A hands-on tutorial on using **git**

Ç. Çöltekin, University of Groningen

Introduction

The aim of the exercises below is to get you started with using, `git` a powerful and popular *version control system* (VCS). The exercises walks you through creating a simple web-based application using PHP & MySQL. The application is used as a case study, the aim of this tutorial is not to teach you PHP and/or MySQL. PHP/MySQL tutorials are abundant on the Internet, and there are too many books to list here. Pick the one that suits your preferences best.

This tutorial only aims to get your hands dirty, and start using `git`. You will probably need other sources as you go along. There are quite a few good books or online tutorials on `git` as well. To name a few,

- http://www.ralfebert.de/tutorials/git/
- http://www.vogella.de/articles/Git/article.html
- and pointers to more: http://sixrevisions.com/resources/git-tutorials-beginners/
- and a reminder: all `git` commands accept the option `--help` to display detailed usage information.

For the sake of exercises below, we will work through a simple web-based application: a birthday calendar. All exercises are presented using UNIX command line and assuming you are using a text editor on the system where the PHP-based web page is served. The exercises should work without modification on the server `siegfried`. In fact, they should work (with minor modifications) on any UNIX-like operating system with `git`/PHP/MySQL. If you are on a different operating system, for example Windows, you will need to figure out how to use the replacement path names or the commands to do common tasks.

You can use various other methods to achieve the same effect. The official `git gui` or some other GUI for git may be used. Or, you can even use an integrated development environment, such as Eclipse, in combination with `git`, to develop PHP applications. However, these are not covered in this exercise set.

Exercises

We start with initializing our repository. This is done with the command `git init`. But we first need to create a directory for our repository. So, here is the list of commands we run:

```
$ cd public_html  # this is normally where your home directory is
$ mkdir bdcal     # we call the project BDcal
$ cd bdcal
$ git init
Initialized empty Git repository in /home/cagri/public_html/bdcal/.git/
```

NOTE: A bar on the left or right margin marks the actual exercises: a set of commands you need to type or a question you need to answer.

`git init` creates an empty repository in the subdirectory `.git` in the project directory for storing all repository related information. You will typically not need to deal with any of the files there (we will see a few exceptions below).

NOTE: To mark the difference between the commands and the output of the commands in the listings, the commands you need to type on the command line is prefixed with a `$` sign. Commands should be typed without `$`.

Before starting real work, we'll digress a little bit, and tell git who we are. To do that, type

```
$ git config --global user.name "My name"
$ git config --global user.email "my@email"
```

You should, of course, replace `My name` with your name, and `my@email` with your email address. Git will use these to identify every change you make in the repository. If you do not supply them it will try to do its

best to guess. But most of the time you will like to set these to some identity that you are happy to share with your audience.

`git config` updates various configuration options for git. The `--global` option here tells that this is a global setting. Global settings are kept in `$HOME/.gitconfig` on UNIX-like systems. There is another file `.git/config` in the project directory where configuration options local to a project are stored. If the same configuration option is specified as both global and local options, the local one will take precedence.

We will use two fake identities for the sake of demonstration of two people working on the same project. Since the exercises are prepared for a single person, we will make use of the local configuration to see this difference. The names we will use for the exercises are 'Linus' and 'Richard'. The exerceses that you need to do as **L**inus will have a bar on the **l**eft margin, and the exercises you need to do as **R**ichard will have a bar on the **r**ight margin. We will first start as Linus and type the following commands in our project directory.

```
$ git config user.name "Linus"
$ git config user.email "linus@example.net"
```

Note that we do not usethe `--global` option here.

Now, we are ready to do some work, and record it in the git repository. Being a good programmer, we first start documenting our design.

Create a file `design-doc.txt` with the following content using your favorite text editor:

```
       BDCal -- a personal birthday calendar application

This application keeps track of birthdays of friends & family. It has
two screens.
    - First, a web-page to add or modify a birthday for a person.
    - Second, another web page to list the birthdays.

Database desgin is simple: a sigle table with two columns "name" and
"birthdate".
```

Now that we have some work done, we'd like to record that work in the our repository. But let's check the status of our project first. To do that we use the command `git status`.

```
$ git status
# On branch master
#
# Initial commit
#
# Untracked files:
#   (use "git add <file>..." to include in what will be committed)
#
#       design-doc.txt
nothing added to commit but untracked files present (use "git add" to track)
```

If you use git from the command line, you will use this command often. `git` reports that we have an untracked file, and actually suggests us to user `git add` to start tracking it.

```
$ git add design-doc.txt
$ git status
# On branch master
#
# Initial commit
#
# Changes to be committed:
#   (use "git rm --cached <file>..." to unstage)
#
```

```
#       new file: design-doc.txt
#
```

This time git reports that it knows about the file. However, the change (new file) is still not recorded in the repository. `git` has a two-stage mechanism for committing changes, first you need to `add` the change, and you can then `commit` them. This may be confusing at first, but in the long run you will find that it is actually a useful feature.

Now, lets do the real commit:

```
$ git commit -m "added the design document"
Created initial commit f5d62dd: added the design document
 1 files changed, 9 insertions(+), 0 deletions(-)
 create mode 100644 design-doc.txt
```

`git` tels us what it did: this is an initial commit with one file, and 9 lines in total. The option `-m` allows us to give a comment on the command line. If you do not, git will start a text editor for you to type the comment.

Check if you have any untracked or not-committed changes.

NOTE: A black bar on one of the margins indicate an exercise for which you need to figure out the answer yourself.

Now that we have an initial design, we'd like to create our database on the MySQL server running on the 'localhost'. Assuming a database called 'bdcal' was already created, use MySQL command-line utility to connect to the database, and create the table.

```
$ mysql -s -p bdcal
Enter password:
mysql> create table birthdays(name varchar(50) not null,
                              day int not null,
                              month int not null,
                              year int);
```

A common practice useful for application that use a database is to record the steps to recreate the database. So, we'd like to store this information in our project.

Using your favorite text editor, create a file `db-init.sql` which includes the above `create table` statement.

Next time we need to create the database from start, we could just type `mysql -p bdcal < db-init.sql`, to create the necessary table(s). You could also use `mysqldump` command to get a complete backup of the database which would include the SQL statements necessary to create the database and insert the data in it.

Here, we decided to use separate `day`, `month` and `year` fields, so that we can record the birthday even if we do not know the year of birth. However, since our initial design assumed a single `birthday` column, this creates a discrepancy between the implementation and our documentation. Before progressing further, we'd like to fix this.

Using a text editor, change the following lines in `design-doc.txt`,

```
Database desgin is simple: a sigle table with two columns "name" and
"birthdate".
```

to

```
Database desgin is simple: a sigle table with columns "name",
"day", "month" and "year", where "year" can be null.
```

Now run `git status` again. `git` will report that there is an untracked file `db-init.sql` and a modified file `design-doc.txt`, but none of them are scheduled for commit yet.

Use `git add` to add the new file, and `git commit -a` to commit the changes. Use a descriptive message for the commit. Remember that `-a` option of `commit` adds the changes on tracked files automatically, but you have to add a new file using `git add`.

Now that our database is ready, we can start writing our web-based application. First, we will start with displaying the birthdays in the database. However, since it would be nice to have a few birthdays already in the database to test. We'd like to add a few example birthdays.

Connect to your database using `mysql` command line client (or `phpMyAdmin`) and add birthdays for at least two friends.

Now we are ready to start coding.

Create a file `display.php` in the project directory with the following content

```
1  <html>
2  <head></head>
3  <body>
4  <?php
5     mysql_connect('localhost', 'dbuser', 'dbpass');
6     mysql_select_db('bdcal');
7     $res = mysql_query('select *, (year(now()) - year) as age
8                         from birthdays
9                         order by (12 - month(now()) + month) % 12');
10
11    while ($row = mysql_fetch_assoc($res)) {
12         echo "${row['name']} ";
13         echo "${row['month']}-${row['day']} ";
14         if ($row['age'])
15             echo "(${row['age']})";
16         else
17             echo "(??)";
18         echo "<br>";
19    }
20
21    mysql_close();
22
23 ?>
24 </body>
25 </html>
```

Of course, you need to replace `dbuser` with your database user name, `dbpass` with your database password, and if necessary `bdcal` with the name of your database. We will skip the description of this code segment. However, you are encouraged to study the code and understand it.

What does `order by` statement on line 9 of the above PHP code do?

Test the application using a web browser. If you followed the previous steps the web page should simply names and birthdays of people you entered in the MySQL database.

Add the new file, and commit it with a descriptive log message.

Until this point, all hands-on work has been done by Linus, even though Richard was helping along the way, now they want to work in parallel on different tasks. For that, Richard wants to get his own working copy. In `git` a working copy comes with the complete repository. This allows all developers to have a complete history of the project, with the expense of some additional storage space (which generally is not much).

To a get a copy of a repository we use the command `git clone`. For these exercises you will use another project directory in your own `$HOME/public_html`. Opening another terminal window, and marking each with the respective fake identities (e.g., changing the terminal title bar or shell prompt may help you distinguish which commands to type on which window) may be helpful.

The following lists the commands for creating a clone of the current state of the repository.

```
$ cd public_html
$ git clone bdcal bdcar
```

```
$ cd bdcar
```

NOTE: Remember that a bar on the right margin means we type these commands as Richard. Starting with the above commands we will switch hats rather frequently, so watch out.

These commands create a clone of the repository under the directory `bdcal` as `bdcar` (note the change in the last character of the directory name). Note that, typically, you will this operation is done over network, using a protocol like `http`, `ssh` or `git`s own protocol.

Set our new fake identity with name `Richard` and email `richard@example.org` in the newly cloned repository `bdcar`. Make sure **not** to use `--global` option. We only want to use this identity for these exercises.

Richard plans to work on the database update page. Nevertheless, he wants to know what was done so far. Of course, he could read the source files and understand it, but `git` logs provide a more concise option. You can look at the history of the changes with the command `git log`.

```
$ git log
commit 7159818aad74e062baebcf83bd9a567cadb81fe4
Author: Linus <linus@example.net>
Date:   Sun Nov 20 00:58:57 2011 +0100

    added PHP code for displaying birthdays

commit 29e27c07151599d44c8c079eef664a793cb3ee0c
Author: Linus <linus@example.net>
Date:   Sat Nov 19 22:19:34 2011 +0100

    added db-init.sql & updated docs for minor DB design change

commit f5d62ddeb71d36b816146a4046f53d3320d00204
Author: Linus <linus@example.net>
Date:   Sat Nov 19 03:01:49 2011 +0100

    added the design document
```

We see three changes in the logs. Each change is listed with the commit message, the date/time of the commit, the identity of the person who did the commit, and an arbitrary commit ID in the form of a hexadecimal number.

We realize that we do not know what the design change was about, and want to see what was done with the second commit with id `29e27c07151599d44c8c079eef664a793cb3ee0c` is about. There are a couple of ways to check these changes, but the most convenient in this case is probably using the `git show` command. If you type `git show` with the commit ID as parameter, you should get something similar to the following:

```
1  $ git show 29e27c0
2  commit 29e27c07151599d44c8c079eef664a793cb3ee0c
3  Author: Linus <linus@example.net>
4  Date:   Sat Nov 19 22:19:34 2011 +0100
5
6      added db-init.sql & updated docs for minor DB design change
7
8  diff --git a/db-init.sql b/db-init.sql
9  new file mode 100644
10 index 0000000..3274659
11 --- /dev/null
12 +++ b/db-init.sql
13 @@ -0,0 +1,4 @@
14 +create table birthdays(name varchar(50) not null,
```

```
15 +                        day int not null,
16 +                        month int not null,
17 +                        year int);
18 diff --git a/design-doc.txt b/design-doc.txt
19 index 3e37386..5e3be5f 100644
20 --- a/design-doc.txt
21 +++ b/design-doc.txt
22 @@ -5,5 +5,5 @@ two screens.
23      - First, a web-page to add or modify a birthday for a person.
24      - Second, another web page to list the birthdays.
25
26 -Database desgin is simple: a sigle table with two columns "name" and
27 -"birthdate".
28 +Database desgin is simple: a sigle table with columns "name",
29 +"day", "month" and "year", where "year" can be null.
```

`git show` gives you the commit message followed by the changes to all files listed in `diff` format The lines added by the commit starts with a `+` and lines removed by the commit starts with a `-` in the very first column of the `diff` output. All changes are listed together with the surrounding context (of three lines above). Also note that we did not use the complete commit ID. For any git command that uses commit IDs only the first few digits are enough (as long as they are distinctive).

Study the output of `git show` above. Are the lines 24 and 25 added, removed or context lines?

Knowing what our project partner has been working hard, now, time to work on our part.

Here is a simple PHP code for adding new birthdays. Type it in using a text editor, and save as `add.php`.

```
1  <?php
2
3      if (isset($_POST['submit'])) {
4          mysql_connect('localhost', 'dbuser', 'dbpass');
5          mysql_select_db('bdcal');
6
7          mysql_query("insert into birthdays values('${_POST['name']}',
8                                                    ${_POST['day']},
9                                                    ${_POST['month']},
10                                                   ${_POST['year']})");
11         echo "Birthday added.<br>";
12         mysql_close();
13     }
14 ?>
15     <form action="add.php", method="post">
16     Name: <input type="text" name="name"/><br>
17     Birthdate (YYYY-MM-DD): <input type="text" name="year"/>-
18                             <input type="text" name="month"/>-
19                             <input type="text" name="day"/><br>
20     <input type="submit" name="submit" value="submit"/>
21     </form>
```

WARNING! For the sake of brevity, the above code does many things you should **never** do. For example, not checking whether the update completed successfully or not, and even worse, using user input in a query without validating it, which may cause serious security problems.

Check whether the code works as expected, `add` and `commit` to the repository.

While Richard is busy with he above code, Linus wants to improve the display page. He thinks that instead of putting data on separate lines with spaces in between, a table may look better, and header lines, such as `name`, `birthday` and `age`, indicating what the column is about would also be useful.

Modify `display.php` such that the result is displayed in an HTML table. Test the result, and `commit` your changes.

NOTE: Pay attention to change of identity.

Now, the two copies of the project have diverged. Each project member has one commit that is only available on their own tree. You can see that by running `git log` on both copies of the project.

Richard wants to do a few more changes. However, he expects that his hard-working project partner have some additions at this point, so he wants to incorporate these additions to his repository first. The command `git pull` takes updates from another repository.

Run the following command

```
$ git pull
remote: Counting objects: 5, done.
remote: Compressing objects: 100% (3/3), done.
remote: Total 3 (delta 2), reused 0 (delta 0)
Unpacking objects: 100% (3/3), done.
From /home/cagri/public_html/bdcal/
   7159818..6dadefa  master     -> origin/master
Merge made by recursive.
 display.php |   12 ++++++++++-----
 1 files changed, 7 insertions(+), 5 deletions(-)
```

`git pull` `fetch`es the changes from the original repository, and `merge`s to your repository. Now, Richard has all the changes Linus had, and his own changes up to this point. However, Linus' copy of the project does not have the changes Richard made. Since in our particular setup Richard has read-write access to Linus' repository (since we are using the same UNIX user), he can `push` his changes as well.

Now, run the following command to push changes

```
$ git push
Counting objects: 7, done.
Compressing objects: 100% (5/5), done.
Writing objects: 100% (5/5), 915 bytes, done.
Total 5 (delta 2), reused 0 (delta 0)
Unpacking objects: 100% (5/5), done.
To /home/cagri/public_html/bdcal/.git
   6dadefa..a80f93e  master -> master
```

At this point, both repositories should be identical.

For a distributed VCS like `git` there is no 'original' repository. Each repository copy is independent, except sharing the history. Nevertheless, when you `clone` a repository, `git` marks the source repository as `origin`, and `pull` `push` use this repository as the default repository to interact with. You can use any repository you have access to for fetching changes you do not have, and there are other possibilities of updating a repository with other. We will not go into details of many ways two or more repositories may interact. However, we note that it is a common practice to have a dedicated repository writable by all project members. This a similar method to client-server VCSes. However, benefits of distributed system remains. Each individual developer can commit their changes without access to the VCS server and share the changes when they are ready.

Now, happy that the changes merged nicely without trouble, Richard decides to work on a problem he has realized earlier. Both PHP scripts use the following two lines for connecting to the database:

```
mysql_connect('localhost', 'dbuser', 'dbpass');
mysql_select_db('bdcal');
```

First problem her is that the code is repeated unnecessarily. If, for example, we change the password, we have to do the same change in multiple locations (currently two). Like repeated data in the databases, repeated code in programming is a bad practice, and easy to avoid.

There is another problem: putting passwords inside the code is a bad idea. Anybody who has access to the code will see the password. Being open source software enthusiasts, both our programmers like to share their code, but certainly not their passwords. Furthermore, all the values, host name, user name, password and database name, seem to be configurable values. Ideally, we should separate the code from the configuration. The solution Richard comes up with is to place these variables into a file `dbconfig.php` and use the variables (instead of the values) in each script.

Since the change requires editing multiple files, and possibly introducing new bugs, Richard does not want to do it on the working system. VCSes provide a nice mechanism for these cases: you can create multiple branches without taking complete copies of the whole project. Furthermore, branches in `git` are cheap and fast. So, we want to crate a temporary branch called `exp`, and do the changes in this branch.

Type the following two commands to crate the new branch, and switch to it.

```
$ git branch exp
$ git checkout exp
```

`git branch` command, if a branch name is given, creates a new branch. `git checkout` switches to the new branch. The initial branch `git` crates when you do `git init` is called `master`, which you have already seen many times in the output of `git status`. If you check the status now, it should tell you that you are on branch `exp`. Another way to check the branch you are on is to run `git branch` without any arguments, which will list all branches, and mark the current branch with an asterisk '`*`'. You are encouraged to try both commands and see the output.

Now, we are ready to make our changes on our new experimental branch.

Create a new file, `dbconfig.php` with the following content.

```
<?php
    $dbhost = 'localhost';
    $dbuser = 'dbuser';
    $dbpass = 'dbpass';
    $dbname = 'bdcal';
?>
```

Modify both `display.php` and `add.php`, replacing the lines,

```
        mysql_connect('localhost', 'dbuser', 'dbpass');
        mysql_select_db('bdcal');
```

with

```
        include_once('dbconfig.php');
        mysql_connect($dbhost, $dbuser, $dbpass);
        mysql_select_db($dbname);
```

After testing both display and add functionality, we would like to commit this change. Note that we do not want to commit the `dbconfig.php` file. It is not part of our source code, it is a configuration file. It is a common practice to include an example configuration file, but we will skip this here.

Commit the changes to `display.php` and `add.php` with a descriptive log message.

Now, Richard is happy that the project is easier to maintain, and the leak of DB credentials is also prevented in case source code is opened to others. However, since he types `git status` every now and then, he is annoyed with the fact that `git status` complains about `dbconfig.php` as an untracked file every time. Luckily there is an easy way to prevent git from complaining files we do not want to track.

Create a text file `.gitignore` in the project directory with a single line containing the text `dbconfig.php`. Check the output of `git status` again.

If everything went well, you should see `git status` to stop complaining about `dbconfig.php`. However, this time it will complain about the file `.gitignore`. It is possible to add `.gitignore` to itself. However we chose to add it to the repository.

Add `.gitignore` to the repository, and commit it with a descriptive log message.

The `.gitignore` file can also contain 'wild cards', for example if you were working with a compiled language and wanted to ignore all object files, you could add a line '`*.o`' to tell git to ignore all files with extension '`.o`'.

Now that the experimental code seems to be working fine, Richard wants to apply these changes to the `master` branch. This is done using the command `git merge`.

To merge the branch `exp` to the branch `master` you first need to switch to master branch.

```
$ git checkout master
```

Before merging, if you want to see the changes on the `exp` branch, you can type,

```
$ git log exp
```

Naturally, log messages for `exp` will be the same with the branch `master` until the point of branching. However, you should see two additional commits messages in the `exp` branch that are not on the `master` branch.

If you wanted to see the actual changes between two branches, you could type

```
$ git diff exp
```

which would bring you all the changes in `diff` format.

After seeing that all changes are what we want to include, we are ready to merge `exp` into `master`.

```
$ git merge exp
Updating a80f93e..1f93f89
Fast forward
 .gitignore  |    1 +
 add.php     |    5 +++--
 display.php |    5 +++--
 3 files changed, 7 insertions(+), 4 deletions(-)
 create mode 100644 .gitignore
```

Now both branches are identical. Since we wanted the branch `exp` until we applied the changes to the `master` branch, we can now delete it,

```
$ git branch -d exp
Deleted branch exp.
```

Branching and merging are useful for many other cases. In these exercises we used a temporary branch to make sure that we do not modify the stable code until we are confident about our modifications. Branches does not have to be temporary, in some cases you may want to maintain multiple (permanent) branches. For example, we could create a new branch for using another database management system.

The merge in the above exercise is called a fast-forward merge; there were no changes on the master branch during time development was going on in the experimental branch. If there were, then we could have a possible *conflict*. When conflicts happen, `git` informs you about which files include conflicts, and marks the conflicting regions in these files. In case of a conflict, you will need to resolve the conflicts manually, and commit the result of the merge. We will not demonstrate a conflict in this tutorial, however you are encouraged to try yourself.

In our exercises so far, Richard `colened` repository of Linus. So, his repository knows where the repository `origin` is, and can `pull` the changes when he wants. And since he has write access to Linus' repository (due to our peculiar setup), he can also `push` his changes. In real world, this mode of sharing is rare, it is more common to use a common repository for the aim of only sharing the code. Various source code hosting services (such as GitHub or BitBucket) may come handy in these cases. Of course, you can also run your own service if you have the means and motivation for it.

For the exercises here, we will stick to our model, and to see how you can add other repositories for using `pull` and `push` we will do one last exercise. In our setup, Linus cannot pull directly from Richard's repository, or push to it.

`git` uses the file `.git/config` in the prject directory to store the information about the other repositories. If you examine the differences between `.git/config` files in both repositories, you will find that Richard's repository configuration includes some additional lines roughly corresponding to:

```
[remote "origin"]
        url = /home/cagri/public_html/bdcal/.git
        fetch = +refs/heads/*:refs/remotes/origin/*
[branch "master"]
        remote = origin
        merge = refs/heads/master
```

In a nutshell, these configuration directives instruct `git` that there is a remote repository `origin` located in `/home/cagri/public_html/bdcal/.git`, and the branch `master` in the local repository, in a sense, 'mirrors' the remote `master` branch. We will not go into details here, but note a few possibilities. First, note that the remote is identified by an URL. As a result, you can use any URL that git understands, such as `http`, `ssh`. Second, the branch names in remote and local does not have to match. You can for example record a remote repository, and track their `master` (or any other) branch with any local branch name you like. Also note that you can edit this file using a text editor, or you can use appropriate `git config` commands to update the values.

We now return the problem that Linus wants to be able to pull and push to Richard's repository whenever he likes, instead of asking him every time to make sure both repositories are in sync.

Using a text editor edit `.git/config` file, and add the following lines:

```
[remote "richard"]
        url = /home/cagri/public_html/bdcar/.git
        fetch = +refs/heads/*:refs/remotes/origin/*
[branch "master"]
        remote = richard
        merge = refs/heads/master
```

Of course you need to change the `url` line to match the repository you have created as Richard.

After this, we should be able to pull from Richard's repository whenever we want. However, we did not name the repository `origin`, so we need to add the repository name to the `git pull` and `git push` commands. Here is an example:

```
$ git pull richard
remote: Counting objects: 10, done.
remote: Compressing objects: 100% (6/6), done.
remote: Total 7 (delta 3), reused 0 (delta 0)
Unpacking objects: 100% (7/7), done.
From /home/cagri/public_html/bdcar/
 * [new branch]      master     -> origin/master
Updating a80f93e..1f93f89
```

This exercise concludes our tutorial. Hopefully, these exercises were useful to get you started. We have just scratched the surface of version management with `git`. `git` is a sophisticated tool, you will likely to use only a subset of features for your projects. And, like many other sophisticated tools, the best way to learn `git` is to use it.

Even though a VCS may seem too abstract at first, as you use it you will realize that it is an indispensable tool for software development. However, it can also be useful for many other cases, for example a web page, a PhD dissertation, or your list of cooking recipes.