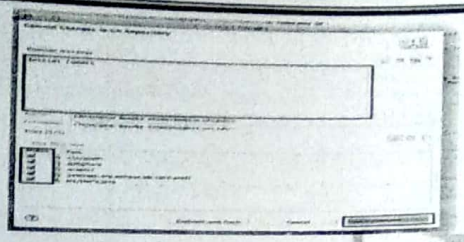


Note: adding a folder (or the entire working directory) to git's index adds all files and subfolders within that folder. You can instead, highlight individual files if you want to be more precise or intentional with each commit.

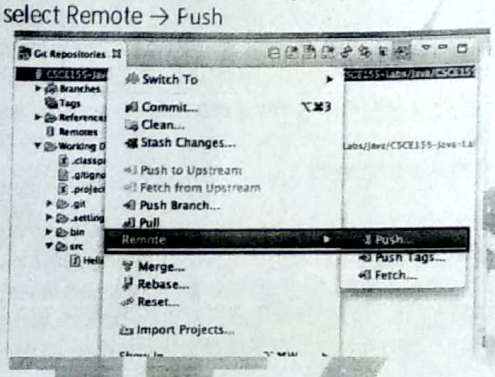
6. Right click the repo and select "Commit.."



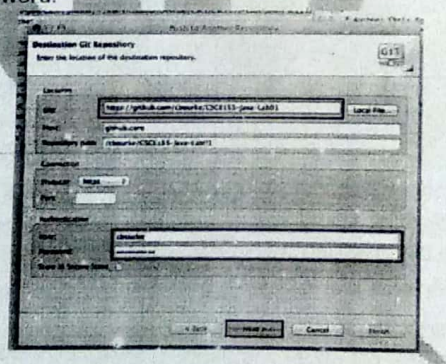
7. Enter a commit message: Initial Commit is good for the first commit, but each commit message should be descriptive and document the changes that have been made. Select the checkboxes of all the files you wish to commit. Click "Commit"



Note: you can see the differences in each file if you double click the file. 8. Right click the repo again and select Remote → Push

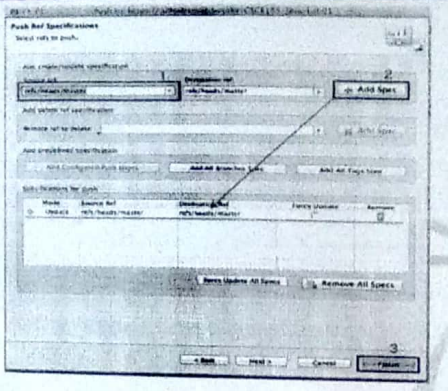


9. Enter the URL of the repo you created on http://github.com. Then enter your Github user name and password.



Note: this will only work for *your* repositories or repos on which you are a collaborator and have been granted write access.

10. Select "master" from the "Source ref" menu and click "Add Spec". The branch should now appear in the "Specifications for push" menu. You can now click "Finish".



11. If successful, a new dialog confirming the push should appear and your files should be updated on Github.

4.3 Making, Committing & Pushing Changes

Now that your code is committed to Github's servers, you'll eventually want to make changes to current files and/or add/remove files and commit these changes. Once you have made your changes, you can essentially repeat part of the process above; however, steps 1–4 will not be necessary.Note that you don't need to push every commit to Github. You can make as many local commits as you want. The entire history and all the diffs (differences) are tracked between each commit.

5 Working With Others

As previously mentioned, you will not be able to pull from a private repo. Nor will you be able to push to a repo that you do not own or that you are not a collaborator of. However, you will want to do this when you work with other individuals either as partners for an assignment or as a group in a group project (assuming that your course instructor's policies allow this of course).

To do this, simply create a repository that will be used by the group as outlined by one of the methods above. You can then grant read/write access to your others by making them *collaborators* on the project. You can easily do this in Github by following the instructions at this link:

https://help.github.com/articles/adding-collaborators-to-a-personal-repository/

Once you've all been added, each of you should be able to push/pull from the same repository.

Practical of GIT HUB in the TEAM environment:-

Step1:-

Create Remote Repository in Git Hub server i.e. github.com
→Www.github.com →signup→+→Create new repository→public☑ README→
→create Repository→download copy

Step 2:-

Configure TL machine eclipse with GIT and create local repository by taking separate eclipse workspace.

i) Configure username and email id in Eclipse for GIT Eclipse window→preferences→team→ ---
→GIT→Configuration→user→TL@gmail.com→name RajaTL→Apply→Ok

ii)Launch the following tags from Showview option of eclipse→window→showview menu →GIT repository→GIT Staging→GIT Tree compare

iii)Clone remote repository to create local repository

GIT repositorY's tab→clone GIT repository→URL paste i.e. collected from server→username →password→Next→Next→Local GIT repository in the desktop(Directory)→Finish

Step 3:-

Make TL to create project and to keep project(commit) in local repository and remote repository(push).

create project→Right click→Team→share project→GIT→select local repository→finish

Right click→project name→team→commit→refresh→select all from unstaged→ drag them to staged→write commit message→TL kept project(Test project 1)→ commit

Right click on branches→Local→master→push branch→next→finish→ok

step 4:-

Make developer1 having separate eclipse workspace and ask him to create his local repository in others configuration like step 2.

Step 5:-

Make developer1 to perform pull operation to bring project to local repository and as working copy in eclipse project Explorer.

Right click→developer1 local repository→pull→file menu→import→GIT→project from GIT→ Existing local repository→developer1 repository location→import existing project→select project → next→finish

Step 6:-

Make developer2 having eclipse workspace and ask him to local repository and ask him to pull the project from report remote repository to local repository to working copy.

Step 7:-

Make developer2 to change SampleApp.Java file→commit→push.

Step 8:-

Make developer1, TL Fetching, pulling the changes in remote repository→Right click on project → team→fetch from upstream/pull

Note:- Before you commit or edit it's recommended to perform fetch and pull.

Step 9:-

Ask developer2 to add new resources (.Java) to project and commit→push

Step 10:-

Make or ask TL, developer1 performing Fetching, Pulling operation.

Step 11:-

Ask developer1 to delete one.Java file and commit→push

Step 12:-

Ask developer2, TL to Fetch and Pull the changes.

Step 13:-

Creating and Resolving conflicts,

i) Make developer1 to modify Line3 of sample.Java→commit→push

ii) Make developer2 to modify same Line3 without pull→commit→push→give error (rejected -non-fast-forward)

iii) Make developer2 →right click on project→pull→change code manually in sampleApp.java →commit→push

Step 14:-

To see the history of the file.... (anywhere)
right click→file→team→show in history

Stay 15:-

To compare workspace copy with remote repository.
Right click→Local master branch/Remote origin branch→Synchronized with workspace

Step 16:-

Creating branch

i) Keep master branch in stable mode
ii) Create branch in TL
Right click→Local master→create branch→name:mod1Branch1
iii) Make TL modifying resources and adding resources in the branch.
iv) Make developer want to pull in the branch. Right click on project→fetch choose mod1Branch1. Go to remote tracking→origin/Mod1Branch1→right click→check out
v) Make developer1 to add additional things in branch.
vi) Make TL to pull Mod1 branch.
vii) Make TL to merge mod1 branch to local/master
→Checkout local/master→right click on local/master→merge→select mod1 branch→ok

STASHING:-

→ It is all about saving uncommitted, unstable changes done to the branch to local repository. It allows to move to another branch or to merge current branch with another branch without applying those uncommitted changes. Later we can apply stash changes back to the branch.

Example:-

i) Add new code to mod2 branch but don't commit.
ii) Perform stashing→right click on project→team→stashes→.....
iii) Merge or do something on branch or go to another branch.
iv) Come back to mod2 branch→right click→team→stashes→choose the stash name→apply stash chances.

Q:-How to stop certain files or folders going to GIT repository?
Ans:-Right click→folder/file→ignore
Generally we do this for .exe, .class files

Tag:-

Creating tag (generally it will be done for releases) Go to GIT repository view→Tags→right click→create tag→tag name:release 1.0→choose commit message to put in tag, tag message→created tag.