# COMP XXX Tools of the Trade Documentation

***Release 0.1***

**Peter Parente**

**Sep 02, 2017**

# Contents

> "If the only tool you have is a hammer, you tend to see every problem as a nail."
>
> —Abraham Maslow

# CHAPTER 1

## The Gist

We met most Fridays in the Spring 2014 semester at UNC-CH to practice using a variety of software development tools to build our experience and confidence. This site captures all of the prepatory materials (e.g.,narrated screencasts) and coding exercises we used.

We don't have a solid plan at the moment about hosting TotT again at UNC. If you're interested in helping out, please speak up in our Gitter chat channel.

CHAPTER 2

# What

Tools of the Trade (TotT) is a recurring meet-up for students who want more practice finding, learning, applying, and evaluating tools used in modern software development. The goal is to build experience and confidence in a friendly, fun, collaborative environment. Every week we will:

1. Pick a tool or topic of interest (e.g., Git, Ruby, Backbone, Pandas, BDD, ...)
2. Do a bit of background reading or video watching as time permits to prep for our meeting.
3. Meet face-to-face to hack on practice problems and small projects together.
4. Help one another and share our experience in-person and online.

We currently have meet-ups planned on the topics listed in the navbar with many more possibilities for future sessions. Content and pages will continue to appear over time.

## Setting Up

You need to prepare your laptop to use the numerous tools in our planned meet-ups. To do so, read this page and follow its instructions, step-by-step. You will find a a verification procedure at the end of the document that will ensure that you have a working development environment.

At a glance, you will configure all of the following. Don't download these now, just glance at the list to get a sense of what you're setting up.

1. *tottbox* (v2013-12-31), an Ubuntu virtual machine (VM) prepared to run all of our tools
2. VirtualBox (v4.3.6) to run *tottbox*
3. Vagrant (v1.4.1) to control *tottbox*
4. Git for version control
5. A GitHub account for code backup, collaboration, and submission
6. SublimeText for code editing (optional)
7. Package Control for SublimeText and some useful extensions (optional)

8. Google Chrome for web development (optional)

Some of the steps will vary depending on your operating system (e.g., Windows, Mac, Linux). Make sure you follow the appropriate instructions.

---

**Note:** If you are following these steps during one of our meet-ups, you can save time by borrowing a thumbdrive with all of the required installers (Windows and OS X, sorry Linux users) and VM images on it. If you do, copy all of the files from the thumbdrive to a temporary folder somewhere on your laptop. Then use them in place of fresh downloads throughout this document.

---

## VirtualBox

VirtualBox is an open source virtualizer, an application that can run an entire operating system within its own virtual machine. For instance, you can create a virtual machine running Ubuntu Linux and bring up that machine right on your Windows 7 desktop. There are many interesting uses and advantages of virtual machines, two of which will benefit us greatly:

1. A common, consistent environment for running code and tools
2. The ability to "reset-to-zero" at any time

Do the following to install VirtualBox 4.3.6, the latest stable version tested for this meetup.

1. Download the installer for your laptop operating system using the links below.
    - VirtualBox 4.3.6 for Windows hosts
    - VirtualBox 4.3.6 for OS X hosts
    - VirtualBox 4.3.6 for Linux hosts (requires that you pick your distro)
2. Run the installer, choosing all of the default options.
    - Windows: Grant the installer access every time you receive a security prompt.
    - Mac: Enter your admin password.
    - Linux: Enter your root password if prompted.
3. Reboot your laptop if prompted to do so when installation completes.
4. Close the VirtualBox window if it pops up at the end of the install.

## Vagrant

Vagrant is an open source command line utility for managing reproducible developer environments. While we could use the VirtualBox GUI to juggle virtual machines, their settings, and their distribution, Vagrant hides the complexity as you'll see in the next section.

First, however, you need to install Vagrant 1.4.1, the latest stable version tested for this meetup.

1. Download the installer for your laptop operating system using the links below.
    - Vagrant 1.4.1 for Windows hosts
    - Vagrant 1.4.1 for OS X hosts
    - Vagrant 1.4.1 for Linux hosts (requires that you pick your distro)
2. Run the installer, choosing all defaults.

3. Reboot your laptop if prompted to do so when installation completes.

### SSH for Windows Users

If you are running Windows on your laptop and have not installed Cygwin or the like, you'll need to perform a few additional steps before Vagrant will be useful to you. Namely, you need to get a command line SSH (Secure SHell) client in order to connect to the virtual machine running on your laptop.

Installing Cygwin just to get SSH is overkill for our needs. A lower-overhead solution is to install git for Windows. This Windows installer includes a few common Unix command line utilities including the necessary `ssh`. (We're not actually going to use git on Windows: we just want this package for its bundled copy of ssh.)

1. Visit http://git-scm.com/download/win.
2. If the installer does not download automatically, click to download it.
3. Run the installer.
    - Choose the defaults **until prompted about adjusting your PATH.**.
    - Pick *Run Git and included Unix tools from the Windows Command Prompt*.
    - Continue choosing defaults until the installer completes.

## tottbox

With VirtualBox and Vagrant installed, you're now ready to bring up the virtual machine (VM) running we'll be using in our meet-ups, affectionately named *tottbox*. The VM has Ubuntu Linux Server 12.04 installed along with many of the tools we need to boostrap our explorations. It will be our common developer environment.

---

**Note:** To make it clear where we are running commands, from now on this doc will call the operating system running on your laptop the *host box* and the virtual machine, *tottbox*.

---

1. Open a terminal window.
    - Windows: In the Start Menu, search for and run the Command Prompt application (cmd.exe). If you have Cygwin installed, you can run the Cygwin Bash Shell instead.
    - Mac: Run Terminal in the Applications folder.
    - Linux: You know what to do.
2. Create a folder that will serve as the container for all of your practice work on your host box. In these instructions, we'll call this the `tott_dir` from now on. Enter the following to create the folder.
    - Windows: `mkdir \Users\your_username\projects\tott`
    - Mac/Linux: `mkdir -p ~/projects/tott`
3. Switch to the folder you just created in the terminal.
    - Windows: `cd \Users\your_username\projects\tott`
    - Mac/Linux: `cd ~/projects/tott`
4. Download the TotT Vagrantfile, a config that tells Vagrant how to run *tottbox*. Do not give the file any extension if your browser prompts you for a download file name.

5. Double-check the file name after downloading. Strip any `.txt` or other extension your browser gives to the `Vagrantfile`. You can do this using the Windows/OSX/Linux desktop environment or when you move the file via the terminal in the next step.
6. Place the `Vagrantfile` in the `tott_dir` you created. You can do this by downloading it directly to `tott_dir`, using the Windows/OSX/Linux desktop environment to drag/drop it in, or using the move command in your terminal. For example:
   - Windows: `move \Users\your_username\Downloads\Vagrantfile \Users\your_username\projects\tott`
   - Mac/Linux: `~/Downloads/Vagrantfile ~/projects/tott/`
7. If you copied files off the borrowed thumbdrive, copy the file ending in `.box` to the `tott_dir` as well.
8. If have **not** borrowed the thumbdrive, pause here until you have a stable Internet connection and time to leave your laptop downloading the *tottbox* virtual machine image (~700 MB) in the next command.
9. Enter the following command: `vagrant up`. You **must** be connected to the Internet whenever you issue this command.
   - Vagrant will download the *tottbox* virtual machine image or copy it off from `tott_dir` for safe keeping.
   - It will make a hidden copy of the image in the folder you created.
   - It will launch and configure an instance of the virtual machine.
   - After some log messages and scary looking (but OK!) text, Vagrant returns you to the command prompt.
10. Type `vagrant ssh` in the terminal.
11. After a moment, you should land at a prompt like `vagrant@tottbox:~$`.

---

**Important:** On Windows, if you get an error about the VM being halted right after bringing it up, you likely need to enable support for virtualization on your laptop. This involves rebooting it, going into the BIOS setup, and finding the setting that says something like "Enable Virtualization Support". Unfortunately, the steps for doing this vary widely across machines. Try to look for it, but ask for help if you can't find it.

---

### The *tottbox* shell

You are now in a shell running on your copy of *tottbox*. When you `vagrant ssh`, you are in the home directory of the `vagrant` user on the virtual machine. You can change to other directories using the shell command `cd` and list the contents of directories using the command `ls`. (We'll cover these command and others in the sesson on *Bash, Screen, Vi, and SSH*).

Leave this shell open for the remainder of the steps in this tutorial. If you close your laptop or reboot it, you can reconnect to *tottbox* by opening a new terminal, returning to `tott_dir` using the `cd` command, typing `vagrant up`, and then running `vagrant ssh`.

If you want to explore, feel free. Anything you do on the VM file system is temporary. You can reset your *tottbox* at any time by running `vagrant destroy` followed by `vagrant up` on your host box.

There is one exception to the reset rule: the `/vagrant` directory on *tottbox* is a synchronized mirror of the `tott_dir` in which you ran `vagrant up` on your host box. Anything you do in `/vagrant` on the VM will also happen in the corresponding folder on your host box. Likewise, anything you do in the `tott_dir` on your host box will appear in the `/vagrant` folder on *tottbox*. **This feature is critical**: it will allow us to edit code and view web apps in our desktop environment, but run them in the stable *tottbox* environment. You'll get to see this in action in a few minutes down below.

## git

Git is an open source, fast, modern distributed version control system. Many high-profile projects have adopted Git for version control, and, according to the GitHub stats quoted on the front page of this site, many more are starting life in Git. We will practice using Git in almost everything we do.

Right now, you just need to tell Git who you are before we proceed. In the *tottbox* terminal, enter the following commands, replacing my name and email address with your own.

```
git config -f /vagrant/.gitconfig user.name "Peter Parente"
git config -f /vagrant/.gitconfig user.email "parente@cs.unc.edu"
```

This information will appear on all code changes you make. Make sure it is accurate.

## GitHub

GitHub and BitBucket are two sites offering version control as a service. GitHub is by far and away the most popular site for social coding, but BitBucket offers unlimited private repositories to users with academic email addresses (i.e., you). Since we're not concerned about keeping our practice code private, we will focus on GitHub. But keep in mind you can get free, private hosting on BitBucket if you need it for other course work.

1. Visit the GitHub home page.
2. Click Sign up for GitHub.
3. Enter the required information.

At this point you've got a GitHub account, but no way to push code to it for version control. To finish the setup, you need to create a public-key pair. You will store the public half of the key on GitHub and keep the private half local for use in your *tottbox*.

1. Click the Account settings (tools icon) in the top right.
2. Enter your first and last name at least.
3. Click SSH keys on the left.
4. Click Add SSH key.
5. Enter *tottbox public key* in the Label field.
6. Switch to your *tottbox* terminal and enter the following commands in the *tottbox* shell.

```
mkdir -p /vagrant/.keys
cd /vagrant/.keys
ssh-keygen -f /vagrant/.keys/github
```

8. When prompted, enter a password of your choosing to protect the key pair. You'll only need to enter it once each time you bring up a new *tottbox* instance, so giving it a password is not painful and it's The-Right-Thing-To-Do (TM).
9. Run `less github.pub` in the *tottbox* terminal.
10. Copy the entire output, the public key, to the clipboard.
11. Back on the GitHub site, paste the entire output into the Key field.
12. Click Add key.

Your GitHub account is now ready for use. We'll test it in a few minutes to confirm your environment is configured properl. For the moment, check that the `/vagrant` directory on your *tottbox* has the proper files.

1. Run the command `find /vagrant` in the *tottbox* terminal.
2. Verify the output looks something like the following. (It's OK if there are other files too.)

```
vagrant
vagrant
vagrant/Vagrantfile
vagrant/.keys
vagrant/.keys/github.pub
vagrant/.keys/github
vagrant/.gitconfig
```

---

**Note:** Typically, keypairs live in a `.ssh` directory in your home folder. We deviate from the norm here because we want our keys to continue to exist even if we destroy and recreate *tottbox*. So, instead, we store the keys in the `/vagrant` folder which keeps them synced with our host box. When the you run `vagrant up` a little script copies the keys from the `/vagrant/.keys` folder into the right location in your *tottbox* instance.

Vagrant does support agent forwarding which would allow us to store the keys more securely on our host box. Setting up forwarding is a bit of a pain on some OSes, however, so we'll stick with the sync'ed folder approach.

---

## SublimeText (Optional)

SublimeText is a cross-platform programmer's text editor with a powerful extension system. To get a sense of what it can do, visit http://www.sublimetext.com/, watch the animation on the front page, and read some of the features further down the page. While I will not go so far as to require that you use a particular editor, I highly recommend it. I've been through Emacs, Vim, Eclipse, TextMate, and others: I've been the most productive with Sublime.

1. Visit the SublimeText home page.
2. Click the download link for your operating system below the animation or visit the Download tab.
3. Install SublimeText.
    - Windows: Double-click the downloaded installer and follow its instructions.
    - Mac: Double-click the downloaded disk image and drag SublimeText to your Applications folder.
    - Linux: `tar xjf Sublime*.bz2` and make sure the `sublime_text` executable is in your `$PATH`.
4. Run SublimeText.
    - Windows: Click the SublimeText icon in the Start menu.
    - Mac: Double click the SublimeText icon in your Applications folder.
    - Linux: Run `sublime_text` in a terminal in your desktop environment.

Take a few minutes to try some of the features noted on the SublimeText home page before continuing. Pay extra attention to the Goto Anything and Command Palette features.

### SublimeText Package Control

Package Control is an extension for SublimeText that lets you easily install a host of additional extensions from within Sublime.

1. Visit the Package Control home page.
2. Click the Installation tab.

3. Follow the instructions to install Package Control for the version of SublimeText you installed.

Once you have Package Control installed, do the following to install some extensions that will benefit you.

1. Press Ctrl-Shift-P (Windows/Linux) or Cmd-Shift-P (Mac) to open the SublimeText Command Palette.
2. Start typing *install* until Package Control: Install Package is the selected item.
3. Press Enter.
4. Start typing *GitGutter* until that package is selected.
5. Press Enter to install it.

Voila. You've installed a package that can show you which lines in your code you've changed since you last committed your code to version control. (If the last sentence was gibberish, don't fret. We're going to cover version control with git and these extensions will make a lot more sense in context.)

Repeat the procedure you just followed to install GitGutter for the following additional packages:

- SublimeLinter
- SidebarEnhancements
- HTML5

After installing these, take a few minutes to browse the Package Control community repository to get a sense of the tools available.

## Google Chrome (Optional)

The desktop browser scene is not as messy as it was some years back. The big browser vendors are largely converging on a common feature set defined by HTML5, CSS3, and so on. Firefox, Safari, Google Chrome, *Opera*, and even recent versions of Internet Explorer are all fine for browsing the web. Most are pretty good for web development too. I recommend using Google Chrome for its excellent developer tools, but any modern browser should suffice.

1. Download the Chrome installer.
2. Follow the instructions that appear one you accept the license agreement to get it installed.
3. Run Chrome.
    - Windows: Click the Chrome icon in the Start menu.
    - Mac: Double click the Chrome icon in your Applications folder.
    - Linux: Run `chrome` in a terminal in your desktop environment.

Chrome will prompt you to create or login to a Google Account. You do not need to do so for the purposes of our meetings, but you can if you wish.

## Verification

We'll now run a quick test of your environment. We won't test everything, but we will at least kick the tires.

By following these steps, you'll start with a fresh *tottbox* instance, fork the repository I created on GitHub for this test, clone the repository locally, fill in a little README text file template with some basic information, run a test suite I wrote to check your work, commit your changes to the repository, and push the changes back up to GitHub.

Again, don't let the jargon scare you: we're going to get lots of practice using git for version control and cover all of these terms. If you want to jumpstart your understanding, start reading the first two chapters of the Pro Git book and playing with git on *tottbox*.

### Destroy

1. In the *tottbox* terminal, type `exit` to terminate the SSH connection to the `tottbox`.
2. Destroy, rebuild, and then connect to a fresh *tottbox* instance by running the following commands in the `tott_dir` on your host box.

```
vagrant destroy
vagrant up
vagrant ssh
```

3. Enter the passphrases you assigned to the GitHub key you created when prompted on login.

### Create and Clone

1. Visit GitHub and login.
2. Visit https://github.com/parente/tott-verify.
3. Click the Fork button.
4. Clone your *tott-verify* fork for local editing with the following commands on *tottbox*, replacing `your_username` with your GitHub username.

```
cd /vagrant
git clone git@github.com:your_username/tott-verify.git
```

### Edit and Test

1. Open SublimeText on your host box.
2. Use it to open the README.md file in the `tott-verify` directory git created in the `tott_dir`.
    - On Windows, if you followed my `tott_dir` suggestion, it's in `\Users\your_username\projects\tott\tott-verify\README.md`
    - On Mac/Linux, if you followed my `tott_dir` suggestion, it's in `~/projects/tott/tott-verify/README.md`.
3. Review the contents of the README.md file.
4. Replace the information about me with the equivalent information about you.
    - If you're using SublimeText and have installed GitGutter, you should see little markers in the left gutter of the editor when you save. These are the lines you've modified in comparison with the latest copy of the README in version control.
    - You don't have to put your real name and email. It's just a test case for pushing code to GitHub.
5. Back at the *tottbox* prompt, do the following to execute a test suite checking the README.md file and *tottbox* environment against the specs.

```
cd /vagrant/tott-verify
behave
```

6. Address any README.md failures reported by fixing your the file until the tests pass.
7. Address any *tottbox* failures by asking for help. (They're probably my bugs, not yours.)

**Note:** For this exercise, specifications and tests are overkill. However, I want you to get a glimpse of behavior-driven development (BDD), a topic we will cover later. If you're curious about what's going on, open `features/*.features` files in your editor and review their contents. They open the associated `features/steps/*.py` files and match up code with specification.

### Commit and Push

1. In the *tottbox* terminal, run the following commands to commit your changes to your local git repository and then push them to the copy of your repository on GitHub.

```
cd /vagrant/tott-verify
git commit -a -m "Replaced user info in README"
git push origin master
```

2. Visit your GitHub dashboard again.
3. Confirm that the front page of your dashboard shows the README with the changes you just made.

### What Happened?

You might wonder what just happened behind the scenes. Here's the gist.

- You destroyed your *tottbox* VM instance and brought up a new one.
- You created a read-write copy, a *fork*, of the read-only parente/tott-verify git repository on GitHub.
- You made a read-write clone of your fork in your `tott_dir` on your laptop for local editing.
- You edited the README.md to note your personal information.
- You ran the test suit I provided to check that your README.md and environment conforms to a simple spec.
- You committed your edits to the README.md in your local clone of the repository.
- You pushed the commit from your local clone up to your fork on GitHub.

Don't worry if the above description leaves you with even more questions. We have an upcoming session on *Git and GitHub*.

## Cleanup

If you borrowed a thumbdrive, you can delete everything you copied to your hardrive, **except the Vagrantfile**. You **can** delete the `.box` file you copied into `tott_dir`. Vagrant has safely stashed it away in its own directory.

## Success

You just setup a virtually indestructible development environment on your laptop with numerous interesting, useful tools pre-installed. Play with it. Break it. Put it back together. Read more about the pieces. Have fun.

We'll exercise all of the pieces during our coming sessions.

# Virtualization and Vagrant

**Author** Peter Parente

## Goals

- *Setup our shared development environment*
- Realize the benefits of a virtual development environment
- Practice managing a dev environment using Vagrant

## Introduction

Vagrant is a tool for creating consistent working environments for teams of software developers using a virtual machine.

To get started, watch the Vagrant slidecast (~20 minutes) describing Vagrant, virtual machines, and their uses. The slidecast includes a live demo of Vagrant basics on side 9 (~10 minutes).

If time permits, review these additional pages:

- Read Hardware virtualization (~10 minutes)
- Read How Vagrant Benefits You (~5 minutes)

## Exercises

You will need to complete the *Setting Up* instructions before you proceed with these exercises.

### Take notes in Gist

GitHub runs a service called Gist. It's useful for storing versioned, plain text files and sharing them online.

As you tackle the exercises in this session, and many others in our meetups, you should post your solutions in gists. I encourage you to share them in our TotT community so you can learn from one another. At the very least, if they're online somewhere, you can always point to them as work you've done or refer to them later as examples.

Vist the Gist site now and play with it for a few minutes to get familiar with it.

### Practice Vagrant commands

On your host machine, open a new terminal window. Change to the `tott_dir` folder you created when you set up *tottbox*, the one containing the Vagrantfile you downloaded. For example, if you used my suggested defaults, you would type the following:

- Windows: `cd \Users\your_username\projects\tott`
- Mac/Linux: `cd ~/projects/tott`

(Hint: After typing a few characters, press the Tab key on your keyboard liberally. The terminal should autocomplete what you are typing. Press it multiple times to cycle through multiple matches.)

Once you're in the proper directory in the terminal, run `vagrant --help` to get a list of commands it supports. Try using the following against the Vagrantfile for *tottbox*, not necessarily in this order:

- `vagrant reload`
- `vagrant halt`
- `vagrant suspend`
- `vagrant resume`
- `vagrant destroy`
- `vagrant up`

What does each do? Document what you find in a gist.

### SSH into *tottbox*

After you finish playing with Vagrant, bring up a new *tottbox* instance and use SSH to connect to it. Refer to the setup instructions for a reminder of how to do it.

### Explore Virtualbox

Vagrant hides much of the complexity of working with a virtual machine (VM). However, it's good to understand a bit more about what's happening under the covers.

Run the Virtualbox desktop application on your machine. Find the *tottbox* instance in the list on the left. Right click it and choose Settings. Poke around in the dialog that appears and try to answer these questions.

- How many processors does *tottbox* have?
- How much RAM does it have?
- How disk space?
- What folders are shared with the VM?

Can Vagrant change any of these? (Hint: Look in the Vagrant documentation.)

### Explain Vagrant networks

NodeJS is a runtime environment for the JavaScript programming language. (It's OK if you don't know JavaScript yet.) The NodeJS package manager lets you install libraries and utilities for your JavaScript programs.

Use the NodeJS package manager to install the `http-server` package globally on *tottbox*. (Hint: `npm` is preinstalled on the VM. Run `npm --help` and/or Google for help on using it.)

Once you have it installed, change directories to `/vagrant` and start `http-server` running in there. Visit http://192.168.33.10:8080/ in your web browser. What do you see? What does *http-server* do? What is 192.168.33.10? Doc what you learn.

### Explain Vagrant ports

Modify the *tottbox* Vagrantfile to forward port 8080 on *tottbox* to port 8080 on your host box, *localhost*. (Hint: Find the Vagrant documentation about config files. It's a one liner.)

After you make the change, destroy the *tottbox* instance and bring up a new one for the change to take effect. (For bonus points: figure out the magic Vagrant command that updates the *tottbox* configuration without destroying the running instance completely.)

Install and run the `http-server` NPM package again. Now visit http://localhost:8080/ in your browser. What do you see? Why? Write it up.

### Try simple provisioning

Modify the *tottbox* Vagrantfile so that it automatically installs `http-server` when you run `vagrant up`. Note and share how you achieve it. (Hint: You need to include something like the commands you've been typing at the prompt in the Vagrantfile somewhere.)

### "Up" multiple boxes

Vagrant can manage and configure multiple boxes from a single Vagrantfile. This feature is handy when you want to simulate a true production deployment, say, where your database runs on one machine, your web server on another, and your job queue on yet another.

Try modifying the *tottbox* Vagrantfile to start and configure a second instance of the *tottbox* image. Share your resulting Vagrantfile in a gist.

### Try complex provisioning

Puppet, Chef, Ansible, and SaltStack are all popular orchestration packages used to configure and manage virtual machines, typically on a large scale. Vagrant ships with plug-ins supporting software provisioning using most of these tools.

Configure one or more of these popular provisioners to install MongoDB on `vagrant up`. What does it take? Why might you use these more advanced tools over simple bash scripts? Document what you find.

## References

**VirtualBox** VirtualBox hypervisor homepage

**Vagrant Documentation** Documentation about the command line tools, builders, providers, configuration, etc.

**Virtual machine** Wikipedia article defining *virtual machine* and how their general implementation

**Vagrantbox.es** A list of base boxes for Vagrant

# Bash, Screen, Vi, and SSH

**Author** Peter Parente

## Goals

- Get comfortable working at the command line
- Gain experience with common Unix command line tools
- Understand the typical use of ssh
- Practice writing simple bash scripts using vi
- Practice using basic features of GNU screen

- Understand pipes and redirects
- Recognize the benefits of the Unix Philosophy

## Introduction

The command line is a means for interacting with a computer by entering simple commands and combining them to express more powerful concepts. Bash is a command line shell for issuing such commands. Vi, Screen, and SSH are all useful programs that operate from the command line.

To get started, watch the Command line interface slidecast (~30 minutes) introducing the Bash shell and its basic concepts as well as the useful screen, Vi, and SSH utilities. The slidecast includes live demos of the following:

- Bash Basics on slide 8 (~6 minutes)
- Bash Basics++ on slide 10 (~4 minutes)
- Vi Simple Editing on slide 17 (~3 minutes)
- Working in GNU Screen on slide 22 (~8 minutes)

When you're comfortable with the basics of the command line, watch the TotT Unix Philosophy screencast (~25 minutes) explaining the design principles behind Unix-like systems and why it behaves the way it does. The slide casts includes live demos of the following:

- Small Superlatives on slide 8 (~3 minutes)
- Text Everywhere on slide 10 (~5 minutes)
- Pipes and Redirects on slide 14 (~8 minutes)

If time permits, review these additional pages:

- Read Learning the Shell "Why Bother?" and "What is The Shell?" sections on linuxcommand.org (~5 minutes)
- Read the vi intro and Interface sections on Wikipedia (~5 minutes)
- Read the GNU Screen Intro and Screen details sections on aperiodic.net (~5 minutes)
- Read Unix Philosophy on Wikipedia (~10 minutes)
- Read Unix Pipeline on Wikipedia (~10 minutes)

## Exercises

You will need to complete the *Setting Up* instructions before you proceed with these exercises. Once you are set up, SSH into *tottbox* using the `vagrant ssh` command from the setup instructions. Then tackle the problems below. Document what you find in a gist and share it with the TotT community later.

### Learn to navigate

The shell prompt is stateful. We often say we are "in" such-and-such a directory and that we "go to" other directories. How we refer to files at the command prompt depends on our current working directory.

Enter the following commands exactly as shown. As you go along, document what each command does and where you wind up as a result.

```
cd /vagrant
ls
ls -l
ls -al
cd ..
pwd
cd .
pwd
cd ~
pwd
ls
ls ..
ls ../..
cd -
pwd
```

### Count on `man`

Run the following command:

```
wc /vagrant/Vagrantfile
```

What does the `wc` command do? What does the output mean? Try `man wc` or Googling.

Now use the `mkdir` command to create a directory `fun` in `~/a/b/c/d/e/f/g/h/i/j/k/l/m/n/o/p`. Where should look for how to do this easily? (Hint: If you're making them one at a time, you're doing it wrong.)

### Run in the background

Run the following commands:

```
cd ~
git clone https://github.com/ether/etherpad-lite.git
cd etherpad-lite
git checkout 1.3.0
bin/run.sh &
```

Wait a bit. When the console finally states "You can access your Etherpad-Lite instance at http://0.0.0.0:9001/", visit http://192.168.33.10:9001 in your web browser. Enter a pad name. Click new pad and enter some text. (Bonus: What happens when you try to access 0.0.0.0? What is 0.0.0.0? Why does it tell you this?)

The ampersand (&) on the last command you entered tells bash to run the command in the background. Control over the terminal returns to you immediately and the command continues to run in the background. Enter `ls` in the console to prove it.

Now type `exit` in the *tottbox* terminal. What do you see in your browser? What does this tell you about background tasks?

Open a new ssh connection to *tottbox*. Run the commands:

```
cd ~/etherpad-lite
screen -S etherpad -d -m ./bin/run.sh
```

Refresh your web browser. What happens? Type `exit` in the *tottbox* terminal. What do you see in the browser? What's different this time?

SSH back into *tottbox* and type `screen -S etherpad -X quit`. Try etherpad in your browser again. What does this command do? Where should you look if you can't figure it out?

### Automate with bash

Start `screen`. Create a second screen window (Ctrl-A, c). Start `vi`. Practice flipping back and forth between the `vi` editor and prompt with the screen hotkey: Ctrl-A, Space.

When you're comfortable, use `vi` to write a script named `etherpad.sh` that automates the cloning and running steps you performed in the last section (without screen or &). Use the terminal in the other screen window to try running your script. Flip back and forth between the two windows to debug any problems.

### Provision on `vagrant up`

Check if you have etherpad running in a screen still using `screen -ls`. If so, kill it before continuing.

Open your `/vagrant/Vagrantfile` in `vi`. Modify it so that when *tottbox* starts, it executes your etherpad clone-and-run script in a screen session. Test to see if it works using the `vagrant provision` on your laptop (**not** on *tottbox*). What does `vagrant provision` do again? When might provisioning be useful?

### Provision from a gist

Revert your Vagrantfile back to its original state. If you destroy it, just download it again from the link in the setup assignment.

Look at the heredoc at the top of the Vagrantfile. What is it doing? What are some pros and cons of this approach?

### Extend the script

Extend your script to support any or all of the following. Share your solutions in your gist.

1. If the etherpad-lite repository already exists, execute `git pull` within it instead of cloning a new copy on top of it. (Hint: Google for "bash file test operator".)
2. Accept one command line argument: a string having value "start" or "stop". Do the right thing for each value, including checking to make sure a etherpad is not already running when starting or stopped when stopping. Some hints:

- Google for "bash command line arguments" or "bash getopts" for help parsing command line options.
- Google for "last command exit code" for help detecting if certain commands worked or failed.

3. Print a short line about how to use your script if the user does not provide the start or stop argument:

```
usage: etherpad.sh [start|stop]
```

### Play with pipes

Install the American wordlist on your *tottbox* like I did in the prep screencast.

```
sudo apt-get install wamerican
```

Now run the following commands and explain what each one computes. (Hints: `man` is your friend. So are experimentation and Google. So is `screen` if you want to flip between help and a prompt.)

```
cat /usr/share/dict/words | cut -c4- | uniq | wc -l
cat /usr/share/dict/words | cut -c2- --complement | uniq | wc -l
```

What other interesting analyses can you perform?

### Generate passwords

The `openssl` tool has a myriad of functions related to encryption. One of its many abilities is the generation of pseudo-random bytes. Try running:

```
openssl rand 10 -base64
```

One use for this ability is the generation of passwords. Say you had to generate a pseudo-random password that was 12 characters long containing only letters and numbers. How would you do it starting from the `openssl` command above? (Hint: Pipe the output to commands that can delete characters from strings and chop them down to the desired size.)

### Inspect logs

The `/var/log/syslog` is the system log for *tottbox*. Have a look at its contents with `less`. It should look something like the following:

```
Aug 23 06:25:01 tottbox rsyslogd: [origin software="rsyslogd" swVersion="5.8.6" x-pid=
↪"791" x-info="http://www.rsyslog.com"] rsyslogd was HUPed
Aug 23 07:08:45 tottbox dhclient: DHCPREQUEST of 10.0.2.15 on eth0 to 10.0.2.2 port 67
Aug 23 07:08:45 tottbox dhclient: DHCPACK of 10.0.2.15 from 10.0.2.2
Aug 23 07:08:45 tottbox dhclient: bound to 10.0.2.15 -- renewal in 35457 seconds.
Aug 23 07:17:01 tottbox CRON[3771]: (root) CMD (   cd / && run-parts --report /etc/
↪cron.hourly)
Aug 23 08:17:01 tottbox CRON[3782]: (root) CMD (   cd / && run-parts --report /etc/
↪cron.hourly)
Aug 23 09:17:01 tottbox CRON[3785]: (root) CMD (   cd / && run-parts --report /etc/
↪cron.hourly)
Aug 23 10:17:01 tottbox CRON[3796]: (root) CMD (   cd / && run-parts --report /etc/
↪cron.hourly)
```

Each row is a log message. Each message has a fixed set of fields. In this case, the fields are date, time, host, process, message text.

Say you wanted to count the number of duplicate entries in the message text field, sort them from most dupes to least, and write the results to a file named `analysis.txt`. What tools could you pipe together to do so? How do you write the results to a file? (Hint: I covered everything you need to cut the lines into fields and count unique values. We didn't talk about how to sort. Take a guess what that tool is called.)

### View and save

Change any of the commands you worked on today to pipe output both to a file and display it in the terminal. (Hint: Google.)

## Projects

If you want to try your hand at something larger than an exercise, consider one of the following.

### Weather on the prompt

Write a Bash script that retrieves weather information from OpenWeatherMap's API and displays it in the terminal. Support the current conditions and forecasts via different command line options. Consider using the jq JSON library to handle the API responses.

### Define input

Write a command line program that reads words on stdin, calls the DuckDuckGo definition API to define each word, and writes them to stdout. Make sure it can be used in conjunction with other tools (e.g., *cat words.txt | define*).

## References

**Learn vim Progressively** "You start by learning the minimal to survive, then you integrate all the tricks slowly."

**The Command Line in 2004** Garrett Birkel's response to Neal Stephenson's 1999 *In the Beginning...was the Command Line* essay, interspersed in the original text

# Git and GitHub

**Author** Peter Parente

## Goals

- Know what version control is
- Realize the benefits of distributed version control
- Know the basics of using git at the command line
- Practice a personal Git workflow
- Practice a collaborative Git workflow
- Get acquainted with GitHub

## Introduction

Git is a tool for version control, a system for recording changes to files over time with the ability to return you to any version. Version control is a cornerstone of modern software development: it enables multiple developers contribute to a project without fear of losing work.

To get started, watch the Git Basics slidecast (~35 minutes) demonstrating the use of Git to track files in version control. The slidecast includes demos of the following:

- Personal, single-branch workflow on slide 14 (~14 minutes)
- Personal, multi-branch workflow on slide 18 (~12 minutes)

Now watch the Social Git slidecast (~40 minutes) showing how Git can enable version control and code sharing across a team. The slidecast contains demos of the following:

- Central Repository Workflow on slide 8 (~17 minutes)

- Pull-Request Workflow on slide 10 (~11 minutes)

If time permits, review these additional pages:

- Read Why Should I Use Source Control? (~5 minutes)
- Read Pro Git Chapter 1, Sections 1.1 through 1.3 (~10 minutes)
- Read the highly opinionated Why Git is Better Than X (~5 minutes)
- Read Pro Git Chapter 5 Section 5.1 (~5 minutes)
- Read Pro Git Chapter 5 Section 5.2 (~30 minutes)

## Exercises

You will need to complete the *Setting Up* instructions before you proceed with these exercises. Once you are set up, SSH into *tottbox* using the `vagrant ssh` command from the setup instructions. Then tackle the problems below. Document what you find in a gist and share it with the TotT community later.

### Immerse yourself

Visit the Git Immersion web site. Starting on lab #3, complete all the labs up through #28, skipping #22 and #23. The labs will take you through cloning, staging, committing, branching, and merging in a practice Ruby project. It will also teach you some handy commands supporting a typical git workflow.

You should perform all the lab exercises in a subfolder under `/vagrant` on *tottbox*. For instance, `/vagrant/version/git_immersion` or similar will work. Remember, this directory also appears on your laptop in the folder you setup for your course work in the first assignment. Feel free to use SublimeText or your preferred text editor to create and edit the files in the tutorials, but execute the git commands in a *tottbox* terminal.

### Diagram git

Draw a little diagram of your interactions with git in the Git Immersion tutorial so far. What effect do your commands have on your working directory, the stage, the master branch, and other branches?

Use plain text to create the diagram or an online tool like actdiag.

### Continue the Immersion

Visit the Git Immersion web site again. Start on lab #29 and work through lab #49. These labs will take you through branching, merging, pulling, and pushing.

### Practice pull requests

GitHub supports the integration manager Git workflow by way of *pull requests*. Practice this model by visiting https://github.com/parente/tott-roster and following the instructions there.

### Update your diagram

Update the diagram you drew in the last class session to include the concepts of pulling and pushing. How do these impact your local repository? The remote one?

### References

**Try Git** Interactive Git tutorial right in your browser

**Ten Git Tutorials for Beginners** A nice top-10 list of Git tutorials to review if you want alternatives to the prep material.

**Interactive Git Cheatsheet** Visualization show what components common Git commands affect

**A Visual Git Guide** Visual reference of how common Git commands work

## JavaScript and NodeJS

**Author** Peter Parente

### Goals

- Learn to read and write JavaScript code
- Recognize the gotchas of JavaScript
- Know what NodeJS is
- Understand how to tap into the NodeJS ecosystem
- Practice writing simple NodeJS applications
- Practice using the Node Package Manager (npm)

### Introduction

JavaScript is the programming language of the web. Web browser support for JavaScript has improved drastically over the past decade, giving developers the ability to create slick frontends GMail, Facebook, Instagram, and Twitter. More recently, thanks to NodeJS, JavaScript has found a foothold in creating applications outside the browser, particularly on web servers.

To get started, watch the JavaScript slidecast (~35 minutes) introducing the JavaScript programming language. The slidecast describes the following language features using short, interactive code samples:

- Syntax on slide 4 (~5 minutes)
- Rich Types on slide 5 (~2 minutes)
- Protypical Inheritence on slide 6 (~5 minutes)
- Closures on slide 7 (~3 minutes)
- Anonymous Functions on slide 8 (~2 minutes)
- Variadic Functions on slide 9 (~2 minutes)
- Working with NodeJS on slide 13 (~7 minutes)

Now watch the JavaScript Ecosystem slidecast (~25 minutes) which demonstrates how to find and install useful JavaScript libraries to save you time in building more advanced NodeJS applications. The slidecast includes demos of the following:

- NPM Basics on slide 6 (~7 minutes)
- package.json Pizzazz on slide 10 (~7 minutes)

If time permits, review these additional pages:

- Skim A Re-Introduction to JavaScript (~15 minutes).
- Read The node.js Community is ... (~10 minutes)

## Exercises

You will need to complete the *Setting Up* instructions before you proceed with these exercises. Once you are set up, SSH into *tottbox* using the `vagrant ssh` command from the setup instructions. Then tackle the problems below. Document what you find in a gist and share it with the TotT community later.

---

**Important:** If you are on a Windows machine, are working in your `/vagrant` folder, and get an error when using `npm` in the exercises below, include the option `--no-bin-links` after the name of the package. See this StackExchange question and answer for an explanation if you're curious.

---

### Guess a number

Write a JS guessing game that picks a random, secret number between 1 and 100, lets the user take up to 5 guesses, and states if the secret number is equal to, higher, lower than a guess. Read input from stdin and print to stdout.

### Guess a word

Install the American English word list in *tottbox* again if you have destroyed the VM since last using it. Remember, it installs into `/usr/share/dict/words`.

```
sudo apt-get install wamerican
```

Now write a JavaScript program that chooses a random word from the list, reveals it one random letter at a time to the user, and asks the user to input guesses.

### Read and visualize

JSON is a lightweight, text-based, human-readable, open data exchange format. It is commonly used for communication between a web server backend and application frontend. It is also a valid subset of the JavaScript language.

Download this list of U.S. capitals to *tottbox*. The list is in JSON format with the capitals sorted east to west by longitude.

Write a JS program to compute the longitude distance between each capital and the next. Output the distances to stdout along with some summary stats for the entire dataset (e.g., min distance, max, mean, median, standard deviation). Try to find a creative way to represent the relative magnitude of the distances in your output.

### Fetch a quote

I Heart Quotes provides a web service for fetching random quotations. You can test it by visiting http://www.iheartquotes.com/api/v1/random in your browser.

Write a JS program that pulls a quote from this site and displays it on stdout in the terminal. (Hint: Look at the http module in the NodeJS standard library, particularly the request() function).

### Serve quotes

Write a tiny web server using the NodeJS `http` module that fetches a quote from I Heart Quotes and returns it to the requesting client. Run the web server in *tottbox* on port 9000 and test it by pointing Google Chrome on your laptop to http://192.168.33.10:9000. (Hint: Google for or look on the NodeJS site for the few lines of code you need to create a web server in NodeJS.)

### Paint a rainbow

Make a new directory in your shared *tottbox* folder. Change to that directory and use `npm` to install the `colors` module locally into that folder.

```
mkdir -p /vagrant/js/rainbow
cd $!
npm install colors
```

Google for `nodejs colors`. Read about the features the module provides and view the examples. Now write a JS program that iterates over all the colors provided and outputs their names in their respective colors.

### Show time til "freedom"

Make another folder and install the `moment` module using `npm`. Look at the university calendar for the date that classes end this semester. Write a JS program using `moment` that output a human friendly description of the time left til classes end. (Hint: Look at `moment.duration` and its functions.)

### Handle args

In the same "freedom" folder, `npm install optimist`. Find its documentation and study the examples. Now use it to add support for command line arguments that let the user specify:

1. The date of interest, with the end of semester date as the default.
2. If the output should be humanized or not, with yes, humanize, as the default.

### Make it repeatable

If you completed the two exerices directly above, your application now depends on `moment` and `optimist`. Write a `package.json` file that installs these prerequisites when you type `npm install`. (Hint: Refer to the interactive package.json cheatsheet).

### Analyze sentiment

Sentiment analysis is an attempt to determine subjective information from text. For example, identifying the *polarity* of a statement, whether it is a positive or negative opinion, has almost become synonymous with with "doing sentiment analysis."

Make another directory and install the `natural` NPM module. Find its documentation, read its summary, and focus on the section about classifiers.

Download the sample movie reviews polariy dataset v2.0 and extract it in the folder you created:

```
cd /vagrant/whatever_folder_you_created
wget http://www.cs.cornell.edu/people/pabo/movie-review-data/review_polarity.tar.gz
tar xzf review_polarity.tar.gz
```

Spend a moment poking around in the contents of the extracted data. Then, write a JS program that reads in 50 positive reviews, 50 negative reviews, and trains a Naive Bayes classifier using them. Use the classifier example in `natural` as a guide. After training the classifier, test the classifier against a few more positive and negative examples from the dataset or your own custom test cases.

### Explore common libs

Use NPM to install the `underscore` and `async` modules, two very popular JavaScript libraries. Read their documentation. Come up with an example of where one or both might be effective. What do the alternatives look like? Why might you prefer use of these libraries?

### Explore node_modules

Install a bunch of modules using NPM. Poke around in the `node_modules` directory. Read about how NPM works on the web. What can you deduce about how NPM and NodeJS manage packages and their dependencies?

## Projects

If you want to try your hand at something larger than an exercise, consider one of the following.

### Markdown slides

Write a utility that can take a Markdown document and convert it into a complete reveal.js slidedeck without forcing the user to write all of the boilerplate. Support slides, subslides, and incremental builds. Decide and document what valid markup will indicate these features.

### .jsjobs cron replacement

Write a JavaScript program that executes a `run()` function exported by any JS module located in a folder named `~/.jsjobs` on an `interval` also exported by each module. Make the program support millisecond intervals to start, but then extend it to support human-readable intervals using a library like Moment.js.

## References

**Eloquent JavaScript** Introduction to programming in JavaScript

**JavaScript on the Mozilla Developer Network** Comprehensive reference for all things JavaScript

**NodeJS Docs** API reference for the NodeJS standard library

**JavaScript Style Guide** A JS style guide from Airbnb

**Principles of Writing Consistent, Idiomatic JavaScript** Another JS style guide

# Express and Jade

**Author** Peter Parente

**Builds-on** *JavaScript and NodeJS*

## Goals

- Know what the Express framework is
- Know what the Jade template language is
- Learn how to bootstrap a basic web application
- Understand Express routes, views, and middleware
- Practice implementing web APIs using Express
- Practice implementing static web UIs using Jade

## Introduction

Express is a JavaScript framework that aids the development of web servers and web APIs. Jade is a templating language for web pages with dynamic values. Together, they enable NodeJS developers to create dynamic web sites.

To get started, watch the Express slidecast (~35 minutes) demonstrating the use of Express to build a simple web application. The slide cast includes live demos of the following:

- Starting an Express Application and Server on slide 5 (~9 minutes)
- Handling Requests and Responding in Express Routes on slide 7 (~9 minutes)
- Using Express Middleware on slide 9 (~10 minutes)

Now watch the Jade slidecast showing the use of Jade to generate web pages with little markup and variable fields. The slidecast includes live examples of all of the following:

- Rendering Static HTML on slide 3 (~3 minutes)
- Embedding JS in Templates on slide 4 (~2 minutes)
- Jade Mixins on slide 5 (~3 minutes)
- Rendering Jade Views in Express on slide 10

If time permits, review these additional pages:

- Read the RESTful Web APIs section of the Representational state transfer (REST) page on Wikipedia. Then skim the rest of the page. (~10 minutes).
- Read Understanding Express.js (~20 minutes).
- Watch Learning the Jade Templating Engine Syntax (~15 minutes). Note the play button and speed controls are in the bottom right of the CSSDeck web app.

## Exercises

You will need to complete the *Setting Up* instructions before you proceed with these exercises. Once you are set up, SSH into *tottbox* using the `vagrant ssh` command from the setup instructions. Then tackle the problems below. Document what you find in a gist and share it with the TotT community later.

### Understand a dead-drop

Imagine you want to build a semi-secure dead-drop web site. Two parties can use your site to exchange private messages without authenticating as long as they both agree upon a secret key ahead of time. For example, Alice and Bob might decide, face-to-face, that Alice will post the first message to Bob under the alias `qwerasidfj98324wer` on the site tomorrow at 11 PM. Alice posts her message to Bob at `http://thesite.com/messages/qwerasidfj98324wer`, includes another secret alias for Bob to use when responding, and a time at which Alias will check for Bob's reply. Bob visits the URL including the agreed upon alias and retrieves Alice's message, at which point it is promptly deleted from the site. Bob posts a message back to Alice at a new alias, including yet another new alias and retrieval time in his message. Message drops continue in the same manner indefinitely. Of course, all traffic to and from the site is encrypted.

In the following exercises, you'll build such a site.

### Bootstrap the code

Express has a command line utility that will lay out a suggested skeleton for an Express-based web application. I used it in the Jade screencast. Use `npm` to install the Express module now (globally or locally, your choice) and then use the `express` command line to generate a new skeleton for your dead-drop. (Hint: Don't forget to run `npm install` in the project skeleton too. See the first demo of the Express slidecast if you don't know what I'm talking about.)

Test that the skeleton works by running `node app.js` in the project directory and visiting http://192.168.33.10:3000 in your web browser.

### Secure the site

The Express skeleton supports HTTP connections out of the box. This setup sends all traffic to and from the web server across the Internet in clear-text. For a dead-drop, this is a big no-no because it allows anyone to sniff the message text on the wire when it is posted or retrieved. (Read about man-in-the-middle-attacks if you are interested.)

Modify the generated boilerplate to use HTTPS instead of HTTP. Google for an example of how to do it or refer to the NodeJS documentation. (Hint: You'll need to generate a self-signed SSL certificate for the site. Again, Google.)

### Add message routes

Look at the application skeleton Express generated for you. Open the `routes/index.js` file. Remove any boilerplate functions from the file. Then define the following functions for creating and retrieving messages.

```
exports.get_message = function(req, res) {
   // TODO: get the message ID from the request, check if we have that message
   // in memory, return it if so or respond with a 404 error if not, delete the␣
↪message
};

exports.post_message = function(req, res) {
   // TODO: get the message ID and text from the request, store the message in memory
   // keyed by the ID
};
```

Now open `app.js` in the root of the project. Look for the calls to register the boilerplate routes. Remove them and register your new functions like so.

```
app.get('/messages/:id', routes.get_message);
app.post('/messages/:id', routes.post_message);
```

Now implement the `get_message` and `post_message` functions as described in the comments. Manually test your POST route using `curl` and some sample data at the command line. For example, to test posting a new message:

```
curl -k -X POST --data-urlencode "message=the cheese flies at midnight; next @12 pm␣
↪tmw under code 123dfjer3" https://192.168.33.10:3000/messages/qwerasidfj98324wer
```

Manually test your GET route by visiting https://192.168.33.10:3000/messages/qwerasidfj98324wer in your browser, replacing the last part of the URL with the message ID to retrieve.

Remember to restart your Express application when you make changes to it. (Hint: Google for ways to automate the restart if it gets tedious.)

### Add message UI

Posting messages using `curl` works, but we can do better. Use Jade to build a view for adding a message under a user-provided ID, mapped to a URL path. Show the UI when the user GETs the home page of the site. In other words, modify the *views/index.jade* to show a form the user can complete to submit a message.

If you plan to use a plain old HTML form, you'll probably want to add a request handler that gets the message ID out of the form data. If you want to keep the web API as it is, you'll need to use some Javascript to either modify the form action or submit the request using AJAX.

### Add linking

Now users can submit messages easily. Retrieving them is a bit clunky. The user has to visit */messages/:id*, read the message, edit the URL in the address bar, and fill out the form.

Instead of simply returning the message text, render a Jade template that shows the message and includes a link back to the home page. Similarly, render a Jade template that reports success after posting a message and includes a link back to the home page.

### Add stats middleware

Implement an Express middleware function that tracks basic site stats in memory. Count the number of messages posted, messages retrieved, attempts to retrieve messages more than once, and any other statistic you find interesting. Make sure none of your statistic reveal unique identifying information about the users of the site, however.

### Add stats UI

Show the stats the dead-drop site has collected when a user visits */stats* or an equivalent resource. Do some basic formatting of the information, say in justified tables, to make it somewhat simple for users to consume. Use Jade to template the page.

### Add expiration

Add an automatic expiration to all messages posted to the site. That is, if Alice posts a message and Bob fails to retrieve it within one hour after a time mentioned in the message, the site should delete the message automatically. If no time is mentioned, the post should self-destruct within one hour of its posting.

You could implement this feature yourself, or you could scour NPM or Google for existing solutions for storing key-value data in memory with an optional expiration. (Hint hint).

## Projects

If you want to try your hand at something larger than an exercise, consider one of the following.

### Improve security

Think about the security flaws of the dead-drop site. What attack vectors exist? Think about how you might improve the security of the dead-drop site without forcing users to authenticate to post or retrieve messages. Document improvements and try to implement them.

For instance, in our running example, Alice and Bob must agree upon an alias to use for the first drop. If this alias is weak and compromised, an impostor might pose as Bob without Alice's knowledge thereafter. Is there a way around this problem?

### Improve UI

Make the dead-drop site easier on the eyes for would-be users. Consider simple styling fixes to start or maybe go as far as using Bootstrap. If you're collecting many stats, improve the way they are rendered as well, perhaps using d3.js or another visualization library.

### Add Features

There are many possibilities for making the dead-drop site more useful. Be creative. Show off.

## References

**Express API** Express API documentation

**Jade Reference** Jade language reference

# Python

**Author** Peter Parente

## Goals

- Gain the ability to read and write Python code
- Recognize Python's strengths as a language
- Understand how to tap into the Python ecosystem
- Practice writing Python
- Practice using pip and virtualenv

## Introduction

Python is language built around clarity, expressiveness, and all around general utility. It is often recommended as a language for beginners, but is equally useful to experienced developers in creating scripts, gluing together large systems, building web sites, and analyzing data.

To get started, watch the Python slidecast (~45 minutes) introducing the Python programming language. The slidecast describes the following language features using short, interactive code samples:

- Syntax on slide 4 (~3 minutes)
- Rich Types on slide 5 (~2 minutes)
- Classical Inheritence on slide 6 (~5 minutes)
- Generators on slide 7 (~3 minutes)
- Comprehensions on slide 8 (~6 minutes)
- Keyword Arguments on slide 9 (~4 minutes)
- Variadic Functions on slide 10 (~3 minutes)
- Working with Python on slide 13 (~9 minutes)

Now Watch the Python Ecosystem slidecast (~45 minutes) which sows how to find and install libraries that can save you time in building more advanced Python applications. The slidecast includes demos of the following:

- Using pip on slide 14 (~8 minutes)
- Using virtualenv on slide 21 (~4 minutes)

If time permits, review these additional pages:

- Skim the Python Wikipedia article (~10 minutes).
- Read Python Enhacement Proposal (PEP) #20: The Zen of Python (5 minutes).
- Read The Basics at learnpython.org and try a few of the exercises (~30 minutes).
- Read the pip package manager Wikipedia article (5 minutes).
- Read the virtualenv front page (~10 minutes).

## Exercises

You will need to complete the *Setting Up* instructions before you proceed with these exercises. Once you are set up, SSH into *tottbox* using the `vagrant ssh` command from the setup instructions. Then tackle the problems below. Document what you find in a gist and share it with the TotT community later.

### Guess a number

Write a Python guessing game that picks a random, secret number between 1 and 100, lets the user take up to 5 guesses, and states if the secret number is equal to, higher, lower than a guess. Read input from stdin and print to stdout.

### Run python -m

Run the following command at the prompt.

```
python -m SimpleHTTPServer
```

What does it output? What is SimpleHTTPServer? Where does it live? What does the `-m` command line flag do? Document what you discover.

### Compare pip and npm

How are `pip` and `npm` similar? Different? What about `pip` plus `virtualenv`? Write down your observations.

### Resize images

Pillow is an image manipulation library for Python.

Create a virtualenv and use `pip` to install Pillow. Write a Python script that uses Pillow to resize all of the images in a directory. Let the user specify the name of the input directory, name of the output directory, and desired width or height of the resulting images on the command line. Maintain the aspect ratio of each image processed. (Hint: Look into the `optparse` or `argparse` packages for Python to help with command line parsing.)

### Bottle it up

Bottle is a web microframework for Python. (It's a single .py file!)

Use `pip` to install Bottle into the same virtualenv you created for the image resize utility. Then define a web service API for your image resize utility. Support upload of images in common formats and specification of the desired width or height while maintaining aspect. For example, I might access your completed service using `curl` like so:

```
curl -X POST -d @myimage.png http://example.com/resize?width=250 --header "Content-
↪Type:image/png"
```

### Show HackerNews

HackerNews is a social new web site for geeks. The site has a web API for hackers to, well, hack.

Use `pip` to install the thekarangoel/HackerNewsAPI, a Python library simplifying access to the HN API. Use it to write a command line tool that can print the top N front page items or newly posted items where the user can specify N.

Add more options to your command line tool as it suits your fancy (e.g., show comments for a given news story and pipe to less for navigation.)

### Spoon the Web soup

BeautifulSoup makes it easy to parse messy HTML markup. It's very useful in screen scraping applications.

Find a public web page of interest to you that provides no clean web API (e.g., http://durhamnc.gov/Pages/NNList.aspx). Using BeautifulSoup, scrape the page for notices. Compare what you extract on the current run to the last run, and notify the user of any differences.

Think about how you might make your site checker run on an interval and notify the user of changes unobstrusively.

### Mechanize the web

Install the `Mechanize` library using `pip`. What does it do? How might it be useful? Build something using it. (Hint: A CLI for your favorite search engine?)

### Explore PyPI

The Python Package Index (PyPI) is host to quite a few libraries. Browse through it. Get a feel for what exists. Pick one or more libraries that interests. Write an example application demonstrating what they do. Write up a little blog post explaining how to use it. Share it with the world.

## Projects

If you want to try your hand at something larger than an exercise, consider one of the following.

### Create *The Daily Dose*

Create a web application that generates spoken summaries of select web sites for users to download for offline listening, say during a commute, while working out, on a bike ride, etc. Allow users to pick what sites they would like included in their summary and in what order.

Don't worry about user customization initially. Offer each visiting user the list of sources, allow him or her to pick and order, generate the summary (perhaps cached on an interval), and offer a link to download it.

Choose a web framework, text-to-speech library, and new sources to support. (Hint: HackerNews please.) Also consider if a text summarizer like sumy would help, depending on the types of sites and pages you choose to summarize.

## References

**PyPI** Official Python package index

**Hitchiker's Guide to Python** Opinionated best-practice guide for Python developers

# BDD and Behave

**Author** Peter Parente

**Builds-on** *Python*

## Goals

- Define test driven development (TDD)
- Define behavior driven development (BDD)
- Understand the use cases for Gherkin
- Understand the red-green-refactor cycle
- Practice reading and writing spec with Gherkin
- Practice BDD with Behave

## Introduction

Behavior driven development (BDD) is a way of writing software. It starts with describing the behavior software in a story-like format. It then proceeds with developers doing the following repeatedly, what is known as the red-green-refactor cycle:

- Writing a failing test case for a behavior (red)
- Implementing the minimum code required for the test to pass (green)
- Refactoring the code to meet project standards (refactor)

There are many libraries supporting behavior- or test-driven development: Cucumber for Ruby, Mocha for JavaScript, easyb for Groovy/Java, etc. In this session, we'll learn using Behave for Python.

To get started, watch the TDD / BDD slidecast (~45 minutes) introducing test- and behavior-driven development using the Gherkin language for specification and Behave for testing. The slidecast includes a demo writing specs, tests, and code for the following:

- Fibonnaci Numbers on slide 13 (~30 minutes)

If time permits, review these additional pages:

- Skim Test-driven Development on Wikipedia. (~20 minutes)
- Skim Behavior-driven Development on Wikipedia. (~20 minutes)
- Read the Gherkin wiki page in the CucumberJS project. (~5 minutes)
- Read the Behave tutorial. (~20 minutes)
- Read Cucumber: When to Use It, When to Lose It and Please don't use Cucumber . (~10 minutes)

## Exercises

You will need to complete the *Setting Up* instructions before you proceed with these exercises. Once you are set up, SSH into *tottbox* using the `vagrant ssh` command from the setup instructions. Then tackle the problems below. Document what you find in a gist and share it with the TotT community later.

### Read the rules of RPSLK

Rock-Paper-Scissors-Lizard-SpocK is an extension of the classic rock-paper-scissors game. Read and understand the rules.

### Gather requirements

Imagine I am paying you to implement a version of this game in Python. The game should declare a winner of the best of 5 rounds. In each round, the game should prompt the user to choose a gesture. The game should then generate a random gesture of its own and report the winner of the round, computer or human. If the round results in a tie, it should not count toward the 5 round limit and should be replayed. The game should announce the winner as soon as the victory is decided (e.g., the first player to win 3 rounds wins the game).

The game should work at the command line for now. Later, I might decide to port it to the web. I need you to design the game to make this port easy. I ask that you separate the game logic (model) from the CLI (view and controller). Two separate modules, `model.py` for the game logic and `rpskl.py` for the CLI the user runs, should do the trick.

### Bootstrap the project structure

Create the following simple project layout on disk in new shared folder called `rpslk`:

```
rpslk/
    features/
        steps/
            model.py
            cli.py
        cli.feature
        environment.py
        model.feature
    model.py
    rpslk.py
```

Initialize a git repository there and git commit the empty files as a seed. Commit your work as you move through these exercises.

### Write model.feature

Given the requirements above, write the Gherkin feature file for the game model including the scenarios below. Use the three completed scenarios, the examples in the Behave documentation, the feature tests you ran when setting up tottbox, and any other Gherkin examples you can find on the web as references. Don't spend too much time on them now as you will tighten them up when you start writing the test code.

```
Feature: RPSLK game logic

    Scenario: User inputs a supported gesture RPSLK
        Given a user gesture
        When the game processes the round
        Then it returns the result of the round

    Scenario: User beats computer in a round
        # TODO

    Scenario: Computer beats user in a round
        # TODO

    @wip
    Scenario Outline: User and computer tie in a round
        Given the user gesture <user_gesture>
        And the computer gesture is the same
        When the game processes the round
        Then it reports the result as a "tie"

        Example: Gestures
            | gesture |
            | rock    |
            | paper   |
            | scissors|
            | lizard  |
            | spock   |

    Scenario: User wins the whole game
        Given the user has won 2 rounds
        And the user gesture is "rock"
        And the computer gesture is "scissors"
```

```
        When the game processes the round
        Then it indicates the user has won the game

    Scenario: Computer wins the whole game
        # TODO
```

Test the syntax of your feature file by doing the following on *tottbox*

```
cd /vagrant/rpslk
behave
```

The command should output your scenario text and mark each one failing because it is not yet implemented. It will also give (poor) code samples you can use to start implementing the test cases. Have a look at them and then move on. (I say poor because behave makes every test step explicit without considering test code reuse. Other libs are better at these suggestions.)

### Test and implement one scenario

Add the following test code to your `features/steps/model.py` file. It completely implements the *User and computer tie in a round* scenario test case. Read the docstrings for each function to get an idea of what is going on.

```
from behave import given, when, then

@given(u'the user gesture {user_gesture}')
def step_impl(context, user_gesture):
    '''
    Store the user's gesture in the context for later steps.
    '''
    context.user_gesture = user_gesture


@given(u'the computer gesture is the same')
def step_impl(context):
    '''
    Dictate that the game Model instance must have a method named
    generate_gesture() that will return the random computer gesture for the
    round. Replace that method here with a function that returns the
    same gesture as the user gesture. This is called "mocking".
    '''
    context.model.generate_gesture = lambda: context.user_gesture


@when(u'the game processes the round')
def step_impl(context):
    '''
    Dictate that the game Model instance must have a method named
    process_round() that takes the user gesture for the round as a parameter.
    Save the return value in the context for later steps.
    '''
    context.result = context.model.process_round(context.user_gesture)


@then(u'it reports the result as a {result}')
def step_impl(context, result):
    '''
    Assert that the result of the round matches what the spec stated should
```

```
    happen.
    '''
    assert context.result == result
```

Notice that `context.model` is assumed to exist. That is, the test steps assume a game model is available for testing. We can ensure this is the case for each scenario by adding the following code to the `features/environment.py` file. (Yes, the *environment.py* file goes in *features/* not in *features/steps*.)

```
from model import Model

def before_all(context):
    context.model = Model()
```

For this import to succeed, you must add a class named `Model` to the `model.py` file in the root of the project. Add the following empty class to that file.

```
class Model:
    pass
```

Now run `behave` in `/vagrant/rsplk`. Notice the lengthy output. Somewhere near the top you should see *When the game processes the round* in red ink and below that a stack trace indicating that the `process_round()` method is missing.

Welcome to the red-green-refactor cycle! You now have a red test. Your goal is to turn it green by fixing the implementation.

Implement the shell of the missing method and run `behave -t @wip` again. If you got the message signature right, that line of text should become green and the next one should show red. If not, the line will remain red but the stack trace will change. Continue in this fashion until the entire scenario is green. (Hint: Implement a `generate_gesture()` method for `process_round()` to invoke and the test to mock. Then add the game logic to compare the user and generated gesture in `process_round()`.)

### Learn about behave options

Have a look at `behave --help`. Investigate the use of tags such as `@wip` and the various formatting options of behave. Customize your future invocations of `behave` to suit your liking.

### Test and implement the other scenarios

(Re)Using the above test steps, the Behave documentation, steps you ran to verify your tottbox setup, and examples you find on the web, test and implement the remaining scenarios. Work each one as a pair: first write the test code, then code the implementation, and then debug the test/implementation pair. When the test passes, move onto the next scenario, refactoring your game or test code when needed.

Don't forget to move the `@wip` to the current scenario you're working or remove it all together when you're done.

### Fill the gaps

Review your game model scenarios, tests, and implementations. Can you think of any other behaviors that your spec should capture or your test cases check? If so, spec, test, and implement them if you haven't already. (Hint: Can anything go wrong?)

### Spec, test and implement the CLI scenarios

At this point, you have an API for the RPSLK game, but you have no user interface. You need to implement the CLI. Write the scenarios, tests, and implementation for the CLI following the pattern you practiced for the game model. (Hint: Keep it simple.)

### Document your experience

What are the pros and cons of behavior-driven development? Test-driven development? Gherkin? When might you follow this process to a T? When might you seek "shortcuts"? What are some alternative workflows you might envision?

## Projects

If you want to try your hand at something larger than an exercise, consider one of the following.

### Compare Behave with unittest

Look into the classic unittest package in the Python standard library. Try porting a few tests to it. What are the differences? When might you use one over the other? Write about it.

### Port it to JavaScript and Mocha

Mocha is a highly popular test framework for JavaScript. It is a unique blend of specification and test implementation that "feels right" in JavaScript.

Port your specs, tests, and implementation from Behave and Python to Mocha and JavaScript. Document your experience. What's different due to language? Library? Test philosophy? What's the same?

## References

**Behave** Behavior-driven development, Python style

**CucumberJS** A port of the Cucumber BDD library from Ruby to JavaScript

**Mocha** Test-driven development, JavaScript style

# Data Science and IPython

**Author** Peter Parente

**Builds-on** *Python*

## Goals

- Stick a definition on "data science"
- Learn about Python libraries for data science
- Understand the usefulness of IPython Notebook

- Practice manipulating data using Pandas
- Practice plotting using matplotlib
- Practice machine learning with scikit-learn

## Introduction

Data Science is the burgeoning study and practice of extracting knowledge from data. It combines ideas and techniques from many fields including machine learning, statistics, mathematics, data warehousing, parallel and distributed computing, visualization, and many others. As the amount of data available to businesses and researchers continues to grow, so does the need for creative teams and powerful tools to draw insight from it.

Python has a large ecosystem of tools relevant to data science. In this session we will look at combining a small but very powerful set of them to explore (Pandas), model (scikit-learn), and visualize (matplotlib) information in a web environment for reproducible research (IPython Notebook).

To get started, watch the IPython Notebook slidecast (~77 minutes) showing the use of these tools in a basic exploration of the wine dataset from the UCI Machine Learning Repository. The slidecast includes demos of the following:

- IPython Notebook basics on slide 4 (~16 minutes)
- Data exploration with Pandas on slide 9 (~14 minutes)
- Graphing with Matplotlib on slide 12 (~19 minutes)
- Classification with scikit-learn on slide 16 (~17 minutes)

You can view and/or download a version of the wine analysis notebook built throughout the slidecast via the IPython Notebook Viewer.

If time permits, review these additional pages:

- Read What is Data Science? at Kaggle. (~1 minute)
- Read the Data Science article on Wikipedia. (~10 minutes)
- Watch the 10-minute whirlwind tour of pandas by Wes McKinney. (~10 minutes)
- Watch the IPython Notebook intro screencast by Titus Brown starting at 2:17. (~6 minutes)
- Skim the introductory IPython Notebooks.
  - Part 1 - Running Code in the IPython Notebook
  - Part 2 - Basic Output
  - Part 3 - Plotting with Matplotlib
  - Part 4 - Markdown Cells
  - Part 5 - IPython's Rich Display System

## Exercises

You will need to complete the *Setting Up* instructions before you proceed with these exercises. Once you are set up, SSH into *tottbox* using the `vagrant ssh` command from the setup instructions. Then tackle the problems below. In this particular session, you can post your IPython Notebooks as gists, view them in NBViewer, and share their NBViewer URLs in the TotT community later.

### Start a notebook

Create a shared *tottbox* folder to store your notebooks and start the IPython Notebook server running like so:

```
mkdir -p /vagrant/notebooks
cd !$
ipython notebook --ip=0.0.0.0
```

Open your web browser and visit it on the *tottbox* IP and port printed, http://192.168.33.10:8888. Create a new notebook in the web UI.

### Load the Tar Heel Reader book data

Gary runs a web site called Tar Heel Reader (THR). It hosts a collection of community-contributed, easy-to-read, accessible books. If you haven't seen it, visit the site now and read a book. In the following exercises, we will analyze a November, 2013 snapshot of the books hosted on the site (approximately 30,000).

THR books reside in a SQL database. To ease our exploration, I've converted the `books` SQL database table into a Pandas DataFrame and serialized it to disk. Download a zipped copy of the DataFrame to your laptop. Unzip it and move the `*.dataframe` object into a shared *tottbox* folder (e.g., putting it in the `notebooks` folder you created is fine for now).

In the notebook you created, import the `pandas` package and load the DataFrame with the code below.

```
import pandas
pandas.read_pickle('thr.dataframe')
```

### Take some basic measurements

With the DataFrame loaded, use the many methods and properties of the DataFrame to explore the data. Try to answer the following questions. (Hint: Use the Pandas documentation. Hit the Tab key repeatedly after `.` or `(` in the notebook for autocompletion and function help.)

- What are the columns in the DataFrame?
- What does each row represent?
- How many total rows are there?
- How many total books are there?
- How many books have been reviewed? Haven't?
- Books are written in how many different languages?
- What is the mean number of pages per book? Median? Minimum? Max? Variance?
- How many different authors have written books?

### Prep words per page (wpp) data

Say we want to understand how the length of the pages in the Tar Heel Reader books have changed or not changed over time. To do so, we first have to chunk the page text into words based on some definition. Choose a definition and write it down in your notebook in a Markdown cell. Then use the `apply` method on the `text` column (a Series) of the DataFrame to do so. Pass it a function that splits each page of text into a list of words according to your definition. Save the return value in a variable called `words`.

After producing the `words` Series, create another series called `wpp`. Use the `apply` method again, but this time compute the number of words per page instead of the words themselves.

### Plot wpp over time

Return to the original DataFrame. Inspect some of its rows using the `head` and `tail` methods. Is it ordered in some way? Write your assumptions in a Markdown cell in your notebook.

Now plot the `wpp` Series you created in the prior step using the `Series.plot` method. The y-axis should represent the number words on a page and the x-axis should represent a page in a book. The pages should be sorted in ascending chronological order as x increases.

Can you spot a trend in the plot? What if you play with the plotting parameters? Try a scatter plot instead? Take Markdown notes in your notebook.

### Plot the rolling, expanding wpp mean

Pandas has quite a few functions for computing *moving statistics*, stats computed over an ordered sample of data. Try using the moving mean function on the `wpp` Series and plot the results. Try a few more times with different parameter values. What does it do? What do you see? Write it in the notebook. (Hint: http://en.wikipedia.org/wiki/Moving_average)

Pandas also has support for *expanding windows*, stats computed over an ordered sample of data up to and including each datum in the order. Try using the expanding mean on the `wpp` Series. Try a few more times with different parameter values. What do you see? Write it in the notebook?

Is there anything interesting to report from these plots?

### Consider pages per book (ppb) over time

Say we now want to understand how the pages per book (ppb) metric varies over time. Prepare a `ppb` Series and study it. Note any interesting findings in your notebook. (Hint: The `DataFrame.groupby` method will get you started with preparing the data.)

### Learn about clustering

THR authors can assign one or more fixed categories to their books. Nothing dictates that books must fit the available categories, and so it's quite possible that additional categories or alternative organization schemes exist. One way to discover such patterns is to cluster books according to some measure of similarity and then simply study the books in a cluster.

The scikit-learn package has many clustering algorithms available. The basic one that we'll use is called k-means clustering. Given an integer `k` number of clusters, k-means will attempt to partition our `n` books so that each book belongs to the cluster with the nearest mean-value for some property of our books. We need to choose a value for `k` and decide what property we'll use to cluster them.

Picking `k` is empirical. We'll try a few values and see what results we get. Deciding what property we'll use to cluster requires more thinking. If we want to discover common themes or topics across books, we might try clustering our books based on their titles. However, we have to remember that THR has books written in many languages. If we try running the clustering algorithm across all books at once, it's not clear how books written in different languages will or will not relate. To simplify our task, we'll focus on books written in English alone for the time being. (We can always try clustering on other languages independently or across languages later.)

### Prep English titles

Use Pandas to get a Series of unique English book titles from the books DataFrame you loaded. This step amounts to a one-liner in which you:

1. Select rows in the DataFrame that have language equal to "en"
2. Select the title column from the remaining rows
3. Drop duplicate titles

Once you have the title Series, you need to transform the titles into feature vectors on which the k-means algorithm can operate. The `sklearn.feature_extraction.text` package has a number of classes that can do this with minimal effort. Add the following imports to your notebook:

```
from sklearn.feature_extraction.text import CountVectorizer, HashingVectorizer,␣
↪TfidfVectorizer
```

Now read the scikit-learn doc about these three classes and use each of them to transform your title Series into a new, independent series: `count`, `hash`, `tfidf`.

Start simply and use defaults where possible. Until you can visualize how the clustering is working, it makes little sense to start turning random knobs.

### Cluster English titles

We'll now run the k-means clustering algorithm over each one of your transformed title Series. The immediate goal is to get a sense of how our choice of parameters affects the ability of k-means to decompose the entire set of books into clusters of books related by title.

Add the following import to your notebook:

```
from sklearn.cluster import KMeans
```

Construct an instance of the class called `km`. Configure it to create 20 clusters. Then `fit` the class to the first of your three title transformation Series, `count`. Once you've fit the model, create a new DataFrame that pairs the human-readable book titles with the assigned cluster IDs like so:

```
# where titles is your untransformed title Series
en_titles = pandas.DataFrame(titles)
en_titles['count_cluster'] = km.labels_
```

Re-fit the `km` algorithm to your `hash` and `tfidf` Series. Add each one to `en_titles` as a new column.

Now, for each of the three `*_cluster` columns you created, determine how many books fall into each of the 20 clusters. (Hint: `groupby` should help you here.)

Does the clustering algorithm appear to work better or worse for any of the transformations? What if you choose to create fewer or more clusters? What if you play with other options to the Vectorizer constructors or the KMeans constructor? Try turning some knobs and document what you discover in your notebook.

### Visualize your clusters

The k-means algorithm assigns each book title to a cluster identified by an integer. That is all. Interpreting the cluster assignments in light of the book titles is the responsibility of the analyst (i.e., you).

Start this task by printing some of the tiles in a cluster with the following code:

```
en_titles[en_titles.count_cluster == 0].head(25)
```

Vary the column name, cluster integer, method of sampling, and sample size. Do you see any patterns within your clusters? Can you assign a category name to any cluster (e.g., books about X).

Studying clusters in this manner is inefficient at best and biased at worst. For instance, just because you look at the first 25 titles in a set of 900 books doesn't mean those 25 are representative of the full set.

Find a way to better visualize and interpret your clusters. Consider manipulations of the titles and clusters using Pandas to show cluster contents compactly and without bias. Consider using matplotlib to display the information graphically in some way. Demonstrate your technique and document its pros and cons.

### Interpret your results

Do your clusters experiments reveal any patterns in book titles? Do they suggest any complementary categorizations or tags for books on the THR site? Do they suggest common topics addressed by THR authors?

Are there clusters that are not easy to explain? Are there books that seem to befuddle clustering? Do you have any ideas about how to study and understand these books better?

## Projects

If you want to try your hand at something larger than an exercise, consider one of the following.

### Find books misclassified by language

Gary says that some number of books on the Tar Heel Reader site are marked as having the wrong language. Manually finding these misclassifications is a pain. A language classifier could help alleviate these problems. Using the data provided, we could:

1. train the classifier on a set of books with known-to-be-correct language assignments (the *ground truth*),
2. evaluate the accuracy of the classifier on a hold-out test set of books by comparing its language predictions with the ground-truth,
3. apply a well-performing classifier to the entire set of books, and
4. review those books where the classifier predicts a language that mismatches the language assigned by the human author.

The text document classification example in the scikit-learn documentation might help get you started. So might the sample pipeline for text feature extraction and evaluation in the scikit-learn doc. In fact, there are many ways to skin this cat using scikit-learn. The key is setting up your notebook to quickly try new experiments in defining features, in picking a classifier algorithm, in choosing classifier parameters, and in evaluating performance.

If you want to tackle this project in earnest, talk with Pete. He has some feature selection code that might help.

### Build a recommendation engine

Gary has a second dataset derived from the Tar Heel Reader site that captures what books were read by what visitors to the site over time. This data can be used to train a recommendation engine based on collaborative filtering. Talk with Gary if you are interested in playing with this dataset and building a recommendation engine for the THR site.

### Improve the IPython Notebook UI

jtyberg writes:

> I love IPython notebook for ad-hoc analysis. However, there are a few shortcomings of the web UI that lessen my user experience. Among them is the tedious nature of reordering cells (moving them up or down) within a notebook. I would like to be able to select multiple cells and move them up/down the page all at once.
>
> A possible solution would be to enable grouping of cells. Can we modify the underlying DOM structure by adding cell elements into the same parent? Then we can manipulate the parent element.
>
> Another idea would be a gutter view within the notebook that shows a condensed view of the notebook content (think Sublime text editor). What if we could select individual cells or cell groups and move them up/down the page by dragging and dropping from within the gutter? That would be sweet.

This is more of a JavaScript project and is posted again in the *jQuery session project list*. The IPython Notebook has an unstable but working JavaScript API that might be useful in accomplishing either or both of these.

## References

**Choosing the right estimator** A rough guide for choosing the right scikit-learn algorithm for your machine learning task

**A gallery of interesting IPython Notebooks** Gallery of IPython Notebooks

**Matplotlib gallery** Gallery of matplotlib examples

**Scikit-learn examples** Gallery of scikit-learn examples

**Python Scientific Lecture Notes** Tutorial material on the scientific Python ecosystem

**Parallel Machine Learning with scikit-learn and IPython** Tutorial on machine learning over "big data"

# HTML, CSS, and Bootstrap

**Author** Peter Parente

## Goals

- Define HTML5
- Learn about HTML markup
- Learn about CSS styling
- Understand HTML and CSS work together
- Understand the pros and cons of Bootstrap
- Practice creating basic HTML / CSS sites
- Practice using Bootstrap to build web sites quickly

## Introduction

The fifth version of the HyperText Markup Language (HTML5) is a standard defining a markup language and APIs for structuring and presenting content on the Web. An HTML document consists of nested sets of elements represented by tags like <*body*>, <*p*>, and <*video*> which define the semantics of a Web page.

The third version of the Cascading Style Sheets (CSS3) standard defines defining a style language used to express the look and formatting of documents written in a markup language. A CSS file contains blocks of style declarations for colors, fonts, sizes, positions, and so on that can apply to elements in a HTML documents.

Together with JavaScript, HTML and CSS define what can be accomplished on the Web today, and experienced by users of Web browsers.

Bootstrap is a front-end framework that makes bootstrapping visually pleasing, mobile and desktop friendly, potentially dynamic Web sites easy. It contains a set of CSS styles that create a consistent, skinnable look-and-feel; a set of components that go beyond the basic constructs of HTML; and a set of JavaScript utilities that enable common, dynamic Web page behaviors.

To get started, watch the Bootstrap slidecast (~39 minutes) which shows uses for Bootstrap styles, layouts, components, and JavaScript behaviors. The slidecast includes demos of the following:

- Awesome Co. Homepage on slide 4 (~8 minutes)
- The Grid on slide 5 (~6 minutes)
- Some JavaScript on slide 6 (~5 minutes)
- Semantic Bootstrap on slide 12 (~13 minutes)

If time permits, review these additional pages:

- Skim the HTML5 article on Wikipedia. (~15 minutes)
- Read Chapters 1 through 4 of CSS Basics. (~10 minutes)
- Read the HTML Dog Beginner Tutorial. (~10 minutes)
- Read the Bootstrap article on Wikipedia. (~10 minutes).
- Read Building Twitter Bootstrap. (~15 minutes)
- Read Bootstrap without all the debt. (~10 minutes)

## Exercises

You will need to complete the *Setting Up* instructions before you proceed with these exercises. Once you are set up, SSH into *tottbox* using the `vagrant ssh` command from the setup instructions. Then tackle the problems below. Document what you find in a gist and share it with the TotT community later.

### Run a tiny web server

Create a project folder in the *tottbox* shared directory. On *tottbox*, change to that directory and run `python -m SimpleHTTPServer`. Back on your host machine, open your web browser and visit http://192.168.33.10:8000. You should see an empty directory listing.

### (Re-)design the TotT web site

Create a simple HTML web page for the TotT meet-up in the `index.html` file in your project folder. Use the content on the current front-page of the TotT meet-up site or make up your own description. Use appropriate HTML5 tags to

indicate headings, sections, navigation, and so on. Make sure it renders properly in your browser. Don't spend much time on making it look spiffy yet.

### Get Bootstrap

Download the assets from the Bootstrap site. Or, get fancy and try using a frontend package manager like bower. Your choice. In either case, get the Bootstrap CSS and JS files into the project folder you created.

### Style your page with Bootstrap

Add the Bootstrap CSS and JS to your `index.html`. Refresh your browser and make sure you see the Bootstrap typography styles take hold.

Now design a layout for the page content that highlights the key facts about the meet-up. Use appropriate styles and components from Bootstrap to do so. For example, you might consider using the grid feature to put simple summaries of what, when, and where above the fold instead of one after the other down the page.

### Try it on mobile

If you have a mobile device, try visiting your site on it. If you want to keep it running on *tottbox* and visit it there, you'll need to modify your *tottbox* Vagrantfile to forward port 8000 on your *tottbox* VM to your host machine and `vagrant reload` your VM. Then you'll need to visit the IP address of your host box in your web browser to view it.

Alternatively, if you have a Dropbox account or other static site hosting, you can dump your site there and view it on your mobile.

### Apply a new theme

Switch the Bootstrap theme to one of the free offerings from Pixelkit. Follow their instructions on how to make the change.

### Add CSS3 eye-candy

Learn about CSS3 transitions, transforms, and animations by searching the web. Think about where you might apply such eye-candy tastefully on the TotT site. Then try your hand at adding the feature. For example, you might try transitioning in more details about any of the basic facts about the meet-up when the user mouses over the initial text. (Hint: Check out these original hover effects for inspiration.)

### Ensure keyboard accessibility

Try navigating the page you built using the keyboard alone. Learn the basic keyboard navigation commands for your browser and OS. Try moving focus among the interactive elements you added to the page (e.g., buttons, links). Try activating all of them. Are there any holes? If so, fix them. (Hint: Did you add mouseovers? Can you trigger them without a mouse?)

## Projects

If you want to try your hand at something larger than an exercise, consider one of the following.

### Revisit the dead-drop

If you attended the *Express and Jade* meet-up, use Bootstrap to style your dead-drop web app.

## References

**HTML5 Rocks** Tutorials, articles, demos, and sample code for HTML5 related technologies

**Dive Into HTML5** "... elaborates on a hand-picked selection of features from the HTML5 specification and other fine standards."

**Can I use ...** Browser compatibility tables for HTML5 and related features

# JavaScript and jQuery

**Author** Peter Parente

**Builds-on** *JavaScript and NodeJS*; *HTML, CSS, and Bootstrap*

## Goals

- Learn about the browser DOM
- Learn about AJAX
- Know what jQuery is
- Understand the history of jQuery
- Practice traversing and manipulating the DOM with jQuery
- Practice invoking REST APIs with jQuery
- Use the basic panels of the Chrome Developer Tools

## Introduction

jQuery is a library that makes writing client-side JavaScript easier than using the standardi HTML document object model (DOM). jQuery presents a simple API for accessing, navigating, and manipulating HTML; handling web page events; and communicating with web servers, all in a manner consistent across web browsers. Although jQuery arose nearly a decade ago, long before HTML5 and wholesale support for Web standards, it remains at the foundation of many web sites, web applications, and web libraries today.

To get started, watch the jQuery slidecast (~25 minutes) introducing the primary features of jQuery. The slidecast includes demos of the following:

- Selections on slide 3 (~3 minutes)
- Event handling on slide 4 (~4 minutes)
- Effects on slide 5 (~2 minutes)
- Traversals on slide 6 (~5 minutes)
- XHR (Ajax) on slide 7 (~7 minutes)

If time permits, review these additional pages:

- Skim DOM Introduction on the Mozilla Developer Network site (~15 minutes)
- Read the AJAX Wikipedia article (15 minutes)
- Read the jQuery Wikipedia article (10 minutes)
- Skim some jQuery Examples and Best Practices (~15 minutes)
- Skim 5 Ways to Make Ajax Calls with jQuery (~15 minutes)
- Read Introduction to Chrome Developer Tools, Part One on HTML5 Rocks (~30 minutes)
- Read Do You Really Need jQuery? (~10 minutes)
- Solve some of the problems on tryJQuery

## Exercises

You will need to complete the *Setting Up* instructions before you proceed with these exercises. Once you are set up, SSH into *tottbox* using the `vagrant ssh` command from the setup instructions. Then tackle the problems below. Document what you find in a gist and share it with the TotT community later.

### Clone the ShortSheet project

I've seeded a project called ShortSheet (SS) on GitHub at https://github.com/parente/tott-shortsheet. It is a starting point for building a basic spreadsheet web application. We will build out this application throughout the following exercises.

To get started, clone SS to your *tottbox* shared folder. Start a bash session on *tottbox*, change to the project directory, and run `make vendor`. The build will download dependencies of the project, namely Bottle, jQuery, and Bootstrap. When the build completes, run `make server` in the same directory. Then visit http://192.168.33.10:8080 in your browser.

Spend a few minutes looking at the application UI and its structure on disk. Open the `Makefile` in the root of the project and look at its contents. What did `make vendor` and `make server` do? Open the `public/index.html` file. What does it contain? How about the `public/vendor` folder? The `public/js/shortsheet.js`? Share what you find.

### Add CSV file import

Right now, the SS shows a sample set of data and nothing more. An *Import* button exists at the top of the page. Clicking it shows a file picker. Selecting a file causes no change. Yet.

Open the `shortsheet.js` file. Find the `TODO` comment about supporting import. Add code using jQuery that attaches a `change` event handler to the `import-csv'` file input element and invokes the function mentioned in the comment. (Hint: Look in the jQuery doc.)

Save this second sample CSV file with a `.csv` extension on your machine. Try importing it into your spreadsheet. The spreadsheet should render the new data if your code is working properly.

If you hit problems, use the Chrome Developer Tools (or equivalent in your browser of choice) to debug the problem. (Hint: Adding simple `console.log` statements to your code and looking for their output in the developer tools *Console* tab can go a long way.)

### Make cells editable

The SS cells are read-only at the moment. We want to make them editable so that you can change values and, later, enter formulas.

Look for the `TODO` comment about making cells editable in the `shortsheet.js` file. Add code using jQuery that listens for double click events on all cells in the spreadsheet table (i.e., all <td> HTML elements in the <tbody> element). In the event listener function, set the `contenteditable` attribute on the clicked table cell to `true`. Then give the cell keyboard focus. (Hint: Again, use the jQuery doc or Google.)

If your code is working properly, the browser will highlight the cell, show a text caret in it, and let you edit its content.

### Make cells read-only again

Allowing edits to a cell is only half the battle. You must also add code to take the cell out of edit mode when you want to cease editing. There are many ways you might do so. Handle two of them for now.

1. If you double click another cell while in editing mode, the current cell should become read-only and the newly clicked cell should be editable.
2. If the you press the Enter key, the current cell should become read-only.

Again, use jQuery to set event handlers for these conditions. Tracking which cell is currently in editing mode in a variable might help in resetting it.

### Support adding rows and columns

The spreadsheet is still pretty static at the moment. The row and column count is fixed at the dimensions of the data you loaded. Add UI to allow addition or removal of rows and columns. Add the appropriate jQuery event handlers to monitor for these elements. When an event occurs, add the appropriate HTML elements:

- For a row, add a <tr> containing a number of <td> elements equal to the current number of columns.
- For a column, append a <td> element to each row <tr> element currently in the table.

Start by supporting additions at the end of the last row or column. Once you have that code working, consider changing the UI and code to support additions anywhere in the sheet.

### Support cell formulas

All spreadsheets have support for formulas. Think about a syntax for arithmetic operations in ShortSheet. Maybe a subset of JavaScript? Maybe something custom? Should it support individual cells? Cell ranges?

Add code to `shortsheet.js` to parse and execute formulas when a cell changes from editable to read-only. Store the formula in a `data-formula` attribute on the cell. Parse and execute the formula. Store the result of the formula in the cell itself.

Re-evaluate any formulas in the sheet whenever a new row or column is added. Change the CSV loading code to add any formulas present in the CSV as `data-formula` attributes and evaluate them all.

Consider editing the `sample.csv` file to include a few formulas to test your code.

### Think about your design

Take a moment and think about the data model of SS. What happens when you want to implement support for saving a spreadsheet? How would you gather up the formulas and plain, old values? Does storing everything in the HTML make things hard in the long-term? (Hint: This is the topic we'll address in the *MV* and Backbone* session.

### Support row and column removal

Add UI and code for removing entire rows and columns from the spreadsheet. Remember to re-execute any formulas after adding either. (Hint: Have you put the code for formula execution in its own reusable function yet?)

### Add CSV URL import

Looking back, it's silly that you had to download a CSV file from a GitHub Gist just to load it from your local machine into your web browser. Why not just fetch it directly from the Gist URL?

One complicating factor is that JavaScript running in a web browser can only send requests to the same origin that served up the HTML page that includes it. This security precaution is known as the same origin policy and is meant to prevent cross-site scripting attacks. Web applications have ways of working around this limitation, one of is to simply make such requests on the server side, not the client-side.

The Python web server hosting the SS web assets already has a `/gist/:userid/:gistid` resource. Sending an HTTP GET request to this resource with a valid GitHub username and Gist ID will cause the server to respond with the raw text of the Gist.

Add elements to the ShortSheet UI to collect this information, and a trigger to send it to the Python server. Add jQuery code to listen for the trigger event and to send a GET request (AJAX request) with the requisite information. Populate the spreadsheet with the response CSV in the same manner as when the file existed locally.

Test your code with the gist you downloaded previously with user ID `parente` and gist ID `7965617`. Or choose another CSV gist located on GitHub as a test.

### Add more features

Consider other features most spreadsheets have (or don't have). Implement whatever you wish. Here are some starting ideas.

- Show errors loading spreadsheets, evaluating formulas, and so on using Bootstrap alerts.
- Support column and row sorting by value.
- Support column and row re-ordering via drag and drop.
- Support keyboard navigation of the sheet.
- Support more formula operations.
- Support progressive loading of large CSV files.
- Set columns to a fixed, but adjustable, width.
- Allow users to download modified sheets as CSV files.
- Add spreadsheet persistence on the server side.
- Make sheet display more attractive with better styling.
- Show a busy spinner while loading data.

## Projects

If you want to try your hand at something larger than an exercise, consider one of the following.

### Stateless Book Builder

See Gary's Stateless Server Idea blog post.

### Slidecast Framework

Pete hacked together a little JS module for reveal.js to support the self-narrating slidecasts you see on the TotT session pages. Extract this code out of the TotT GitHub repository and migrate it to its own repo. Then spend some time cleaning it up, making it more general purpose, and documenting it so others can use it to build their own slidecasts.

### Hosted Slidecasts

Take the slidecast framework mentioned above and build a cloud-hosted version. One approach could be:

1. A user signs in.
2. The user links her slidecast account to her DropBox account.
3. The user enters Markdown to construct her slides.
4. The user records audio right on the site via the HTML5 getUserMedia API.
5. The site persists the slideshow in the user's DropBox account.

This project would be a large undertaking, but unique on the web at the moment, as best as I can tell.

### Improve the IPython Notebook UI

jtyberg writes:

> I love IPython notebook for ad-hoc analysis. However, there are a few shortcomings of the web UI that lessen my user experience. Among them is the tedious nature of reordering cells (moving them up or down) within a notebook. I would like to be able to select multiple cells and move them up/down the page all at once.
>
> A possible solution would be to enable grouping of cells. Can we modify the underlying DOM structure by adding cell elements into the same parent? Then we can manipulate the parent element.
>
> Another idea would be a gutter view within the notebook that shows a condensed view of the notebook content (think Sublime text editor). What if we could select individual cells or cell groups and move them up/down the page by dragging and dropping from within the gutter? That would be sweet.

The IPython Notebook has an unstable but working JavaScript API that might be useful in accomplishing either or both of these.

## References

**Learn jQuery** Explanations, workarounds, best practices, how-tos

**Chrome Developer Tools Documentation** Official documentation from Google

# Automation and Gulp.js

**Author** Michael Coalman

**Builds-on** *JavaScript and NodeJS*; *HTML, CSS, and Bootstrap*; *JavaScript and jQuery*

## Goals

- Learn what a task runner is
- Learn how to install gulp and access gulp plugins
- Edit an example gulp file to build an example project

## Introduction

Gulp.js is a new task runner that automates repetitive, frontend build tasks. It can automate the process of linting your JavaScript, compressing images, minifying JavaScript, and many other common web dev tasks.

So why should we care about task runners? Well, currently frontend development requires a lot of build tasks. We need to minify css/html, compile templates, compress images, compile TypeScript/CoffeeScript/Dart/Sass/LESS/Stylus files, and do all kinds of other things. Task runners can automate all of these tasks. Once you configure a task runner, it can save you huge amounts of time.

If time permits, try to review some of these resources before the meet-up. If you can't, come anyway!

- Watch this video on using Grunt (~2 minutes): You don't need to watch all of it, but it gives a good idea of what a task runner does. Gulp.js solves the same problems, so the first two minutes are still useful.
- Make sure you feel somewhat comfortable with npm, nodejs, and javascript before attempting this. Some knowledge of jade and markdown can help too.
- Read this simple tutorial (~5 minutes)

## Exercises

You will need to complete the *Setting Up* instructions before you proceed with these exercises. Once you are set up, SSH into *tottbox* using the `vagrant ssh` command from the setup instructions. Then tackle the problems below.

### Install Gulp.js

The first step is to install the gulp npm package globally. If you aren't sure how to do this, enter `npm help install` or ask for help.

This session will involve an example project for you to work with. If you already have a website or some kind of web related project (your project should probably have something to do with front-end web dev), you can try to integrate gulp into your project instead of using my example project. If you want to use the example project, just git clone this repository. Make sure to read the project's readme to get a sense of the folder structure and complete the initial setup.

### Copy Bootstrap CSS Files

After running `bower install`, you should notice a `bower_components` folder. In the `gulpfile.js` file, you should see a function that looks like this:

```
gulp.task('vendor', function() {
    console.log('Edit the gulpfile.js file!');
});
```

Implement this function to copy the `bower_components/bootstrap/dist/css/bootstrap.css` file to the `dist/vendor` directory (gulp will create this directory for you if it is not already there). You may find the following link useful: gulp api.

### Render Jade Templates

If you read the example project `readme.md`, you'll know that the `src/` directory has some `.jade` files. It's time to compile these files into html. Inside the `gulpfile.js` file you should notice a *build* gulp task. Implement this function to use the jade plugin to compile the `src/*.jade` files to `dist/`.

If you want to preview the html files after they are created, feel free to use the http-server utility we installed in an earlier meetup to view the files.

### Pass Data to the Jade Templates

If you viewed the files, you'll notice a pleasant Domo. Lets change that. If you did the last exercise correctly, you will have a line like this:

```
.pipe(jade())
```

Unfortunately, this step is not well documented on the `gulp-jade` repository so I'll walk you through it. The `jade` function can take an object with some options, one of them is a `data` option that my templates can use to generate the html with something other than a pleasant domo. The following code snippet is similar to what your code should look like after this exercise:

```
.pipe(jade({
    data: {
        title: 'About Me',
        name: 'Michael Coleman', // use your own name, this one is mine :P
        portrait: 'url to a pic of you', // more on this in a bit
        github: 'https://github.com/Coalman', // use your own github url
        email: 'your email address'
    }
}
```

There is an interesting choice to make when it comes to your portrait. You could choose to use gravatar (github uses this service to get your github profile picture) to generate the url of your gravatar portrait or insert the link here manually. It's purely up to you to choose.

Also, feel free to change the Markdown text in the `index.jade` file to say a little bit about yourself!

### Setup Local Web Server

I've included a small static file server using Express in the `preview.js` file in the root folder. It's been imported to the `preview` variable in your gulpfile. If you call the `listen` function on this `preview` object (and pass in a port number), it will serve your files locally. I recommend putting this code in the `server` gulp task.

#### Watch files and LiveReload

This exercise is a bit challenging. When you make changes to the jade files, you have to manually run `gulp build` to see the changes. Use gulp.watch in the gulpfile to rebuild the jade files when a change is detected.

Also, you can setup livereload to refresh your browser when these files are rebuilt. You may find this link helpful.

#### Compress your HTML, CSS, and Image Files

This exercise is a bit challenging. There is a gulp plugin called gulp-gzip that will gzip your files. Use this plugin to compress your generated html/css files. This will reduce the size of the files which decreases bandwidth usage and increases transfer speed. You can *read this <http://betterexplained.com/articles/how-to-optimize-your-site-with-gzip-compression/>* for more information on why you should compress your assets.

### References

**Gulp.js plugins** List of gulp.js plugins. If you don't see one you need here, consider writing one. If you don't need to interact with a stream, you could consider looking for normal npm modules that could be adapted or used.

**Gulp.js docs** Official docs. At the time of writing, they are a tad small, but there are examples in the repository. It's documented well enough.

## First Sequence: Retrospective

**Author** Peter Parente

The first incarnation of the Tools of the Trade (TotT) meet-up ran from January 10th, 2014 until April 11th, 2014. We met on eleven of the fourteen available Fridays. Most of our meetings ran for approximately two hours, from 3 PM until 5 PM. Participants were free to leave at any time. The majority stayed the duration.

Roughly fifty or so students attended the first session to learn more about the meet-up. About half as many returned the following week seeking to participate in earnest. Attendance leveled off at about ten to fifteen regulars each week for the rest of the semester with additional people joining based on interest and availability. The participants were mostly UNC-CS students but a handful came from other departments and nearby universities.

Some people came to TotT with friends to work collaboratively. Others attended alone. Most attempted the session exercises, with or without assistance. Some of these were willing to share what they learned with the entire group. One student even contributed and lead a session himself.

All said, I was pleased with the first offering of TotT. When polled, most students echoed my sentiment by expressing gratitude for our meetings and interest in future sessions. But while there were certainly aspects of TotT that worked well there were also those that could be improved. I wish to bring both the good and the bad to light here in order to make future TotT offerings better.

### What worked well

**The concept.** Without a doubt, attendees were excited by the prospect of having time and support for practicing software development in a no-consequence environment. I heard repeatedly from students who believed TotT was their first real opportunity to learn about the variety of tools used in industry. While I don't agree with their viewpoint, their enthusiasm remains as evidence of why I assembled this meet-up and why I hope it continues.

**The pace and variety.** I took a gamble in jumping between topics each week. On one hand, I wanted to give students a glimpse of the bevy of tools used in modern software development and start building their self-confidence in learning

them with agility. On the other, I didn't want the weekly switching of topics to cause chaos and confusion. I believe my bet paid off in the end: no one expressed displeasure in studying a new topic each week and only a few found issue with the particular topics we covered. One attendee suggested that maybe the sessions could all build upon one long-running software project. I agree that such a setup would be ideal, but struggle to identify a single, reasonably sized project that could logically encompass such a diverse set of tools.

**Starting all-together.** Until the fifth session or so, I would briefly introduce a topic, have the attendees start working on the exercises, and travel around the room assisting when asked. I could sense this was not working well: the room would typically go quiet at this point as students found themselves thrown off the deep-end, especially since most did not have time to review the prep materials and some preferred to work solo. I polled the students and found a handful of them asking that we work an exercise or two together at the outset of each session to get everyone rolling along together. We tried this at our next meet-up and it appeared to work very well. Thereafter, we would typically start working together as one big group, then split off into smaller groups, reconvene as one group again to share work, split again, and so on. Students who needed help caught up at the full-group checkpoints while those who wanted to race ahead did so unabated.

## What didn't work well

**The introduction.** As in all things, the TotT elevator pitch could be better. I fear some of the first-session attendees did not return, not because of a lack of interest, but because they thought TotT would be beyond their ability. Some students expressed this concern to me directly and I tried to allay their worries but failed to do so. Next time around, the introduction must explicitly state that anyone with an interest and basic understanding of programming can benefit from TotT. At the very least, simply sitting through sessions and entering the revealed exercise solutions would offer awareness of a tool and its application heretofore unknown to novice students.

**The meeting space.** We met in one of the UNC campus lecture halls with risers for students and a podium at the front. I think the room created an unintentional speaker-listener dynamic in our sessions whereas I wanted to promote small group collaborations among students. A room with a large round table or many individual tables might foster more student interaction in future renditions, and properly demote the leader's role to that of advisor rather than lecturer.

**Assuming attendees had time to prep.** Most of my preparation for TotT went into the creation of the prep materials: the narrated slidecasts, the example code, the screencast videos, the links to additional information, etc. I knew that attendees would not have time to review everything before every session, but I assumed they would at least have ten or so minutes to skim the introduction to each topic. I was wrong. Most people came to the sessions without having read or watched anything in advance. Future offerings of TotT should plan for this outcome, perhaps by scheduling a from-scratch introduction in the first 15 to 30 minutes of each session for those that need it, followed by the typical group activity there after for those with a basic grasp of a subject. Alternatively, some incentive could be given to encourage students to review the prep material beforehand.

I do not feel the the time I spent creating the prep materials was wasted, however. Some students informed me that they used the TotT website, links, videos, and so on as a reference after our sessions. Since everything is available online, it's potentially benefiting others too without my knowledge.

**The VM setup and use.** Getting a dozen or more disparate laptops to run even one piece of software consistently is difficult, let alone the large variety of tools we studied in TotT. The *tottbox* VM was my attempt at creating a consistent development environment meant to alleviate the inevitable discrepancies across student computers. It was imperfect, however, and still required OS and even machine specific tweaks. Some students found working with it difficult enough to try to install the tools on their host OS (and most of these met with less than desirable success).

I see no grand solution to this problem, and still believe a VM is still the best solution for the purposes of TotT, better than having students configure tens of tools on their host machines and better than forcing students to work with the tools on some VM in the cloud. More investment in making *tottbox* easier to setup, understand, and use is warranted.

### What's next?

Where TotT goes from here is an open question. I'd like to see three things happen and will try to assist in all of them:

1. The creation of sessions on additional topics, by myself and others. I've identified some already and put placeholders on the TotT site for them.
2. A repeat offering of the first sequence of topics in the UNC-CS department. It would be wonderful if this were a student-backed effort.
3. The further dissemination of the TotT materials. I believe there is a lot educational value tucked away on the TotT website, particularly in the slidecasts. Surfacing this information so others can find it more easily is probably worth the effort.

If you or someone you know is interested in helping in any of these efforts, or others, please reach out.

## MV* and Backbone

**Author** Peter Parente

**Builds-on** *JavaScript and jQuery*, *HTML, CSS, and Bootstrap*

### Goals

- Learn a bit about MVC, MVP, MVVM, etc. design patterns
- Learn the basic concepts of the Backbone framework
- Recognize the MV* pattern in Backbone
- Understand how Backbone can help structure client-side code
- Practice writing web applications using Backbone

### Introduction

Backbone is a JavaScript framework that lends structure to frontend code. Its concepts of models, views, collections, and routes help separate the data and logic of a web application from its necessary representation as DOM nodes in a web browser. This separation is critical to keeping the code base of a rich client-side application intelligible and its runtime data in-sync between a backend data store, backend API, and frontend user interface.

To get started, review these materials online:

- Read the Wikipedia Model-view-controller article (~10 minutes)
- Read the Introduction section of the Backbone doc (~5 minutes)
- Read Why you should use MV* frontend frameworks like Backbone, Ember, and Angular (~10 minutes)
- Read the annotated Todos.js Backbone example code (~15 minutes)

### Exercises

You will need to complete the *Setting Up* instructions before you proceed with these exercises. Once you are set up, SSH into *tottbox* using the `vagrant ssh` command from the setup instructions. Then tackle the problems below. Document what you find in a gist and share it with the TotT community later.

### Clone the Kvetch project

I've seeded a project called Kvetch on GitHub at https://github.com/parente/tott-kvetch. It is a starting point for building a place for people to post anonymous gripes, just like in the Daily Tar Heel. We will build out this application throughout the following exercises.

To get started, clone the project to your *tottbox* shared folder. Start a bash session on *tottbox*, change to the project directory, and run `make build`. The build will download dependencies of the project, namely Bottle, jQuery, Bootstrap, and Backbone, and Underscore. When the build completes, run `make server` in the same directory. Then visit http://192.168.33.10:8080 in your browser.

Spend a few minutes looking at the application UI and its structure on disk. Note the following:

- The `Makefile` installed the project prerequisites. What libraries did it install when you ran `make build`? What did it run when you typed `make server`?
- `server.py` contains a complete implementation of the REST API. What routes does it support? What data does it take as input and respond with as output?
- `views/index.tpl` defines the one HTML document used by the app. What JavaScript files does it load? What JS code executes when the page finishes loading?
- The project contains placeholders for Backbone components. Where are they? What placeholders are defined?

Document what you find.

### Understand the requirements

When complete, I would like Kvetch to behave as follows:

- The newest 25 posts appear when the page loads in the *Latest* column
- The the top 25 posts with the most upvotes appear in the *Favorites* column when the page loads, sorted from most votes to least.
- Any user may enter a post on the site up to 140 characters in length.
- The user's post should appear at the top of the *Latest* column when entered.
- Each post should show the post body text, the date the post was posted, and the number of upvotes the post has received.
- Each post should also show a `+1` button to allow a visiting user to vote for the post *once per page load*.
- If a user upvotes a post, the *Favorites* column should re-sort appropriately.
- A page refresh should result in the latest information from the server being shown.
- Posts and upvotes should persist across server restarts.

The HTML, CSS, REST API, and database persistence are already implemented. You need to add the JavaScript code necessary to support the above.

Before proceeding, think about how you might implement the necessary application logic without a MV* framework, say using vanilla JS or jQuery alone. Jot down your initial thoughts.

### Understand the components

Think about how you might structure the frontend code for Kvetch in terms of Backbone models, views, and collections. What data and logic might go together in models? Are there natural collections of models? What must be shown to the user in a application view? Does it help to decompose the app into multiple views?

Write down your thoughts. Draw a diagram. Depict how the frontend components would interact in your design. Then, compare your design with the one represented by the files and folders in `public/js`. (Hint: There is no one right answer to designing this application. But this rest of this tutorial leads you down the path I chose.)

### Define a post model

Open the `public/js/models/post.js` file. In it you'll see a call to the function `Backbone.Model.extend` with an empty object as its parameter. You need to populate this object with the following to define a simple model for a post on the kvetching board:

- The default value of the `body` model property, an empty string
- The default value of the `vote` model property, zero
- The attribute attribute name that will serve as the unique ID of the post, `rowid`

(Hint: Look in the Backbone documentation in the Models section for what properties you need to set. Or refer to the TodoMVC Backbone example.)

### Define a posts collection

Open the `public/js/collections/post.js` file. In here, you'll see a call to the function `Backbone.Collection.extend`. You need to populate its argument with the following to define a collection of posts on the kvetching board:

- The model to store as elements in the collection
- The URL path on the server that represents the posts collection resource in the REST API

### Define the post view

Open the `views/index.tpl` file. Look at lines 45 through 50 in the file. This section contains markup for an Underscore template. When rendered as HTML, it will display the body, vote count, and timestamp of a post model on the kvetching board.

Now open the `public/js/views/post-view.js` file. Look for the call to `Backbone.View.extend`. Populate its object with the following properties to use the Underscore template as the view for a post:

```
app.PostView = Backbone.View.extend({
   // html tag to use for each post
   tagName: 'div',
   // css class name to use on each post
   className: 'post',
   // template to use for each post
   template: _.template($('#post-template').html()),

   render: function() {
      // TODO
   }
});
```

Look in the Backbone documentation in the View section and the Underscore doc for the `template` function to understand what these properties mean. Once you do, implement the `render` function so that it does the following:

1. Renders the Underscore template as HTML using the properties of `this.model` (Hint: Look in the Underscore doc for an example of how to render the template.)

2. Puts the rendered HTML on the page in the HTML element bound to the view. (Hint: Look in the Backbone doc for the view instance variable name containing a reference to the element on the page.)
3. Returns the view instance for use by the caller of the `render` function. (Big Hint: `return this;`.)

### Define a list of posts view

Open the `public/js/views/posts-view.js` file. Look for the call to `Backbone.View.extend`. Populate its object argument with the following functions:

```
app.PostsView = Backbone.View.extend({
   initialize: function(options) {
      // reference to the posts collection
      this.posts = options.posts;

      // listen to add and reset events on the collection
      this.listenTo(this.posts, 'add', this.on_add_one);
      this.listenTo(this.posts, 'reset', this.on_add_all);

      // force the collection to fetch exists
      this.posts.fetch({reset: true});
   },

   on_add_one: function(post) {
      // TODO
   },

   on_add_all: function() {
      this.$el.html('');
      this.posts.each(function(post) {
         this.on_add_one(post);
      }, this);
   }
});
```

Review the `Backbone.View` documentation about the `initialize` and `listenTo` functions. Understand when Backbone will invoke the `on_add_one` and `on_add_all` callback functions.

Now implement the `on_add_one` function so that it does the following:

1. Creates a new instance of a `app.PostView` and passes it the `post` argument as the `model` for the view.
2. Calls the `render` function on the view instance and appends the result to this view's (the `app.PostsView`) element. (Hint: Did you find the documentation about where a view stores its element reference?)

### Put it all together

At this point, you've created a simple post model, a post collection, a view for a post, and a view for a collection of posts. Now you need to wire all these pieces together in an application view.

Open the `views/index.tpl` file again. Find the following:

- The ID of the <input> element.
- The ID of the submit <button> element
- The ID of the <div> under the *Latest* heading.

Now open the `public/js/views/app-view.js` file. Add the following to it. Then handle the TODOs in the code using the Backbone documentation and the element IDs you looked up in the `index.tpl` file.

```
app.AppView = Backbone.View.extend({
   el: '#app',

   events: {
      // TODO: register for click event on submit button and call on_submit
      // TODO: register for keypress event on the input element and call on_keypress
   },

   initialize: function() {
      // get a jQuery reference to the input element
      this.$input = $('#input');

      // TODO: create a new instance of the app.Posts collection
      //       and store it in an instance variable
      // TODO: create a new instance of the app.PostsView bound
      //       to the latest column, with a reference to the
      //       posts collection
   },

   on_submit: function() {
      // get the input text
      var val = this.$input.val().trim();
      if(val) {
         // TODO: create a new post in the collection with the
         // value as the body of the post

         // reset the text box to empty
         this.$input.val('');
      }
   },

   on_keypress: function(e) {
      // invoke submit when user presses enter
      if(e.which === app.ENTER_KEY) {
         this.on_submit();
      }
   }
});
```

When you're done, start the web server again if it isn't already running and refresh the browser page. If everything is working, you should be able to submit a new post and see it appear in the list of latest posts. Also, if you refresh the page or restart the server, you should still see all your posts.

Like in our jQuery session, if you hit problems, use the Chrome Developer Tools (or equivalent in your browser of choice) to debug the problem.

### Show the timestamp

When the user adds a new post, Backbone sends the post body to the server for inclusion in the application database. The server backend inserts the post body, current date and time, and initial vote count (zero) in the database. It responds with all of this information plus the unique ID of the post, namely the `rowid` from the database.

Update the `app.PostView` to listen to this server response. When received, re-render the post so that it includes the server generated information. (Hint: Look in the Backbone documentation for the model event the view needs to `listenTo`.)

### Support post upvotes

Supporting upvotes requires changes in both the post model and view.

- Add an event listener to `app.PostView` for clicks on the *+1*.
- Add an event callback that invokes an `upvote` function on the post model for the view.
- Add the `upvote` function to the `app.Post` model that uses jQuery to POST an empty request to `/upvote` on the server.
- Add a success callback to the jQuery AJAX request that updates the vote count on the model to the `response.votes` count the server returns.
- Add a listener to the `app.PostView` that updates the `#count` element in the view when the model's `vote` property changes.

Play with the application a bit once you get all this working. Is there any other logic you should add to the upvote feature? (Hint: How many upvotes should a user get?)

### Define a favorites collection

With the app receiving upvotes, it's now possible to show a collection of favorite posts: those with the most upvotes. Create a new `app.Favorites` collection that extends `app.Posts`. Point it to the `/favorites` URL of the server. Then instantiate a new `app.PostsView` in `app.AppView`. Pass this instance a reference to the ID of the favorites column in the page template and a reference to the favorites collection instance.

If all goes to plan, you shouldn't need to make any other changes. Why? (Hint: Are you getting benefits from reuse?)

### Sort the favorites by votes

Per the requirements, the favorites view should sort its posts from most votes to least. Add the necessary logic to make this happen to the `app.Favorites` collection. Then add an event listener to the `app.PostsView` that orders the post views accordingly. (Hint: Backbone supports sorting via a model `comparator` function. The harder part is getting the views sorted properly after the collection sort.)

### Keep the views consistent

A given post may appear twice on the page, both in the latest and favorites columns. If you upvote one of these posts, you'll notice that its counterpart does not update accordingly. Ideally, it should.

Currently, if a post appears in two columns, it means two post views are attached to two separate model instances representing the same post. Instead, we want the two views to share the same post model instance representing the post. You can accomplish this change by overriding how Backbone constructs new post instances and checking if an instance already exists for a given post `rowid`. If it does, you should reuse that instance instead of creating a new one.

(Hint: I overrode the `constructor` for `app.Post`.)

### Ask for my version

I do have a local git branch with a version of the Kvetch app meeting all the requirements set forth on this page. If you put significant effort into building the app, but get stuck, talk to me and I'll share my version with you. I will ask to see what you've done before I hand over my solution, however.

## Projects

If you want to try your hand at something larger than an exercise, consider one of the following.

### TotT gamification

I'd really like to recognize students as they complete exercises or projects throughout the TotT sessions. A web site that gamifies TotT might be cool. For instance, if you attend 10 sessions in a row, perhaps you receive the *Omnipresent* badge. If you attempt all the bash exercises, maybe you get the *Bash basher* badge. If your NodeJS dead-drop passes a set of tests you get the *007* badge. You get the idea.

The difficulty with such an undertaking is in the validation of achievements. How would the site know a student attended 10 sessions, tried all the bash exercises, and passed all dead-drop unit tests? I think all of these are solvable, but leave it to you to come up with creative solutions.

If you do, design and implement such a site using Backbone or another MV* framework. I'll gladly host it somewhere if you succeed.

## References

**TodoMVC** A TODO list web app implemented in numerous MV* frameworks (and not)with all of their source on GitHub for educational purposes

**Backbone Tutorials** A collection of Backbone related tutorials

CHAPTER 3

# When

We met on most Fridays in the Spring 2014 semester starting at 3 PM.

CHAPTER 4

# Where

We met in Sitterson Hall, Room 011 on the UNC-CH campus.

# CHAPTER 5

## How

In our first session, we will *set up a consistent, shared development environment* using Vagrant and VirtualBox. If you join us after the first session, please give the setup instructions a shot before you attend. If you get stuck, don't be afraid to show up and ask for help.

CHAPTER 6

# Who

Consider joining us if you:

- want a gently guided introduction to a variety of software tools,
- want dedicated time to practice coding in a collaborative environment,
- want to improve your ability to tame unfamiliar languages and libraries,
- think learning a bit about tools such as git, Python, jQuery, NodeJS, MongoDB, and Sinatra sounds like fun.

# CHAPTER 7

# Why

Like most artisans, we software developers use many specialized tools to practice our craft. Tools to make our wares available to millions of users. Tools to store, search, analyze, and visualize vast amounts of data. Tools to convey information and insight to others. Tools to automate the repetitive and error prone. Tools to express our thought-stuff as functioning software. Tools that beget new tools. And like in other professions, not a one is a golden hammer. To build successfully, we must use appropriate tools.

Choosing the right instruments requires awareness of their existence, knowledge of their function, practice in their application, and evaluation of their alternatives. Here, our discipline presents both a unique opportunity and challenge: the population of tools at our immediate disposal is mind-boggling and growing at an accelerating pace. There is almost certainly a perfect combination of tools for every software development project, and there is almost certainly a project using any particular combination of tools. We cannot hope to master, let alone know about, all of them a priori. We must be nimble in our ability to find, learn, apply, and evaluate tools as the situation (our problem, our team, our client, our employer, etc.) demands.

Fig. 7.1: Graph from the 10 Million Repositories post on the GitHub Blog

I strongly believe practice "hacking" builds this agility. Taking time to discover a new tool, install it, run its "hello world", read its documentation, think about its use, create small examples, apply it to some pet project, compare it to other tools, and so on provides us invaluable experience. It builds our confidence so that we might step-up to unfamiliar tools, learn them quickly, and master them eventually. It adds tools to our belts, albeit few out of millions. It fulfills our desire to learn and build new things. It entertains and provides a chance for collaboration.

Most importantly, it improves our ability to wield the endless tools of our trade.

CHAPTER 8

# Attribution

- Peter Parente conceived the Tools of the Trade meet-up and created materials for most of the sessions on this site.
- Michael Coalman contributed a session on Gulp.js.
- Gary Bishop publicized and sponsored the meet-up at UNC, and helped Pete refine the approach to TotT.
- The University of North Carolina at Chapel Hill Computer Science Department offered room for the TotT meet-up.
- The Emerging Technology Team in IBM Software Group gave Pete time to participate.
- Numerous citizens of the web published code, screencasts, tutorials, articles, documentation, and tools referenced from the TotT web site.