Unit Testing, MVC Pattern, and Git

Nazmus Saquib

March 10, 2016

Contents

1 Unit Testing

Knowingly or unknowingly, we have been performing testing right from the beginning of our coding life. When we finish up writing a code and check whether it does what it is supposed to do, we are performing kind of a testing. When we debug through our code using various debug tools, we are also performing some kind of a testing. The problem with this approach is a lot of manual labor is involved, we need to step through our statements one by one. It would have been great if we could automate the process. Sometimes we insert arbitrary statements to spit out values of different variables in our code, or simply to check whether our code enters a certain code block - that's also testing. But these statements are not really productive, they don't add any functionality to our code. Moreover we need to go through our code again and either remove them or comment them out. Once again, we are stuck with manual labor. In general, we could say testing is something that we do to lift our confidence level higher regarding the fact that our code does what it should do - from the perspective of the coder and/or the end-user. If we code and test incrementally, we can figure out whether a new piece of code has somehow unearthed a previous bug at the quickest possible time. In this lecture we are concerned about a very specific type of testing - **unit testing**. There are other kinds of testing too - functional testing, UI testing, performance testing, etc. But those are out of our current scope. In your term project you are going to have to do a bit of unit testing, exactly which part(s) will have to be tested will be determined later.

1.1 What is Unit Testing

From a linguistic point of view, *unit* means the smallest building block of something. For example, in case of human body the unit is a *cell*. So what might be the unit of our source code? Our code might have a number of classes, so the answer might be a *class*. Can we go to finer granularity? Well, our classes are made up of a bunch functions, so a better answer would be a *function*. And if we want to attain an even finer granularity, we could say the various execution paths a function takes. In general, through unit testing we test each of our functions - testing perhaps all or most of the branches our functions take. While we are at it, there is a related term known as the **path coverage**. It simply denotes the percentage of paths / branches that are covered through our tests.

1.2 Some Terminologies

When we are working with unit testing, a few terminologies always seem to creep up:

Test case A test case is a function that tests another function in our source code. Generally one test case tests only one feature of a function. So one function can actually have a number of test cases associated with it.

Test suite A test suite is simply a collection of test cases. Test suites might be nested. For example, we might have a class with two functions in it. So each of the functions will have a test suite. These two test suites will make up one test suite for the class.

Assertions Generally, when we say we are asserting a statement, we are confidently saying that the statement is true. The statement might be positive or negative. When it comes to unit testing, assertions are simply functions which tell us whether a particular incident is true or not. For example, we might want to assert whether a result is equal to/greater than/less than another given value. We might simply want to assert whether a function gets called along the way of our execution. We might even want to assert whether a particular member has been defined. There is a huge array of assertions that we can perform throughout our unit testing procedure.

1.3 Unit Testing Frameworks

Irrespective of which language we are using, we will probably be able to find a number of unit testing frameworks for that language. For Java the leading framework is JUnit, for C# NUnit, for PHP PHPUnit, so on and so forth. These frameworks make our life a lot easier when we are performing unit testing. It's a good idea to have a clear concept regarding at least one unit testing framework for your primary language. As the underlying methodologies of the frameworks are quite similar, one should not face that much of a problem in switching frameworks.

1.4 Equivalence Classes and Boundary Value Analysis

Equivalence classes simply divide all possible inputs to our programs into a number of classes. To make the idea a bit more concrete, suppose we are interested in writing a simple function that is going to toggle the cases of its string argument. So basically all lower case alphabetic characters will be converted into upper case alphabetic characters, all upper case alphabetic characters will be converted into lower case alphabetic characters, and non-alphabetic characters will remain unchanged. Let us name this function *ChangeCase*. Here's a question for you, what are the different types of inputs we might have? A brief consideration yields that broadly speaking, we might have three types of inputs:

1. lower case alphabetic characters,
2. upper case alphabetic characters, and
3. non-alphabetic characters.

The utility of equivalence classes is that once we see a few inputs from a particular equivalence class is giving the expected result, we can sort of achieve a pretty high confidence level regarding the fact that the other inputs from this class will also give expected results.

Now equivalence classes alone are not really sufficient to give us that much of a confidence level, we need another concept known as the **boundary value analysis** that complements our equivalence classes. It works with the values at the extreme ends of our equivalence classes. In our example, for the first equivalence class (lower case letters) the boundary values are *a* and *z*. For the second equivalence class (upper case letters) the boundary values are *A* and *Z*. The third equivalence class does not really require any boundary value analysis. As most of the time we delineate our area of interest in our code through the boundary values of an input range, it is important that we perform boundary value analysis. Equivalence class coupled with boundary value analysis solidifies the outcome of our testing.

1.5 A Hands-on Example

In this subsection we are going to perform unit-testing for our *ChangeCase* function. I am going to use JavaScript (actually NodeJS, which can be thought of as JavaScript on the server side) for this short project. Be advised that this tutorial is not about "unit testing in JavaScript", rather it is about "basic workflow of unit testing". I have simply chosen to show it in JavaScript, that's all. At this point you have two options:

1. You can follow along with me, I'll show you everything from installation of frameworks to the very end of testing.

2. You can read this whole subsection first, understand the basics, and then get on coding with your preferred language and framework.

1.5.1 Installation of NodeJS and Jasmine

Jasmine is a unit testing framework for JavaScript. If you are on a UNIX machine, you can install NodeJS using your distro's package manager. For example if you are using Ubuntu, you would do something like this:

```
1  $ sudo apt-get install nodejs
2  $ sudo ln -s /usr/bin/nodejs /usr/bin/node
3  $ sudo apt-get install npm
4  $ sudo npm install -g jasmine
```

The second command is simply to create an alias for nodejs, as a lot of NodeJS packages internally use the name node instead of nodejs. The third command is for installing npm - node package manager. Basically npm is to NodeJS what apt-get is to Ubuntu. We can install various packages for NodeJS through npm. Through the last command we are installing Jasmine globally. The package can be installed locally on a project basis, or globally on system basis. Here I just chose to install on a system basis.

If you are a Windows user, you can go to *nodejs.org* and download binaries from there. Be sure to update your environment variables according to your installation folder, otherwise you won't be able to execute commands from the

cmd. Recent binaries should contain npm packaged with NodeJS by default, just in case it is missing, you can download npm from *npmjs.com*.

1.5.2 Directory Structure

It is a good practice to keep our source codes in one directory and our test codes in another directory. Let us create a directory named *ChangeCase*, and inside that directory we are going to create another directory named *src*. We still haven't created a directory for our test codes. We will generate a directory through jasmine while we are inside our *ChangeCase* directory:

```
$ jasmine init
$ sudo chmod 777 spec
```

The first command creates a folder named *spec* along with a few files, which we won't generally have to modify. The second command simply grants permission to all users to read, write and execute the spec folder. We are going to create a file named *ChangeCase.js* within our *src* folder - this is where we are going to write our source code. We are also going to create a file named *ChangeCaseSpec.js* within our *spec* folder - this is where we are going to write our test codes. Our final directory structure looks like this:

```
ChangeCase
├── src
│   └── ChangeCase.js
└── spec
    ├── support
    │   └── jasmine.json
    └── ChangeCaseSpec.js
```

These are the points to take away from this subsection:

1. We need to keep our source codes and test codes separate.
2. How we are going to generate the test folder and what its name is going to be depends on the framework we are using.
3. Generally we maintain one test file for one source file, and words like *spec* or *test* (depending on the framework) are appended to the name of the test file. Most of the time our framework is still going to work if we do not follow this step strictly, but it is considered to be a good practice to do so.

1.5.3 The Source Code

Our *ChangeCase* function is going to accept a *string* argument and return another string with the cases beign toggled. Let us get right into coding and then I will explain each of the major lines.

```
var ChangeCase = function(str){
  var strChanged = "";
```

```
3   for(var i = 0; i < str.length; i++){
4     if(str.charCodeAt(i) >= 97 && str.charCodeAt(i) <= 122){
5       strChanged += String.fromCharCode(str.charCodeAt(i) - 32);
6     }else if(str.charCodeAt(i) >= 65 && str.charCodeAt(i) <= 90){
7       strChanged += String.fromCharCode(str.charCodeAt(i) + 32);
8     }else{ //non-alphabetic characters
9       strChanged += str[i];
10    }
11  }
12  return strChanged;
13 };
14
15 module.exports = {
16   ChangeCase: ChangeCase
17 };
```

In line number 1 we are creating a handler for our function through the variable *ChangeCase*. So basically the function is callable by writing *ChangeCase* followed by parameter within parentheses. In line number 2 we are initializing the variable *strChanged* to the empty string, this will be returned by the function. We start iterating over each character of our parameter in line number 3. We can get the ascii value of the character in ith position of a string by the function *charCodeAt*. We can create a character from a specific ascii value through the function *fromCharCode*. In line numbers 4-6 we check if a character is lower cased, if it is we convert it to upper case. In line numbers 6-8 we check if a character is upper cased, if it is we convert it to lower case. In line numbers 8-10 we don't make any modifications to the character as it is non-alphabetic. In line number 12 we return *strchanged*.

In the last three lines we are basically making our JavaScript file "includeable". We are creating an object which has the name *module.exports* - it's important that we use this exact name. We have an item named *ChangeCase* within this object (the first one before ':' in line number 16) - technically we could have used any name here. The second *ChangeCase* refers to the variable we created in the very first line. If in the first line we used a different name like toggleCase, we would have written that instead after the ':'.

1.5.4 The Test Code

We start by first including our source file:

```
var lib = require("../src/ChangeCase.js");
```

Here we are simply creating a reference to whatever is being returned from our source file to the variable *lib*. Recall that our source file and test file are in two separate directories in the same level - so we need to go one level up, go into the *src* directory and then grab the source file. The first thing that we need to do is create a test suite for our function. In jasmine we do this through a function named *describe* which takes two parameters - a string and a callback function. The string can be anything we want, but it is generally the name of the function. Our test cases reside within the callback function. A test case is in turn created

by a function named *it*. Test cases take two parameters - a string and a callback function. The string can again be anything we want, and generally takes a very descriptive form - expressing what we are testing through this test case. Finally our assertions reside within the callback function. Assertions are simply functions named *expect*. Let us take a look at the full test code to get a better idea.

```
var lib = require("../src/ChangeCase.js");
describe("ChangeCase -", function(){
  it("changes lowercase letters to uppercase letters", function(){
    expect(lib.ChangeCase("asz")).toEqual("ASZ");
  });
  it("changes uppercase letters to lowercase letters", function(){
    expect(lib.ChangeCase("ASZ")).toEqual("asz");
  });
  it("does not change non alphabetic characters", function(){
    expect(lib.ChangeCase("$#!")).toEqual("$#!");
  });
});
```

We can see within our *describe* block we have three *it* blocks - denoting three test cases. In each of these test cases we have one assertion - respectively checking expected result of passing lower case, upper case, and non-alphabetic characters to our *ChangeCase* function. Notice the use of boundary value analysis within our equivalence classes. Points to take away from this subsection:

1. One test case should test for one particular feature / branch of our function.
2. Our test code will have one or more test suites, and those in turn are going to have one or more test cases. How we define test suites and test cases will depend on the framework we are testing.
3. Test cases contain assertions, there are a wide range of assertions apart from "assert whether something is equal to some other thing". Explore your framework's documentation to check them out.
4. While constructing test cases pay special attention to boundary value analysis and equivalence classes.

1.5.5 Running the Test Cases

If we go to our parent directory i.e. *ChangeCase* directory and run the following command:

```
jasmine
```

we should see three green dots denoting successful completion of our three test cases. If for some reason a test case fails, we will see the 'F' character in red - denoting unsuccessful completion of the test case. It will provide us a message from which we will be able to understand exactly which test case failed. Here's

an **exercise** for you - deliberately fail a test case and see whether you can figure out which test case failed from the message.

So this was a pretty simple example on how to perform unit testing. In real life scenario you will most probably have to perform more complex assertions. But the underlying concepts are pretty much the same.

1.6 Test Driven Development - TDD

When it comes to testing we often hear the term TDD - test driven development. The philosophy behind it is that our testing should drive our development. Even before we write our source code, we are encouraged to think about what our code should do (one feature at a time), and write test cases for that feature (one test case at a time). Obviously if we write our test case before any kind of source code and run them, our test case will fail (the red 'F' will show up). Once we see the red message, we write our code that implements the feature we are testing. If everything goes according to our plan, we should see green dot when we run the test cases. As a final step we can "refactor" our code - meaning we can optimize our code, remove duplicate codes, etc. Once we are done with one particular feature we move on to the next feature and perform the same cycle - write test (see it fail), write code (watch the test pass), refactor (improve our code). So in short we can come up with the following steps of test driven development:

1. Write test.
2. See the test fail.
3. Write code.
4. See the test pass.
5. Refactor code.
6. Start again from step 1.

This cycle is commonly termed as the "red-green-refactor" cycle. The main motivating factor behind TDD is that when we are writing the test cases first, we are forced to think about what our code should do - this creates a clear vision on our mind. Moreover as we are concentrating on only one feature at a time, it is easier for us to employ our whole energy in solving that single problem.

2 MVC (Model View Controller) Design Pattern

At the core of MVC design pattern is separation of responsibility. As the name suggests, we are talking about the interaction among three components - the model, the view, and the controller.

The **model** deals with persistent storage, such as database, json file, xml file, etc. Doesn't matter where you have decided to store data, if you are planning on manipulating (create, read, update, delete - CRUD) that data, it should be done through the model.

The **view** deals with the front end, whatever the end users are seeing. So in general we can say the user interface is created by the view.

The **controller** is the think tank of our software. It maintains liaison between the view and the model. When should one view be invoked, when should a model be sent to fetch data, etc. - these kinds of shots are all called by the controller.

To make the concept a bit more concrete, let's consider a scenario. Suppose you want to login to mail.google.com. When you type in the url you are presented with a form to enter your username and password. Once you've done that your credentials are checked against the database to see if a user with the supplied credentials actually exists. If so you are logged in, otherwise not. Here are a few questions:

- Who created the outline (like how many fields should be there, how they should be arranged, etc.) of the form that you are shown during logging in? **View**.
- Who checked the supplied credentials by actually hitting the database? **Model**.
- Who planned out the whole thing that the application should grab the info supplied in the form and send it to the model for checking? **Controller**.

So that's pretty much it on the MVC pattern. Let us take a look at a few industry standard MVC frameworks next.

2.1 Framework Review

If you are planning on developing for the web you have quite a lot of options depending on your preferred language. If your preferred language is PHP you have the following options:

Codeigniter An extremely fast and a very lightweight framework, capable of doing all kinds of heavy duty job. The major problem is that it is also extremely outdated. It hasn't been updated in a long time and lacks a lot of features of modern PHP. It is a great framework, no way to deny that - and it will survive a few more years as a lot of projects are still running on it. But if anyone is starting fresh, it is better to choose something else.

Laravel Perhaps currently the most popular framework. Compared to other giants it is quite new. A very user friendly framework with a lot of cool features. The community is very strong. A lot of codeigniter users have switched to using laravel in the recent past (including me - I moved from codeigniter to laravel in early 2013). It is also a pretty fast framework - but not as fast as codeigniter.

Symfony, Yii Laravel took a lot of features from Symfony, if someone compares the codebase he/she is going to find that a lot of their codebase are quite similar. But somehow Laravel is way faster than Symfony.

PhalconPHP Currently the fastest PHP framework out there. It is implemented as a C extension directly in the PHP laguage, so surpasses any other framework in speed.

Zend, CakePHP In the past both of them have been in high demand. Problem with Zend is it is not that user-friendly. CakePHP is extremely slow. I wouldn't suggest anyone starting out with these two.

If your preferred language is Python:

Django A great framework able to perform every sort of work. If you are going with Python, just choose this.

Flask A lightweight framework for Python, if you are developing a website with simple features - go with flask.

We can't really ignore another language when it comes to web development - Ruby:

Ruby on Rails RoR for short - is the framework that actually revolutionized the MVC concept in regards to web development. Not a very big community, but a pretty strong one.

Sinatra A lightweight framework for Ruby, kind of like what Flask is to Python.

If you are thinking of developing desktop applications you simply can't ignore **Qt** and its variants. Qt is a great framework to develop desktop applications in C++. It has been around for quite a while and the community is pretty strong. If you are thinking of developing in Python you can go with **PyQt**. So pretty much whatever you decide to work with - chances are there are going to be more than one industry standard MVC framework for it. In case you can't find a framework, organize your code in such a way that you can differentiate your model, view, and controller.

3 Git

Git is what we call a version control system (VCS). It is exactly what it sounds like - it maintains various versions of our code base. Technically it doesn't have to be "codes", any text file can be tracked by git. Even binary files, say images, can be tracked - although this is not commonly seen. Git also allows us to collaborate with our team members in a feasible manner. In your term project you are going to work in groups, make sure that you use git and collaborate through a remote server, like github. Another aspect of git which many of us do not pay much attention to is git is actually a great way to advertise your expertise. It lets others know what you have been working on, and a great way to assess someone is through his/her work.

3.1 The Three Tier Architecture of Git

Git follows what is known as a three tier (level) architecture:

1. working directory,
2. staging index, and
3. repository.

When you first write something in your file, it is in the working directory. After that you can invoke the `git add` command to copy it to staging index. Finally a `git commit` puts all your staged work in the repository ("commits" your work to repository). So once in a while when you are satisfied with your progress, make sure you perform these two (add and commit) commands.

3.2 Installation

For unix users, again installation is super simple. For example if you are using Ubuntu the following command will install git:

```
$ sudo apt-get install git
```

If you are using Windows, go to *git-scm.com* to download the binaries.

After installation, be sure to set some configuration information, namely username and email address:

```
$ git config --global user.name <your_name>
$ git config --global user.email <your_email_address>
```

The best way to learn git is to get your hands dirty straight away, so let's do that.

3.3 Creating a Repository - git init

Suppose you want to create a repository inside a directory named *git-prac*. You can invoke the following commands to do so:

```
$ mkdir git-prac
$ cd git-prac
$ git init
```

You will see a message stating that an empty repository has been initialized. If you invoke `ls -a` command you will see that git actually created a hidden directory named *.git*. Chances are, you will never have to deal with this directory directly. Rather the various commands that you invoke will do that for you. Bear in mind that removing a repository simply comes down to removing this *.git* directory.

3.4 Man Page for Git Commands - git help ¡command¿

Anytime you need to know a bit more about any git command, simply invoke `git help <command_name>`.

3.5 Status Report - git status

`git status` shows the current status of the three tiers. **Read the status reports carefully** - you will get a bettern grip on git this way. Recall that we are still in an empty directory. Invoke the status command and see what the output is. Notice the report gives you a small hint that you can create/copy a file and invoke the add command. Let us create a file and see the status report again:

```
$ touch hello.txt
$ git status
```

The status report shows the file in red and says that the file is not being tracked. We will not be able to store various versions of this file unless we track it. It also gives us a hint as to what we can do to make it "trackable".

3.6 Copy Files to Staging Index - git add

Let us invoke `git add` command and see the status report again:

```
$ git add hello.txt
$ git status
```

Here we are invoking add with a specific file name, if we supply '.' it will simply add all untracked/modified files. The status report will let you know that changes can be committed now, i.e. moved to the repository. It also says something about a command named "rm".
Exercise: find out what *git rm* does by invoking `git help rm`. Specifically concentrate on the "cached" option.

3.7 Copy from Staging Index to Repository - git commit

`git commit -m "message"` commits all the changes in the staging index to our repository. Let us invoke the command and see the status report:

```
$ git commit -m "initial commit"
$ git status
```

After commit you are going to see a message containing a sequence of seven hexadecimal characters. This is the starting of a 40 character hash value that is created by passing the contents of your file, the current timestamp, etc. to the SHA1 algorithm (a hashing algorithm). Every commit in your repository can be uniquely identified through this 40 character value. Generally these values are so diversified that specifying only the first 5-7 characters is enough to uniquely identify a commit. Git has a special pointer named **HEAD** which simply points to the latest commit on the current branch (more on branching in subsection 3.11).
Exercise: Find out what `git commit -am <commit message>` does.

3.8 Review Commits - git log

`git log` allows us to see all the commits we've made so far. It shows the hash value and commit message of each of the commits made. Notice that it shows the commits in chronologically reverse order i.e. the latest commit is shown first.
Exercise: how can we see only the last 3 commits we've made?

3.9 Ignoring File - .gitignore

There are certain files we would not generally want to keep track of. For example, when we compile a latex file with pdflatex, it generates a log file, an aux file, and a pdf file. We would not really want to maintain versions of these files, as we can easily generate these from the tex file. So basically we would want to instruct git to ignore these files. This is where the *.gitignore* file comes in. In this case we would simply create a .gitignore file with the following lines in it:

```
*.log
*.aux
*.pdf
```

Notice the use of wild card matcher "*", which would make git ignore any files ending with the stated extensions.

3.10 Find Differences between Versions - git diff

To get a taste of this command let's create some changes to our *hello.txt* file:

```
$ echo "first line - commit 2" >> hello.txt
$ echo "second line - commit 2" >> hello.txt
$ git add hello.txt
$ git commit -m "add two lines to hello.txt"
```

By the first two commands we are simply appending some texts to our files. Let us make some more changes and commit again:

```
$ echo "third line - commit 3" >> hello.txt
$ echo "fourth line - commit 3" >> hello.txt
$ git add hello.txt
$ git commit -m "add second set of two lines to hello.txt"
```

Let us make a change and stage the change. This time we will not commit.

```
$ echo "this line has been staged only" >> hello.txt
$ git add hello.txt
```

Let us make a final change - this time we will not even stage our changes.

```
$ echo "this line is only in the working directory" >> hello.txt
```

Now do a `git log` and note down the first five characters of the hash value of each of the three commits.
Exercise: Execute the following commands and try to understand the messages in the terminal:

```
$ git diff <short-hash-of-commit1> <short-hash-of-commit2>
$ git diff <short-hash-of-commit2> <short-hash-of-commit1>
$ git diff <short-hash-of-commit1> <short-hash-of-commit3>
$ git diff <short-hash-of-commit3> <short-hash-of-commit1>
$ git diff <short-hash-of-commit2> <short-hash-of-commit3>
$ git diff <short-hash-of-commit3> <short-hash-of-commit2>
$ git diff
$ git diff --cached
$ git diff --staged
```

Is there any difference between the last two commands? What is the difference between `git diff` and the last two commands?

3.11 Branches in Git

Branches can be thought of as a particular line of development. Suppose you are developing a software and at a certain point you need to create a fix for a bug that came to your knowledge. But you are a bit worried that if you keep on working with the current files your present workflow might get hampered. In this case you can create a "branch" and work on that branch. Once you are satisfied with your bugfix, you can simply "merge" the changes in your two branches - i.e. combine them. The default branch you are working on from the beginning is known as the *master* branch. When you create a branch, all of your current commits sort of get copied to the new branch, and you can work on that branch separately. Be advised that whatever commits you make on one branch will not affect the other (unless you are performing a merge). Let us create a branch named bugfix and "checkout" (i.e. switch to) that branch.

```
$ git branch bugfix
$ git checkout bugfix
$ git branch
```

The first command creates a branch named bugfix, the second one switches to the new branch, and the last one simply shows a list of branches. The current branch is highlighted in green with an asterisk on the side. The `git checkout` command is actually quite versatile. Suppose you want to overwrite the current *hello.txt* file with the one in the second commit. You can do so by the following command.

```
$ git checkout <short-hash-of-commit2> -- hello.txt
```

Notice that this will also overwrite the file in staging index. The two dash preceeding the filename is considered a good practice, it denotes that we are indeed checking out a file and not a branch. In case our filename has the same name as that of a branch, git would become confused. When you are satisfied with your work in bugfix and feel that you are ready to merge the changes to the master branch, you can perform the following steps:

1. Commit all changes while you are in the bugfix branch. In general it is better to always commit changes while switching branches.

2. Switch to the master branch.

3. Invoke the command `git merge bugfix`.

Exercise: What does `git checkout -b <branchname>` do? How can you delete a branch? How can you compare two branches?

3.12 Working with Remotes - Intro to Github

There are a lot of remote servers that let us host our repositories and collaborate with others. Github is certainly the most popular one among them (BitBucket, GitLab, etc. are also pretty cool). Suppose we want to store our local repository that we've been working on to a remote server - in this case a github account. The first thing you need to do is create an account in *github.com* - it's free. The only downside is all the repositories will be public. But as long as you are working on an open source project or say your term project, it shouldn't hurt. If you are looking for private repos for free checkout BitBucket. After our account has been created, we need to create a new repository. You will be presented with a *url* after successful creation of the repo. Then you will have to invoke the following commands:

```
$ git remote add origin <url>
$ git push origin master
```

In the first line you are recording the url of your remote repository. You are also supplying an alias (i.e. a nickname) to your remote - origin. This is a traditional name given to the primary remote, but it could be anything you want. In the second line you are pushing (i.e. uploading) the master branch of your repository to your remote. At this point you will be prompted for your username and password, just type in your credentials and you will be good to go. Notice that the password will not be visible on the screen as you type it in. Now if you refresh your webpage you are going to see that the repo in your local drive has been uploaded to your remote.

3.13 Copying a Repo - git clone and Forking

Suppose you are loitering around github (or any such website) and you come across a really awesome repo. You would really love to work on this repo. You can invoke `git clone <url>`, this will create a copy of the repo on your local hard drive. There is something known as "forking". It simply means copying a repo on the server side. Basically if you press the fork button on a user's repo in github, that repo will be copied to your github account.

3.14 Collaboration through Github

It is really easy to collaborate with your group members through github. When you create a public repo, anyone can copy that repo and work on his/her copy. But no one can modify your copy - only you have that option. But to collaborate in a group you must grant some permissions to your groupmates. You can do this by adding "collaborators" for your repo. Go to the settings page of your repo

and find the "Collaborators" option. From there you can add your groupmates as the collaborators. They will be able to modify your repo, but will not be able to delete it.

While we are on the topic of collaboration, let's talk about `git pull`. Consider a scenario where two users, User1 and User2 are working on a project. User1 does some coding and uploads the repo to the remote server. Now User2 does not have anything on his local drive, so he performs a `git clone`. Suppose he starts working on the local copy. After some time he wants to push the changes to the remote. Should he straight away start pushing everything to the remote? Well, no; because while User2 was making some changes, perhaps User1 pushed some changes to the server too. So he first needs to update his local copy to reflect those changes, and he does so by the command `git pull <url>`. This command simply combines two other commands - `fetch` and `merge` - the contents are first fetched from the remote and then merged to the local copy. Now that User2 is confident that he has all the updates, he can push the changes by invoking `push`. Same thing applies for all other users.