

# 1 CLEAN CODE

---

Based on the many definitions, main Characteristics of Clean Code includes:

Elegant: Clean code is pleasing to read, should make you smile.

Readability: Clean code should read like well-written prose.

Simple: Do one thing with the Single Responsibility Principle (SRP).

Testable: Run all the tests.

## 1.1 MEANINGFUL NAMES

Everything in an application has a name, from variables, functions, arguments, classes, modules, to packages, source file, directories.

Names are the powerful way you communicate your code's intent to developers who read your code, including yourself. Choosing good names takes time but make your code better and cleaner. It makes your code easy to read by other developers, as well as yourself in the future.

Use intention-revealing Names

That means choosing names that reveal intent, the name should tell you why it exists, what it does, and how it is used. If a name requires a comment, then the name does not reveal its intent. It makes your code easier to understand and change later.

# bad

`t = Date.today` # the name `t` reveals nothing.

# good

`today = Date.today`

### 1.1.1 Use Pronounceable Names

Humans are good at words and words are, by definition, pronounceable.

If you can't pronounce it, you can't discuss it. This matter because programming is a social activity.

# bad

`ymdhms = Time.current` # date, year, month, day, hour, minute, second

# good

`today_timestamp = Time.current`

### 1.1.2 Class Names

Classes and objects should have noun or noun phrase names like Customer, WikiPage, Account, and AddressParser. Avoid words like Manager, Processor, Data, or Info in the name of a class.

A class name should not be a verb.

### 1.1.3 Method Names

Methods should have a verb or verb phrase names. Examples: createUser, deletePhoto or save

### 1.1.4 Pick One Word per Concept

One and the same concept in your application should have the same name. For example, it's confusing to have fetch, retrieve, and get as equivalent methods of different classes.

Using the same word per concept will help developers easier to understand the code.

Should NOT:

### 1.1.5 Encodings

Encoded names are seldom pronounceable and are easy to mistype.

### 1.1.6 Mental Mapping

Readers shouldn't have to mentally translate your names into other names they already know. This problem generally arises from a choice to use neither problem domain terms nor solution domain terms.

One difference between a smart programmer and a professional programmer is that the professional understands that clarity is king. Professionals use their powers for good and write code that others can understand.

### 1.1.7 Pun

Avoid using the same word for two purposes. Using the same term for two different ideas is essentially a pun.

### 1.1.8 Functions

How do you make a function communicate its intent? There are some best practices help you write good functions easy to read and change.

### 1.1.9 Small

The first rule of functions is that they should be small. The second rule of functions is that they should be smaller than that.

#### 1.1.10 Do One Thing

Functions should do one thing. They should do it well. They should do it only.

Follow the Single Responsibility Principle (SRP)

#### 1.1.11 Function Arguments

Functions should have  $< 3$  arguments.

The ideal number of arguments for a function is zero (niladic). Next comes one (monadic), followed closely by two (dyadic). Three arguments (triadic) should be avoided where possible. More than three (polyadic) requires very special justification — and then shouldn't be used anyway.

### 1.2 KISS

Keep It Simple Stupid. A design principle originating from the U.S. Navy that goes back to 1960 already. It states that most systems should be kept as simple as possible (but not simpler, as Einstein would have said). Unnecessary complexity should be avoided. The question to ask when you're writing code is "can this be written in a simpler way?".

### 1.3 YAGNI

You Aren't Gonna Need It. A developer should not add functionality unless deemed necessary. YAGNI is part of the Extreme Programming (XP) methodology, which wants to improve software quality and increase responsiveness to customer requirements. YAGNI should be used in conjunction with continuous refactoring, unit testing, and integration.

### 1.4 COMPOSITION OVER INHERITANCE

Composition is favored over inheritance by many developers, because inheritance forces you to build a taxonomy of objects early on in a project, making your code inflexible for changes later on.  
(Komposisi bermain dengan komponen dan inheritance bermain dengan penurunan fungsi atau class)

### 1.5 FAVOR READABILITY

It's not because a machine can read your code that another human can. Particularly when working with multiple people on a project, always favor readability over conciseness. There's no point in having concise code if people don't understand it.

### 1.6 DON'T REPEAT YOURSELF (DRY)

Duplication is a problem in software. Many principles and best practices have been created to reduce duplication code.

(Refactoring is must for generic code)

## 2 DRY

---

### 2.1 WHAT IS DRY PRINCIPLE?

In software engineering, don't repeat yourself (DRY) is a principle of software development, aimed at reducing repetition of information of all kinds, especially useful in multi-tier architectures.

### 2.2 WHAT IS THE BENEFIT OF DRY?

Less code is good: It saves time and effort, is easy to maintain, and also reduces the chances of bugs.

One good example of the DRY principle is the helper class in enterprise libraries, in which every piece of code is unique in the libraries and helper classes.

### 2.3 HOW TO ACHIEVE DRY?

To avoid violating the DRY principle, divide your system into pieces. Divide your code and logic into smaller reusable units and use that code by calling it where you want. Don't write lengthy methods, but divide logic and try to use the existing piece in your method.

### 2.4 THE PROGRAMMING PRINCIPLES

These are some important programming principles that will help us to achieve DRY and to make our code clean.

- Abstraction ([menangani kompleksitas dengan menyembunyikan detail detail](#))

Abstraction is one of the key concepts of object-oriented programming (OOP) languages. Its main goal is to handle complexity by hiding unnecessary details from the user. That enables the user to implement more complex logic on top of the provided abstraction without understanding or even thinking about all the hidden complexity. ([membuat sesuatu yang general dan bisa diimplementasi ke kelas kelas lainnya](#))

- Rule of Three

Simply put: If you have identical code in 3 or more places, it should be abstracted into a common method, class, etc. to be reusable and easily maintainable.

- K.I.S.S. (Keep it simple stupid!)

Don't cram too much into one little method or function. "Clean design leads to clean code". Look at this code below for example. The 2 methods are doing the same, so you must choose which one to remove:

```
public class Example {
    public String weekday1(int day) throws Exception {
        switch (day) {
            case 1:
                return "Monday";
        }
    }
}
```

```

        case 2:
            return "Tuesday";
        case 3:
            return "Wednesday";
        case 4:
            return "Thursday";
        case 5:
            return "Friday";
        case 6:
            return "Saturday";
        case 7:
            return "Sunday";
        default:
            throw new Exception("day must be in range 1 to 7");
    }
}

public String weekday2(int day) throws Exception {
    if ((day < 1) || (day > 7))
        throw new Exception("day must be in range 1 to 7");
    String[] days = { "Monday", "Tuesday", "Wednesday", "Thursday", "Friday", "Saturday", "Sunday" };
    return days[day - 1];
}

public static void main(String[] args) {
    System.out.println("Hello");
}
}

```

- Separation of Concern (tanggung jawabnya atau penugasannya dipisah-pisah)

Separation of concerns is a design principle for separating a computer program into distinct sections, such that each section addresses a separate concern. For example, the business logic of the application is a concern and the user interface are another concern. Changing the user interface should not require changes to business logic and vice versa.

These will Simplify development and maintenance of software applications and when concerns are well-separated, individual sections can be reused, as well as developed and updated independently.

To achieve it you just break program functionality into separate modules that overlap as little as possible.

- Single Responsibility (modulasi-modulasi fungsi sesuai dengan responsibility dan field nya)

Every class should have a single responsibility, and that responsibility should be entirely encapsulated by the class. Responsibility can be defined as a reason to change, so a class or module should have one, and only one, reason to change. Look at this class below.

```

class Employee {
    public double calculatePay() {
        // calculation logic
    }

    public void save() {
        // save to database
    }

    public String reportHours() {
        // string that report hour
    }
}

```

This class has three reason to change:

1. The calculating pays business rules
2. The database schema
3. The string that report hour.

We don't want a single class to be impacted by these three completely different forces. We don't want to modify the Employee class every time the accounts decide to change the format of the hourly report, or every time the DBAs make a change to the database schema, as well as every time the managers change the payroll calculation. Rather, we want to separate these functions out into different classes so that they can change independently of each other.

#### - You Aren't Going to Need It (YAGNI)

Any work that's only used for a feature that's needed tomorrow, means losing effort from features that need to be done for the current iteration. It leads to code bloat; the software becomes larger and more complicated.

Always implement things when you actually need them, never when you just foresee that you need them.

These DRY and all programming principle seems simple, but these are “the principles of a good programming”, so **you must learn how to implement them in your codes.** (OK!)

### 3 TDD

---

Test-driven development (TDD) (Beck 2003; Astels 2003), is an evolutionary approach to development which combines test-first development where you write a test before you write just enough production code to fulfill that test and refactoring. What is the primary goal of TDD? One view is the goal of TDD is specification and not validation (Martin, Newkirk, and Kess 2003). In other words, it's one way to think through your requirements or design before you write your functional code (implying that TDD is both an important agile requirements and agile design technique). Another view is that TDD is a programming technique. As Ron Jeffries likes to say, the goal of TDD is to write clean code that works. I think that there is merit in both arguments, although I lean towards the specification view, but I leave it for you to decide.

A significant advantage of TDD is that it enables you to take small steps when writing software. This is a practice that I have promoted for years because it is far more productive than attempting to code in large steps. For example, assume you add some new functional code, compile, and test it. Chances are pretty good that your tests will be broken by defects that exist in the new code. It is much easier to find, and then fix, those defects if you've written two new lines of code than two thousand. The implication is that the faster your compiler and regression test suite, the more attractive it is to proceed in smaller and smaller steps. I generally prefer to add a few new lines of functional code, typically less than ten, before I recompile and rerun my tests.

**So, we can summarize that test-driven development (TDD) is a development technique where you must first write a test that fails before you write new functional code. TDD does not replace traditional testing, instead it defines a proven way to ensure effective unit testing. A side effect of TDD is that the resulting tests are working examples for invoking the code, thereby providing a working specification for the code. My experience is that TDD works incredibly well in practice and it is something that all software developers should consider adopting. (membangun worst test case scenario sebelum membangun functional code)**

## 4 GIT

---

Git is a distributed versions control system, that log every working history of a repository (storage).

### 4.1 TO USE GIT, WE DO THESE STEPS:

Go to this page <https://git-scm.com/downloads> and choose your operating system. After that follow the steps provided.

After we successfully install git, now open command prompt and type:

```
$ git --version
```

You'll see the git version if it's installed correctly

After that now we need to configure git username and email. On terminal run:

```
git config --global user.name "full_name"  
git config --global user.email "email_address"
```

To check configuration, on terminal run:

```
git config --list
```

That's it we can use git now.

\* All global git configuration will be saved in **~/.gitconfig** file.

### 4.2 LOCAL IMPLEMENTATION

after successfully install and configuring git now we will be using git locally, let's create an empty folder in home directory:

```
$ mkdir ~/learn_git_local
```

Now change directory to that directory:

```
$ cd learn_git_local
```

After that, now we need to initialize 'learn\_git\_local' as our local repository, on terminal run:

```
$ git init
```



Output:

```
Initialized empty Git repository in /home/username/learn_git_local/.git/
```

then check current state repository, run:

```
$ git status
```

Output:

```
On branch master
Initial commit
nothing to commit (create/copy files and use "git add" to track)
```

it means repository is in branch '*master*', and nothing to be commit (save our working to local repository), now add a new file:

```
$ touch learn_git.txt
```

then check current state repository, run:

```
$ git status
```

Output:

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

That means git detect new file that need to be staged, git will give suggestion for what action that we going to do, from that status git suggest for add new file learn\_git.txt, run:

```
$ git add learn_git.txt
```

Then if we check status repository, run:

```
$ git status
```

That will result:

```
On branch master
Initial commit
Changes to be committed:
(use "git rm --cached <file>..." to unstage)
new file: learn_git.txt
```

That means file `learn_git.txt` has been added to index of working tree and prepare for next commit, so every change from this file will be logged. Now try add content using any text editor to `learn_git.txt`, run:

```
$ gedit learn_git.txt
```

Add this text:

```
Learn git  
in local repository
```

Save the file and close the gedit. Now if we check:

```
$ git status
```

That will result:

```
On branch master  
Initial commit  
Changes to be committed:  
(use "git rm --cached <file>..." to unstage)  
new file: learn_git.txt  
Changes not staged for commit:  
(use "git add <file>..." to update what will be committed)  
(use "git checkout -- <file>..." to discard changes in working directory)  
modified: learn_git.txt
```

That means file `learn_git.txt` has changes, if we notice git will give suggestion for git add or git checkout, if we use git checkout `learn_git.txt`, that will remove all changes back to first add that file or when the last commit. To see what changes has been added to the file use:

```
$ git diff learn_git.txt
```

Output:

```
diff --git a/learn_git.txt b/learn_git.txt  
index e69de29..e16794a 100644  
--- a/learn_git.txt  
+++ b/learn_git.txt  
@@ -0,0 +1,2 @@  
+Learn git  
+in local repository
```

If we see `@@ -0,0 +1,2 @@`, it means in the file has add from line 1 until line 2. And `+Learn git +in local repository`, that means the file has an added text "Learn git in local repository". After that use git add, run:

```
$ git add learn_git.txt
```

Then if we check:

```
$ git status
```

Output:

```
On branch master
Initial commit
Changes to be committed:
  (use "git rm --cached <file>..." to unstage)
    new file:   learn_git.txt
```

now we need commit our changes, run:

```
$ git commit learn_git.txt -m "First Commit"
```

That command means that we commit our changes to the local repository with a message "First Commit". Now if we check:

```
$ git status
```

Output:

```
On branch master
nothing to commit, working directory clean
```

It means in branch master nothing changes. Now we want see logs of commit for the last created:

```
$ git log -1
```

Output:

```
commit 785186a77cecace02b9cd413444dd6084972b6e0
Author: jack ranter <jack@kiranatama.com>
Date: Thu Feb 18 14:13:36 2017 +0700
```

After successfully commit for the first time, let's say we want work on new branch so the base project will not change. Let's create new branch using:

```
$ git branch development_1
```

*\* if we want delete branch use: git branch -d development\_1*

Check the all branch that exist for repository, run:

```
$ git branch
```

Output:

```
development_1
* master
```

the mark '\*' means the repository is in branch master, if we want move to branch 'development\_1', run:

```
$ git checkout development_1
```

Output:

```
Switched to branch 'development_1'
```

Now if we check the current branch:

```
$ git branch
```

Output:

```
* development_1
master
```

That means the repository has move to branch 'development\_1'.

*\* git checkout, can be used to remove changes in working file or switching between branches.*

## 4.3 UNDOING CHANGES

### 4.3.1 git checkout

The git checkout command serves three distinct functions: checking out files, checking out commits, and checking out branches. In this module, we're only concerned with the first two configurations.

Checking out a commit makes the entire working directory match that commit. This can be used to view an old state of your project without altering your current state in any way. Checking out a file lets you see an old version of that particular file, leaving the rest of your working directory untouched.

#### 4.3.1.1 Usage

```
git checkout master
```

Return to the master branch. Branches are covered in depth in the next module, but for now, you can just think of this as a way to get back to the "current" state of the project.

```
git checkout <commit> <file>
```

Check out a previous version of a file. This turns the <file> that resides in the working directory into an exact copy of the one from <commit> and adds it to the staging area.

```
git checkout <commit>
```

Update all files in the working directory to match the specified commit. You can use either a commit hash or a tag as the <commit> argument. This will put you in a detached HEAD state.

#### 4.3.1.2 Example

##### Viewing an Old Revision

This example assumes that you've started developing a crazy experiment, but you're not sure if you want to keep it or not. To help you decide, you want to take a look at the state of the project before you started your experiment. First, you'll need to find the ID of the revision you want to see.

```
git log --oneline
```

Let's say your project history looks something like the following:

```
b7119f2 Continue doing crazy things
872fa7e Try something crazy
a1e8fb5 Make some important changes to hello.py
435b61d Create hello.py
9773e52 Initial import
```

You can use git checkout to view the "Make some import changes to hello.py" commit as follows:

```
git checkout a1e8fb5
```

This makes your working directory match the exact state of the a1e8fb5 commit. You can look at files, compile the project, run tests, and even edit files without worrying about losing the current state of the project. *Nothing* you do in here will be saved in your repository. To continue developing, you need to get back to the "current" state of your project:

```
git checkout master
```

This assumes that you're developing on the default master branch, which will be thoroughly discussed in the Branches Module.

Once you're back in the master branch, you can use either git revert or git reset to undo any undesired changes.

##### Checking Out a File

If you're only interested in a single file, you can also use git checkout to fetch an old version of it. For example, if you only wanted to see the hello.py file from the old commit, you could use the following command:

```
git checkout a1e8fb5 hello.py
```

Remember, unlike checking out a commit, this *does* affect the current state of your project. The old file revision will show up as a "Change to be committed," giving you the opportunity to revert back to the previous version of the file. If you decide you don't want to keep the old version, you can check out the most recent version with the following:

```
git checkout HEAD hello.py
```

#### 4.3.2 git revert

The git revert command undoes a committed snapshot. But, instead of removing the commit from the project history, it figures out how to undo the changes introduced by the commit and appends a *new* commit with the resulting content. This prevents Git from losing history, which is important for the integrity of your revision history and for reliable collaboration.

#### 4.3.2.1 Usage

```
git revert <commit>
```

Generate a new commit that undoes all of the changes introduced in <commit>, then apply it to the current branch.

#### 4.3.2.2 Example

The following example is a simple demonstration of git revert. It commits a snapshot, then immediately undoes it with a revert.

```
# Edit some tracked files
# Commit a snapshot
git commit -m "Make some changes that will be undone"
# Revert the commit we just created
git revert HEAD
```

#### 4.3.3 git reset

If git revert is a “safe” way to undo changes, you can think of git reset as the *dangerous* method. When you undo with git reset (and the commits are no longer referenced by any ref or the reflog), there is no way to retrieve the original copy—it is a *permanent* undo. Care must be taken when using this tool, as it’s one of the only Git commands that has the potential to lose your work.

Like [git checkout](#), git reset is a versatile command with many configurations. It can be used to remove committed snapshots, although it’s more often used to undo changes in the staging area and the working directory. In either case, it should only be used to undo *local* changes—you should never reset snapshots that have been shared with other developers.

##### 4.3.3.1 Usage

```
git reset <file>
```

Remove the specified file from the staging area, but leave the working directory unchanged. This unstages a file without overwriting any changes.

```
git reset
```

Reset the staging area to match the most recent commit, but leave the working directory unchanged. This unstages *all* files without overwriting any changes, giving you the opportunity to re-build the staged snapshot from scratch.

```
git reset --hard
```

Reset the staging area and the working directory to match the most recent commit. In addition to unstaging changes, the --hard flag tells Git to overwrite all changes in the working directory, too. Put another way: this *obliterates* all uncommitted changes, so make sure you really want to throw away your local developments before using it.

```
git reset <commit>
```

Move the current branch tip backward to <commit>, reset the staging area to match, but leave the working directory alone. All changes made since <commit> will reside in the working directory, which lets you re-commit the project history using cleaner, more atomic snapshots.

```
git reset --hard <commit>
```

Move the current branch tip backward to <commit> and reset both the staging area and the working directory to match. This obliterates not only the uncommitted changes, but all commits after <commit>, as well.

#### 4.3.3.2 Examples

##### Unstaging a File

The git reset command is frequently encountered while preparing the staged snapshot. The next example assumes you have two files called hello.py and main.py that you've already added to the repository.

```
# Edit both hello.py and main.py
# Stage everything in the current directory
git add .
# Realize that the changes in hello.py and main.py
# should be committed in different snapshots
# Unstage main.py
git reset main.py
# Commit only hello.py
git commit -m "Make some changes to hello.py"
# Commit main.py in a separate snapshot
git add main.py
git commit -m "Edit main.py"
```

As you can see, git reset helps you keep your commits highly-focused by letting you unstage changes that aren't related to the next commit.

##### Removing Local Commits

The next example shows a more advanced use case. It demonstrates what happens when you've been working on a new experiment for a while, but decide to completely throw it away after committing a few snapshots.

```
# Create a new file called `foo.py` and add some code to it
# Commit it to the project history
git add foo.py
git commit -m "Start developing a crazy feature"
# Edit `foo.py` again and change some other tracked files, too
# Commit another snapshot
git commit -a -m "Continue my crazy feature"
# Decide to scrap the feature and remove the associated commits
git reset --hard HEAD~2
```

The `git reset HEAD~2` command moves the current branch backward by two commits, effectively removing the two snapshots we just created from the project history. Remember that this kind of reset should only be used on *unpublished* commits. Never perform the above operation if you've already pushed your commits to a shared repository.

#### 4.3.4 `git clean`

The `git clean` command removes untracked files from your working directory. This is really more of a convenience command, since it's trivial to see which files are untracked with `git status` and remove them manually. Like an ordinary `rm` command, `git clean` is *not* undoable, so make sure you really want to delete the untracked files before you run it.

The `git clean` command is often executed in conjunction with `git reset --hard`. Remember that resetting only affects tracked files, so a separate command is required for cleaning up untracked ones. Combined, these two commands let you return the working directory to the exact state of a particular commit.

##### 4.3.4.1 *Usage*

```
git clean -n
```

Perform a “dry run” of `git clean`. This will show you which files are going to be removed without actually doing it.

```
git clean -f
```

Remove untracked files from the current directory. The `-f` (force) flag is required unless the `clean.requireForce` configuration option is set to false (it's true by default). This will *not* remove untracked folders or files specified by `.gitignore`.

```
git clean -f <path>
```

Remove untracked files, but limit the operation to the specified path.

```
git clean -df
```

Remove untracked files *and* untracked directories from the current directory.

```
git clean -xf
```

Remove untracked files from the current directory as well as any files that Git usually ignores.

##### 4.3.4.2 *Example*

The following example obliterates all changes in the working directory, including new files that have been added. It assumes you've already committed a few snapshots and are experimenting with some new developments.

```
# Edit some existing files
# Add some new files
# Realize you have no idea what you're doing
# Undo changes in tracked files
git reset --hard
# Remove untracked files
git clean -df
```



After running this reset/clean sequence, the working directory and the staging area will look exactly like the most recent commit, and git status will report a clean working directory. You're now ready to begin again.

Note that, unlike the second example in git reset, the new files were not added to the repository. As a result, they could not be affected by git reset --hard, and git clean was required to delete them.

## 4.4 REMOTE SERVER IMPLEMENTATION

There many remote servers for git in market like github.com, bitbucket.org. The difference between those is github cannot create private repository for free account, that means github focus on open source project, whereas bitbucket allow create private repository for free account, that means bitbucket it more focus for enterprise.

### 4.4.1 Generate SSH key

SSH is a cryptographic network protocol for secure data communication, remote command-line login, remote command execution, and other secure network services between two networked computers.

To create new ssh key in ubuntu, on terminal run:

```
$ ssh-keygen -t rsa -C <wer@email.address>
```

For example:

```
$ cd ~  
$ ssh-keygen -t rsa -C "someone@gmail.com"
```

After we run that command, we will be asked for file destination, and passphrase.

For destination leave the default value, where it will be put in ~/.ssh folder

passphrase leave it empty.

After finish, it will generate:

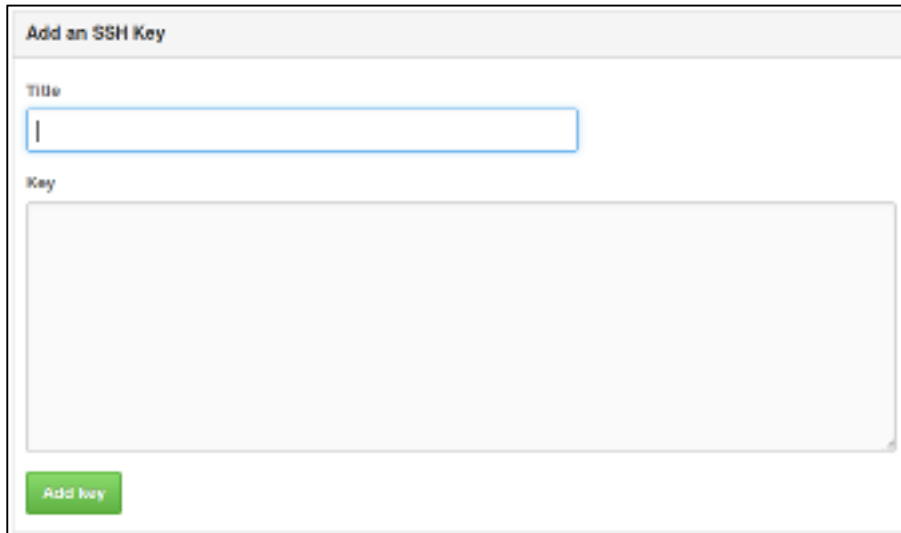
```
- .ssh/  
-- id_rsa  
-- id_rsa.pub  
-- known_hosts
```

'id\_rsa.pub' file contain public key, that we will use for communication between remote server and local.

### 4.4.2 Create Github account

Here we will use github as remote repository, first create account in: <https://github.com/join>

After success creating new account, now we need to add SSH Key to github, so communication between local repository and remote server repository in github can be established, go to: <https://github.com/settings/ssh>, click button “Add SSH key”, that will show input form:



- In input Title, insert email address.
- In input Key, insert public key from '.ssh/id\_rsa.pub'. For example, on terminal run:

```
$ cat ~/.ssh/id_rsa.pub
```

Output:

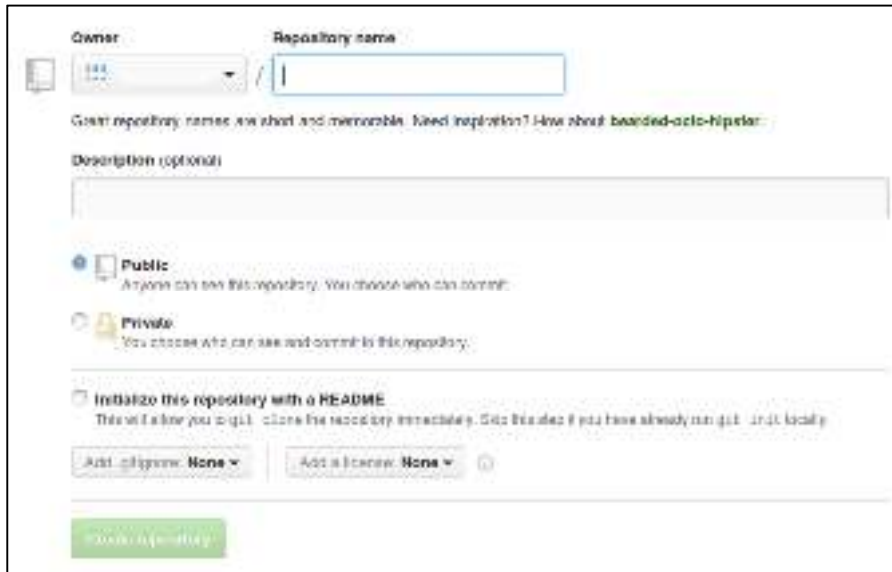
```
ssh-rsa
AAAAB3NzaC1yc2EAAAADAQABAAQCAjEjB6m6voulrnsAgigBGYNZIVkQYylxImIHEImoX2cCk1
kDWp5VVa/rtnbFW6nD8bA58q4/DkGyLZYmwj+Md1IYBE1FtBffbOSD3DsT2l1xbuNyLBfNzlfkc/MakEFb
+HjfW4p0AP7c5gj1CZFaFLDab/tUvssoaT56hRaH2PMjGXZjnJMhv18lapatVy3AyD0eNam7kyNXyhMLzS
1
cetBPSdASOGv/FvpxiChjX+wG/BES4TN4PDX6nv55Rx5A3Lv02dgp3V3Or6E3WRJv/pnZ8Hs6WHmcA8D
Zh3iJaNTPM+xc/hHpAORL8BVNbbmIbVqhaIDg2fmllfKobCh jack@gmail.com
```

Copy that public key to insert field, so it will look like:



Click button “Add key”.

Now create new repository in: <https://github.com/new>



Fill Repository Name the same as Local Repository name '*learn\_git\_local*', choose public so everyone can see. After that click button '*Create repository*', and then we will be redirected to:



The description above is how to prepare local repository, so we can send all commit in local into remote server github. This is example to setup in local, open console:

```
$ cd learn_git_local
```

After that run:

```
$ git remote add origin git@github.com:jack/learn_git_local.git
```

*\*If we want to modify remote repository use `git remote set-url origin <link_to_remote>`  
example: `git remote set-url origin https://jack@github.com/jack/learn_git_local.git`*

That git command is for setup remote repository to github, if we want see detail can be found in file '`learn_git_local/.git/config`'. 'Origin' is default name repository github.

#### 4.4.3 Implement Github

After successfully configuring, now we need to push (send local repository to github) the files. On terminal, and make sure we're in '`learn_git_local`' directory, run:

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

If we setup passphrase when generate ssh key, it will be asking passphrase.

If the process finished successfully, it will show message:

```
Counting objects: 5, done.  
Writing objects: 100% (3/3), 303 bytes | 0 bytes/s, done.  
Total 3 (delta 0), reused 0 (delta 0)  
To git@github.com:jack/learn_git_local.git  
785186a..29e0386 master -> master
```

now if we access [https://github.com/user\\_name/learn\\_git\\_local/commits/master](https://github.com/user_name/learn_git_local/commits/master), we can see lists of committed files in our local that logs date, author, then the detail changes.

Next if we want to push to github again use:

```
$ git push
```

*\*if 'git push' not work use this: `'git push origin master'`*

That is because we use '-u' option the first time we pushed to github (this command allow us to not include origin and master).

After successfully push to github, if other person modified our work and push it to github, that means our local work need to be updated with changes from the remote repository, inside '`learn_git_local`' run:

```
$ git pull origin master
```

The 'git pull' will successfully run when our local repository is the same as the remote one, if not git will give suggestion what we must do before pull. Make sure every commit that has push to remote server is never rollback, because it will result bad behavior, so make sure our work tested or create new branch for backup before pushing to remote server.

Make sure we're not modifying the same file with other members, because it will cause conflict (redundancy in repository that will cause confusion to git engine to choose which one the valid changes), to prevent this we must pull first before push to git server, but if conflict still happens we must manually modify the changes and choose the updated or both changes that we will commit.

## 4.5 GITIGNORE

Gitignore is a file where we list of all file and folder that won't be committed, this file resides in root project folder or root repository folder. This is an example where to put the file gitignore:

```
-- latihan /  
---- app/  
---- db/  
---- config/  
---- .git/  
---- .gitignore  
---- tmp/  
---- public/  
---- log/
```

for the case above we won't commit all the folders listed. Now let's try to add some changes to .gitignore, and then add:

```
#ignore this file when git commit  
.version  
tmp/  
public/  
log/  
config/
```

Save that file and when we check with git status command, the file or folder that we added in gitignore will not be detected as modification.

For complete resources: <http://git-scm.com/docs>

## 4.6 CHANGING A REMOTE'S URL

The git remote set-url command changes an existing remote repository URL

The git remote set-url command takes two arguments:

- An existing remote name. For example, origin or upstream are two common choices.
- A new URL for the remote. For example:
  - If you're updating to use HTTPS, your URL might look like:

```
https://[hostname]/USERNAME/REPOSITORY.git
```

- If you're updating to use SSH, your URL might look like:

```
git@hostname:USERNAME/REPOSITORY.git
```

#### 4.6.1 Example

List your existing remotes in order to get the name of the remote you want to change.

```
$ git remote -v
origin git@hostname:USERNAME/REPOSITORY.git (fetch)
origin git@hostname:USERNAME/REPOSITORY.git (push)
```

Change your remote's URL from SSH to HTTPS with the `git remote set-url` command.

```
$git remote set-url origin https://hostname/USERNAME/REPOSITORY.git
```

Verify that the remote URL has changed.

```
$ git remote -v
# Verify new remote URL
origin https://hostname/USERNAME/REPOSITORY.git (fetch)
origin https://hostname/USERNAME/REPOSITORY.git (push)
```