

Java Basics

Lesson 2: Basic Walkthrough



SOFTWARE-GUILD

Credits and Copyright

Copyright notices

Copyright © 2016 by The Learning House.

All rights reserved. No part of these materials may be reproduced, distributed, or transmitted in any form or by any means, including photocopying, recording, or other electronic or mechanical methods, without the prior written permission of The Learning House. For permission requests, write to The Learning House, addressed "Attention: Permissions Coordinator," at the address below.

The Learning House
427 S. 4th Street #300
Louisville KY 40202

Lesson 4: Git Quick Start

Java Cohort

Lesson 2: Basic Walkthrough

Installation and Setup

Git can be installed on Windows, Linux or Mac. Instructions for installation on each of the operating system can be found on the git website located at <http://git-scm.com>. The key to a successful installation is installing the Git Bash command line tools. All exercises and labs will use the git bash command line to accomplish our goals.

Once you have installed Git you can set your email address and username so that this is used with each of the repositories you create locally. This will be saved in the git configuration at the machine level and used for all repositories on that machine.

Steps

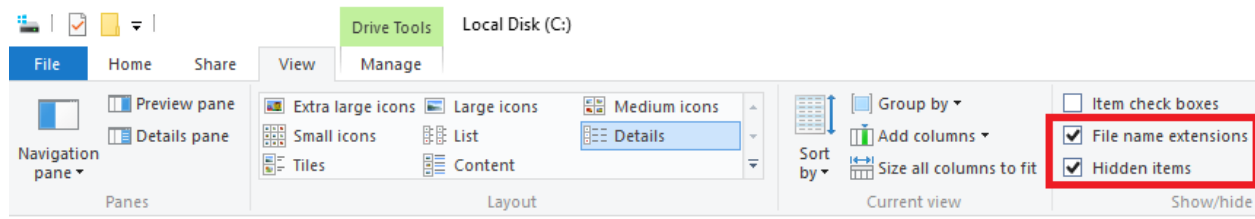
- Open a command prompt or terminal window. This varies by operating system and it is assumed you know how to do this.

Note for Windows users: If you installed the explorer integration you can also right click and select "git bash here" to open a command window with the current directory set as the current context.



- Run the command to set your username
`git config --global user.name "<USERNAME>"`
- Run the command to set your email address
`git config --global user.email "<EMAIL>"`

We also recommend that you enable viewing hidden files and file name extensions on your computer. In Windows, you can find these options in the View tab of the File Explorer window:



At this point you are ready to use git on your local machine.

Git Structure

The structure of a git repository is the same as the structure any file directory. At its core, there is a root folder that is at the top of the structure. This folder is where the actual git repository is created and will continue to reside. The name of the parent folder is irrelevant as all the information we need to know about the repository will be stored in a hidden folder in the root folder going by the name `.git`. The `.git` folder is the repository and its presence controls if the folders and files adjacent to and within will be tracked and stored within the repository.



While using a repository, it is important to remember that structure of the repository relies on the parent folder being the repository. There is no need to create repositories within any of the sub folders. Doing so would cause issues unless the sub folder is properly setup as a sub module. In the beginning sub modules are a topic we do not need to review and should not use. If you find that you have accidentally made a commit with a sub folder as a repository, no worries there are ways to remove the sub repository and have the parent repository continue to track the files.

How do I know my Git repository is set up?

If you do not have a `.git` folder in your root directory, then it is not set up correctly!



`.gitignore`

`.gitignore` is a file that is part of git but resides outside of the `.git` folder created when we create a repository. By default this file is not present in the repository and either has to be manually created, downloaded from the web (there are several sites that provide common ignore files for common use) or some developer tools will create this file for you.

The importance of this file lies in its ability to automatically exclude certain files from the git repository. Earlier, we mentioned that the Add command is used to start tracking changes to files. Unadded files will still show up in some commands as information to you, the user. However, the `.gitignore` file can override this behavior so that you never see them in your commands.

Any file extensions or directories specified in this file will be ignored and not added to the repository. They can continue to exist in the folder and sub folders of the repository, but will not be included in the repository ever. It is in the best interest of the person initially creating

the repository to generate the .gitignore file before the first commit, thus making the gitignore file part of the initial commit. This way as the project continues you will already be ignoring the appropriate files and won't have to later remove them.

Some examples of files and folders that are often included in .gitignore:

- Files that are created as part of compiling a program
 - Any other developer can recreate these files, they do not have to be stored
- Files that are specific to your computer
 - Local settings, file paths, configuration information
- Developer tool-specific files
 - For example, a program for writing code like Visual Studio or Eclipse will often have local settings files about your project that other developers would not want

Getting Started with a Repository

There are two ways to get started with a repository. Both of these methods are valid and with both you will end up with a working repository.

1. Use the git init method if you want to create a repository on your computer first and then later link it to a git server in the cloud (ex: on bitbucket).
2. Use the git clone method if you have already created a repository in the cloud and simply want to create your local copy on your machine.

git init method

1. Create a repository on the git host provider. This should be done according to the steps for the provider. Generally there will be a menu option to create a new repository.
2. Note the URL of the repository as this will be used later.
3. Create a folder on the local file system where you would like to store all repositories on the system, Ex: _repos. This folder is optional but is a best practice as then you can store all repositories you clone and work with to one location. Otherwise it may become difficult to track where you have stored all of them.
4. Create a folder for the repository. It should be named according to the name you would use on the remote host.
5. Open a terminal/git bash window from this new directory
6. From the bash window type the following command creating a new git repository in the folder location

```
git init
```
7. The next command should be to add the remote repository as follows:

```
git remote add origin <URL>
```

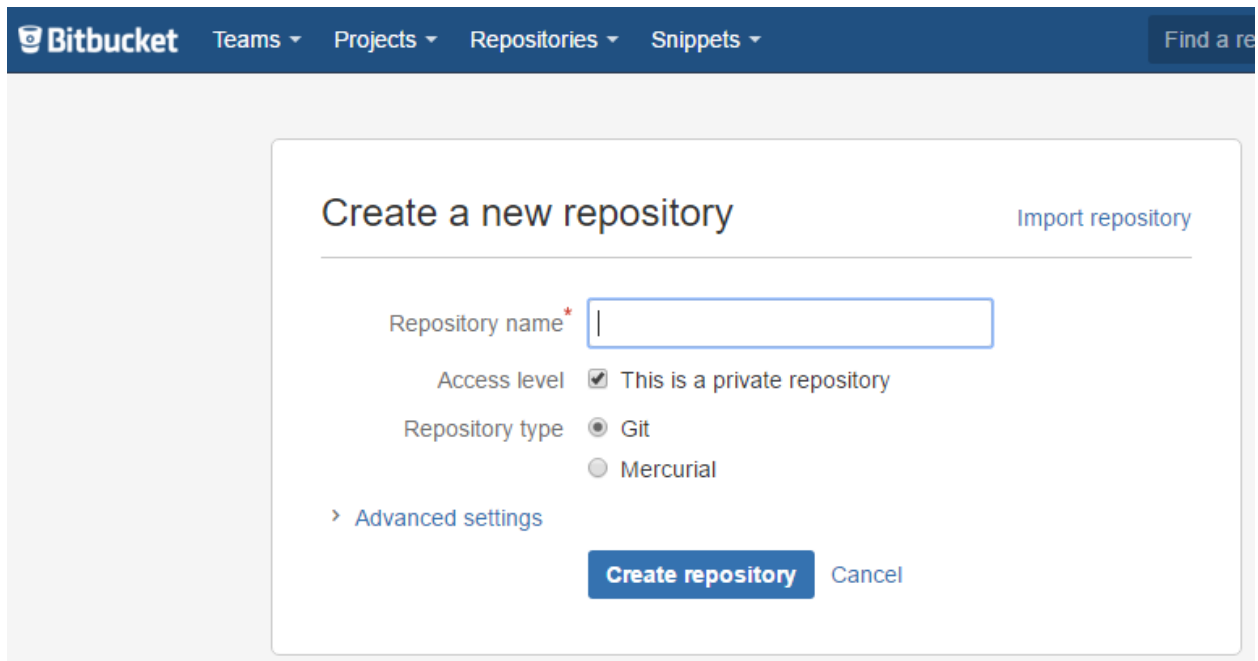
where the <URL> is the same as noted in step #2

- Now you are ready to use the repository. The only other special command to run isn't the command but the option. The first time you push to the repository use the `-u` option to ensure you continue to track against the remote repository.

```
git push -u origin master
```

git clone method

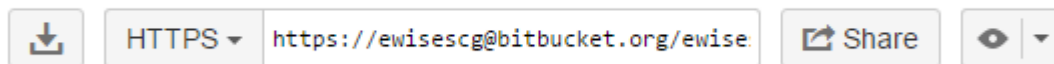
- Create a repository on the git server provider. This should be done according to the steps for the provider. Generally there will be a menu option to create a new repository.



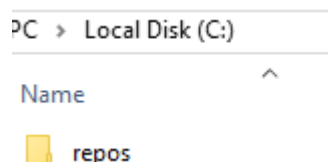
The screenshot shows the Bitbucket web interface for creating a new repository. The top navigation bar includes 'Teams', 'Projects', 'Repositories', and 'Snippets'. The main heading is 'Create a new repository' with a link to 'Import repository'. The form contains the following fields and options:

- Repository name***: A text input field.
- Access level**: A checkbox labeled 'This is a private repository' which is checked.
- Repository type**: Radio buttons for 'Git' (selected) and 'Mercurial'.
- Advanced settings**: A link to expand more options.
- Create repository**: A blue button to submit the form.
- Cancel**: A text link to cancel the operation.

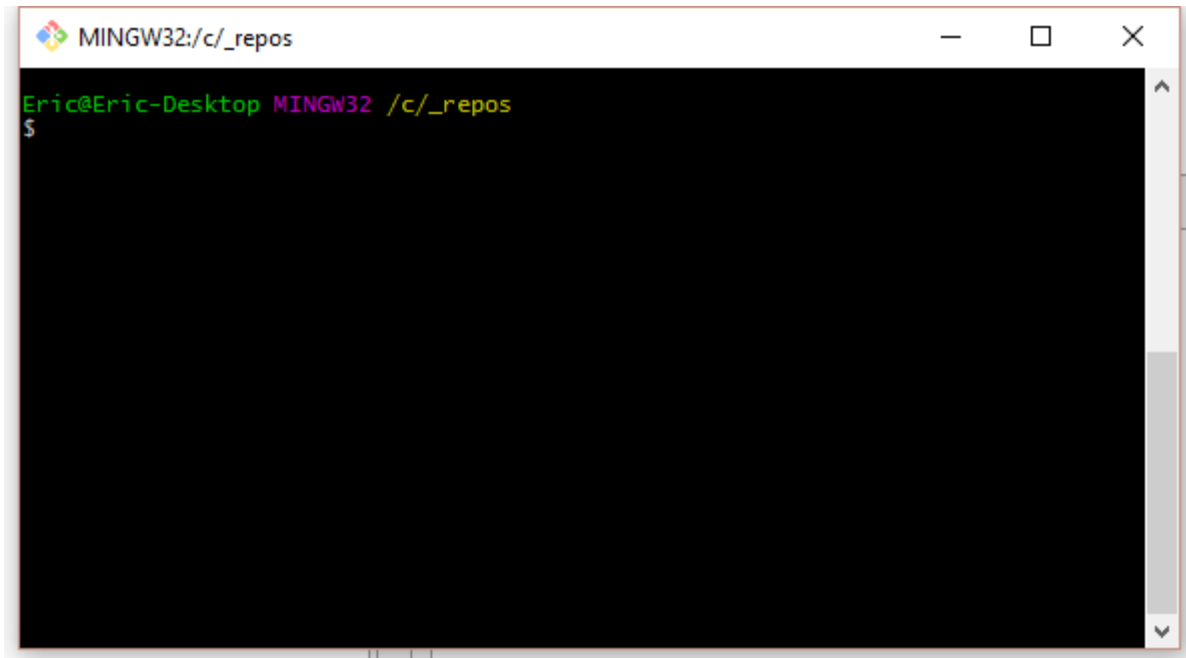
- Note the URL of the repository, as this will be used later. You can usually find this information on the overview page for the repository:



- Create a folder on the local file system where you would like to store all repositories on the system, Ex: `_repos`. This folder is optional but is a best practice as then you can store all repositories you clone and work with to one location. Otherwise it may become difficult to track where you have stored all of them.



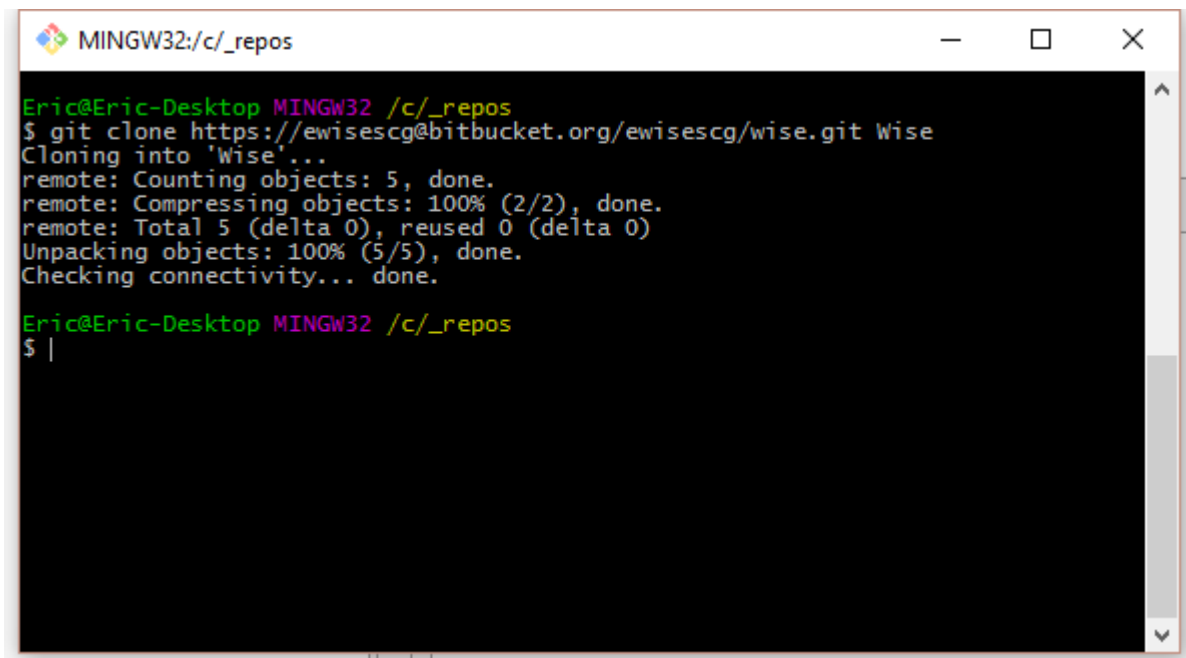
4. Open a terminal\git bash window from this _repos directory



```
MINGW32:/c/_repos

Eric@Eric-Desktop MINGW32 /c/_repos
$
```

5. Next you can run the command to clone the repository to the local system.
git clone <URL> <DIR>
where the <URL> is the URL noted in Step #2.
<DIR> is an OPTIONAL parameter that will allow you to specify the name of the directory created for the repository. If this parameter is omitted the directory will be named the same as the repository on the host provider.

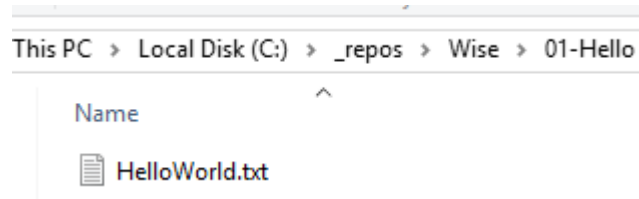


```
MINGW32:/c/_repos

Eric@Eric-Desktop MINGW32 /c/_repos
$ git clone https://ewisescg@bitbucket.org/ewisescg/wise.git Wise
Cloning into 'Wise'...
remote: Counting objects: 5, done.
remote: Compressing objects: 100% (2/2), done.
remote: Total 5 (delta 0), reused 0 (delta 0)
Unpacking objects: 100% (5/5), done.
Checking connectivity... done.

Eric@Eric-Desktop MINGW32 /c/_repos
$ |
```

6. This directory is ready to use as-is. The remote is tracked already since you cloned the remote repository, hence the connection was made then. The Wise repository we cloned in this example already had a folder named 01-Hello with a file called HelloWorld.txt on the server, so the clone command automatically copied them to the local machine into the Wise folder.



Of these two methods, the clone method is the easiest to understand and set up. Regardless of which method you use, you are now ready to add new files, make changes to existing files, and then commit and push your changes back up to the server.

Adding Files and Committing

The purpose of a repository is to add files to the repository and commit them creating a save point in the history of the repository. These save points are how we ensure that files do not get lost, changes are saved and developers can continue to collaborate on a single project.

To add a file to the repository we can use the add command. An example of this command is:

```
git add --all
```

The use of the --all parameter ensures that all files will be staged including new files, modified files as well as those files removed from the repository. This command is good to get in the habit of calling in this manner. Another way to approach adding files is to call them one by one such as:

```
git add <FILE>
```

where <FILE> is the path to the file you wish to add to the repository.

It is important to note that the git add command is the only way to add new files to the repository. There is an option on the commit command to stage modified and removed files but that option will not add new files.

Speaking of commit, the commit command will create a new save point in the history of the repository. This is the points we review and care about when it comes to the repository. They provide the save point in the repository where we can get to the exact state of the files as they stood at that moment.

```
git commit -m "<MESSAGE>"
```

where <MESSAGE> is a descriptive label for the save point in the repository. This message is required when creating a new save point and if we were to leave off the option `-m` we would be prompted with a command line editor and a file that contains more information about the save point. Within this editor the cursor will be placed near the top of the document where we can add a message. This prompt is a linux-based editor and can be difficult to navigate. To enter the message you may have to hit the 'i' key to begin inserting characters. To exit you should hit 'esc' and then 'shift' + ':' and it would then be wq to write the file and quit.

As mentioned previously there is an option on the commit command to include adding modified and removed files to the stage area before the commit is invoked. This is the `-a` option and could be combined with `-m` as follows:

```
git commit -am "<MESSAGE>"
```

The Git Staging Area

Files within a git repository exist in one of three states:

- **Modified** – files have been modified or created within the working directory and are not being tracked by the repository currently.
- **Staged** – files have been flagged by the repository to be included in the next commit. This state is not a save point in the repository but a holding place for files before committing them. This is also referred to as the Index. The Index is important to understanding other aspects of git that are more advanced.
- **Committed** – files have been saved as a point within the history of the repository. This point is able to be referenced at any time and will display in the history.

In order to save files we must take the file through the different stages until a commit point is created. Only then can we be sure that everything is saved and if disaster happens we can recover the files in their current state.

Git Diagnostics

git status

Git status is your lifeline into the repository and what the current state of the repository is. This method can tell you what has been staged and what files does the repository not yet track. The status also includes where the repository stands compared to the remotes that are being tracked. This can tell you if you have a push to complete or if you have the possibility of a merge conflict.

In order to run the git status command simply call it as follows:

```
git status
```

a possible return from this command could be:

```
On branch master
Initial commit
Untracked files:
  (use "git add <file>..." to include in what will be committed)
    Pre-Work/
nothing added to commit but untracked files present (use "git add" to
track)
```

In this case we have untracked files. The status will even give us hints as to the commands to run to resolve or move forward with the repository. In this case the git add command to add the directory to the repository's tracked files.

In another case, git status could return more information such as:

```
On branch master
Your branch is up-to-date with 'origin/master'.
nothing to commit, working directory clean
```

This return shows the status being up to date with the origin/master. The current repository could be ahead, behind or diverged from the current line of history.

- **Ahead** – implies we need to pull the changes and update our local repository
- **Behind** – implies we need to push our changes locally to the remote repository
- **Diverged** – implies there are changes locally that the remote does not have and changes on the remote that our local repository does not have. This is a state in which we may have conflicts when we pull the changes in the remote locally. In this case we may need to resolve conflicts in any files that were changed in both places. See merging for more information.

git log

Sometimes it becomes necessary to review the history of the repository. In these cases we can run the git log command to see that history. With a few select options we can even make the history easier to follow and view. The command will return something like:

```
$ git log --oneline --graph --decorate
*   ad2564d (HEAD -> master, origin/master) fixed the merge conflict
|\
| * 4dc020b added a new line to the file
* | 93a6b42 second line from Ward added
|/
* cd375ca initial commit - adding file1
```

where each commit is listed with a single line using the unique id of the commit and the message entered. The graph provides a nice dotted display of the history. Lastly, decorate provides a view of the master and origin/master branches and where they are currently residing.

Working with Remote Repositories

Pulling changes from a remote repository

Pulling changes from a remote is quite easy. We run the git pull command. Now in some cases you may see developers leave the remote name and branch name off the command but it is a good idea to get use to using them. Therefore the command will appear like:

```
git pull origin master
```

where origin is the default name of the remote repository.

Master is then the name of the branch we wish to pull from. If we remove master, we will pull the branch information and merge into the current branch we are on.

Pushing new changes to a repository

Pushing is similar to pulling in that we can specify the remote repository and the branch. It is a good habit to get into. Thus our command would appear as follows:

```
git push origin master
```

where again, if we leave off the remote (origin) and the branch (master), then the default remote for the current branch will be used. In either case, this command will take the local changes and attempt to save them on the remote repository.

With this command there is also a `-u` (`--set-upstream`) option will mark the branch to be tracked on the remote going forward. This is important in cases where you would like to establish a link with the remote for future pushes. The `git status` command will also track against the remote branch, so you can see how your local repository differs from the latest information copied from the remote repository during the last fetch command.