

Linux Development Environment

In this lecture we will set up a Linux environment for writing and debugging SSJS code using **screen**, **emacs**, and **git**. Then we will show how to set up a new machine in a scriptable and reproducible way via **setup.git** and **dotfiles.git**.

Development Environment

screen: Tab manager for remote sessions

When Google Chrome or Firefox crashes, it offers to restore your tabs upon reboot. The **screen** program is similar: when your network connection to the remote host drops out, you can reconnect and resume your session. This is more than a convenience; for any long-running computation, **screen** (or background processes) is the only way to do anything nontrivial. For the purposes of this class, let's use a custom **.screenrc** file which configures **screen** to play well with the **emacs** text editor (to be introduced shortly). You can install this config file as follows:

```
1 $ cd $HOME
2 $ wget spark-public.s3.amazonaws.com/startup/code/screenrc -O .screenrc
3 $ head .screenrc
4 $ screen
```

Let's go through an interactive session to see the basics of **screen**. If you want to learn even more about **screen**, go through this excellent [tutorial](#) and [video tutorial](#); if you want to use **tmux** instead, [this](#) is a reasonable starting place.

emacs: An environment for writing code

What is a text editor? A text editor like **emacs** or **nano** is different from a word processor like Microsoft Word in that it allows you to directly manipulate the raw, unadorned string of bytes that make up a file. That's why text editors are used by programmers, while word processors are employed by end users (Figure 1).

Many people use GUI-based text editors like [Textmate](#), [Sublime Text](#), [Visual Studio](#), [XCode](#), or [Eclipse](#). These editors have several good properties, but don't support all of the following properties simultaneously:

- free
- open-source
- highly configurable

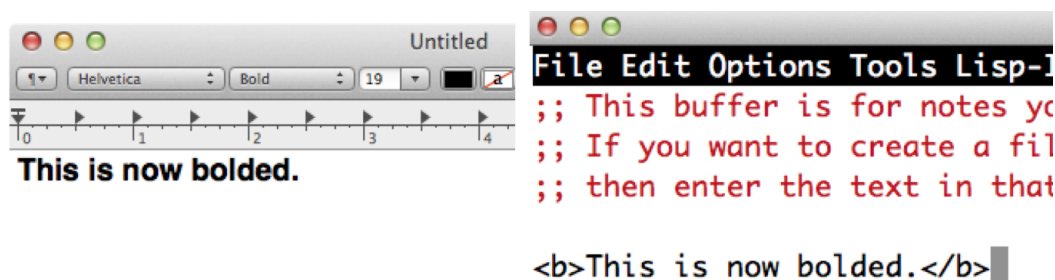


Figure 1: A word processor (left) vs. a text-editor (right). The character escapes to bold text are explicit in a text editor and hidden in a word processor.

- large dev community
- editing modes for virtually every language
- external tool integration
- ultra cross platform
- embeddable in other programs
- fast boot time
- lightweight memory footprint

Two editors that do have all these features are `vim` (vim.org) and `emacs` (gnu.org/software/emacs). If you are doing a technology startup, you probably¹ want to learn one of these two. In general, `vim` is more suitable for sysadmins or people who must manually edit many little files. It has excellent defaults and is very fast for pure text-editing. By contrast, `emacs` is greatly preferred for the data scientist or academic due to features like `orgmode` and strong read-evaluate-print-loop (`REPL`) integration for interactive debugging sessions. For intermediate tasks like web development, one can use either editor, but the `REPL` integration features of `emacs` carry the day in our view.²

OS X and Windows: Control and Meta Keys. Many keyboard sequences³ in Unix depend on pressing multiple keys simultaneously. We use the following notation for key combinations using Control or Meta:

¹Of course there are smart people who don't use `emacs` or `vim`; Jeff Atwood of StackOverflow used `Visual Studio` and built it on the .NET stack, and DHH of Ruby on Rails fame famously uses `TextMate`. Still, even Jeff Atwood used Ruby for his [latest project](#), and the top technology companies ([Google](#), [Facebook](#), et alia) tend to use `vim` or `emacs` *uber alles*.

²One editor out there that might be a serious contender to the `vim` and `emacs` duopoly is the new `Light Table` editor by Chris Granger. A key question will be whether it can amass the kind of open-source community that `emacs` and `vim` have built, or alternatively whether its business model will provide enough funds to do major development in-house.

³The `emacs` keybindings are used by default in any application that uses the `readline` library (like `bash`), while `vim` keybindings are used in commands like `less`, `top`, and `visudo`. The major advantage of investing in one of these editors is that you can get exactly the same editing environment on your local machine or any server that you connect to regularly.

- C-a: press Control and A simultaneously
- M-d: press Meta and D simultaneously
- C-u 7: press Control and U together, then press 7
- C-c C-e: press Control and C simultaneously, let go, and then press Control and E simultaneously

Given how frequent this is, to use **emacs**, **bash**, and **screen** with maximum efficiency you will want to customize your control and meta keys to make them more accessible.

1. *Windows: Keyboard Remapper:* On Windows you will want to use a [keyboard remapper](#) and possibly use [Putty](#) if you have troubles with Cygwin.
2. *Mac: Terminal and Keyboard Preferences:* On a Mac, you can configure Terminal Preferences to have “option as meta” and Keyboard Preferences to swap the [option](#) and [command](#) keys, and set Caps Lock to Control, as shown in Figures 2 and 3.

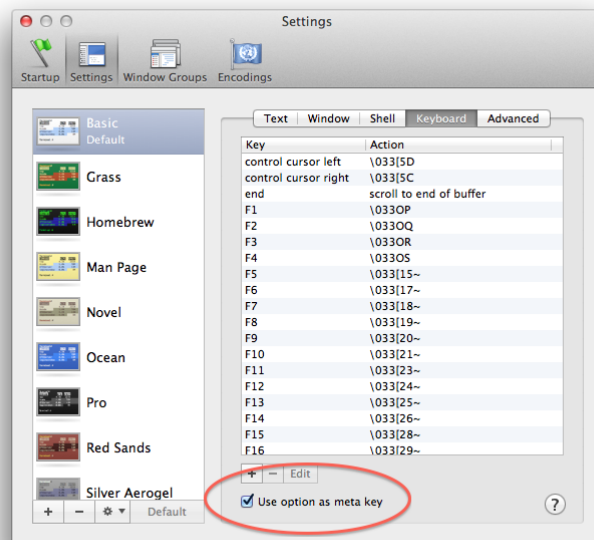


Figure 2: Click Settings (in the top row of buttons) and then “Keyboard”. Click “Use option as meta key”. You’ll need this for *emacs*.

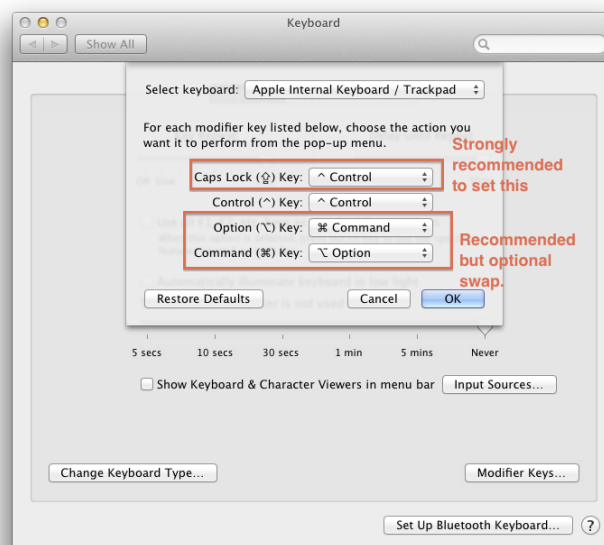


Figure 3: *Switch Caps-lock and Control, and (at your discretion) switch Option and Command. The latter will make emacs much easier to use as you'll be hitting Option/Meta a lot.*

When complete, now we are done with the preliminaries of configuring our local keyboard mappings.

Installing Emacs. Let's now install the latest version of **emacs** using these instructions⁴ for [Ubuntu](#):

```
1 sudo apt-add-repository -y ppa:cassou/emacs
2 sudo apt-get update
3 sudo apt-get install -y emacs24 emacs24-el emacs24-common-non-dfsg
```

Emacs basics. To get started with **emacs**, watch the video lecture for this section for an interactive screencast. This shows you how to launch and quit, and how to start the online tutorial. Notice in particular the command combinations streaming by in the lower right corner. The first thing you want to do is go through M-x **help-with-tutorial** and start learning the keyboard shortcuts. With **emacs**, you will never again need to remove your hands from the touch typing position. You will also want to keep this [reference card](#) handy as you are learning the commands.

Emacs and the node.JS REPL. Now we get to the real payoff of doing all this setup. The real win of **emacs** comes from the ability to run an interactive REPL next to your code on any machine. First, execute the following commands to download some emacs-specific configuration (see Scripting the setup of a developer environment).

```
1 cd $HOME
2 git clone https://github.com/startup-class/dotfiles.git
3 ln -sb dotfiles/.screenrc .
4 ln -sb dotfiles/.bash_profile .
5 ln -sb dotfiles/.bashrc .
6 ln -sb dotfiles/.bashrc_custom .
7 mv .emacs.d .emacs.d~
8 ln -s dotfiles/.emacs.d .
```

The configuration files we just symlinked into ~/.emacs.d enable us to write and debug JS code interactively. Let's try this out with the Fibonacci code from last time:

```
1 wget https://spark-public.s3.amazonaws.com/startup/code/fibonacci.js
2 emacs -nw fibonacci.js
```

When in emacs, hit C-c! to bring up a prompt in the *js buffer. Then hit C-cC-j to send the current line from the **hello.js** buffer to the prompt in the *js* buffer, or select a set of lines and hit C-cC-r to send an entire region. In this manner you can write code and then

⁴If you want to run emacs locally, on OS X you can install this from [emacsformacosx.com](#). On Windows you can install it from [here](#) or from within the Cygwin [package installer](#); see also [here](#) and [here](#) for more tips.

rapidly debug it⁵ with just a few keystrokes. Watch the video lecture for this section for details.

Emacs and jshint. Javascript is a language that has many idiosyncracies and stylistic warts; Douglas Crockford's book [Javascript: The Good Parts](#) and [JS Wat](#) go through many of these issues. Perhaps the easiest way to confirm that you haven't done something silly (like missing a semicolon) is to integrate `jshint` (jshint.com) into `emacs` as a compilation tool. First let's install `jshint` globally:

```
1 npm install -g jshint
```

Using the configuration from the last time, we can now hit `C-cC-u` to automatically check our JS code for errors. Watch the screencast in this section of the lecture to see how to interactively find and debug JS issues within emacs with `jshint`.

Troubleshooting Emacs. Emacs is controlled by Emacs Lisp ([1](#), [2](#)), which is a language in its own right. If you have problems loading up emacs at any time, execute the command `emacs --debug-init`. That will tell you the offending line in your emacs configuration. You can also use `M-x<RET>ielm` to get an interactive Elisp REPL prompt where you can type in commands, `M-x<RET>describe-mode` to see the commands in the current mode, and `M-x<RET>apropos` to search for help on a given topic.

Going further with Emacs. We have only scratched the surface. There are many things you can do with `emacs`, including:

1. *Pushing and pulling with magit:* You can always push and pull to `git` from the command line. But it's a little easier if you use `magit` to do so. See [here](#).
2. *Searching into a function with etags:* Once you get to the point that you have multifile codebases, you will want to jump to the point of a given function definition. You can do this by generating an index file with the `etags` command; see [here](#).
3. *Tab autocompletion and snippets:* With [hippie-expand](#) and [yasnipet](#), you can use tab to autocomplete any symbol which is present in any open buffer. This is fantastic for working with long variable or function names, and reduces the chances of accidentally misspelling the name of a constant.

And these are only for coding; Emacs also has tools like:

- `org-mode` for note-taking and reproducible research.
- `gnus` for rapidly reading email.
- `dired-mode` for directories
- `tramp` for remote access

⁵This can be generalized to any language; indeed, we have code for setting up a similar REPL environment for many other languages and interpreters (R, Matlab, python/ipython, haskell/ghc), using the `emacs comint-mode` features ([1](#), [2](#), [3](#), [4](#)).

- `patch-mode` for looking at diffs

Indeed, virtually every programming task has an Emacs major mode. And if you master `Elisp` (1, 2) you can configure your development environment such that you can do anything in one or two keystrokes from within `emacs`.

Emacs Summary. To summarize, we just explored the basics of how to launch `emacs` at the command line, opening and switching windows, finding online help (with `M-x describe-mode`, `M-x describe-command`, `M-x apropos`), opening/creating/saving files, and the use of `emacs` for editing SSJS with a REPL. You should practice editing files and print the `emacs` [reference card](#) to make sure you master the basics.

git: Distributed Version Control Systems (DVCS)

To understand the purpose of `git` and Github, think about your very first program. In the process of writing it, you likely saved multiple revisions like `foo.py`, `foo_v2.py`, and `foo_v9_final.py`. Not only does this take up a great deal of redundant space, using separate files of this kind makes it difficult to find where you kept a particular function or key algorithm. The solution is something called *version control*. Essentially, all previous versions of your code are stored, and you can bring up any past version by running commands on `foo.py` rather than cluttering your directory with past versions.

Version control becomes even more important when you work in a multiplayer environment, where several people are editing the same `foo.py` file in mutually incompatible ways at the same time on tight deadlines. In 2005, Linus Torvalds (the creator of Linux) developed a program called `git` that elegantly solved many of these problems. `git` is a so-called *distributed version control system*, a tool for coordinating many simultaneous edits on the same project where byte-level resolution matters (e.g. the difference between `x=0` and `x=1` is often profound in a program). `git` replaces and supersedes previous version control programs like `cvs` and `svn`, which lack distributed features (briefly, tools for highly multiplayer programming), and is considerably faster than comparable DVCS like `hg`.

The SHA-1 hash. To understand `git`, one needs to understand a bit about the [SHA-1](#) hash. Briefly, any file in a computer can be thought of as a series of bytes, each of which is 8 bits. If you put these bytes from left to right, this means all computer files can be thought of as very [large numbers](#) represented in binary (base-2) format. Cryptographers have come up with a very interesting function called SHA-1 which has the following curious property: any binary number, up to 2^{64} bits, can be rapidly mapped to a 160 bit (20 byte) number that we can visualize as a 40 character long number in hexadecimal (base-16) format. Here is an example using `node`'s [crypto](#) module:

```
1 $ node
2 > var crypto = require('crypto')
3 > crypto.createHash('sha1').update("Hello world.", 'ascii').digest('hex')
4 'e44f3364019d18a151cab7072b5a40bb5b3e274f'
```

Moreover, even binary numbers which are very close map to completely different 20 byte SHA-1 values, which means `SHA-1(x)` is very different from most “normal” functions like

$\cos(x)$ or e^x for which you can expect [continuity](#). The interesting thing is that despite the fact that there are a finite number of SHA-1 values (specifically, 2^{160} possible values) one can assume that if two numbers are unequal, then their hashes are also extremely likely to be unequal. Roughly speaking this means:

$$x_1 \neq x_2 \rightarrow P(\text{SHA-1}(x_1) \neq \text{SHA-1}(x_2)) \geq .9999\dots$$

Here is how that works out in practice. Note that even small perturbations to the “Hello world.” string, such as adding spaces or deleting periods, result in completely different SHA-1 values.

```
1 $ node
2 > var crypto = require('crypto')
3 > var sha1 = function(str) {
4   return crypto.createHash('sha1').update(str, 'ascii').digest('hex');
5 }
6 > sha1("Hello world.")
7 'e44f3364019d18a151cab7072b5a40bb5b3e274f'
8 > sha1("Hello world. ")
9 '43a898d120ad380a058c631e9518c1f57d9c4edb'
10 > sha1("Hello world")
11 '7b502c3a1f48c8609ae212cdfb639dee39673f5e'
12 > sha1(" Hello world.")
13 '258266d16638ad59ac0efab8373a03b69b76e821'
```

Because a hash can be assumed to map 1-to-1 to a file, rather than carting around the full file’s contents to distinguish it, you can just use the hash. Indeed, you can just use a hash to uniquely⁶ identify *any* string of bits less than 2^{64} bits in length. Keep that concept in mind; you will need it in a second.

A first git session. Now let’s go through a relatively simple interactive session to create a small git repository (“repo”). Type in these commands at your Terminal; they should also work on a local Mac, except for the `apt-get` invocations.

```
1 sudo apt-get install -y git-core
2 mkdir myrepo
3 cd myrepo
4 git config --global user.name "John Smith"
5 git config --global user.email "example@stanford.edu"
6 git init
7 # Initialized empty Git repository in /home/ubuntu/myrepo/.git/
8 git status
9 echo -e 'line1\nline2' > file.txt
10 git status
11 git add file.txt
```

⁶Is it really unique? No. But finding a collision in SHA-1 is hard, and you can show that it’s much more likely that other parts of your program will fail than that a nontrivial collision in SHA-1 will occur.


```

12 git status
13 git commit -m "Added first file"
14 git log
15 echo -e "line3" >> file.txt
16 git status
17 git diff file.txt
18 git add file.txt
19 git commit -m "Added a new line to the file."
20 git log
21 git log -p
22 git log -p --color

```

That is pretty cool. We just created a repository, made a dummy file, and added it to the repository. Then we modified it and looked at the repo again, and we could see exactly which lines changed in the file. Note in particular what we saw when we did `git log`. See those alphanumeric strings after `commit`? Yes, those are SHA-1 hashes of the entire commit, conceptually represented as a single string of bytes, a “diff” that represents the update since the last commit.

```

1 ubuntu@ip-10-196-107-40:~/myrepo$ git log
2 commit 5c8c9efc99ad084af617ca38446a1d69ae38635d
3 Author: Balaji S <balajis@stanford.edu>
4 Date: Thu Jan 17 16:32:37 2013 +0000
5
6     Added a new line to the file.
7
8 commit 1b6475505b1435258f6474245b5161a7684b7e2e
9 Author: Balaji S <balajis@stanford.edu>
10 Date: Thu Jan 17 16:31:23 2013 +0000
11
12     Added first file.

```

What is very interesting is that `git` uses SHA-1 all the way down. We can copy and paste the SHA-1 identifier of a commit and pass it as an argument to a `git` command (this is a very frequent operation). Notice that `file.txt` itself has its own SHA-1.

```

1 ubuntu@ip-10-196-107-40:~/myrepo$ git ls-tree 5c8c9efc99ad084af617ca38446a1d69ae38635d
2 100644 blob 83db48f84ec878fbfb30b46d16630e944e34f205 file.txt

```

And we can use this to pull out that particular file from within `git`’s bowels:

```

1 ubuntu@ip-10-196-107-40:~/myrepo$ git cat-file -p 83db48f84ec878fbfb30b46d16630e944e34f205
2 line1
3 line2
4 line3

```

The point here is that `git` tracks *everything* with a SHA-1 identifier. Every alteration you make to the source code tree is tracked and can be uniquely addressed and manipulated with a SHA-1 identifier. This allows very interesting kinds of computations using `git`.

Pushing to Github.com While you can use `git` purely in this kind of local mode, to collaborate with others and protect against data loss you'll want to set up a centralized repository. You can certainly host your own repository, but it will be bare bones and yet another service to maintain. Or you can use one of many `git` hosting services, such as Bitbucket, Google Code, Sourceforge, or Github. The last will be our choice for this class. You can think of github.com as a web frontend to `git`, but it is now much more than that. It's one of the largest open source communities on the web, and the github.com website is a big deal⁷ in its own right. With that background, let's create a web version of our repo and push this content to github. Go to github.com/new and initialize a new repo as shown:

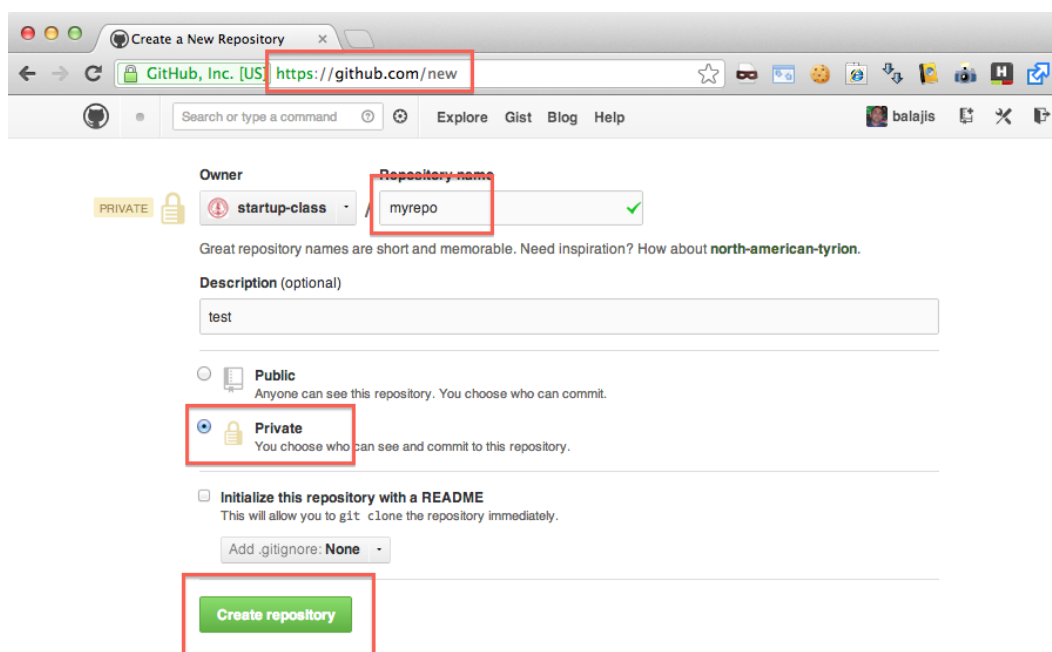


Figure 4: Creating a new repo at github.com/new.

⁷At some point you should go through github's [online tutorial](https://github.com/learn) content to get even more comfortable with `git`; the [learn.github.com/p/intro.html](https://github.com/learn) page is a good place to begin, following along with the commands offline. This will complement the instructions in this section. Once you've done that, go to help.github.com and do the four sections at the top to get familiar with how the github.com site adds to the `git` experience.

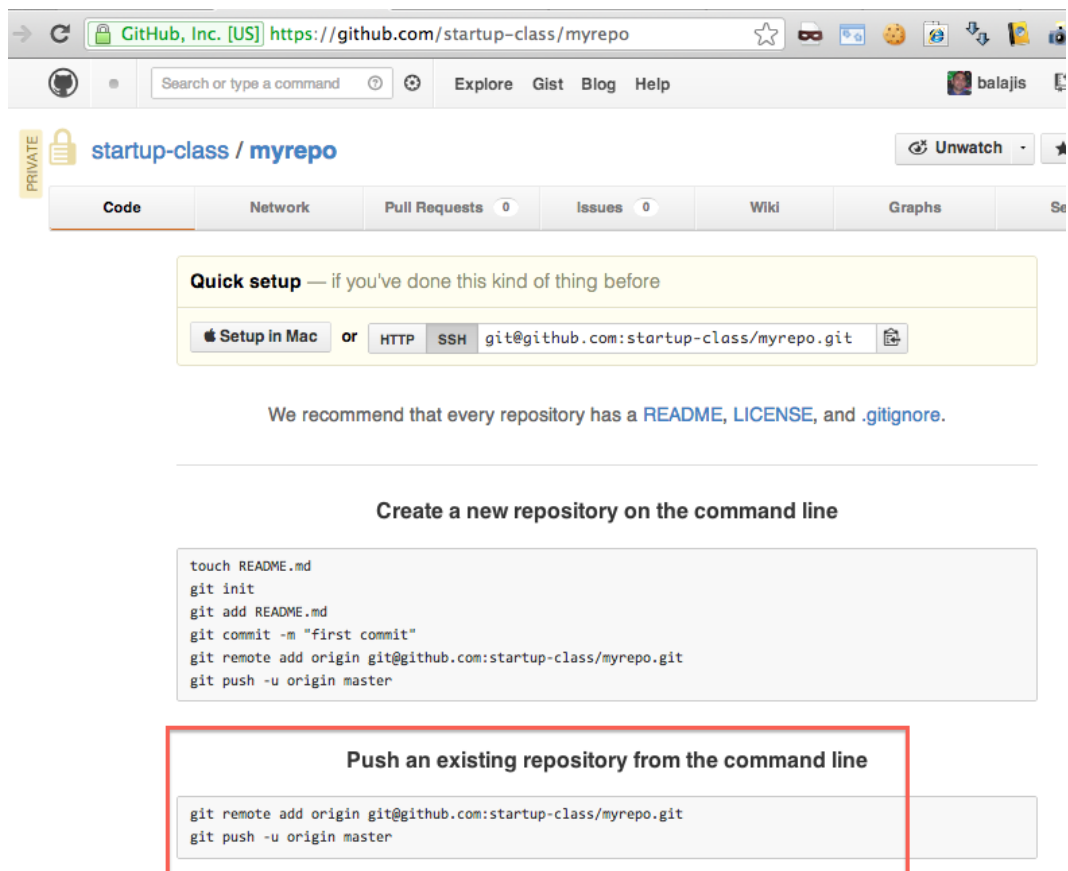


Figure 5: Instructions after a new repo is created. Copy the first line in the highlighted box.

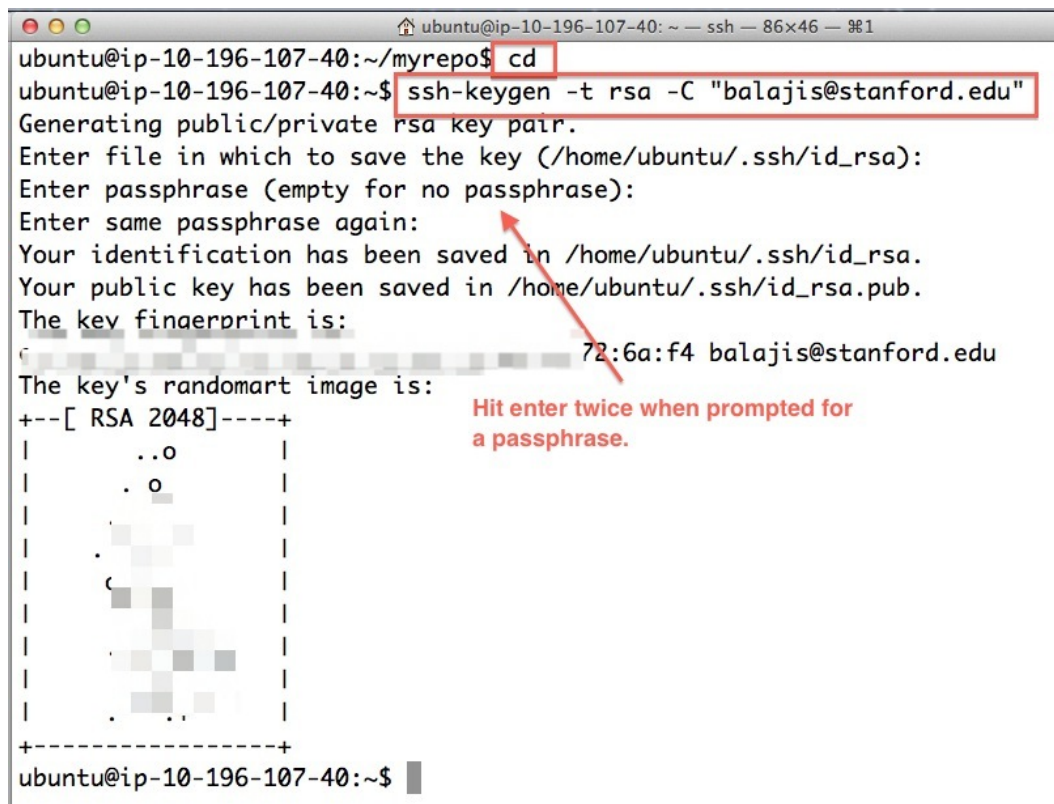
Now come back to EC2 and run these commands:

```
1 git remote add origin git@github.com:startup-class/myrepo.git
2 git push -u origin master # won't work
```

The second command won't work. We need to generate some `ssh` keys so that github knows that we have access rights as [documented here](#). First, run the following command:

```
1 cd $HOME
2 ssh-keygen -t rsa -C "foo@stanford.edu"
```

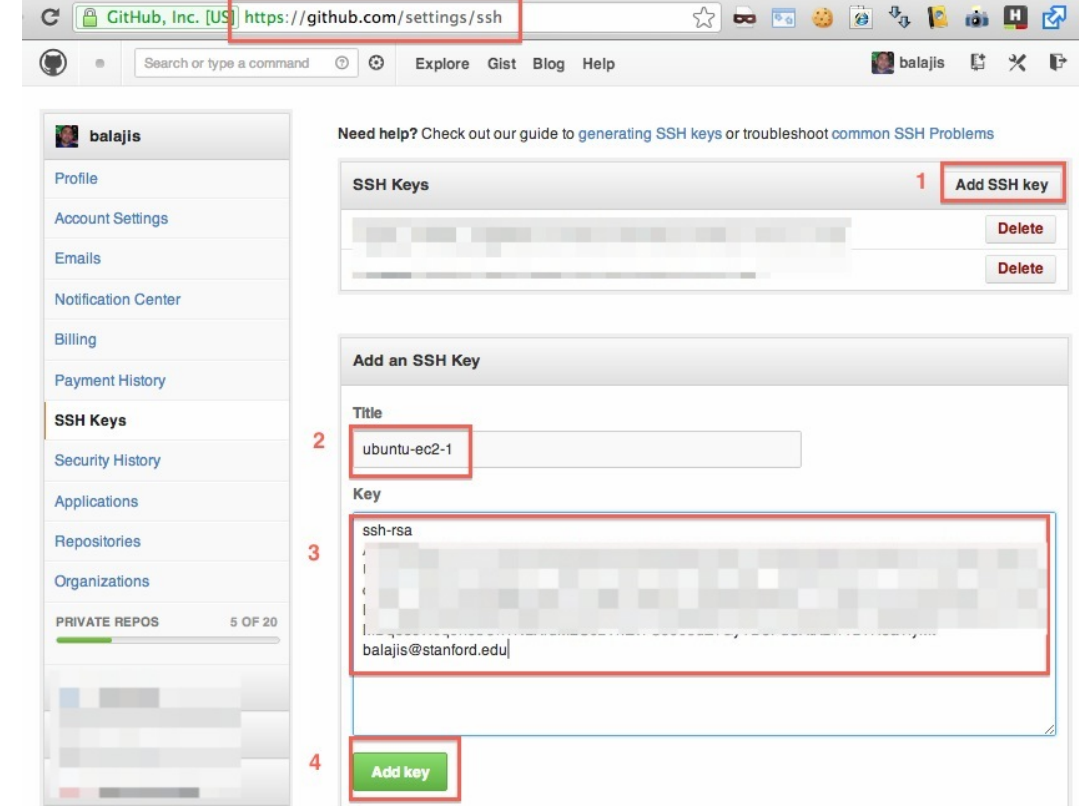
Here's what that looks like:



```
ubuntu@ip-10-196-107-40: ~ — ssh — 86x46 — 81
ubuntu@ip-10-196-107-40:~/myrepo$ cd
ubuntu@ip-10-196-107-40:~$ ssh-keygen -t rsa -C "balajis@stanford.edu"
Generating public/private rsa key pair.
Enter file in which to save the key (/home/ubuntu/.ssh/id_rsa):
Enter passphrase (empty for no passphrase):
Enter same passphrase again:
Your identification has been saved in /home/ubuntu/.ssh/id_rsa.
Your public key has been saved in /home/ubuntu/.ssh/id_rsa.pub.
The key fingerprint is:
72:6a:f4 balajis@stanford.edu
The key's randomart image is:
+--[ RSA 2048 ]-----+
|      ..o             |
|      . o            |
|      .              |
|      .              |
|      .              |
|      .              |
|      .              |
|      .              |
|      .              |
|      .              |
+-----+
ubuntu@ip-10-196-107-40:~$
```

Figure 6: *Generating an ssh-key pair.*

```
ubuntu@ip-10-196-107-40:~$ cd .ssh/
ubuntu@ip-10-196-107-40:~/..ssh$ cat id_rsa.pub
ssh-rsa AAAAB3NzaC1yc2EAAAADAQABAAQDNgDElEraSAZAVrkcj6gYYAnZkEefuMn/V0py6yfl4sdU8+
f
V
Evr3005u24py1b3f45Rt247/121msu1rymc balajis@stanford.edu
ubuntu@ip-10-196-107-40:~/..ssh$
```



Confirm password to continue

Password (Forgot password)

Confirm password

Figure 9: *You will see a prompt for your github password.*

Now come back to the command line and type in:

```
1 ssh -T git@github.com
```

```
ubuntu@ip-10-196-107-40:~/.ssh$ ssh -T git@github.com
Hi balajis! You've successfully authenticated, but GitHub does not provide shell access.
```

Figure 10: *You should see this if you've added the SSH key successfully.*

Now, finally you can do this:

```
1 git push -u origin master # will work after Add Key
```

```
ubuntu@ip-10-196-107-40:~/.ssh$ cd ..
ubuntu@ip-10-196-107-40:~$ cd myrepo/
ubuntu@ip-10-196-107-40:~/myrepo$ git push -u origin master
Counting objects: 6, done.
Compressing objects: 100% (2/2), done.
Writing objects: 100% (6/6), 458 bytes, done.
Total 6 (delta 0), reused 0 (delta 0)
To git@github.com:startup-class/myrepo.git
 * [new branch]      master -> master
Branch master set up to track remote branch master from origin.
```

Figure 11: *Now you can run the command to push to github.*

And if you go to the URL you just set up, at `https://github.com/$USERNAME/myrepo`, you will see a git repository. If you go to `https://github.com/$USERNAME/myrepo/commits/master` in particular, you will see your Gravatar next to your commits. Awesome. So we've just linked together EC2, Linux, git, Github, and Gravatar.

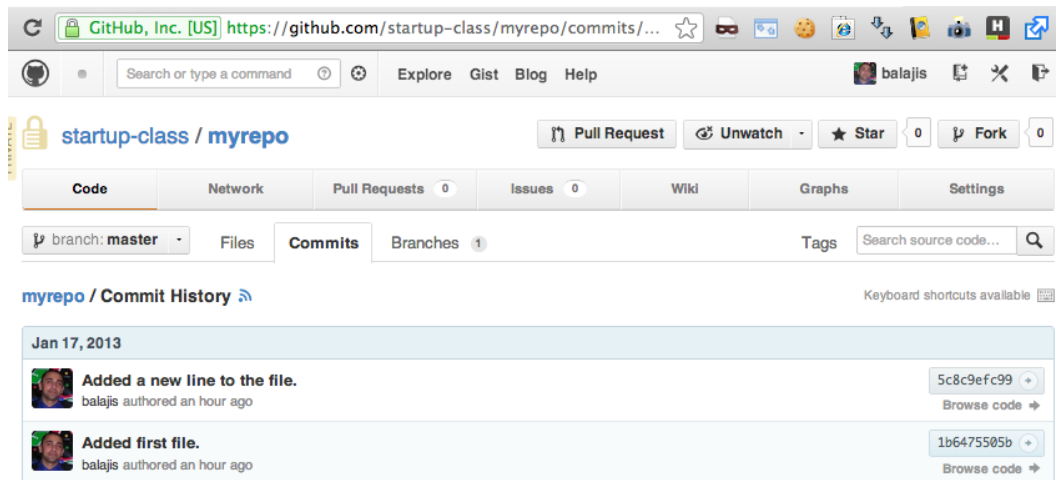


Figure 12: Now you can see your command line commits on the web, and share them with classmates.

Congratulations. That's the basics of `git` right there: creating a repository, adding files, making commits, and pushing to a remote repository. We will be using `git` quite a bit over the course, and will get into intermediate usage soon, but in the interim it will be worth your while to go a bit deeper on your own time. There are a number of excellent references, including the official [book](#) and [videos](#) and this [reference](#) for advanced users.

Managing Setup and Configuration as Code

The concept of `setup.git` and `dotfiles.git`

One of the things you may have started to realize is that it is not completely trivial to set up a working environment in which to write, edit, and debug code. We keep running `apt-get` commands all the time, and we needed to download some files to get `emacs` to work. A key concept is that the shell commands issued during the setup process constitute code in their own right, as do the configuration files used for `bash`, `screen`, `emacs`, and the like. By saving these files themselves in a `git` repository we can rebuild a development environment with one command. Indeed, this idea of scriptable infrastructure is one of the core ideas behind *DevOps* (1, 2), and there are tools like `chef` and `puppet` that facilitate automation of very sophisticated deployments. But we'll start simple with just a shell script.

Scripting the setup of an EC2 instance

Let's try this out with `setup.git` on a fresh EC2 instance.

```
1 cd $HOME
2 sudo apt-get install -y git-core
3 git clone https://github.com/startup-class/setup.git
4 ./setup/setup.sh
```

And just like that, we just ran a script that installed everything we've been using on the local machine.

Scripting the setup of a developer environment

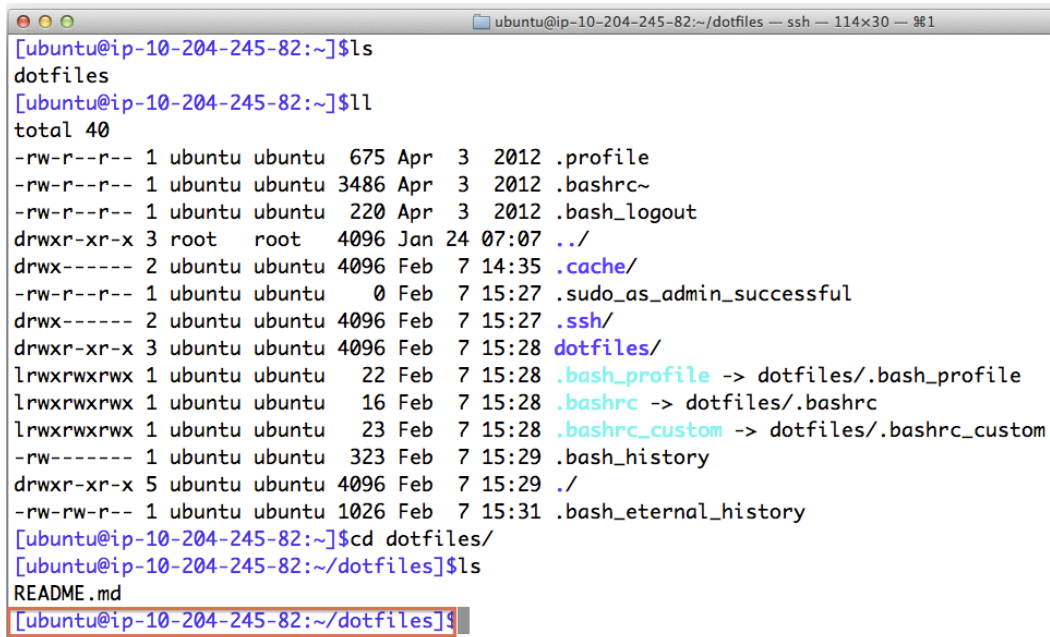
Within `setup.git` are the commands to pull down your developer environment via a second repo called `dotfiles.git`. If you look at `setup.sh`, it calls the commands we ran previously:

```
1 cd $HOME
2 git clone https://github.com/startup-class/dotfiles.git
3 ln -sb dotfiles/.screenrc .
4 ln -sb dotfiles/.bash_profile .
5 ln -sb dotfiles/.bashrc .
6 ln -sb dotfiles/.bashrc_custom .
7 mv .emacs.d .emacs.d~
8 ln -s dotfiles/.emacs.d .
```

To understand what is happening here: the `ln -s` command is used to symbolically link in configuration files to the home directory, and the `-b` flag indicates that we will make a backup of any existing files. Thus `.bashrc~` is created in the directory if a `.bashrc` already exists.. However, the `-b` flag only works on files, not directories, so we need to move `.emacs.d` to `.emacs.d~` with an explicit `mv` command. After executing these commands, if you now log out (by hitting `C-d` or typing `exit` in any directory) and log back in to the instance, you will see your prompt transform into something more useful:

```
ubuntu@ip-10-204-245-82:~$ git clone git://github.com/startup-class/dotfiles.git
Cloning into 'dotfiles'...
remote: Counting objects: 18, done.
remote: Compressing objects: 100% (13/13), done.
remote: Total 18 (delta 3), reused 18 (delta 3)
Receiving objects: 100% (18/18), 9.85 KiB, done.
Resolving deltas: 100% (3/3), done.
ubuntu@ip-10-204-245-82:~$ cd dotfiles/
```

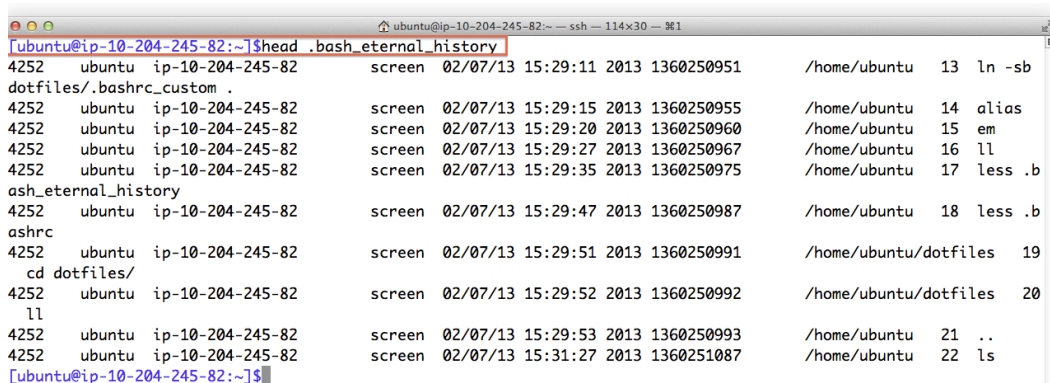
Figure 13: Prompt before `.bashrc` and `.bash_profile` configuration



```
ubuntu@ip-10-204-245-82:~/dotfiles — ssh — 114x30 — %1
[ubuntu@ip-10-204-245-82:~]$ ls
dotfiles
[ubuntu@ip-10-204-245-82:~]$ ll
total 40
-rw-r--r-- 1 ubuntu ubuntu 675 Apr 3 2012 .profile
-rw-r--r-- 1 ubuntu ubuntu 3486 Apr 3 2012 .bashrc~
-rw-r--r-- 1 ubuntu ubuntu 220 Apr 3 2012 .bash_logout
drwxr-xr-x 3 root root 4096 Jan 24 07:07 ../
drwx----- 2 ubuntu ubuntu 4096 Feb 7 14:35 .cache/
-rw-r--r-- 1 ubuntu ubuntu 0 Feb 7 15:27 .sudo_as_admin_successful
drwx----- 2 ubuntu ubuntu 4096 Feb 7 15:27 .ssh/
drwxr-xr-x 3 ubuntu ubuntu 4096 Feb 7 15:28 dotfiles/
lrwxrwxrwx 1 ubuntu ubuntu 22 Feb 7 15:28 .bash_profile -> dotfiles/.bash_profile
lrwxrwxrwx 1 ubuntu ubuntu 16 Feb 7 15:28 .bashrc -> dotfiles/.bashrc
lrwxrwxrwx 1 ubuntu ubuntu 23 Feb 7 15:28 .bashrc_custom -> dotfiles/.bashrc_custom
-rw----- 1 ubuntu ubuntu 323 Feb 7 15:29 .bash_history
drwxr-xr-x 5 ubuntu ubuntu 4096 Feb 7 15:29 ./
-rw-rw-r-- 1 ubuntu ubuntu 1026 Feb 7 15:31 .bash_eternal_history
[ubuntu@ip-10-204-245-82:~]$ cd dotfiles/
[ubuntu@ip-10-204-245-82:~/dotfiles]$ ls
README.md
[ubuntu@ip-10-204-245-82:~/dotfiles]$
```

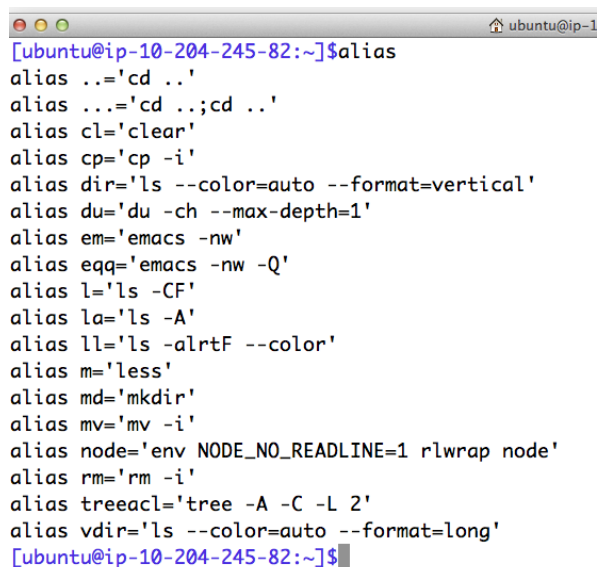
Figure 14: Prompt after `.bashrc` and `.bash_profile` configuration

Note that the username, hostname, and current directory are now displayed in the prompt. This provides context on your current environment. Go back and reread previous lectures if you need to refresh on bash. In addition to this, we have defined a `.bash_eternal_history` file in your home directory which will record all past commands, along with a few useful aliases:



```
ubuntu@ip-10-204-245-82:~$ head .bash_etcnal_history
4252  ubuntu  ip-10-204-245-82      screen  02/07/13 15:29:11 2013 1360250951    /home/ubuntu 13 ln -sb
dotfiles/.bashrc_custom .
4252  ubuntu  ip-10-204-245-82      screen  02/07/13 15:29:15 2013 1360250955    /home/ubuntu 14 alias
4252  ubuntu  ip-10-204-245-82      screen  02/07/13 15:29:20 2013 1360250960    /home/ubuntu 15 em
4252  ubuntu  ip-10-204-245-82      screen  02/07/13 15:29:27 2013 1360250967    /home/ubuntu 16 ll
4252  ubuntu  ip-10-204-245-82      screen  02/07/13 15:29:35 2013 1360250975    /home/ubuntu 17 less .b
ash_etcnal_history
4252  ubuntu  ip-10-204-245-82      screen  02/07/13 15:29:47 2013 1360250987    /home/ubuntu 18 less .b
ashrc
4252  ubuntu  ip-10-204-245-82      screen  02/07/13 15:29:51 2013 1360250991    /home/ubuntu/dotfiles 19
cd dotfiles/
4252  ubuntu  ip-10-204-245-82      screen  02/07/13 15:29:52 2013 1360250992    /home/ubuntu/dotfiles 20
ll
4252  ubuntu  ip-10-204-245-82      screen  02/07/13 15:29:53 2013 1360250993    /home/ubuntu 21 ..
4252  ubuntu  ip-10-204-245-82      screen  02/07/13 15:31:27 2013 1360251087    /home/ubuntu 22 ls
ubuntu@ip-10-204-245-82:~$
```

Figure 15: *Bash eternal history*



```
ubuntu@ip-10-204-245-82:~$ alias
alias ..='cd ..'
alias ...='cd ../;cd ..'
alias cl='clear'
alias cp='cp -i'
alias dir='ls --color=auto --format=vertical'
alias du='du -ch --max-depth=1'
alias em='emacs -nw'
alias eqq='emacs -nw -Q'
alias l='ls -CF'
alias la='ls -A'
alias ll='ls -alrtF --color'
alias m='less'
alias md='mkdir'
alias mv='mv -i'
alias node='env NODE_NO_READLINE=1 rlwrap node'
alias rm='rm -i'
alias treeacl='tree -A -C -L 2'
alias vdir='ls --color=auto --format=long'
ubuntu@ip-10-204-245-82:~$
```

Figure 16: *List of aliases*

You can define more aliases in your `~/.bashrc_custom` file. Feel free to fork the `dotfiles.git` repository into your own github account and edit. The last thing to note is that we added these lines to your `~/.bashrc`:

```
1 # 2.6) node.js nvm
2 # http://nodejs.org/api/repl.html#repl_repl
3 # sudo apt-get install -y rlwrap
4 alias node="env NODE_NO_READLINE=1 rlwrap node"
5 alias node_repl='node -e "require(\'\''repl\'\'').start({ignoreUndefined: true})"'
6 export NODE_DISABLE_COLORS=1
7 if [ -s ~/.nvm/nvm.sh ]; then
8     NVM_DIR=~/.nvm
9     source ~/.nvm/nvm.sh
10     nvm use v0.10.12 &> /dev/null # quiet the nvm use command
11 fi
```

The `alias` line in this configuration snippet redefines the `node` command to use `rlwrap`, which makes it much easier to retrieve and edit previous commands in `node` ([see here for more](#)). It also defines a `node_repl` convenience command which turns off the default `node` practice of echoing `undefined` to the prompt for commands that don't return anything, which can be a bit distracting. The other lines turn off coloration in the `node` interpreter and automatically use `nvm` to initialize the latest version of node.

As an exercise for the reader, you can [fork](#) and update your `dotfiles.git` to further customize your dev environment. And you can even further automate things by using the [EC2 command line tools](#) ([zip](#)) from AWS, which in conjunction with a [user-data-script](#) would let you instantiate an EC2 instance from your local Terminal.app (OS X) or Cygwin (Windows) command line rather than going to `aws.amazon.com` and clicking "Launch Instance". But at this point you get the idea: any setup or configuration of a machine is itself code and should be treated like code, by keeping the install scripts and config files under version control in a `git` repository.

Summary

In this lecture you set up a development environment on Ubuntu using `emacs` and `screen`, learned `git` and pushed some of your first repos to github, and learned how to orchestrate a reproducible dev machine setup.