# Git for Robotics

Arjun Gandhi

March 2020

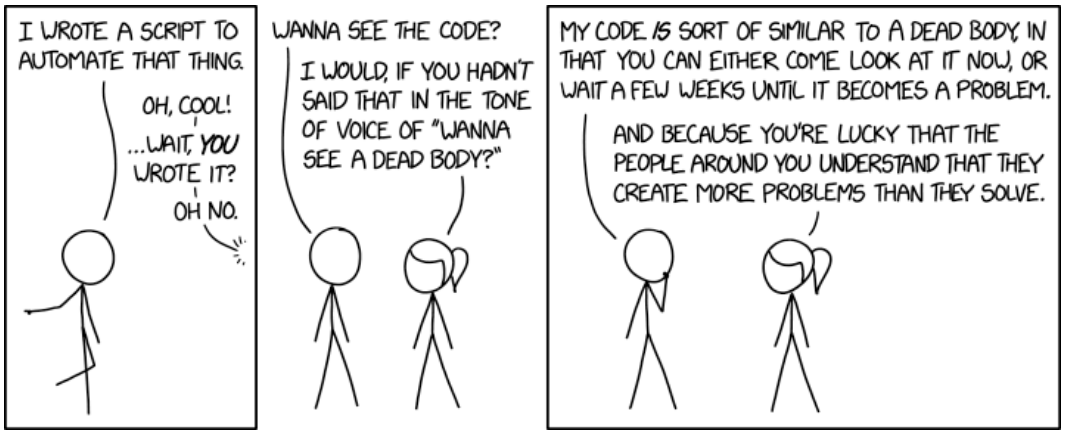



Figure 1: This is how all of my robotics project go.

# Contents


**1 What is version control? 4**
- 1.1 Why use version control . . . 4
  - 1.1.1 Storing Versions (Properly) . . . 5
  - 1.1.2 Restoring Old Versions . . . 5
  - 1.1.3 Understanding What Changes Were Made . . . 6
  - 1.1.4 Backup . . . 6
  - 1.1.5 What is Git? . . . 6

**2 Setting Up 6**
- 2.1 Setting Up Git on Your Computer . . . 7
  - 2.1.1 Installing Git on Windows . . . 7
  - 2.1.2 Installing Git on MacOS . . . 7
  - 2.1.3 Installing Git on Linux . . . 8
- 2.2 Basic Bash Commands . . . 9
- 2.3 Setting up GitHub . . . 9
  - 2.3.1 Configuring Git . . . 9

**3 Basic Workflow 9**
- 3.1 Creating a local git repository . . . 11
  - 3.1.1 Ignoring Files . . . 12
  - 3.1.2 Making Your First Commit . . . 13

**4 Starting with an Existing Project on a Server 13**
- 4.1 Creating a repository on GitHub . . . 13

**5 Working on your Project 16**
- 5.1 The Staging Area . . . 16
- 5.2 Getting an Overview of Your Changes . . . 17
- 5.3 Getting Ready to Commit . . . 18
- 5.4 Committing Your Work . . . 19
- 5.5 Sending your Changes to the Cloud . . . 20
- 5.6 Inspecting the Commit History . . . 20
- 5.7 Time to Celebrate . . . 23

**6 Branching Can Change Your Life 23**
- 6.1 Working in Contexts . . . 23
- 6.2 A World Without Branches . . . 24
- 6.3 Branches to the Rescue . . . 25

**7 Working with branches 25**

**8 Saving Changes Temporarily 27**

**9 Checking Out a Local Branch 28**

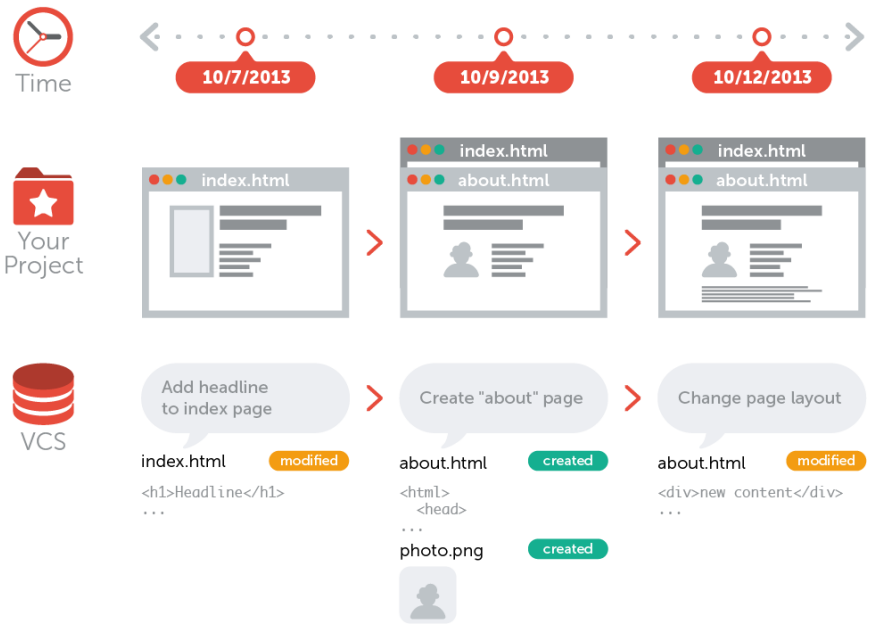
**10 Merging Changes** **30**
10.1 Dealing with Merge Conflicts . . . . . 32
10.1.1 You Cannot Break Things . . . . . 32
10.1.2 How a Merge Conflict Occurs . . . . . 32
10.1.3 How to Solve a Merge Conflict . . . . . 33
10.1.4 How to Undo a Merge . . . . . 34

**11 Working with remote branches** **34**
11.1 Local / Remote Workflow . . . . . 35
11.2 Integrating Remote Changes . . . . . 35

**12 Fun Things, Foot Notes and Personal Favorites** **36**
12.1 EGit(Eclipse) . . . . . 36
12.2 GitHub Desktop . . . . . 36
12.3 GitKraken . . . . . 37
12.4 Learn More About Git . . . . . 37
12.5 GitHub Student Developer Pack . . . . . 37
12.6 Sign Off & Acknowledgments . . . . . 37

# 1 What is version control?

You can think of a version control system (short: "VCS") as a kind of "database". It lets you save a snapshot of your complete project at any time you want. When you later take a look at an older snapshot (let's start calling it "version"), your VCS shows you exactly how it differed from the previous one.




Version control is independent of the kind of project / technology / framework you're working with:

- It works for a website as well as it does for a robotics project
- It lets you work with any tool you like; it doesn't care what kind of text editor, graphics program, file manager or other tool you use

At its core, a VCS records the changes you make to your project's files. This is what version control is about. It's really as simple as it sounds.

## 1.1 Why use version control

Without a VCS you are probably working together in a shared folder on the same set of files. Texting your teammates that you are currently working on file "xyz" and that, meanwhile, your teammates should keep their fingers off. This is an awful idea. It's extremely error-prone as you're essentially doing open-heart surgery all the time: sooner or later, someone will overwrite someone else's changes.

With a VCS, everybody on the team is able to work absolutely freely - on any file at any time. The VCS will later allow you to merge all the changes into a common version. There's no question where the latest version of a file or the whole project is. It's in a common, central place: your version control system.

Other benefits of using a VCS are even independent of working in a team or on your own.

### 1.1.1 Storing Versions (Properly)

Making a save of your project after you make critical changes is an important and necessary habit. It prevents you from losing your new work and allows you to go back to your old work easily in case you need it. Without a VCS your save system might look like the following awful ideas (I have personally seen someone do every one of these).

- Saving the changes as a new file and appending a number/date to the end.
- Making a copy of the entire code and pasting it (commented out) below the actual code for each version of the code. (This was done by an old rho beta officer)
- Copying the code and sending it in an email. (This was done by a rho beta vice president)
- Saving pictures of the code and putting them in Dropbox.
- Sending code to each other via Facebook Messenger.
- Use InstructAssist as a VCS. (This was done by a rho beta officer)
- Saving code on a flashdrive. (This was done by an old RBE1001 SA)

The core of this however is that the problem gets out of hand fast.

A VCS solves this problem. A version control system acknowledges that there is only one project. Therefore, there's only the one version on your disk that you're currently working on. Everything else - all the past versions and variants - are neatly packed up inside the VCS. When you need it, you can request any version at any time and you'll have a snapshot of the complete project right at hand.

### 1.1.2 Restoring Old Versions

Being able to restore older versions of a file (or even the whole project) effectively means one thing: you can't mess up! If the changes you've made lately prove to be garbage, you can simply undo them in a few clicks. Knowing this should make you a lot more relaxed when working on important bits of a project.

### 1.1.3 Understanding What Changes Were Made

When working in large teams understanding what changes other members have done quickly is crucial to efficient working.

Every time you save a new version of your project, your VCS requires you to provide a short description of what was changed. Additionally (if it's a code / text file), you can see what exactly was changed in the file's content. This helps you understand how your project evolved between versions.

### 1.1.4 Backup

It's happened to all of us before: you accidental deleted a critical file in a moment of panic. Luckily a side-effect of using a distributed VCS like Git is that it can act as a backup; every team member has a full-blown version of the project on their disk - including the project's complete history. Should your beloved central server break down (and your backup drives fail), all you need for recovery is one of your teammates' local Git repository.

### 1.1.5 What is Git?

By far, the most widely used modern version control system in the world today is Git. Git is a mature, actively maintained open source project originally developed in 2005 by Linus Torvalds, the famous creator of the Linux operating system kernel. A staggering number of software projects rely on Git for version control, including commercial projects as well as open source. Developers who have worked with Git are well represented in the pool of available software development talent and it works well on a wide range of operating systems and IDEs (Integrated Development Environments).

# 2 Setting Up

There are two main ways of working with Git: either via its Command Line Interface(CLI) or with a Graphical User Interface(GUI) application. Neither of these are right or wrong.

On the one hand, using a GUI application is likely easier at first and can help new users with the basic features.

On the other hand, however, I recommend learning the basics of Git on the command line first. It helps you form a deeper understanding of the underlying concepts and makes you independent of any specific GUI application.

In this guide we will be using a CLI not only because of the foundations it provides but also as it is what you will be using in the later robotics classes.

## 2.1 Setting Up Git on Your Computer

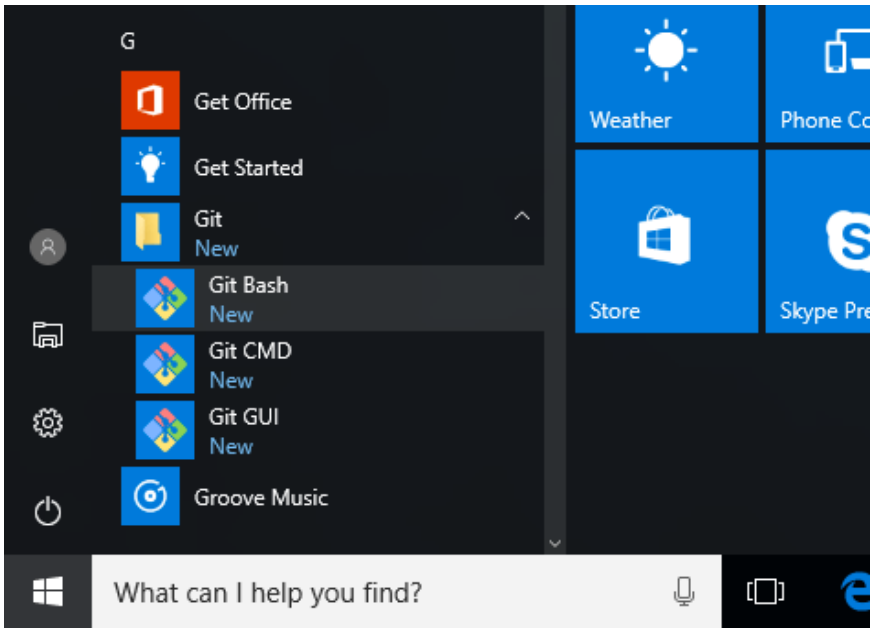
Installing Git has become incredibly easy in recent times. There are one-click installers for both Mac and Windows.

In this tutorial, like in many others, the "$" sign represents the prompt of the command line interface (you don't have to type this character in your commands!). Therefore, any time you see a line starting with the "$" sign, it means we're executing commands in "Terminal" or "Git Bash".

### 2.1.1 Installing Git on Windows

On Windows, you can download the "Git for Windows" package from here: https://git-for-windows.github.io/ Allow popup: yes

When running the installer EXE, you should choose the default options in each screen. After finishing the installation, you can begin working with Git by starting the "Git Bash" application. You'll find it in the Windows START menu, inside the "Git" folder:



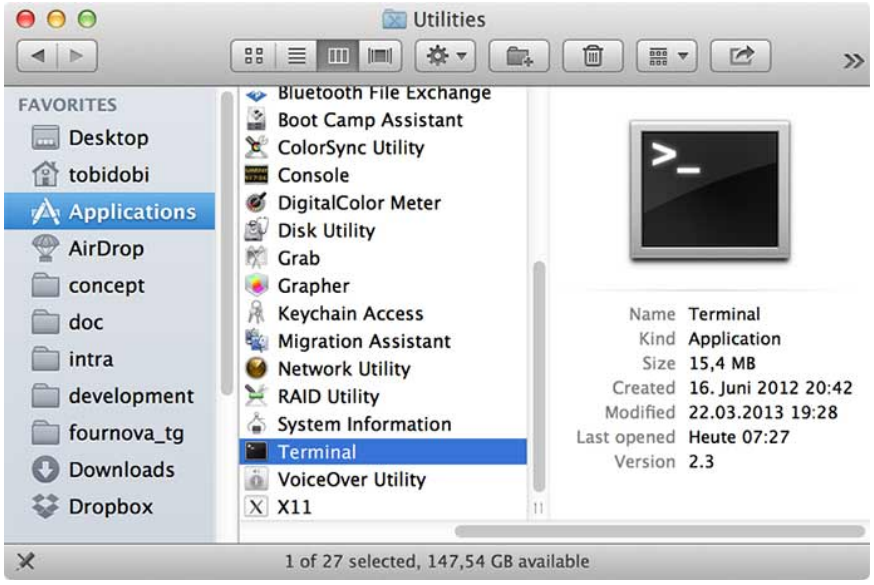

### 2.1.2 Installing Git on MacOS

Download and run the installer from here. Follow the prompts to install Git.

**Note** *Note: if you are already using Homebrew, feel free to use that instead with the following command:*

```
$ brew install git
```

Then open up the terminal you can do this by starting "Terminal.app" on your Mac. You'll find this in the "Utilities" subfolder of your "Applications" folder in Finder:




### 2.1.3 Installing Git on Linux

Use your the appropriate package manager for your linux system to install git:

Some of the common systems are listed below.

Open terminal and run the following commands:
**Ubuntu/Debian**

```
$ sudo apt update
$ sudo apt install git
```

**Redhat Based System**

```
$ sudo yum install git
```

**Arch Linux**

```
$ sudo pacman -S git
```

## 2.2 Basic Bash Commands

This is a list of basic bash commands that will help you navigate directories during this tutorial. It's recommended you play with these commands in the terminal to gain familiarity with them. These commands will work across systems

```
$ cd name/of/target/directory #moves you into the target
↪  directory
$ cd .. #moves you up a directory
$ ls #lists the contents of the current directory
$ pwd #list the path from root to current directory
```

## 2.3 Setting up GitHub

To proceed with this process you need to make a GitHub account.

If you don't already have one you can sign up for it here:

### 2.3.1 Configuring Git

A couple of very basic configurations should be made before you get started. You should set your name and email address as well as enable coloring to pretty up command outputs:

Make sure to use the same email that you used for your GitHub account.

```
$ git config --global user.name "John Doe"
$ git config --global user.email "john@doe.org"
$ git config --global color.ui auto
```

# 3 Basic Workflow

Before we get lost in Git commands, you should understand what a basic workflow with version control looks like. We'll walk through each step in detail later in this book. But first, let's get an understanding of the workflow in general.

The most basic building block of version control is a "repository".

**Definition** *Repository:*

*Think of a repository as a kind of database where your VCS stores all the versions and metadata that accumulate in the course of your project. In Git, the repository is just a simple hidden folder named ".git" in the root directory of your project. Knowing that this folder exists is more than enough. You don't have to (and, moreover, should not) touch anything inside this magical folder*

Getting such a repository on your local machine can be done in two ways:

- If you have a project locally on your computer that is not yet under version control, you can initialize a new repository for this project.
- If you're getting on board of a project that's already running, chances are there is a repository on a remote server (on the internet or on your local network). You'll then probably be provided with a URL to this repository that you will then "`$ git clone`" (download / copy) to your local computer.

(1)As soon as you have a local repository, you can start working on your files: modify, delete, add, copy, rename, or move files in whatever application (your favorite editor, a file browser, ...) you prefer. In this step, you don't have to watch out for anything. Just make any changes necessary to move your project forward.

(2)It's only when you feel you've reached a noteworthy state that you have to consider version control again. Then it's time to wrap up your changes in a commit.

**Definition** *Commit:*

*A commit is a wrapper for a specific set of changes. The author of a commit has to comment what he did in a short "commit message". This helps other people (and himself) to understand later what his intention was when making these changes.*

*Every set of changes implicitly creates a new, different version of your project. Therefore, every commit also marks a specific version. It's a snapshot of your complete project at that certain point in time (but saved in a much more efficient way than simply duplicating the whole project...). The commit knows exactly how all of your files and directories looked and can therefore be used, e.g., to restore the project to that certain state.*

(3) However, before you commit, you'll want to get an overview of what you've changed so far. In Git, you'll use the "`$ git status`" command to get a list of all the changes you performed since the last commit: which files did you

change? Did you create any new ones or deleted some old ones?

(4) Next, you tell Git which of your local changes you want to wrap up in the next commit. Only because a file was changed doesn't mean it will be part of the next commit! Instead, you have to explicitly decide which changes you want to include. To do this, you "`$ git add`" them to the so-called "Staging Area".

(5) Now, having added some changes to the Staging Area, it's time to actually commit these changes. You'll have to add a short and meaningful message that describes what you actually did. The commit will then be recorded in your local Git repository, marking a new version of your project.

(6) From time to time, you'll want to have a look at what happened in the project - especially if you're working together with other people. The "`$ git log`" command lists all the commits that were saved in chronological order. This allows you to see which changes were made in detail and helps you comprehend how the project evolved.

(7) Also when collaborating with others, you'll both want to share (some of) your changes with them and receive the changes they made. A remote repository on a server is used to make this exchange possible.

**Definition** *Local & Remote Repositories:*

*There are two kinds of repositories:*
*A "local" repository resides on your local computer, as a ".git" folder inside your project's root folder. You are the only person that can work with this repository, by committing changes to it. A "remote" repository, in contrast, is typically located on a remote server on the internet or in your local network. No actual working files are associated with a remote repository: it has no working directory but it exclusively consists of the ".git" repository folder. Teams are using remote repositories to share & exchange data: they serve as a common base where everybody can publish their own changes and receive changes from their teammates.*

## 3.1 Creating a local git repository

Let's start with an existing project that is not yet under version control. Change into the project's root folder on the command line and use the "git init" command to start versioning this project:

```
$ cd path/to/project/folder
$ git init
```

Now take a moment to look at the files in that directory (including any hidden files):

```
$ ls -la
```

You'll see that a new, hidden folder was added, named ".git". All that happened is that Git created an empty local repository for us. Please mind the word "empty": Git did not add the current content of your working copy as something like an "initial version". The repository contains not a single version of your project, yet.

**Definition** *Working Copy:*

*The root folder of your project is often called the "working copy" (or "working directory"). It's the directory on your local computer that contains your project's files.*

*You can always ask the version control system to populate your working copy with any version of your project. But you always only have one working copy with one specific version on your disk - not multiple in parallel.*

### 3.1.1 Ignoring Files

Typically, in every project and on every platform, there are a couple of files that you don't want to be version controlled: Eclipse likes to make a massive .eclipse folder and if version controlled it has a tendancy to cause problems. In other projects, you might have build or cache files that make no sense in a version control system. You'll have to decide yourself which files you don't want to include.

**Note** *What Files Should I Ignore?*

*As a simple rule of thumb you'll most likely want to ignore files that were created automatically (as a "by-product"): temporary files, logs, cache files...*

*Other examples for excluded files range from compiled sources to files that contain passwords or personal configurations.*

*A helpful compilation of ignore rules for different projects and platforms can be found here*

The list of files to ignore is kept in a simple file called ".gitignore" in the root folder of your project. It's highly recommended to define this list at the very beginning of your project - before making your first commit. Because once files are committed, you'll have to jump through some hoops to get them out of version control, again.

Now, let's get going: Create an empty file in your favorite editor and save it as ".gitignore" in your project's root folder. If you're on a Mac, e.g., you'll

want to make sure it contains at least the following line:

```
.DS_Store
```

If there are other files you want to ignore, simply add a line for each one. Defining these rules can get quite complex. Therefore, to keep things simple, I'll list the most useful patterns which you can easily adapt to your own needs:

- Ignore one specific file: Provide the full path to the file, seen from the root folder of your project.
  path/to/file.ext
- Ignore all files with a certain name (anywhere in the project): Just write down the file's name, without giving a path.
  filename.ext
- Ignore all files of a certain type (anywhere in the project):
  *.ext
- Ignore all files in a certain folder:
  path/to/folder/*

### 3.1.2 Making Your First Commit

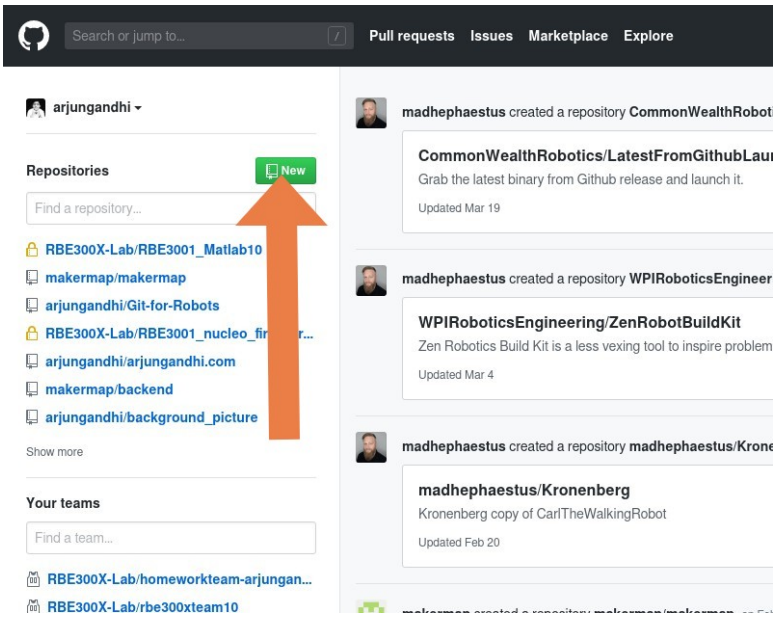
With some ignore rules in place, it's time to make our initial commit for this project. We'll go into great detail about the whole process of committing a little later in this document. For now, simply execute the following commands:

```
$ git add -A
$ git commit -m "Initial commit"
```

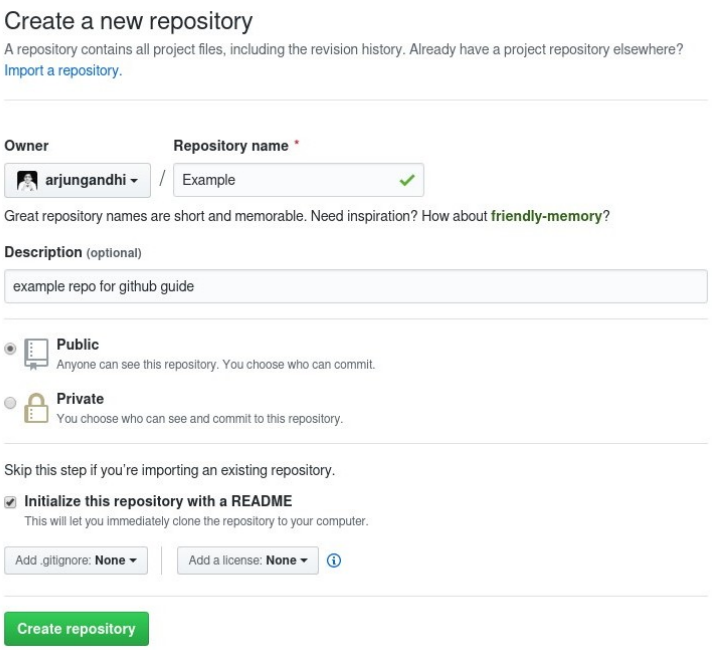
# 4 Starting with an Existing Project on a Server

## 4.1 Creating a repository on GitHub

We are going to go onto GitHub and make our first repository. Log on to GitHub and hit the new repository button:




Go ahead and give your repository a unique name:

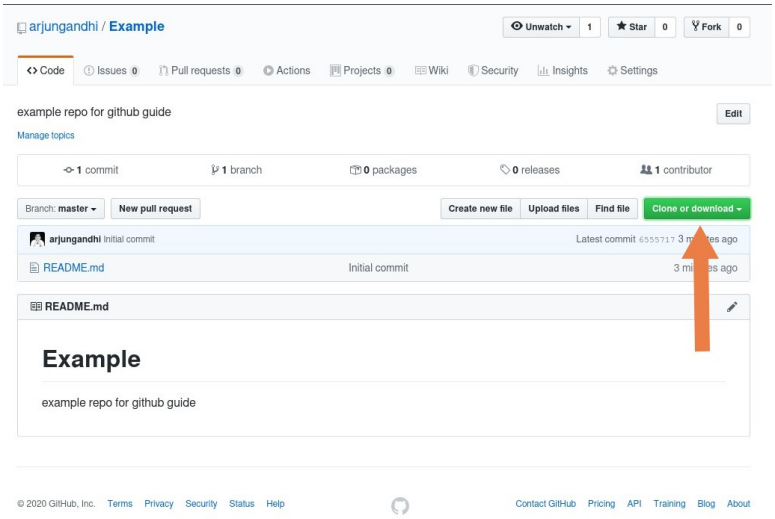



There are a couple options on GitHub.

- Public vs Private: This means exactly what you think, choose public if you want to keep the repository visible to others on the GitHub website and private if you want only you and select friends/teammates to be able to see and access the repository. As a general rule of thumb school work should be kept in a private repository.
- Initialize repository with a ReadMe: This will create an example file called README.md in your new GitHub repository that will be rendered when ever some one visits the repository. This is a good place to put any useful information that someone looking at your project might need to know.

Now we need to download the new GitHub project onto our computer. This is known as cloning a repository.

We first need the URL for the repository this can be found by clicking the clone or download button on the page.




Copy the link that pops up and make sure you have selected the HTTPS option.

Go back to the terminal and cd into the directory where you want to download the repository.

```
$ cd your/development/folder/
```

Now we will clone our new repository with the clone command.

```
$ git clone https://github.com/arjungandhi/Example.git
```

You will have to enter your GitHub username and password here.

Git will now download a complete copy of this repository to your local disk.

# 5 Working on your Project

No matter if you created a brand new repository or if you cloned an existing one - you now have a local Git repository on your computer. This means you're ready to start working on your project: use whatever application you want to change, create, delete, move, copy, or rename your files.

**Concept** *The Status of a File:*

*In general, files can have one of two statuses in Git:*

- *untracked: a file that is not under version control, yet, is called "untracked". This means that the version control system doesn't watch for (or "track") changes to this file. In most cases, these are either files that are newly created or files that are ignored and which you don't want to include in version control at all.*
- *tracked: all files that are already under version control are called "tracked". Git watches these files for changes and allows you to commit or discard them.*

## 5.1 The Staging Area

At some point after working on your files for a while, you'll want to save a new version of your project. Or in other words: you'll want to commit some of the changes you made to your tracked files.

**Golden Rule 1** *Commit Only Related Changes:*

*When crafting a commit, it's very important to only include changes that belong together. You should never mix up changes from multiple, different topics in a single commit. For example, imagine wrapping both some work for your path planning and your drive code in fix #169 the same commit:*

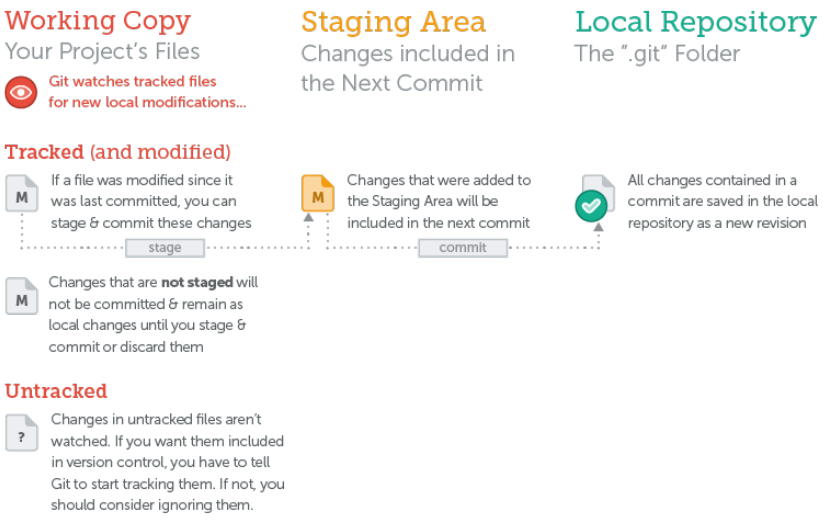
- *Understanding what all those changes really mean and do gets hard for your teammates (and, after some time, also for yourself). Someone who's trying to understand the progress of that new login functionality will have to untangle it from the bugfix code first.*
- *Undoing one of the topics gets impossible. Maybe your login functionality introduced a new bug. You can't undo just this one without*

*undoing your work for fix #169, also!*

*Instead, a commit should only wrap related changes: fixing two different bugs should produce (at the very least) two separate commits; or, when developing a larger feature, every small aspect of it might be worth its own commit. Small commits that only contain one topic make it easier for other members of your team to understand the changes - and to possibly undo them if something went wrong.*

However, when you're working full-steam on your project, you can't always guarantee that you only make changes for one and only one topic. Often, you work on multiple aspects in parallel.

This is where the "Staging Area", one of Git's greatest features, comes in very handy: it allows you to determine which of your local changes shall be committed. Because in Git, simply making some changes doesn't mean they're automatically committed. Instead, every commit is "hand-crafted": each change that you want to include in the next commit has to be marked explicitly ("added to the Staging Area" or, simply put, "staged").




## 5.2 Getting an Overview of Your Changes

Let's have a look at what we've done so far. To get an overview of what you've changed since your last commit, you simply use the "git status" command:

```
$ git status
# On branch master
# Changes not staged for commit:
#   (use "git add/rm <file>... " to update what will be
↪  committed)
#   (use "git checkout -- <file>..." to discard changes in
↪  working
#    directory)
#
#       modified:   robot/drive.py
#       modified:   robot/path.py
#       deleted:    work.py
#       modified:   runofftable.py
#       modified:   dobackflip.py
#
# Untracked files:
#    (use "git add <file>..." to include in what will be
↪  committed)
#       display.py
no changes added to commit (use "git add" and/or "git commit
↪  -a")
```

Thankfully, Git provides a rather verbose summary and groups your changes in 3 main categories:

- "Changes not staged for commit"
- "Changes to be committed"
- "Untracked files"

## 5.3 Getting Ready to Commit

Now it's time to craft a commit by staging some changes with the "git add" command:

```
$ git add runofftable.py display.py robot/*
```

With this command, we added the new "display.py" file, the modifications in "runofftable.py", and all the changes in the "robot" folder to the Staging Area. Since we also want to record the removal of "work.py" in the next commit, we have to use the "git rm" command to confirm this:

```
$ git rm work.py
```

Let's use "git status" once more to make sure we've prepared the right stuff:

```
$ git status
# On branch master
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#       modified:   robot/drive.py
#       modified:   robot/path.py
#       deleted:    work.py
#       modified:   runofftable.py
#       new file:   display.py
#
# 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:   dobackflip.py
#
```

Assuming that the changes in "dobackflip.py" concerned a different topic than the rest, we've deliberately left them unstaged. That way, they won't be included in our next commit and simply remain as local changes. We can then continue to work on them and maybe commit them later.

## 5.4 Committing Your Work

Having carefully prepared the Staging Area, there's only one thing left before we can actually commit: we need a good commit message.

**Golden Rule 2** *Write Good Commit Messages*

*Time spent on crafting a good commit message is time spent well: it will make understanding what happened easier for your teammates (and after some time also for yourself).*

*Begin your message with a short summary of your changes (up to 50 characters as a guideline). Separate it from the following body by including a blank line. The body of your message should provide detailed answers to the following questions: What was the motivation for the change? How does it differ from the previous version?*

The "git commit" command wraps up your changes:

```
$ git commit -m "Got the robot to drive off a table
↪  consistently"
```

If you have a longer commit message, possibly with multiple paragraphs, you can leave out the "-m" parameter and Git will open an editor application for you (which you can also configure via the "core.editor" property).

**Concept** *What Makes a Good Commit?*

*The better and more carefully you craft your commits, the more useful will version control be for you. Here are some guidelines about what makes a good commit:*

- *Related Changes: As stated before, a commit should only contain changes from a single topic. Don't mix up contents from different topics in the same commit. This will make it harder to understand what happened.*
- *Completed Work: Never commit something that is half-done. If you need to save your current work temporarily in something like a clipboard, you can use Git's "Stash" feature (which will be discussed later in the document). But don't eternalize it in a commit.*
- *Tested Work: Related to the point above, you shouldn't commit code that you think is working. Test it well - and before you commit it to the repository.*
- *Short & Descriptive Messages: A good commit also needs a good message. See the paragraph above on how to "Write Good Commit Messages" for more about this.*

*Finally, you should make it a habit to commit often. This will automatically help you to keep your commits small and only include related changes.*

## 5.5 Sending your Changes to the Cloud

If you are working in a repository the commits you made locally aren't pushed to the cloud yet. You can do this with the "`$ git push`" command.

```
$ git push
```

## 5.6 Inspecting the Commit History

Git saves every commit that is ever made in the course of your project. Especially when collaborating with others, it's important to see recent commits to understand what happened.

The "git log" command is used to display the project's commit history:

```
$ git log
```

It lists the commits in chronological order, beginning with the newest item. If there are more items than it can display on one page, the command line indicates this by showing a colon (":") at the end of the page. You can then go to the next page with the SPACE key and quit with the "q" key.

```
commit 2dfe283e6c81ca48d6edc1574b1f2d4d84ae7fa1
Author: Arjun Gandhi <beepboop@arjungandhi.com>
Date: Fri Jul 26 10:52:04 2013 +0200

    Got the robot to drive off a table consistently

commit 2b504bee4083a20e0ef1e037eea0bd913a4d56b6
Author: Arjun Gandhi <beepboop@arjungandhi.com>
Date: Fri Jul 26 10:05:48 2013 +0200

    Got the robot to unplug its own wires

commit 0023cdddf42d916bd7e3d0a279c1f36bfc8a051b
Author: Arjun Gandhi <beepboop@arjungandhi.com>
Date: Fri Jul 26 10:04:16 2013 +0200

    Robot learned about fire! oh no!
```

Every commit item consists (amongst other things) of the following metadata:

- Commit Hash
- Author Name & Email
- Date
- Commit Message

**Definition** *The Commit Hash:*

*Every commit has a unique identifier: a 40-character checksum called the "commit hash". While in centralized version control systems an ascending revision number is used for this, this is simply not possible anymore in a distributed VCS like Git: The reason herefore is that, in Git, multiple people can work in parallel, committing their work offline, without being connected to a shared repository. In this scenario, you can't say anymore*

*whose commit is #5 and whose is #6.*

*Since in most projects, the first 7 characters of the hash are enough for it to be unique, referring to a commit using a shortened version is very common.*

Apart from this metadata, Git also allows you to display the detailed changes that happened in each commit. Use the "-p" flag with the "git log" command to add this kind of information:

```
$ git log -p
commit 2dfe283e6c81ca48d6edc1574b1f2d4d84ae7fa1
Author: Arjun Gandhi <beepboop@arjungandhi.com>
Date: Fri Jul 26 10:52:04 2013 +0200

    Got the robot to drive off a table consistently

diff --git a/robot/drive.py b/robot/path.css
index e69de29..4b5800f 100644
--- a/css/drive.py
+++ b/css/drive.py
@@ -0,0 +1,2 @@
+ for pid in pids:
+     pid.dopid()
\ No newline at end of file
diff --git a/robot/path.py b/robot/path.py
index a3b8935..d472b7f 100644
--- a/robot/path.py
+++ b/robot/path.py
@@ -21,7 +21,8 @@
 +edges=lidar.find_edge
+ target=None
+ for edge in edges:
+     if edge.distance > 1000:
+         target = edge.center()
+         break
+ path=astar(cur_pos,target)
+ return path
diff --git a/work.py b/work.py
deleted file mode 100644
index 78alc33..0000000
--- a/work.py
+++ /dev/null
@@ -1,43 +0,0 @@
- robot.begood()
- robot.disable_murphy()
```

```
- robot.disable_stupid()
```

This shows in detail how each file in the commit was modified or changed.

### 5.7 Time to Celebrate

Congratulations, these are the fundamentals of working with git and version control. Many projects can and have been build on this foundation. However, git can do much more than this. So take a break, maybe grab a beverage, and let's continue on!

I suggest to further cement your knowledge, try doing all the actions by working on a small programming project. A fun one is to try writing an auto solver for the 24 Game. Remember to keep in mind all the things you have learned about committing often and frequently with good commit messages!

## 6 Branching Can Change Your Life

It sounds insane .... But the truth is: it's not an exaggeration. Using branches in your day-to-day work might very well make you a better programmer or designer.

Now, let's look at why branches are so important.

### 6.1 Working in Contexts

In every project, there are always multiple different contexts where work happens. Each feature, bugfix, experiment, or alternative of your product is actually a context of its own: it can be seen as its own "topic", clearly separated from other topics.

This leaves you with an unlimited amount of different contexts. Most likely, you'll have at least one context for your "main" or "production" state, and another context for each feature, bugfix, experiment, etc.

In real-world projects, work always happens in multiple of these contexts in parallel:

- while you're experimenting with two versions of a PID loop to see which one runs faster (contexts 1 & 2)...
- you're also trying to fix an annoying bug (context 3).
- on the side, you also update some set values for your gripper (context 4), while...

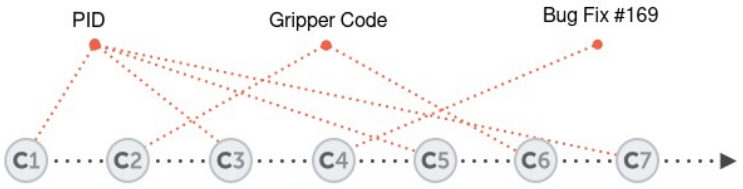
- one of your teammates is working on brand new gripper code (context 5),...
- and another teammate is trying to rewrite the entire codebase (context 6).

## 6.2 A World Without Branches

Not working in clearly separated contexts can (and sooner or later will) cause several problems:

- What happens if your 2nd PID codes turns out to be much faster but in the meantime a bunch of other changes have happened?
- What happens if the new gripper code your teammate wrote turned out to be hot garbage? How do you get all that unwanted code (and only that code!) out?
- What do you do if the entire code rewrite turns out to be impossible to implement? It's already mingled with all of those other changes, being almost impossible to separate out!
- How can you avoid losing track of what actually matters? Most likely, you shouldn't be bothered with all the topics from all of your teammates.

Things will start to get very confusing when you try to handle multiple topics in a single context:



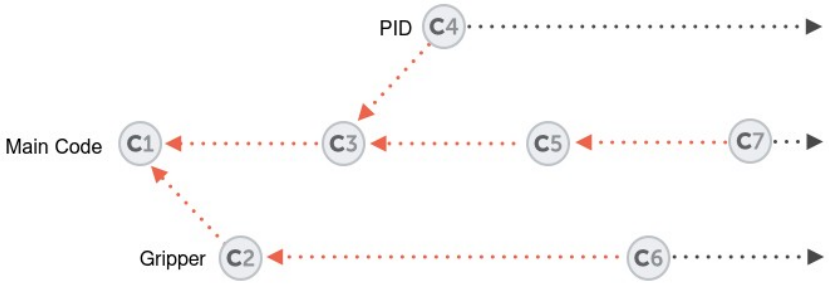

A tempting workaround might be to simply copy your complete project folder for each new context. But this only leaves you with other problems:

- You circumvent your VCS, since those new folders won't be under version control.
- Not being version controlled, you can't easily share & collaborate with others.
- Integrating changes from one context into another (maybe your main context) is difficult and error-prone.

To make a long story short: if your goal is to maintain a level of sanity, you'll have to find a way to deal with multiple contexts in a clean manner.

### 6.3 Branches to the Rescue

You might have already guessed it: branches are what we need to solve these problems. Because a branch represents exactly such a context in a project and helps you keep it separate from all other contexts.




All the changes you make at any time will only apply to the currently active branch; all other branches are left untouched. This gives you the freedom to both work on different things in parallel and, above all, to experiment - because you can't mess up! In case things go wrong you can always go back / undo / start fresh / switch contexts...

Luckily, branches in Git are cheap & easy. There's no reason not to create a new branch when you start working on a new topic, no matter how big or small it might be.

**Golden Rule 3** *Use Branches Extensively:*

*Branching is one of Git's most powerful features – and this is not by accident: quick and easy branching was a central requirement from day one. Branches are the perfect tool to help you avoid mixing up different lines of development. You should use branches extensively in your development workflows: for new features, bug fixes, experiments, ideas...*

# 7 Working with branches

Until now, we haven't taken much notice of branches in our example project. However, without knowing, we were already working on a branch! This is because branches aren't optional in Git: you are always working on a certain branch (the currently active, or "checked out", or "HEAD" branch).

So, which branch is HEAD at the moment? The "git status" command tells us in its first line of output: "On branch master". The "master" branch was created by Git automatically for us when we started the project. Although you

could rename or delete it, you'll have a hard time finding a project without it because most people just keep it. But please keep in mind that "master" is by no means a special or magical branch. It's like any other branch!

Now, let's start working on a new feature. Based on the project's current state, we create a new branch and name it "pid-test":

```
$ git branch pid-test
```

Using the "git branch" command lists all of our branches (and the "-v" flag provides us with a little more data than usual):

```
$ git branch -v
  pid-test     3de33cc Got the robot to drive off a table
  ↪  consistently
* master       3de33cc [ahead 1] Got the robot to drive off a
↪  table consistently
```

You can see that our new branch "pid-test" was created and is based on the same version as "master". Additionally, the little asterisk character (*) next to "master" indicates that this is our current HEAD branch. To emphasize this: the "git branch" command only created that new branch - but it didn't make it active. Before checking out that new branch, it's a good idea to have another look at "git status" to see where we currently are:

```
$ git status
# On branch master
# 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:   dobackflip.py
#
no changes added to commit (use "git add" and/or "git commit
↪  -a")
```

Oh, right: we still have some changes in "imprint.html" in our working copy! Actually, we just wanted to start working on our new "pid-test" branch; but these changes don't belong to this feature. So what do we do with them? One way to get this work-in-progress out of the way would be to simply commit it. But committing half-done work is a bad habit.

**Golden Rule 4** *Never Commit Half-Done Work*

*You should only commit code when it's completed. This doesn't mean you have to complete a whole, large feature before committing. Quite the contrary: split the feature's implementation into logical chunks and remember to commit early and often. But don't commit just to get half-done work out of your way when you need a "clean working copy". For these cases, consider using Git's "Stash" feature instead.*

# 8 Saving Changes Temporarily

A commit wraps up changes and saves them permanently in the repository. However, in your day-to-day work, there are a lot of situations where you only want to save your local changes temporarily. For example, imagine you're in the middle of some changes for feature X when an important bug report comes in. Your local changes don't belong to the bugfix you're going to make. You have to get rid of them (temporarily, without losing them!) and continue working on them later.

Situations like this one happen all the time: you have some local changes in your working copy that you can't commit right now - and you want or need to start working on something else. To get these changes out of your way and have a "clean" working copy, Git's "Stash" feature comes in handy

**Concept** *The Stash*

*Think of the Stash as a clipboard on steroids: it takes all the changes in your working copy and saves them for you on a new clipboard. You're left with a clean working copy, i.e. you have no more local changes.*

*Later, at any time, you can restore the changes from that clipboard in your working copy - and continue working where you left off.*

*You can create as many Stashes as you want - you're not limited to storing only one set of changes. Also, a Stash is not bound to the branch where you created it: when you restore it, the changes will be applied to your current HEAD branch, whichever this may be.*

Let's stash away these local changes so we have a clean working copy before starting to work on our new feature:

```
$ git stash
Saved working directory and index state WIP on master:
   2dfe283 Got the robot to drive off a table consistently
HEAD is now at 2dfe283 Got the robot to drive off a table
↪  consistently
```

```
$ git status
# On branch master
nothing to commit (working directory clean)
```

The local changes in "dobackflip.py" are now safely stored on a clipboard, ready to be restored any time we want to continue working on them.

You can easily get an overview of your current Stashes:

```
$ git stash list
stash@{0}: WIP on master: 2d6e283 Implement the new login box
```

The newest Stash will always be at the top of the list, named "stash@0". Older Stashes have higher numbers.

When you're ready to restore a saved Stash, you have two options:

- Calling "`$ git stash pop`" will apply the newest Stash and clear it from your Stash clipboard.
- Calling "`$ git stash apply <stashname>`" will also apply the specified Stash, but it will remain saved. You can delete it later via "`$ git stash drop <stashname>`".

You can choose to not specify the Stash when using any of these commands. Then, Git will simply take the newest Stash (always "stash@0").

**Concept** *When to Stash*

*Stashing helps you get a clean working copy. While this can be helpful in many situations, it's strongly recommended...*

*...before checking out a different branch.*
*...before pulling remote changes.*
*...before merging or rebasing a branch.*

# 9 Checking Out a Local Branch

Now that we have a clean working copy, the first thing we have to do is switch to (or "check out") our newly created branch:

```
$ git checkout pid-test
```

**Concept** *Checkout, HEAD, and Your Working Copy*

*A branch automatically points to the latest commit in that context. And since a commit references a certain version of your project, Git always knows exactly which files belong to that branch.*


*At each point in time, only one branch can be HEAD / checked out / active. The files in your working copy are those that are associated with this exact branch. All other branches (and their associated files) are safely stored in Git's database.*

*To make another branch (say, "pid-test") active, the "git checkout" command is used. This does two things for you:*

- *It makes "pid-test" the current HEAD branch.*
- *It replaces the files in your working directory to match exactly the revision that "pid-test" is at.*

Running "git status" once more, you'll see that we're now "On branch pid-test". From now on, all of our changes and commits will only impact this very context - until we switch it again by using the "checkout" command to make a different branch active.

Let's prove this by creating a new file called "jumpboy.py" and committing it:

```
$ git add jumpboy.py
$ git commit -m "Robot can now jump"
$ git log
commit 56eddd14cf034f4bcb8dc9cbf847b33309fa5180
Author: Arjun Gandhi <beepboop@arjungandhi.com>
Date: Fri Jul 26 10:56:16 2013 +0200

    Robot can now jump

commit 2dfe283e6c81ca48d6edc1574b1f2d4d84ae7f1
Author: Arjun Gandhi <beepboop@arjungandhi.com>
```

```
Date: Fri Jul 26 10:52:04 2013 +0200

    Got the robot to drive off a table consistently

commit 2b504bee4083a20e0ef1e037eea0bd913a4d56b6
Author: Arjun Gandhi <beepboop@arjungandhi.com>
Date: Fri Jul 26 10:05:48 2013 +0200

    Got the robot to unplug its own wires
```

Looking at the Log, you'll see that your new commit was properly saved. No big surprises, so far. But now let's switch back to "master" and have a look at the Log once more:

```
$ git checkout master
$ git log
commit 2dfe283e6c81ca48d6edc1574b1f2d4d84ae7f1
Author: Arjun Gandhi <beepboop@arjungandhi.com>
Date: Fri Jul 26 10:52:04 2013 +0200

    Got the robot to drive off a table consistently

commit 2b504bee4083a20e0ef1e037eea0bd913a4d56b6
Author: Arjun Gandhi <beepboop@arjungandhi.com>
Date: Fri Jul 26 10:05:48 2013 +0200

    Got the robot to unplug its own wires
```

You'll find that the "Robot can now jump" commit isn't there - because we made it in the context of our HEAD branch (which was the "pid-test" branch, not the "master" branch). This is exactly what we wanted: our changes are kept in their own context, separated from other contexts.

# 10 Merging Changes

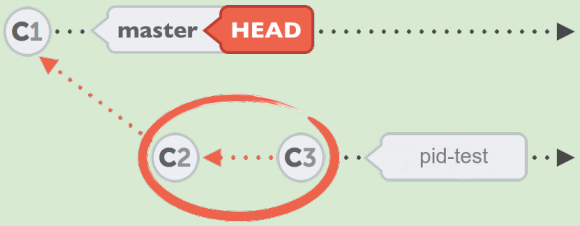
Keeping your commits in the separate context of a branch is a huge help. But there will come a time when you want to integrate changes from one branch into another. For example when you finished developing a feature and want to integrate it into your "production" branch. Or maybe the other way around: you're not yet finished working on your feature, but so many things have happened in the rest of the project in the meantime that you want to integrate these back into your feature branch.

Whatever the scenario may be: such an integration is called "merging" and is done with the "git merge" command.

**Concept** *Integrating Branches - Not Individual Commits*

*When starting a merge, you don't have to (and cannot) pick individual commits that shall be integrated. Instead, you tell Git which branch you want to integrate - and Git will figure out which commits you don't have in your current working branch. Only these commits will then be integrated as a result.*

*Also, you never have to think long and hard about where these changes end up: The target of such an integration is always your current HEAD branch and, thereby, your working copy.*




In Git, performing a merge is easy as pie. It requires just two steps:

- Check out the branch that should receive the changes.
- Call the "git merge" command with the name of the branch that contains the desired changes. Let's integrate the changes from our "pid-test" branch into "master":

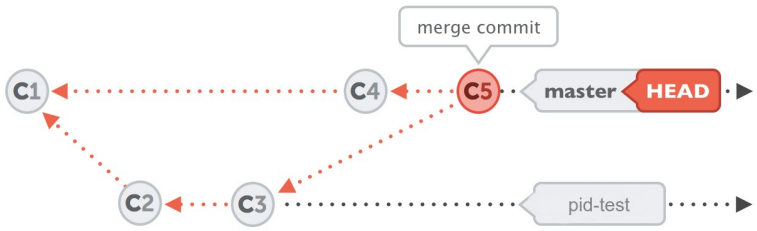
```
$ git checkout master
$ git merge pid-test
```

When you now perform a "git log" command, you'll see that our "Robot can now jump" commit was successfully integrated into master!

```
$ git log
commit 56eddd14cf034f4bcb8dc9cbf847b33309fa5180
Author: Arjun Gandhi <beepboop@arjungandhi.com>
Date: Fri Jul 26 10:56:16 2013 +0200

    Robot can now jump

commit 2dfe283e6c81ca48d6edc1574b1f2d4d84ae7f1
Author: Arjun Gandhi <beepboop@arjungandhi.com>
Date: Fri Jul 26 10:52:04 2013 +0200
```

```
    Got the robot to drive off a table consistently

commit 2b504bee4083a20e0ef1e037eea0bd913a4d56b6
Author: Arjun Gandhi <beepboop@arjungandhi.com>
Date: Fri Jul 26 10:05:48 2013 +0200

    Got the robot to unplug its own wires
```

However, the result of a merge action can't always be displayed that clearly: not always can Git simply add the missing commits on top of the HEAD branch. Often, it will have to combine changes in a new, separate commit called a "merge commit". Think of it like a knot that connects two branches. You can merge




one branch into another as often as you like. Git will again figure out which changes haven't been merged and only consider these.

## 10.1 Dealing with Merge Conflicts

For a lot of people, merge conflicts are as scary as accidentally formatting their hard drive. In the course of this section, I want to relieve you from this fear.

### 10.1.1 You Cannot Break Things

The first thing that you should keep in mind is that you can always undo a merge and go back to the state before the conflict occurred. You're always able to undo and start fresh.

Also, a conflict will only ever handicap yourself. It will not bring your complete team to a halt or cripple your central repository. This is because, in Git, conflicts can only occur on a developer's local machine - and not on the server.

### 10.1.2 How a Merge Conflict Occurs

In Git, "merging" is the act of integrating another branch into your current working branch. You're taking changes from another context (that's what a branch effectively is: a context) and combine them with your current working

files.

A great thing about having Git as your version control system is that it makes merging extremely easy: in most cases, Git will figure out how to integrate new changes.

However, there's a handful of situations where you might have to step in and tell Git what to do. Most notably, this is when changing the same file. Even in this case, Git will most likely be able to figure it out on its own. But if two people changed the same lines in that same file, or if one person decided to delete it while the other person decided to modify it, Git simply cannot know what is correct. Git will then mark the file as having a conflict - which you'll have to solve before you can continue your work.

### 10.1.3 How to Solve a Merge Conflict

When faced with a merge conflict, the first step is to understand what happened. E.g.: Did one of your colleagues edit the same file on the same lines as you? Did he delete a file that you modified? Did you both add a file with the same name? Git will tell you that you have "unmerged paths" (which is just another way of telling you that you have one or more conflicts) via "git status":

```
$ git status
# On branch pid-test
# You have unmerged paths.
#   (fix conflicts and run "git commit")
#
# Unmerged paths:
#   (use "git add <file>..." to mark resolution)
#
#       both modified:   dobackflip.py
#
no changes added to commit (use "git add" and/or "git commit
↪  -a")
```

Let's take an in-depth look on how to solve the most common case, when two changes affected the same file on the same lines. Now is the time to have a look at the contents of the conflicted file. Git was nice enough to mark the problematic area in the file by enclosing it in "«««< HEAD" and "»»»> [other/branch/name]".

```
<<<<<<< HEAD
This code existed in our HEAD branch
=======
This code existed in the branch we tried to merge into HEAD
>>>>>>> [other/branch/name]
```

Handling this is much easier than people suspect all you have to do is open your favorite text editor and clean up the code. Your goal is to make the code exactly like it normally should. Delete/Modify the lines as needed, save the file and commit. It can be necessary to consult the teammate who wrote the conflicting changes to decide which code is finally correct. Maybe it's yours, maybe it's theirs - or maybe a mixture between the two.

There also exists some 3rd party editors that make this process a little easier by having a pretty GUI that you can click on.

### 10.1.4 How to Undo a Merge

You should always keep in mind that you can return to the state before you started the merge at any time. This should give you the confidence that you can't break anything. On the command line, a simple "git merge –abort" will do this for you.

In case you've made a mistake while resolving a conflict and realize this only after completing the merge, you can still easily undo it: just roll back to the commit before the merge happened with "git reset –hard " and start over again.

## 11 Working with remote branches

About 90% of version control related work happens in the local repository: staging, committing, viewing the status or the log/history, etc.

Only when it comes to sharing data with your teammates, a remote repo comes into play. Think of it like a "file server" that you use to exchange data with your colleagues.

Let's look at the few things that distinguish local and remote repositories from each other:

- Location: Local repositories reside on the computers of team members. In contrast, remote repositories are hosted on a server that is accessible for all team members - most likely on the internet or on a local network.
- Features: Technically, a remote repository doesn't differ from a local one: it contains branches, commits, and tags just like a local repository. However, a local repository has a working copy associated with it: a directory where some version of your project's files is checked out for you to work with. A remote repository doesn't have such a working directory: it only consists of the bare ".git" repository folder.
- Usage: It's important to stress that the actual work on your project happens only in your local repository: all modifications have to be made &

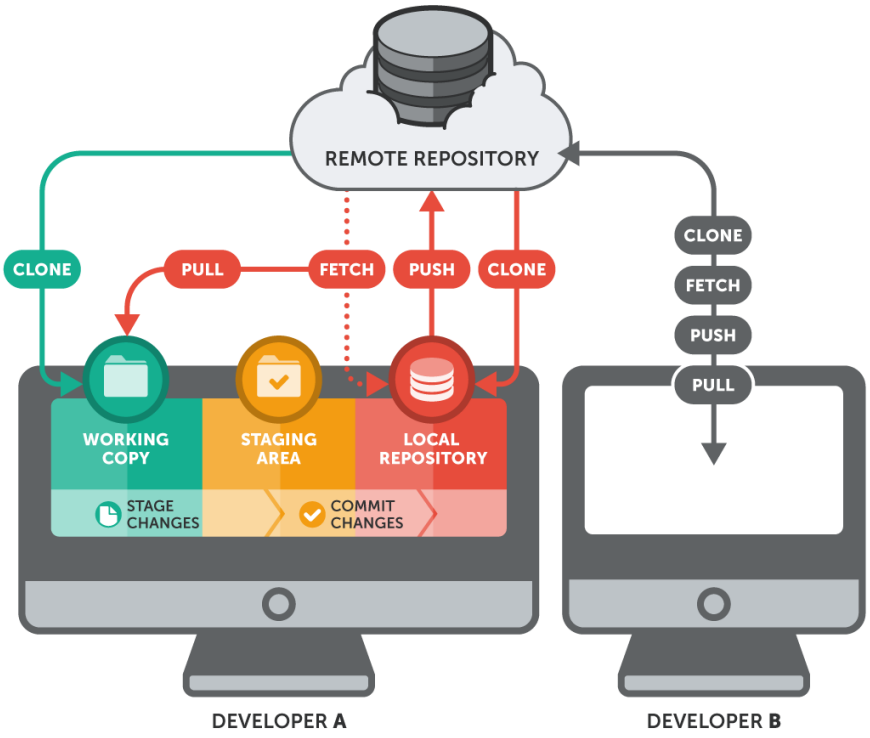
committed locally. Then, those changes can be uploaded to a remote repository in order to share them with your team. Remote repositories are only thought as a means for sharing and exchanging code between developers - not for actually working on files.

## 11.1 Local / Remote Workflow

In Git, there are only a mere handful of commands that interact with a remote repository.

The overwhelming majority of work happens in the local repository. Until this point (except when we called "git clone"), we've worked exclusively with our local Git repository and never left our local computer. We were not dependent on any internet or network connection but instead worked completely offline.

We'll look at each of these commands in the course of the following sections.




## 11.2 Integrating Remote Changes

Sooner or later, one of your teammates will probably also share their changes on your common remote repository. Before integrating these changes into your

local working copy, you might first want to inspect them:

```
$ git fetch origin
$ git log origin/master
```

The "git log" command now shows you the commits that happened recently on the "master" branch of the remote called "origin".
If you decide you want to integrate these changes into your working copy, the "git pull" command is what you need:

```
$ git pull
```

This command downloads new commits from the remote and directly integrates them into your working copy. It's actually a "fetch" command (which only downloads data) and a "merge" command (which integrates this data into your working copy) combined.

As with the "git push" command: in case no tracking connection was established for your local HEAD branch, you will also have to tell Git from which remote repository and which remote branch you want to pull ("git pull origin master", e.g.). In case a tracking connection already exists, a vanilla "git pull" is sufficient.

The target of the integration, however, is independent of a tracking connection and is always the same: your local HEAD branch and, thereby, your working copy.

# 12 Fun Things, Foot Notes and Personal Favorites

This is just a quick section on some of my favorite tools and features for managing and handling git projects.

## 12.1 EGit(Eclipse)

I personally do not use Eclipse as an editor for anything, but I know in the 2000 level RBE courses you work with the Sloeber IDE. There exists a git plugin for the Eclipse IDE by the name of EGit. If you have a hard aversion to command line, you might want to consider installing it. (It may or may not come preinstalled.) You should have no problem figuring out how to push, pull, and commit if you've made it through this guide.

## 12.2 GitHub Desktop

The GitHub Desktop Tool is another GUI tool you can download. It's a clean, simple way to handle 99% of your git tasks and it was written by the dev team over at GitHub. I personally know many people that adore this tool so feel free to download it and give it a try.

## 12.3 GitKraken

This is my all time favorite git GUI tool. It's the one I personally use for my daily workflow, and it's great at handling everything from merge conflicts to stashing. It has a really pretty tree on it that shows your branches and how they have merged and evolved over time. Unfortunately it does not work with the RBE Lab's GitHub due to the way organization permissions work. I do recommend it for personal projects especially if they are of the big and convoluted kind. You can check it out here.

## 12.4 Learn More About Git

Git is a very powerful tool with an incredible amount of features. While I covered the basics needed to be successful in RBE, there is still much more to learn about the subject. Here are some helpful tutorials.

- Git-Tower Guide (Only about half this guide is covered.)
- A Shorter But More Technical Guide
- Here's a great way to practice everything you've learned with actual commands in a virtual terminal.

## 12.5 GitHub Student Developer Pack

Do you like free things? Well if you do you've come to the right paragraph.

The GitHub Student Developer Pack has lots of free stuff for you to get. The only requirement is that you need to be a student!

You do need to send in a picture of your student ID to get approved because WPI email addresses last forever and there's something about that being "abusable".

## 12.6 Sign Off & Acknowledgments

Thanks so much to the following people for looking over this doc and correcting my English:

- Cooper Bennet
- Kristen Andonie
- Sean O'Neil
- Ilyas Salhi
- Anybody else I forgot.

For any questions about this guide, RBE, or just life reach out to me at: agandhi@wpi.edu.

# References

[1] A large portion of this document was "liberated" from: https://www.git-tower.com/learn/git/ebook/en/command-line/basics/why-use-version-control

[2] https://www.atlassian.com/git/tutorials/what-is-git