Full Stack Development Documentation

Release 1.0

Abdulrahman Alotaibi and Abdulaziz Alsaffar

November 22, 2014

Contents

In this course we will explore the concept of Full Stack Development. The course is divided into three main parts:

- Development Tools
- Team Project Workflow
- Complete Web development

The objective of the course is to make you familiar with the terminology and the tools that are used in modern web development.

The first week will be about the essential tools required to build a project and set up a local environment. The focus is on the interaction between you, the developer and your machine. The more you know about your machine the faster you become at fixing problems.

The second week we will focus on the interactions between you and your peers. It is a fact of modern development that all big projects are done in teams and development velocity is measured by the average velocity of all team members, not the velocity of any individual. In this week we will introduce some of the tools that help developers work simulataneously and effectively. Team development is all about increasing the Bus-factor of your team.

The third course is going to take two weeks. We will try to make you more familiar with the tools from the first two weeks by working through a demo project. We will introduce the concpets of automated deployment and replication of the development environment in production machines. This course will discuss best practices when it comes to design architecture of web applications. We will also compare between IAAS and PAAS and how to take advantage of them.

CHAPTER 1

Contents

1.1 Introduction

1.1.1 Definition

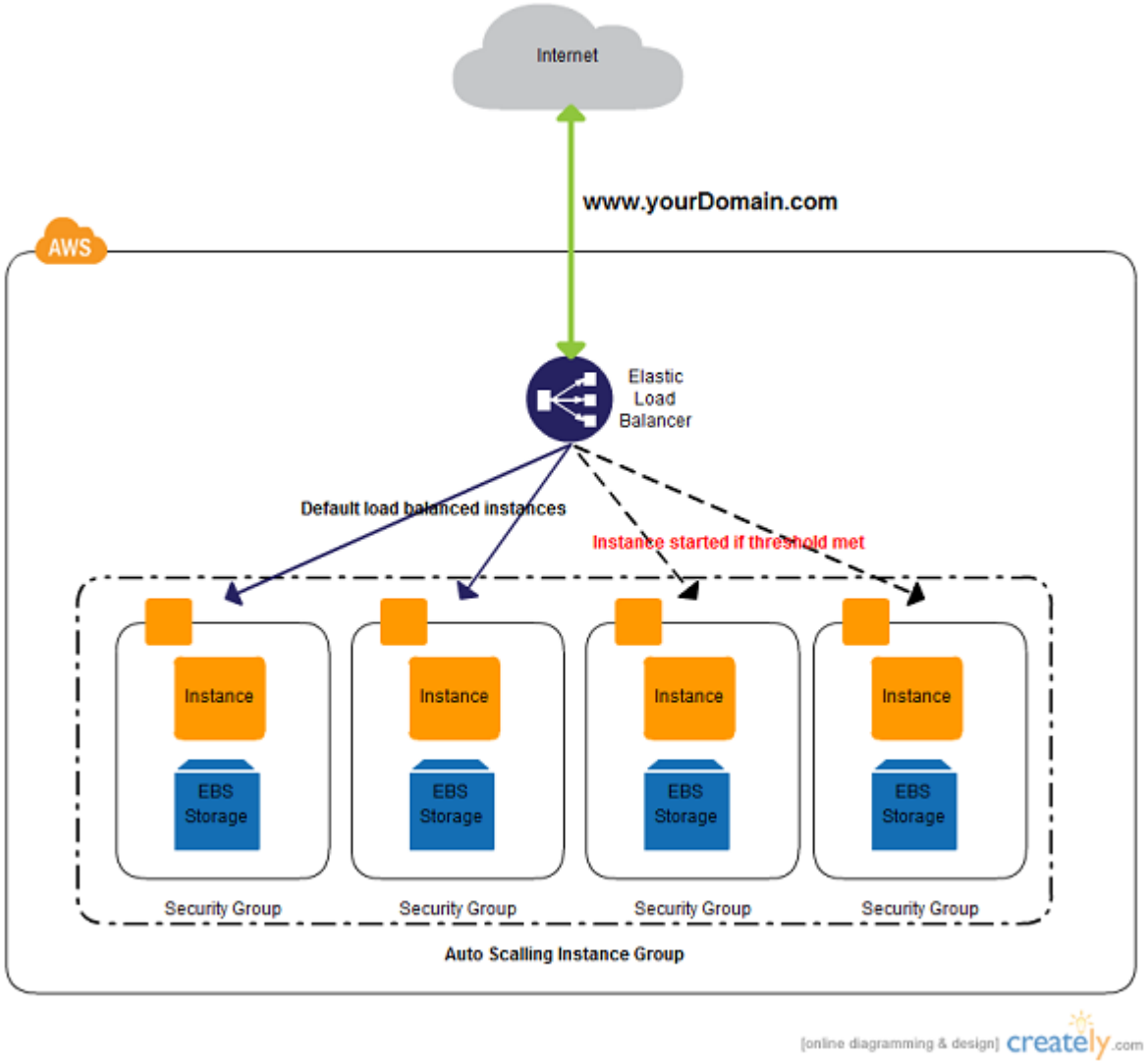
The industry definition of a Full Stack Developer is an engineer who can work on different levels of an application stack. The term stack refers to the combination of components and tools that make up the application. The components could be in the front-end or the back-end of the system.

The main objective of full stack engineer is to keep every part of the system running smoothly. A Full Stack Developer can performs tasks ranging from resizing an image or text in a webpage to patching the kernel.

Figure 1.1: This image shows a job position at stackoverflow definition of the term.

1.1.2 Modern Application Architecture

Modern applications are developed to be installed on mobile devices or hosted on the web. This is a result of trends in faster internet speeds, greater web access and penetration, and the development of more powerful mobile devices. This has created the need to rethink application development. Instead of an isolated desktop or mobile application, modern applications have a distributed back-end infrastructure interactively serving a variety of front-end clients over the web.



System parts

- **Back-End**
  - Operating System (OS)
  - Firewall
  - Web server
  - Database (SQL or NoSQL)

    - Caches
    - Message queuing software
    - Application
- **Front-End**
    - HTML
    - CSS
    - JavaScript
    - Dart

Modern Front-End frameworks

We have seen the big shift in the web from HTML 4 to HTML5 which has built-in APIs to help you accomplish many tasks to built a richer web application. This has resulted in a variety of front-end MVC frameworks such as:

- Polymer
- AngularJS
- React

Modern Development Frameworks

The changing computing world has led to and been led by the fast growing world of web development frameworks such as:

- NodeJS
- Django
- RoR

1.2 Phase I: Development Tools

1.2.1 Introduction

The purpose of the workshop

A workshop/tutorial for software developers who want to explore and learn about some of the tools required to set up a productive development environment and then utilise it effectively.

The workshop will walk partcipants thorugh some of the basic development tools and how to use them and then conclude with manually setting up a basic stack using these tools.

Phase I: Development Tools will help put you in a position to set up a development environment and then interact with that environment effectively to create your projects.

What's covered

Virtual Machines will help you set up a fresh development machine on which you can build the environment that you need for your project.

Command Line Interface will give you the tools you need to interact with your development machine and environment directly or remotely, with or without a GUI.

Software Package Managers will introduce you to the tools that will help you manage and set up the software in your development environment.

Editors and IDEs will give you the ability to create and edit files in your development environment.

Basic Stack Configuration Ecercise - this final section will bring all the tools encountered earlier to help us set up a stack for serving up a blank Django application.

1.2.2 Contents

Virtual Machines

Definition

In computing, a virtual machine (VM) is an emulation of a particular computer system. Virtual machines operate based on the computer architecture and functions of a real or hypothetical computer, and their implementations may involve specialized hardware, software, or a combination of both [1].

Software

- VirtualBox
- VMware
- LinuxKVM
- etc...

We will focus entirely on VirtualBox.

Exercises

1. Download and install the VirtualBox
2. Download an image of an operating system. We will use Ubuntu in this course. Download the ISO image
3. In VirtualBox `Create a Virtual Machine`
4. Configure the RAM to be allocated
5. Configure a hard drive for the image

[1] Wikipedia

Command Line Interface

Command Line Interface or CLI is the fastest way of communicating with a computer. CLI can retrive data from the current system or run a task on a remote server. Also, CLI provides a way to run periodic commands and runs long running commands in the background, `Daemons`,.

In this chapter, we will discover the import commands, and we will teach you to use them. After opening up a terminal in Ubuntu, type in the following command `ls`. As you can see in the terminal, a list of files and directories showed up. `ls` is the command to list the contents of the current directory.

Important commands

As you might expect, anything that you can do in the GUI you can do in the command line. CLI has many Shells e.g. Bourne Shell, Korn SHell, Bourne_Again_SHell ...etc. Shells are program that help you execute commands on the computer. They have a special language, and they provide basic programming capabilities. Below are a list of important commands that you should be comfortable with.

- **pwd**
  Stands for Print Working Directory:
  pwd prints the current working directory from `/` or `root directory`

- **mkdir**
  Make Directory:
  `mkdir foo` creates a directory called foo
  `mkdir -p foo/bar/baz` creates baz and all the missing directories in the path to it
  Exercise:
  - Make a directory called `temp`
  - Make a directory at the path `temp/stuff`
  - Make a directory at the path `temp/stuff/things`
  - Can you create a directory at `temp/stuff/things/frank/joe/alex/john` using a single command?
- **cd**
  Change Directory.
  Once invoked it will change your working directory to a new one:
  `cd temp` to change to the temp directory you created in the last exercise
  Exercise:
  - Change to the `temp` directory. Check where you are using `pwd`.
  - Change to the `stuff` directory. Check where you are.
  - Change directly to the `john` directory in one command.
  - Use the command `cd ..`. Where did you end up?
  - Where does `cd ../../..` take you?
  - What about `cd .`?
  - Just `cd`?
- **ls**
  Lists the contents of the current directory:

`ls -a` to list all files and directory including hidden directory or `dotfiles`
`ls -l` to list the files and directory with more information about their permissions, owner, group that owns it, disk size and creation date
Exercise:

- Navigate back to your home directory `~`.
- Go to `temp` and use `ls` to see what is in it.
- While in `temp` try `ls -lR`. What did it do?
- Use a combination of `cd`, `ls` and `pwd` to explore the files on your machine.

- **touch**
  Creates an empty file in the current directory:
  `touch CaptainAwesomesauce.txt` creates a blank text file called CaptainAwesomesauce.txt

- **cp**
  Copy file or directory from one location to another:
  `cp file1 file2` copies the contents of file1 into file2
  `cp file1 Documents/` copies file1 into the Documents directory
  `cp -r /tmp Documents/tmp` copies the contents of /tmp into Documents/tmp
  Exercise:
  - Inside `temp` create a file called `iamcool.txt`.
  - Make a copy of it called `awesome.txt`.
  - Make a directory called `stuff` and copy `awesome.txt` into it.
  - Without leaving `temp` check the contents of `stuff`
  - Copy `stuff` and all its contents into a new directory called `things`.
  - Without leaving `temp` check the contents of `things`.

- **mv**
  Moves files or directories to different location (path). Also it can be used
  to rename files or directories:
  `mv file file.txt` renames file to file.txt
  `mv Downloads/file.zip Documents/` moves file.zip from Downloads/ to Documents/
  Exercise:
  - Change the name of the file `awesome.txt` to `notawesome.txt`
  - Change the name of the directory `stuff` to `foo`
  - Move the file `iamcool.txt` from `temp` into `foo`

Warning: Be careful when passing paths!
Paths can be:
- **Absolute Paths** relative to root
  e.g. `/etc/init/`, `~/Desktop/bar.py`
- **Realtive Paths** from your current working directory
  e.g. `../foo/bar/`, `temp/stuff/awesome.txt`

- **nano**
  Nano is an easy to use terminal text editor:

`nano file1` opens file1 for editing

- **less**
  Less is a file viewer, and it has search features. The name came from the
  Unix philosophy "Less is more, more is less"[2]:
  `less foo.txt` page through foo.txt

- **cat**
  Concatenate files and prints them to `stdout`:
  `cat file1` spits the content of file1 to `stdout`
  `cat file1 file2` concatenates file1 to file2 then spits the contents to `stdout`
  Exercise:
  - Create a file called `zen.txt` with the following content:

```
Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
Flat is better than nested.
Sparse is better than dense.
Readability counts.
Special cases aren't special enough to break the rules.
Although practicality beats purity.
Errors should never pass silently.
Unless explicitly silenced.
In the face of ambiguity, refuse the temptation to guess.
There should be one-- and preferably only one --obvious way to do it.
Although that way may not be obvious at first unless you're Dutch.
Now is better than never.
Although never is often better than *right* now.
If the implementation is hard to explain, it's a bad idea.
If the implementation is easy to explain, it may be a good idea.
Namespaces are one honking great idea -- let's do more of those!
```

  - View it using `less` and `cat`. What's the difference?

- **rm**
  Removes a file or directory:
  `rm /path/to/file1` to delete file1
  `rm -r /path/to/dir1` to recursively delete dir1 and all its contents
  Exercise:
  - Go to the `temp` directory
  - Remove the file `notawesome.txt`.
  - Remove the directory `things` and all its content.

- **echo**
  Takes a string of text and prints it to `stdout`
  `echo Hello world`

- **|**

[2] Less history

The Pipe character which takes the output of the left command and inputs it to the right command
`ls | grep ""`

- **>**
  Redirect to character; it redirect the output of the command to a file
  `echo Hello > foo.txt`

- **>>**
  Append character; it appends the output to a file
  `echo Hello >> foo.txt`

- **<**
  Input in character; it inputs the text of a file to the command
  `cat < foo.txt`

- **man**
  Return the help manual for any command in the system:
  `man shell-command`

- **find**
  Find is a powerful command. Take a look at the manual of `find` to see all the options that you can use with it:
  `find . -type f -name foo` looks for a file that's named foo

- **diff**
  Differences between two files. The command `diff` prints out the difference between two files:
  `diff v1/foo1 v2/foo1`

- **comm**
  Common is a command that compares two files and print the common bytes between them:
  `comm v1/foo1 v2/foo1`

- **head**
  Head prints out first lines of a file:
  `head foo.txt`

- **tail**
  Tail is simliar to head but it prints out the last lines of a file:
  `tail foo.txt`

- **sort**
  Sort sorts text:

```
sort foo
```

- *** - The Wildcard**
  `*` is known as the wildcard because it matches everything.
  It's great when you want to do a command on a set of files all at once:
  `ls *.py` lists all the files in the current directory ending in `.py`
  `rm -r h*` removes all files and directories beginning with `h`
  `rm h*.*` removes only files beginning with `h`
  Exercise:
  - **Create the following files in `temp`:**
    * `ex12.txt`
    * `ex13.txt`
    * `ex14.py`
    * `stupid.vb`
    * `useless.vb`
    * `wasteoftime.vb`
  - List all the .txt files in `temp`.
  - List all the files that begin with `ex`.
  - Delete all the `vb` files!
  - Use `find` and `less` to see all the `.txt` files under your `home` directory.

Hint: You will need a `|` pipe for that last exercise

- **grep**
  Grep is a pattern search that uses regular expressions to look for a pattern
  in text. It's powerful if you know regular expressions:
  `grep this words.txt` looks for the word `this` inside a file named `words.txt`
  Exercise:
  - Create a file in `temp` called `newfile.txt` with the following text:

    ```
    This is a new file.
    This is a new file.
    This is a new file.
    ```

  - Create another file called `oldfile.txt` but with:

    ```
    This is an old file.
    This is an old file.
    This is an old file.
    ```

  - Search for all occurences of the word `new` in all the `.txt` files in `temp`.
  - Search for all occurences of `old`.
  - Search for all occurences of `file`.
  - How would you search for the words `This is`?

Hint: You can quickly type text into a file using `cat > file.txt`
This will overwrite `file.txt` with whatever you type until you close the file using `CTRL-c`.

See also:

Take a look at the Python Docs for more information

- **env**
  Prints out all the environments variables
  `env`
  Exercise:
  - Print all your environment variables.
  - Use `|` and `grep` to print only the variables that have your username in them.
- **export**
  Export a local varialble to become an environment variable
  `export VAR`
  Exercises:
  - Create an environment variable called `TESTING` and set it to `"1 2 3"`.
  - `echo` your new varialble.

Note: Environment variables are reset every time a new terminal session starts.

- **ssh**
  SecureShell is a program that connects you to remote computers and execute commands on them:
  `ssh alice@foo.com`
- **scp**
  Secure copy like FTP but uses SSH protocol to transmit data:
  `scp words.txt alice@foo.com:Desktop/store`
- **sudo**
  Super User DO is a command that escalates and runs the given command as **root**
- **‘‘** or **${}**
  Backticks command; which executes the command inside it and returns the output:
  `cat ‘ls *txt‘`
- **ifconfig**
  To check the network cards and the ip address
- **alias**
  To alias command and modify them
  `alias l="ls -al"`

See also:

Here is a comprehensive Command Line Cheatsheet

Dot files/directories

Dot files and directories play a big role in the Unix/Linux operating system. Once a file or directory starts with a **.** it will be hidden from the regular **ls** command, and to display it you need to run **ls** with **-a** flag. The flag display the all the files in the current directory. There are many special dotfiles that you need to be aware of. The list below lists couple of them.

1. ~/.bashrc
2. ~/.vimrc
3. ~/.emacs
4. ~/.bash_history

Exercises

run **ls -a ~** to display the files and try to get familiar with them

Software Package Managers

So far in the course we have learned how to set up a fresh Ubuntu virtual machine and how to use some of the basic built in shell commands. Plain Ubuntu is extremely powerful and we can do a lot with it but at some point, whether we are setting up a development machine or a production server, we will need to install and manage software, tools, programming libraries, etc... As a devloper you'll often find yourself rapidly installing, configuring, testing, uninstalling, adjusting, reinstalling multiple version of multiple software packages. And if you don't keep track of them they can conflict with each other. And break each other. And make your life miserable.

Luckily, back in the stone-age of Linux (the 90s), the combined efforts of many distressed developers resulted in the concept of a package manager! Package managers will search for and install or uninstall software on your system, ensure that dependencies and conflicts are taken care of, and generally help you manage the software on your machine.

- **Aptitude**
  This is Linux's very powerful built-in package manager.
  Basic Usage:
  `apt-get install foo-package`
- **PIP**
  PIP is the software manager for Python libraries and packages.
  (PIP stands for "PIP Installs Python". Developers are weird)
  Basic Usage:
  `pip install pyfoo`

Exercises

1. Read the Aptitude man page
2. Install PIP, python3, nginx using Aptitude
3. Read the PIP help file `pip help`

4. Install django, selenium, uwsgi using PIP

More Information:

There are many more package managers than we mentioned in this section depending on what kind of web development you are doing. We would recomment looking at:

- npm (node.js)
- yum (Linux)
- rpm (Linux)
- homebrew (Mac OSX)
- macports (Mac OSX)

Editors and IDEs

There are many editors in the Linux environment such as:

- nano
- pico
- vi
- vim
- emacs
- Sublime
- etc...

They are divided into GUI and terminal, and you can also divide them into advance text editors and normal text editors. Explore them and choose the one that works best for you.

Tip: Combining the power of GUI and non-GUI text editors would give you more power.

In addition to basic text editors, there is also a class of software packages known as Integrated Development Environments such as:

- PyCharm (python)
- Eclipse (java)
- XCode (objective-c and swift)
- Android Studio (android specific java)
- etc...

As you can see, IDEs are highly specialized for specific types of projects and applications. They are full of language and project specific features.

As a very general rule you would use basic editors to sporadically modify individual files from the command line such as when you are SSHed into one of your production machines. And you would use IDEs to set up, manage and develop an entire application and all its files locally such as developing a Django application.

Exercises

1. Navigate to **foo** directory again
2. Launch one of the text editors and create **hello.txt**
3. Write the following in it:

```
Lorem ipsum dolor sit amet, consectetur adipiscing elit.
Cras fermentum dolor purus, quis lobortis arcu volutpat ac.
Nullam consequat dapibus bibendum. Donec at libero at mi pulvinar sagittis.
Mauris et risus molestie, porta justo vel, suscipit erat.
Maecenas sed diam a nisi finibus ultrices eu et felis.
Nulla ornare elementum mi, vel dignissim tellus bibendum sed.
Vivamus diam nulla, hendrerit non dapibus at, pharetra non arcu.
Mauris posuere erat nibh, volutpat venenatis mi ultricies quis.
Duis hendrerit lacus lacus, eget viverra justo rhoncus ac.
Class aptent taciti sociosqu ad litora torquent per conubia nostra, per inceptos himenaeos.
Vestibulum id orci ut tortor fringilla imperdiet volutpat id orci.
Sed bibendum mauris ac dolor efficitur, ut pharetra neque lobortis.
Fusce malesuada ultricies feugiat. Interdum et malesuada fames ac ante ipsum primis in faucibus.
Praesent facilisis ultricies accumsan.
Quisque lectus neque, faucibus in egestas interdum, euismod at lacus.
Sed imperdiet nisl justo, eget porta tortor molestie sed.
Aenean consectetur varius ante, nec malesuada lorem pretium in.
Donec feugiat sapien non justo scelerisque vestibulum sed vel libero.
Suspendisse lobortis arcu nec ultrices vehicula.
Phasellus gravida nulla sed nunc sollicitudin, a vehicula enim commodo.
Sed eu convallis augue. Donec in eros malesuada, pretium sapien quis, eleifend est.
Ut vel venenatis turpis.
```

4. Save and Exit from the text editor.

Vim Exercise

1. Launch your terminal and follow and the instructions.
2. Finish all the 7 chapters.

Stack Configuration Exercise

In this exercise we are going to use many of the tools we have learned about during the week to set up our first stack.

This stack will run on a virtual machine and consist of the following components:

- Application Framework (Django)
- Python WSGI HTTP Server (Gunicorn)
- Web Server (NGINX)
- Database (PostgreSQL)

Step One: Setup Your Virtual Machine

At this point you should already have a virtual machine up and running but if you don't, set one up using the image you created on the first day of this course.

Step Two: Install and Create Virtualenv

In *Phase II* we will go into more details about how to install, configure and use python virtual environments but for now just use the following commands.

Install virtualenv:

```
sudo apt-get install python-virtualenv
```

Create a virtualenv for our python packages:

```
virtualenv ~/myenv
```

Step Three: Install and Configure PostgreSQL

Most Django users prefer to use PostgreSQL as their database server. It is much more robust than MySQL and the Django ORM works much better with PostgreSQL than MySQL, MSSQL, or others.

Install First we need to install dependencies for PostgreSQL to work with Django:

```
sudo apt-get install libpq-dev python-dev
```

Then we install PostgreSQL itself:

```
sudo apt-get install postgresql postgresql-contrib
```

Configure Now we can start configuring PostgrSQL. We need to create a database, create a user, and grant the user we created access to the database we created. Start off by running the following command:

```
sudo su - postgres
```

Your terminal prompt should now say *postgres@yourserver*. Run the following command to create a database:

```
createdb mydb
```

You now have a database named mydb. Now create a database user:

```
createuser -P {{username}}
```

Note: Replace the place holder {{username}} with the new username without {{}}

You will now be met with a series of 6 prompts. The first one will ask you for the name of the new user. Use whatever name you would like. The next two prompts are for your password and confirmation of password for the new user. For the last 3 prompts just enter "n" and hit "enter". This just ensures your new users only has access to what you give it access to and nothing else. Now activate the PostgreSQL command line interface:

```
psql
```

Grant the new user access to the new database:

```
GRANT ALL PRIVILEGES ON DATABASE mydb TO myuser;
```

You now have a PostgreSQL database and a user to access that database with.

Step Four: Install Django and Create a Project

Install We activate our virtualenv before we install any python packages:

```
source ~/myenv/bin/activate
```

You should now see that "(myenv)" has been appended to the beginning of your terminal prompt. This will help you to know when your virtualenv is active and which virtualenv is active.

We can now install Django in our virtualenv using pip:

```
pip install django
```

Note: After running this command you will have Django 1.7 installed in your system

---

Configure a New Project With django installed we can now create a new project to test that our stack is working. First change to the directory where you want your project source to live (we chose home ~):

```
cd ~
```

Now run the following command to create a project directory:

```
djang-admin.py startproject myproject
```

If we want django to talk to our database, we need to install a backend for PostgreSQL:

```
pip install psycopg2
```

Now we can edit the django database settings in the settings.py file using a command line editor:

```
cd ~/myproject/myproject
vim settings.py
```

Find the database settings and edit them to look like this:

```
DATABASES = {
        'default': {
                    'ENGINE': 'django.db.backends.postgresql_psycopg2', # Add 'postgresql_psycopg2',
                    'NAME': 'mydb',                      # Or path to database file if using sqlite3
                    # The following settings are not used with sqlite3:
                    'USER': 'myuser',
                    'PASSWORD': 'password',
                    'HOST': 'localhost',                       # Empty for localhost through domain so
                    'PORT': '',                      # Set to empty string for default.
        }
}
```

Save and exit the file. Then move up to your main project directory and run django's database configurtion tool:

```
cd ~/myproject/
python manage.py syncdb
```

Note: Familiarize yourself wuth `./manage.py makemigrations` and `./manage.py migrate` commands

You should see some output describing what tables were installed, followed by a prompt asking if you want to create a superuser. Just say no for now.

Step Five: Install and Configure Gunicorn

Gunicorn is a very powerful Python WSGI HTTP server.

Install Gunicorn is a python package so activate your virtualenv and install it using pip:

```
source ~/myenv/bin/activate
pip install gunicorn
```

Configure For now we are going to configure gunicorn using the most basic configuration with default settings:

```
gunicorn --bind localhost:8001 myproject.wsgi:application
```

Note:

Please be careful it's a Python import syntax not a file system path
Also to make the command run in the background append **&** to the end of the line

---

Now go to your web browser and visit localhost:8001 and see what you get. You should get the Django welcome screen.

Step Six: Install and Configure NGINX

NGINX is an incredibly fast and light-weight web server. We will use it to serve up our static files for our Django app.

Install To install nginx just run this command:

```
sudo apt-get install nginx
```

Configure Make sure that nginx is running:sten:

sudo service nginx start

We're going to be using NGINX to serve our static files so first we need to decide where our static files will live. Edit the django settings.py file and add STATIC_ROOT setting it to the following:

```
STATIC_ROOT = '/home/{{ user }}/static/'
```

Tip: Remember to replace {{ user }} with your own username on your VM

Now we can set up NGINX to handle the files in our static directory. Open a new NGINX config file:

```
sudo vim /etc/nginx/sites-available/myproject
```

Now add the following to the file:

```
server {
        listen 80; # to listen on the default HTTP port but you need to be root ;)
        server_name {{your ip}};

        access_log off;

        location /static/ {
```

```
                alias /home/{{user}}/static/;
        }

        location / {
                proxy_pass http://127.0.0.1:8001;
                proxy_set_header Host $host;
        }
}
```

Now we need to set up a symbolic link in the /etc/nginx/sites-enabled directory that points to this configuration file. That is how NGINX knows this site is active. Change directories to /etc/nginx/sites-enabled like this:

```
cd /etc/nginx/sites-enabled
sudo ln -s ../sites-available/myproject
```

Warning: Dont' forget to disable the **default** website in `/etc/nginx/site-enabled`

Now restart NGINX:

```
sudo service nginx restart
```

And that's it! You now have Django installed and working with PostgreSQL and your app is web accessible with NGINX serving static content and Gunicorn serving as your app server.

Tip:

If it doesn't work that means you forgot to run gunicorn in the background

---

1.3 Phase II: Team Project Workflow

1.3.1 Introduction

A workshop/tutorial for Python/Django developers who would like to contribute more to the projects they use, but need more grounding in some of the tools required.

The workshop will take participants through the complete cycle of identifying a simple issue in a Django or Python project, writing a patch with tests and documentation, and submitting it.

The purpose of the workshop

Don't be afraid to commit will help put you in a position to commit successfully to collaborative projects.

You'll find it particularly useful if you think you have some good coding ideas, but find that managing the development process sometimes gets in the way of your actual development.

What's covered

virtualenv and **pip** will help you manage your own work in a more streamlined and efficient way.

Git and **GitHub** will also help you manage your own workflow and development, and will make it possible for you to collaborate effectively with others. The Django Project, like many other open projects, uses both.

Automated tests will help you develop your software faster, better and more easily and give other developers more confidence in your contributions.

Documentation - being able to create, manage and publish documentation in an efficient and orderly way will make your work more accessible and more interesting to other people.

Contributing - how to submit your work

1.3.2 Contents

Prerequisites

What you need to know

The tutorial assumes some basic familiarity with the commandline prompt in a terminal.

You'll need to know how to install software. Some of the examples given refer to Debian/Ubuntu's `apt-get`; you ought to know what the equivalent is on whatever operating system you're using.

You'll also need to know how to edit plain text or source files of various kinds.

It will be very useful to have some understanding of Python, but it's not strictly necessary. However, if you've never done any programming, that will probably be an obstacle.

Software

The tutorial assumes that you're using a Unix-like system, but there should be no reason why (for example) it would not work for Windows users.

You'll need a suitable text editor, that you're comfortable using.

Other software will be used, but the tutorial will discuss its installation.

However, your environment *does* need to be set up appropriately, and you *will* need to know how to use it effectively.

If your system already has Django and/or Python packages running that you've installed, you probably already have what you need and know what you need to know. All the same:

Platform-specific notes

GNU/Linux Please make sure that you know how to use your system's package manager, whether it's `aptitude` or `YUM` or something else.

Mac OS X There are two very useful items that you should install.

- Command Line Tools for Xcode, various useful components (requires registration)
- Homebrew, a command line package manager

Python You'll need a reasonably up-to-date version of Python installed on your machine. 2.6 or newer should be fine.

Git Please do check you can install Git:

```
sudo apt-get install git # for Debian/Ubuntu users
```

or:

```
brew install git # for Mac OS X users with Homebrew installed
```

There are other ways of installing Git; you can even get a graphical Git application, that will include the commandline tools. These are described at:

> http://git-scm.com/book/en/Getting-Started-Installing-Git

Virtualenv and pip

In this section you will:

- use pip to install packages
- install virtualenv
- create and destroy, activate and deactivate virtual environments
- use `pip freeze` to show installed items

What is virtualenv?

Virtualenv lets you create and manage virtual Python environments.

If you're running a Python project for deployment or development, the chances are that you'll need more than one version of it, or the numerous other Python applications it depends upon, at any one time.

For example, when a new version of Django is released, you might want to check your project is still compatible. You don't want to have to set up a whole new server with a different version of Django to find out.

With virtualenv, you can quickly set up a a brand new Python environment, and install your components into it - along with the new version of Django, without touching or affecting what you already have running.

You can have literally dozens of virtualenvs on the same machine, all running different versions of your Python software, all independently of each other, and can safely make changes to one without affecting anything else.

pip goes hand-in-hand with virtualenv; in fact, it comes with virtualenv (as well as separately). It's an installer, and is the easiest way to install things into a virtualenv.

Install virtualenv

Try:

```
virtualenv --version
```

Do you have a version lower than 1.9? Upgrade it:

```
sudo pip install --upgrade virtualenv
hash -r # purge shell's PATH, though this may not be necessary for you
```

If you got a "Command not found" when you tried to use virtualenv, try:

```
sudo pip install virtualenv
```

or:

```
sudo apt-get install python-virtualenv # for a Debian-based system - but
it may not be up-to-date
```

If that fails or you're using a different system, you might need more help:

> Virtualenv installation documentation

Create and activate a virtual environment

```
virtualenv my-first-virtualenv
cd my-first-virtualenv
source bin/activate
```

Notice how your command prompt tells you that the virtualenv is active (and it remains active even while you're not in its directory):

```
(my-first-virtualenv)~/my-first-virtualenv$
```

Using pip

pip freeze `pip freeze` lists installed Python packages:

```
(my-first-virtualenv)daniele@v029:~/my-first-virtualenv$ pip freeze
argparse==1.2.1
distribute==0.6.24
pyasn1==0.1.7
virtualenv==1.9.1
wsgiref==0.1.2
```

pip install Earlier, you may have used `sudo pip install`. You **don't** need `sudo` now, because you're in a virtualenv. Let's install something.

```
pip install rsa
```

pip will visit PyPI, the Python Package Index, and will install Python-RSA (a "Pure-Python RSA implementation"). It will also install its dependencies - things it needs - if any have been listed at PyPI.

Now see what `pip freeze` reports. You will probably find that as well as Python-RSA it installed some other packages - they were ones that Python-RSA needed.

And try:

```
(my-first-virtualenv)~/my-first-virtualenv$ python
Python 2.7.2+ (default, Jul 20 2012, 22:15:08)
[GCC 4.6.1] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> import rsa
```

To uninstall it:

```
pip uninstall rsa
```

To install a particular version:

```
pip install rsa==3.0
```

To ugrade the package to the latest version:

```
pip install --upgrade rsa
```

Where packages get installed Your virtualenv has a site-packages directory, in the same way your system does. So now rsa can be found in:

```
~/my-first-virtualenv/lib/python2.7/site-packages/rsa
```

(It's possible that you'll have a different version of Python listed in that path.)

Dependencies Python-RSA doesn't have any dependencies, but if it did, and if those dependencies had dependencies, pip would install them all.

So if all the package authors have done a good job of informing PyPI about their software's requirements, you can install a Django application, for example, and pip will will install it, and Django, and possibly dozens of other pieces of software, all into your virtualenv, and without your having to make sure that everything required is in place.

Managing virtualenvs

Create a second virtualenv

```
cd ~/ # let's not create it inside the other...
virtualenv my-second-virtualenv
```

When you activate your new virtualenv, it will deactivate the first:

```
cd my-second-virtualenv
source bin/activate
```

`pip freeze` will show you that you don't have Python-RSA installed in this one - it's a completely different Python environment from the other, and both are isolated from the system-wide Python setup.

Deactivate a virtualenv manually Activating a virtualenv automatically deactivates one that was previously active, but you can also do this manually:

```
deactivate
```

Now you're no longer in any virtualenv.

–system-site-packages When you create a virtualenv, it doesn't include any Python packages already installed on your system. But sometimes, that *is* what you want. In that case you'd do:

```
virtualenv --system-site-packages my-third-virtualenv
```

remove a virtualenv virtualenvs are disposable. You can get rid of these:

```
cd ~/
rm -r  my-first-virtualenv my-second-virtualenv my-third-virtualenv
```

And that's pretty much all you need to get started and to use pip and virtualenv effectively.

Git and GitHub

Git and GitHub

In this section you will:

- create a GitHub account
- create your own fork of a repository
- create a new Git branch
- edit and commit a file on GitHub
- make a pull request
- merge upstream changes into your fork

What is it? **Git** is a source code management system, designed to support collaboration.

GitHub is a web-based service that hosts Git projects, including Django itself: https://github.com/django/django.

The key idea in Git is that it's *distributed*. If you're not already familiar with version control systems, then explaining why this is important will only introduce distinctions and complications that you don't need to worry about, so that's the last thing I will say on the subject.

Set up a GitHub account

- sign up at GitHub if you don't already have an account

It's free.

Some basic editing on GitHub

Forking

- visit https://github.com/evildmp/afraid-to-commit/

You can do various things there, including browsing through all the code and files.

- hit the **Fork** button

A few moments later, you'll have your own copy, on GitHub, of everything in that repository, and from now on you'll do your work on your copy of it.

Your copy is at `https://github.com/<your github account>/afraid-to-commit/`.

You will typically do this for any Git project you want to contribute to. It's good for you because it means you don't have to sign up for access to a central repository to be permitted to work on it, and even better for the maintainers because they certainly don't want to be managing an small army of volunteers on top of all their other jobs.

Note: Don't worry about all the forks

> You'll notice that there might be a few forks of https://github.com/evildmp/afraid-to-commit; if you have a look at https://github.com/django/django you'll see thousands. There'll even be forks of the forks. Every single one is complete and independent. So, which one is the real one - which one is *the* Django repository?
>
> In a technical sense, they all are, but the more useful answer is: the one that most people consider to be the canonical or official version.

In the case of Django, the version at https://github.com/django/django is the one that forms the basis of the package on PyPI, the one behind the https://djangoproject.com/ website, and above all, it's the one that the community treats as cannonical and official, not because it's the original one, but because it's the most useful one to rally around.

The same goes for https://github.com/evildmp/afraid-to-commit and its more modest collection of forked copies. If I stop updating it, but someone else is making useful updates to their own fork, then in time theirs might start to become the one that people refer to and contribute to. This could even happen to Django itself, though it's not likely to any time soon.

The proliferation of forks doesn't somehow dilute the original. Don't be afraid to create more. Forks are simply the way collaboration is made possible.

Create a new branch Don't edit the *master* (default) branch of the repository. It's much better to edit the file in a new branch, leaving the *master* branch clean and untouched:

1. select the **branch** menu
2. in *Find or create a branch...* enter `add-my-name`
3. hit **Create branch: add-my-name**

Note: Don't hesitate to branch

As you may have noticed on GitHub, a repository can have numerous branches within it. Branches are ways of organising work on a project: you can have a branch for a new feature, for trying out something new, for exploring an issue - anything at all.

Just as virtualenvs are disposable, so are **branches** in Git. You *can* have too many branches, but don't hesitate to create new ones; it costs almost nothing.

It's a good policy to create a new branch for every new bit of work you start doing, even if it's a very small one.

It's especially useful to create a new branch for every new feature you start work on.

Branch early and branch often. If you're in any doubt, create a new branch.

Edit a file GitHub allows you to edit files online. This isn't the way you will normally use Git, and it's certainly not something you'll want to spend very much time doing, but it's handy for very small changes, for example typos and spelling mistakes you spot.

1. go to `https://github.com/<your github account>/afraid-to-commit`
2. find the `attendees_and_learners.rst` file
3. hit the **Edit** button
4. add your name (just your name, you will add other information later) to the appropriate place in the file. If you're following the tutorial by yourself, add your details in the *I followed the tutorial online* section.

Commit your changes

- hit **Commit Changes**

Now *your* copy of the file, the one that belongs to *your* fork of the project, has been changed; it's reflected right away on GitHub.

If you managed to mis-spell your name, or want to correct what you entered, you can simply edit it again.

What you have done now is make some changes, in a new branch, in your own fork of the repository. You can see them there in the file.

Make a Pull Request When you're ready to have your changes incorporated into my original/official/canonical repository, you do this by making a **Pull Request**.

- go back to `https://github.com/<your github account>/afraid-to-commit`

You'll see that GitHub has noted your recent changes, and now offers various buttons to allow you to compare them with the original or make a pull request.

- hit **Compare & pull request**

This will show you a *compare view*, from which you can make your pull request.

When preparing for a pull request, GitHub will show you what's being compared:

```
evildmp:master ... <your github account>:add-my-name
```

On the left is the **base** for the comparison, my fork and branch. On the right is the **head**, your fork and branch, that you want to compare with it.

A pull request goes from the **head** to the **base** - from right to left.

You can change the bases of the comparison if you need to:

1. hit **Edit**
2. select the forks and branches as appropriate

You want your version, the **head branch** of the **head fork** - on the right - with some commits containing file changes, to be sent to my **base repo** - on the left.

1. hit the **Pull Request** button
2. add a comment if you like (e.g. "please add me to the attendees list")
3. hit **Send pull request**

You have now made a pull request to an open-source community project - if it's your first one, congratulations.

GitHub will notify me (by email and on the site, and will show me the changes you're proposing to make). It'll tell me whether they can be merged in automatically, and I can reject, or accept, or defer a decision on, or comment on, your pull request.

GitHub can automatically merge your contribution into my repository if mine hasn't changed too much since you forked it. If I want to accept it but GitHub can't do it automatically, I will have to merge the changes manually (we will cover this later).

Once they're merged, your contributions will become a part of https://github.com/evildmp/afraid-to-commit. And this is the basic lifecycle of a contribution using git: *fork* > *edit* > *commit* > *pull request* > *merge*.

Incorporate upstream changes into your master In the meantime, other people may have made their own forks, edits, commits, and pull requests, and I may have merged those too - other people's names may now be in the list of attendees. Your own version of afraid-to-commit, *downstream* from mine, doesn't yet know about those.

Since your work is based on mine, you can think of my repository as being *upstream* of yours. You need to merge any *upstream* changes into *your* version, and you can do this with a pull request on GitHub too.

This time though you will need to switch the bases of the comparison around, because the changes will be coming from *my version* to *yours*.

1. hit **Pull Request** once more

2. hit **Edit**, to switch the bases
3. change the **head repo** on the right to *my* version, `evildmp/afraid-to-commit`, branch *master*
4. change the **base repo** to yours, and the **base branch** to *master*, so the comparison bar looks like:

   ```
   <your github account>:master ... evildmp:master
   ```

5. hit **Click to create a pull request for this comparison**
6. add a **Title** (e.g. "merging upstream master on Github) and hit **Send pull request**

You're sending a pull request to to *yourself*, based on updates in my repository. And in fact if you check in your **Pull Requests** on GitHub, you'll see one there waiting for you, and you too can review, accept, reject or comment on it.

If you decide to **Merge** it, your fork will now contain any changes that other people sent to me and that I merged.

The story of your work is this: you **forked** away from my codebase, and then created a new **branch** in your fork.

Then you **committed** changes to your branch, and sent them **upstream** back to me (with a **pull request**).

I **merged** your changes, and perhaps those from other people, into my codebase, and you **pulled** all my recent changes back into your *master* branch (again with a **pull request**).

So now, your *master* and mine are once more in step.

Git on the commandline

In this section you will:

- install and configure Git locally
- create your own local clone of a repository
- create a new Git branch
- edit a file and stage your changes
- commit your changes
- push your changes to GitHub
- make a pull request
- merge upstream changes into your fork
- merge changes on GitHub into your local clone

So far we've done all our Git work using the GitHub website, but that's usually not the most appropriate way to work.

You'll find that most of your Git-related operations can and need to be done on the commandline.

Install/set up Git

```
sudo apt-get install git # for Debian/Ubuntu users
brew install git # for Mac OS X users with Homebrew installed
```

There are other ways of installing Git; you can even get a graphical Git application, that will include the commandline tools. These are described at:

> http://git-scm.com/book/en/Getting-Started-Installing-Git

Tell Git who you are First, you need to tell Git who you are:

```
git config --global user.email "you@example.com"
git config --global user.name "Your Name"
```

Give GitHub your public keys This is a great timesaver: if GitHub has your public keys, you can do all kinds of things from your commandline without needing to enter your GitHub password.

- https://github.com/settings/ssh

https://help.github.com/articles/generating-ssh-keys explains much better than I can how to generate a public key.

This tutorial assumes you have now added your public key to your GitHub account. If you haven't, you'll have to use *https* instead, and translate from the format of GitHub's *ssh* URLS.

For example, when you see:

```
git@github.com:evildmp/afraid-to-commit.git
```

you will instead need to use:

```
https://github.com/evildmp/afraid-to-commit.git
```

See https://gist.github.com/grawity/4392747 for a discussion of the different protocols.

Some basic Git operations When we worked on GitHub, the basic work cycle was *fork* > *edit* > *commit* > *pull request* > *merge*. The same cycle, with a few differences, is what we will work through on the commandline.

Clone a repository When you made a copy of the *Don't be afraid to commit* repository on GitHub, that was a *fork*. Getting a copy of a repository onto your local machine is called *cloning*. Copy the *ssh URL* from `https://github.com/<your github account>/afraid-to-commit`, then:

```
git clone git@github.com:<your github account>/afraid-to-commit.git
```

Change into the newly-created `afraid-to-commit` directory, where you'll find all the source code of the *Don't be afraid to commit* project.

Now you're in the **working directory**, the set of files that you currently have in front of you, available to edit. We want to know its status:

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

Create a new branch Just as you did on GitHub, once again you're going to create a new branch, based on *master*, for new work to go into:

```
$ git checkout -b amend-my-name
Switched to a new branch 'amend-my-name
```

`git checkout` is a command you'll use a lot, to switch between branches. The `-b` flag tells it to **create a new branch** at the same time. By default, the new branch is based upon whatever branch you were on.

You can also choose what to base the new branch on. A very common thing to do is:

```
git checkout -b new-branch-name upstream/master
```

This creates a new branch `new-branch-name`, based on `upstream/master`.

Edit a file

1. find the `attendees_and_learners.rst` file in your working directory
2. after your name and email address, add your Github account name
3. save the file

`git status` is always useful:

```
$ git status
# On branch amend-my-name
# Changes not staged for commit:
#   (use "git add <file>..." to update what will be committed)
#   (use "git checkout -- <file>..." to discard changes in working directory)
#
#   modified:   attendees_and_learners.rst
#
no changes added to commit (use "git add" and/or "git commit -a")
```

What this is telling us:

- we're on the *amend-my-name* branch
- that we have one modified file
- that there's nothing to commit

These changes will only be applied to this branch when they're committed. You can `git add` changed files, but until you commit they won't belong to any particular branch.

Note: When to branch

You didn't actually *need* to create your new *amend-my-name* branch until you decided to commit. But creating your new branches before you start making changes makes it less likely that you will forget later, and commit things to the wrong branch.

Stage your changes Git has a **staging area**, for files that you want to commit. On GitHub when you edit a file, you commit it as soon as you save it. On your machine, you can edit a number of files and commit them altogether.

Staging a file in Git's terminology means adding it to the staging area, in preparation for a commit.

Add your amended file to the staging area:

```
git add attendees_and_learners.rst
```

and check the result:

```
$ git status
# On branch amend-my-name
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#   modified:   attendees_and_learners.rst
#
```

If there are other files you want to change, you can add them when you're ready; until you commit, they'll all be together in the staging area.

What gets staged? It's not your files, but the **changes to your files**, that are staged. Make some further change to `attendees_and_learners.rst`, and run `git status`:

```
$ git status
# On branch amend-my-name
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#   modified:   attendees_and_learners.rst
#
# Changes not staged for commit:
#   (use "git add <file>..." to update what will be committed)
#   (use "git checkout -- <file>..." to discard changes in working directory)
#
#   modified:   attendees_and_learners.rst
#
```

Some of the changes in `attendees_and_learners.rst` will be committed, and the more recent ones will not.

- run `git add` on the file again to stage the newer changes

Commit your changes When you're happy with your files, and have added the changes you want to commit to the staging area:

```
git commit -m "added my github name"
```

The `-m` flag is for the message ("added my github name") on the commit - every commit needs a commit message.

Push your changes to GitHub When you made a change on GitHub, it not only saved the change and committed the file at the same time, it also showed up right away in your GitHub repository. Here there is an extra step: we need to **push** the files to GitHub.

If you were pushing changes from *master* locally to *master* on GitHub, you could just issue the command `git push` and let Git work out what needs to go where.

It's always better to be explicit though. What's more, you have multiple branches here, so you need to tell git *where* to push (i.e. back to the remote repository you cloned from, on GitHub) and *what* exactly to push (your new branch).

The repository you cloned from - yours - can be referred to as **origin**. The new branch is called *amend-my-name*. So:

```
$ git push origin amend-my-name
Counting objects: 34, done.
Delta compression using up to 2 threads.
Compressing objects: 100% (21/21), done.
Writing objects: 100% (28/28), 6.87 KiB, done.
Total 28 (delta 13), reused 12 (delta 7)
To git@github.com:evildmp/afraid-to-commit.git
 * [new branch]      amend-my-name -> amend-my-name
```

Note: Be explicit!

Next time you want to push committed changes in *amend-my-name*, you won't *need* to specify the branch - you can simply do `git push`, because now *amend-my-name* exists at both ends. However, it's *still* a good idea to be explict. That way you'll be less likely to get a surprise you didn't want, when the wrong thing gets pushed.

Check your GitHub repository

- go to https://github.com/<your GitHub name>/afraid-to-commit
- check that your new *amend-my-name* branch is there
- check that your latest change to `attendees_and_learners.rst` is in it

Send me a pull request You can make more changes locally, and continue committing them, and pushing them to GitHub. When you've made all the changes that you'd like me to accept though, it's time to send *me* a pull request.

Important: make sure that you send it from your new branch *amend-my-name* (not from your *master*) the way you did before.

And if I like your changes, I'll merge them.

Note: Keeping master 'clean'

You *could* of course have merged your new branch into your *master* branch, and sent me a pull request from that. But, once again, it's a good policy to keep your *master* branch, on GitHub too, clean of changes you make, and only to pull things into it from upstream.

In fact the same thing goes for other branches on my upstream that you want to work with. Keeping them clean isn't strictly necessary, but it's nice to know that you'll always be able to pull changes from upstream without having to tidy up merge conflicts.

Incorporate upstream changes Once again, I may have merged other people's pull requests too. Assuming that you want to keep up-to-date with my changes, you're going to want to merge those into your GitHub fork as well as your local clone.

So:

- on GitHub, pull the upstream changes into your fork the way you did previously

Then switch back to your master branch in the usual way (`git checkout master`). Now, fetch updated information from your GitHub fork (**origin**), and merge the master:

```
git fetch
git merge origin/master
```

So now we have replicated the full cycle of work we described in the previous module.

Note: `git pull`

Note that here instead of `git fetch` followed by `git merge`, you could have run `git pull`. The `pull` operation does two things: it **fetches** updates from your GitHub fork (**origin**), and **merges** them.

However, be warned that occasionally `git pull` won't always work in the way you expect, and doing things the explicit way helps make what you are doing clearer.

`git fetch` followed by `git merge` is generally the safer option.

Switching between branches locally Show local branches:

```
git branch
```

You can switch between local branches using `git checkout`. To switch back to the *master* branch:

```
git checkout master
```

If you have a changed tracked file - a tracked file is one that Git is managing - it will warn you that you can't switch branches without either committing, abandoning or 'stashing' the changes:

Commit You already know how to commit changes.

Abandon You can abandon changes in a couple of ways. The recommended one is:

```
git checkout <file>
```

This checks out the previously-committed version of the file.

The one that is not recommended is:

```
git checkout -f <branch>
```

The `-f` flag forces the branch to be checked out.

Note: Forcing operations with `-f`

Using the `-f` flag for Git operations is to be avoided. It offers plenty of scope for mishap. If Git tells you about a problem and you force your way past it, you're inviting trouble. It's almost always better to find a different way around the problem than forcing it.

`git push -f` in particular has ruined a nice day for many people.

Stash If you're really interested, look up `git stash`, but it's beyond the scope of this tutorial.

Working with remotes

In this section you will:

- add a remote repository to your local clone
- fetch remote information
- checkout a remote branch
- merge an upstream branch

Managing remotes Your repository on GitHub is the **remote** for the clone on your local machine. By default, your clone refers to that remote as **origin**. At the moment, it's the only remote you have:

```
$ git remote
origin
```

```
$ git remote show origin
    * remote origin
      Fetch URL: git@github.com:evildmp/afraid-to-commit.git
      Push  URL: git@github.com:evildmp/afraid-to-commit.git
      HEAD branch: master
      Remote branches:
        amend-my-name tracked
        master        tracked
```

```
    Local branch configured for 'git pull':
      master merges with remote master
  Local refs configured for 'git push':
    amend-my-name pushes to amend-my-name (up to date)
    master        pushes to master        (up to date)
```

It's very useful for Git to know about other remote repositories too. For example, at the end of the previous section, we considered a conflict between your GitHub fork and and the upstream GitHub repository. The only way to fix that is locally, on the command line, and by being able to refer to both those remotes.

Now you *can* refer to a remote using its full address:

> https://github.com/evildmp/afraid-to-commit

But just as your remote is called **origin**, you can give any remote a more memorable name. Your own **origin** has an upstream repository (mine); it's a convention to name that **upstream**.

Add a remote

```
git remote add upstream git@github.com:evildmp/afraid-to-commit.git
```

Fetch remote data The remote's there, but you don't yet have any information about it. You need to **fetch** that:

```
git fetch upstream
```

This means: get the latest information about the branches on **upstream**.

List remote branches `git branch` shows your local branches.

To see a list of them all, including the remote branches:

```
git branch -a
```

Checkout a remote branch You'll have seen from `git branch -a` that there's a branch called *additional-branch* on the **upstream** repository.

You can check this out:

```
git checkout -b additional-branch upstream/additional-branch
```

This means: create and switch to a new local branch called *additional-branch*, based on branch *additional-branch* of the remote **upstream**.

Managing *master* on the commandline Until now, you have only updated your *master* on GitHub using GitHub itself. Sometimes it will be much more convenient to do it from your commandline. There are various ways to do it, but here's one:

```
git checkout master # switch back to the master branch
git fetch upstream # update information about the remote
git merge upstream/master # merge the changes referred to by
upstream/master
```

`git status` will tell you that your local master is ahead of your *master* at **origin**.

Push your changes to master:

```
git push origin master
```

And now your *master* on GitHub contains everything my *master* does; it's up-to-date and clean.

Resolving conflicts

In this section you will:

- encounter a merge conflict on GitHub
- encounter a merge conflict on the commandline
- resolve the conflict in a new temporary Git branch

Encountering a merge conflict on GitHub Sometimes you'll discover that your GitHub fork and the upstream repository have changes that GitHub can't merge.

There's an *unmergeable-branch* at https://github.com/evildmp/afraid-to-commit. It's unmergeable because it deliberately contains changes that conflict with other changes made in *master*.

Using the GitHub interface, try creating a pull request from *unmergeable-branch* to your *master* on GitHub. If you do, GitHub will tell you:

```
We can't automatically merge this pull request.

Use the command line to resolve conflicts before continuing.
```

GitHub will in fact tell you the steps you need to take to solve this, but to understand what's actually happening, and to do it yourself when you need to, we need to cover some important concepts.

Merging changes from a remote branch

```
# make sure you have the latest data from upstream
$ git fetch upstream
# create and switch to a new branch based on master to explore the conflict
$ git checkout -b explore-conflict upstream/master
# now try merging the unmergeable-branch into it
$ git merge upstream/unmergeable-branch
Auto-merging attendees_and_learners.rst
CONFLICT (content): Merge conflict in attendees_and_learners.rst
Automatic merge failed; fix conflicts and then commit the result.
```

When there's a conflict, Git marks them for you in the files. You'll see sections like this:

```
<<<<<<< HEAD
* Daniel Pass <daniel.antony.pass@googlemail.com>
* Kieran Moore
=======
* Kermit the Frog
* Miss Piggy
>>>>>>> upstream/unmergeable-branch
```

The first section, `HEAD` is what you have in your version. The second section, `upstream/unmergeable-branch` is what Git found in the version you were trying to pull in.

You'll have to decide what the file should contain, and you'll need to edit it. If you decide you want the changes from both versions:

```
* Daniel Pass <daniel.antony.pass@googlemail.com>
* Kieran Moore
* Kermit the Frog
* Miss Piggy

$ git add attendees_and_learners.rst
$ git commit -m "fixed conflict"
[explore-conflict 91a45ac] fixed conflict
```

Note: Create new branches when resolving conflicts

It's very sensible *not* to do merging work in a branch you have done valuable work in. In the example above, your *explore-conflict* branch is based on master and doesn't contain anything new, so it will be easy to re-create if it all goes wrong.

If you had a branch that contained many complex changes however, you certainly wouldn't want to discover dozens of conflicts making a mess in the files containing all your hard work.

So remember, **branches are cheap and disposable**. Rather than risk messing up the branch you've been working on, create a new one specially for the purpose of discovering what sort of conflicts arise, and to give you a place to work on resolving them without disturbing your work so far.

You might have conflicts across dozens of files, if you were unlucky, so it's very important to be able to backout gracefully and at the very least leave things as they were.

More key Git techniques

There are a few more key Git commands and techniques that you need to know about.

.gitignore When you're working with Git, there are lots of things you won't want to push, including:

- hidden system files
- .pyc files
- rendered documentation files
- ... and many more

`.gitignore`, https://github.com/evildmp/afraid-to-commit/blob/master/.gitignore, is what controls this. You should have a `.gitignore` in your projects, and they should reflect *your* way of working. Mine include the things that my operating system and tools throw into the repository; you'll find soon enough what yours are.

With a good `.gitignore`, you can do things like:

```
git add docs/
```

and add whole directories at a time without worrying about including unwanted files.

Starting a new Git project You've been working so far with an existing Git project. It's very easy to start a brand new project, or turn an existing one into a Git project. On GitHub, just hit the **New repository** button and follow the instructions.

Combining Git and pip When you used pip to install a package inside a virtualenv, it put it on your Python path, that is, in the virtualenv's `site-packages` directory. When you're actually working on a package, that's not so convenient - a Git project is the most handy thing to have.

On the other hand, cloning a Git repository doesn't install it on your Python path (assuming that it's a Python application), so though you can work on it, you can't actually use it and test it as an installed package.

However, pip is Git-aware, and can install packages *and* put them in a convenient place for editing - so you can get both:

```
cd ~/
virtualenv git-pip-test
source git-pip-test/bin/activate
pip install -e git+git@github.com:python-parsley/parsley.git#egg=parsley
```

The `-e` flag means editable; `git+` tells it to use the Git protocol; `#egg=parsley` tells it what to call it.

(Should you find that this causes an error, try using quotes around the target:

```
pip install -e "git+git@github.com:python-parsley/parsley.git#egg=parsley"
```

)

You can also specify the branch:

```
pip install -e git+git@github.com:python-parsley/parsley.git@master#egg=parsley
```

And now you will find an editable Git repository installed at:

> ~/git-pip-test/src/parsley

which is where any other similarly-installed packages will be, and just to prove that it really is installed:

```
$ pip freeze
-e git+git@github.com:python-parsley/parsley.git@e58c0c6d67142bf3ceb6eceffd50cf0f8dae9da1#egg=Parsley
wsgiref==0.1.2
```

Git is a source code management system, designed to support collaboration.

GitHub is a web-based service that hosts Git projects, including Django itself: https://github.com/django/django.

Git is a quite remarkable tool. It's fearsomely complex, but you can start using it effectively without needing to know very much about it. All you really need is to be familiar with some basic operations.

The key idea in Git is that it's *distributed*. If you're not already familiar with version control systems, then explaining why this is important will only introduce distinctions and complications that you don't need to worry about, so that's the last thing I will say on the subject.

Testing Django applications

The Django Project tutorials include a testing tutorial, which is what we'll follow. It assumes you have built the Polls application over the previous four parts of the tutorial.

To save you having to create it now, I've made the tutorial project application on GitHub. If you haven't followed the Django tutorial right the way through though, you really should.

This section of the workshop will follow roughly the tutorial at https://docs.djangoproject.com/en/1.5/intro/tutorial05/.

Get the code set up

Create a virtualenv

```
$ virtualenv testingtutorial
New python executable in testingtutorial/bin/python
Installing setuptools.............done.
Installing pip...............done.
$ cd testingtutorial/
$ source bin/activate
```

Install Django using pip

```
(testingtutorial)$ pip install django==1.5.1
Downloading/unpacking django
[...]
```

Clone the repository Visit https://github.com/evildmp/django-tutorials-project. Fork the repository.

```
(testingtutorial)$ git clone git@github.com:<your git name>/django-tutorials-project.git
Cloning into 'django-tutorials-project'...
[...]
```

Checkout the code we want

```
(testingtutorial)$ cd django-tutorials-project/
(testingtutorial)$ git checkout tutorial-four
Branch tutorial-four set up to track remote branch tutorial-four from origin.
Switched to a new branch 'tutorial-four'
```

Fire it up

```
python manage.py syncdb # set up the database
```

Note: ImportError: No module named django.core.management

Maybe you don't have Django installed. That's no problem; you already know the answer to this:

```
pip install django # install Django into your virtualenv
```

You *will* need to be a superuser, so enter a username and password.

```
python manage.py runserver 0.0.0.0:8000 # start the runserver
```

Developing tests

The first bug Visit the site's admin, and create a new poll. You'll soon see that you can expose a little bug. In the list of polls in the admin, you'll see that polls published in the last day are considered to have been `published recently`, but so are polls whose `pub_date` is in the future.

We can fix that bug easily enough, but in order to be sure that it *stays* fixed, a test will help. So, we should:

1. create a test that exposes the bug
2. work on the bug
3. run the test, and go back to step 2 until the test passes

https://docs.djangoproject.com/en/1.5/intro/tutorial05/#we-identify-a-bug

Following good Git working practices, *before you change a single byte of code*, create a new branch to work in:

```
git checkout -b test-for-published-recently
```

When you've worked through that section in the tutorial, you'll have fixed the bug and created three tests. This would be a good moment to add your changed files, commit your changes, and push them to your GitHub repository before continuing.

Your next bugfix and tests What about your next bugfix and tests? Should they be in a branch based on *tutorial-four*, or based on *test-for-published-recently*?

That really depends on whether your next bugfix and tests can be regarded as independent of the ones in *test-for-published-recently*, in which case it might be better to start again based on *tutorial-four*:

```
git checkout -b improve-list-view tutorial-four
```

This will mean that you can make a pull request for those changes independently of the previous ones, meaning your pull request will be smaller, simpler and easier to understand - maintainers love that.

On the other hand, if you're going to build on the work you've just done, you'll need:

```
git checkout -b improve-list-view
```

Keep creating new branches as appropriate, as you work through the tutorial.

Following the tutorial

The Django testing tutorial covers a number of key ideas and approaches. It's worth following the whole thing, and making sure you understand the points its making.

Documentation using Sphinx and ReadTheDocs.org

Without documentation, however wonderful your software, other potential adopters and developers simply won't be very interested in it.

The good news is that there are several tools that will make presenting and publishing it very easy, leaving you only to write the content and mark it up appropriately.

For documentation, we'll use **Sphinx** to generate it, and **Read the Docs** to publish it. GitHub will be a helpful middleman.

If you have a package for which you'd like to create documentation, you might as well start producing that right away. If not, you can do it in a new dummy project.

Set up your working environment

The virtualenv As usual, create and activate a new virtualenv:

```
virtualenv documentation-tutorial
[...]
cd documentation-tutorial/
source bin/activate
```

The package or project

If you have an existing package to write documentation for If your package is on GitHub already and you want to start writing documention for, clone it now using Git. And of course, start a new branch:

```
git checkout -b first-docs
```

You can merge your docs into your master branch when they start to look respectable.

If you don't have an existing package that needs docs If you don't have a suitable existing package on GitHub, create a repository on GitHub the way you did before. Call it `my-first-docs`. Then create a Git repository locally:

```
mkdir my-first-docs
cd my-first-docs/
# Converts the directory into a git repository
git init
# Point this repo at the GitHub repo you just created
git remote add origin git@github.com:<your git username>/my-first-docs.git
# Create a new branch in which to do your work
git checkout -b first-docs
```

Create a `docs` directory And either way, create a `docs` directory for your docs to live in:

```
mkdir docs
```

Sphinx

Install Sphinx

```
pip install sphinx
```

It might take a minute or so, it has quite a few things to download and install.

sphinx-quickstart `sphinx-quickstart` will set up the source directory for your documentation. It'll ask you a number of questions. Mostly you can just accept the defaults it offers, and some are just obvious, but there are some you will want to set yourself as noted below:

```
sphinx-quickstart
```

Root path for the documentation `docs`

Project name `<your name>'s first docs`, or the name of your application

Source file suffix `.rst` is the default. (Django's own documentation uses `.txt`. It doesn't matter too much.)

You'll find a number of files in your `docs` directory now, including `index.rst`. Open that up.

Using Sphinx & reStructuredText

reStructuredText elements Sphinx uses reStructuredText. http://sphinx-doc.org/rest.html#rst-primer will tell you most of what you need to know to get started. Focus on the basics:

- paragraphs
- lists
- headings ('sections', as Sphinx calls them)
- quoted blocks

- code blocks
- emphasis, strong emphasis and literals

Edit a page Create an **Introduction** section in the `index.rst` page, with a little text in it; save it.

Create a new page You have no other pages yet. In the same directory as `index.rst`, create one called `all-about-me.rst` or something appropriate. Perhaps it might look like:

```
############
All about me
############

I'm Daniele Procida, a Django user and developer.

I've contributed to:

*   django CMS
*   Arkestra
*   Django
```

Sphinx needs to know about it, so in `index.rst`, edit the `.. toctree::` section to add the `all-about-me` page:

```
.. toctree::
   :maxdepth: 2

   all-about-me
```

Save both pages.

Render your documentation In the `docs` directory:

```
make html
```

This tells Sphinx to render your source pages. *Pay attention to its warnings* - they're helpful!

Note: Sphinx can be fussy, and sometimes about things you weren't expecting. For example, you well encounter something like:

```
WARNING: toctree contains reference to nonexisting document u'all-about-me'
...
checking consistency...
<your repository>/my-first-docs/docs/all-about-me.rst::
WARNING: document isn't included in any toctree
```

Quite likely, what has happened here is that you indented `all-about-me` in your `.. toctree::` with *four* spaces, when Sphinx is expecting *three*.

If you accepted the `sphinx-quickstart` defaults, you'll find the rendered pages in `docs/_build/html`. Open the `index.html` it has created in your browser. You should find in it a link to your new `all-about-me` page too.

Publishing your documentation

Exclude unwanted rendered directories Remember `.gitignore`? It's really useful here, because you don't want to commit your *rendered* files, just the source files.

In my `.gitignore`, I make sure that directories I don't want committed are listed. Check that:

```
_build
_static
_templates
```

are listed in `.gitignore`.

Add, commit and push `git add` the files you want to commit; commit them, and push to GitHub.

If this is your first ever push to GitHub for this project, use:

```
git push origin master
```

otherwise:

```
git push origin first-docs # or whatever you called this branch
```

Now have a look at the `.rst` documentation files on GitHub. GitHub does a good enough job of rendering the files for you to read them at a glance, though it doesn't always get it right (and sometimes seems to truncate them).

readthedocs.org However, we want to get them onto Read the Docs. So go to https://readthedocs.org, and sign up for an account if you don't have one.

You need to **Import** a project: https://readthedocs.org/dashboard/import/.

Give it the details of your GitHub project in the **repo** field - `git@github.com:<your git username>/my-first-docs.git`, or whatever it is - and hit **Create**.

And now Read the Docs will actually watch your GitHub project, and build, render and host your documents for you automatically.

It will update every night, but you can do better still: on GitHub:

1. select **settings** for your project (not for your account) in the navigation panel on the right-hand side
2. choose **Webhooks & Services**
3. enable `ReadTheDocs` under **Add Service** dropdown

... and now, every time you push documents to GitHub, Read the Docs will be informed that you have new documents to be published. It's not magic, but it's pretty close.

Contributing

Having identified a contribution you think you can usefully make to a project, how are you actually going to make it?

For nearly every project, the best first step is:

Talk to somebody about it

You may have been eating, sleeping, dreaming and otherwise living your idea for days, but until you discuss it, no-one else knows anything about it. To them, your patch will come flying out of the blue.

You need - usually - to introduce your idea, and - particularly if you're new to the community - yourself.

In the case of Django, this will typically mean posting to the Django Developers email list, django-developers@googlegroups.com (sign up at http://groups.google.com/group/django-developers), or raising it on #django-dev on irc.freenode.net.

Quite apart from letting people know about what you have to offer, it's an opportunity to get some feedback on your proposal.

Don't automatically expect your proposal to be considered a great idea. Be prepared to explain the need it meets and why you think your solution is a good one. Be prepared to do some more work.

Above all, you will need to be **patient**, **polite** and **persistent**.

File it

If one doesn't exist already, lay down a formal public marker raising the issue your contribution addresses. In Django's case, this will be a ticket on https://code.djangoproject.com/. For others, it's likely to be an issue on GitHub or whatever system they use. *Mention your earlier discussion!*

Do your homework

Every project has its standards for things like code and documentation, and its ways of working. They tend to follow a general pattern, but they often have their own little quirks or preferences - so **learn them**.

If you think that sounds tedious, it's nothing compared to the potential pain of having to manage or use code and documentation written according to the individual preferences of all its different contributors.

- https://docs.djangoproject.com/en/1.7/internals/contributing/
- https://docs.djangoproject.com/en/1.7/internals/contributing/writing-code/working-with-git/

Note that the Django Project's Git guidelines ask contributors to use `rebase` - which is firstly a little unusual, and secondly explained in the documentation above better than I can here - so read that.

And be prepared to get it wrong the first few times, and even the subsequent ones. It happens to everyone.

Submit it

Now you can make your pull request. Having prepared the way for it, and having provided the accompaniments - tests and documentation - that might be required, it has a good chance of enjoying a smooth passage into the project.

What to start with?

Documentation! It's the easy way in.

Everyone loves documentation, and unlike code where incompleteness or vagueness can make it worse than useless, documentation has to be really quite bad to be worse than no documentation.

What's more, writing documentation will help you better understand the things you're writing about, and if you're new to all this, that's going to put you in a better position to understand and improve code.

Some suitable Django Project tickets Have a look at one of the tickets specially selected for people doing this tutorial. They're not all for documentation, though most are.

Cheatsheet

virtualenv

create `virtualenv [name-of-virtualenv]`

include system's Python packages `virtualenv [name-of-virtualenv] --install-site-packages`

activate `source bin/activate`

deactivate `deactivate`

pip

install `pip install [name-of-package]`

install a particular version `pip install [name-of-package]==[version-number]`

upgrade `pip install --upgrade [name-of-package]`

uninstall `pip uninstall [name-of-package]`

show what's installed `pip freeze`

git

tell git who you are `git config --global user.email "you@example.com"`

`git config --global user.name "Your Name"`

clone a repository `git clone [repository URL]`

checkout `git checkout [branch]` switches to another branch

`git checkout -b [new-branch]` creates and switches to a new branch

`git checkout -b [new-branch] [existing branch]` creates and switches to a new branch based on an existing one

`git checkout -b [new-branch] [remote/branch]` creates and switches to a new branch based on `remote/branch`

`git checkout [commit sha]` checks out the state of the repository at a particular commit

current status of your local branches `git status`

show the commit you're on in the current working directory `git show`

commit `git commit -m "[your commit message]"`

add `git add [filename]` adds a file to the staging areas

`git add -u [filename]` - the `-u` flag will also remove deleted files

remote `git remote add [name] [remote repository URL]` sets up remote

`git remote show` lists remotes

`git remote show -v` lists remotes and their URLs

branch `git branch`

`git branch -a` to show remote branches too

fetch `git fetch` gets the latest information about the branches on the default remote

`git fetch [remote]` gets the latest information about the branches on the named remote

merge `git merge [branch]` merges the named branch into the working directory

`git merge [remote/branch] -m "[message]"` merges the branch referred to into the working directory - **don't forget to fetch the remote before the merge**

pull `fetch` followed by `merge` is often safer than `pull` - don't assume that `pull` will do what you expect it to

`git pull` fetches updates from the default remote and merges into the working directory

push `git push` pushes committed changes to the default remote, in branches that exist at both ends

`git push [remote] [branch]` pushes the current branch to the named `branch` on `remote`

log `git log` will show you a list of commits

Notes Throughout Git, anything in the form `remote/branchname` is a reference, not a branch.

Documentation

initialise Sphinx documentation `sphinx-quickstart`

render documentation `make html`

Attendees

This is a record of people who attended a *Don't be afraid to commit* workshop, or followed the tutorial in their own time.

Workshops

Full Stack Development Course in Kuwait, 17th November 2014

- Ghada Alnaqi https://github.com/gnaqi <ghnaqi@gmail.com>
- Nasser Hussain <nasser.bader@outlook.com> https://github.com/nasser-bader
- Musab Alshatti <m93b85@gmail.com> (https://github.com/m93b85)
- Lulwah AlKulaib <lkulaib@kisr.edu.kw> https://github.com/lalkulaib
- Ebtisam (e.al-slama@hotmail.com)https://github.com/ealsalamah
- Muneera Aljeri <mujeri@kisr.edu.kw> (https://github.com/muneera88)
- Abdulaziz Al-Massaeed (abdulaziz.almassaeed@ahliunited.com) (https://github.com/hackfoo)
- Nasser AlSnayen (nasser.lc9@gmail.com) <https://github.com/LC9>

Honor Guests

- Kermit the frog
- Captain Awesomesauce

PyCon Ireland in Dublin, 13th October 2014

- Randal McGuckin <randal.mcguckin@gmail.com>
- Laura Duggan https://github.com/labhra
- Jenny McGee
- Conor McGee <mcgeeco@tcd.ie> https://github.com/mcgeeco
- Nadja Deininger https://github.com/machinelady
- Andrew McCarthy
- Brian McDonnell <https://github.com/brianmcdonnell/>
- Brendan Cahill (https://github.com/brencahill/)
- Adam Dickey
- Paul O'Grady (Twitter: @paul_ogrady; GitHub: paulogrady)
- Jenny DiMiceli - https://github.com/jdimiceli
- Stephen Kerr
- Wayne Tong
- Vinicius Mayer (viniciusmayer@gmail.com) https://github.com/viniciusmayer
- Dori Czapari https://github.com/doriczapari (@doriczapari)
- Karl Griffin (karl_griffin@hotmail.com) https://github.com/karlgriffin

PyCon UK in Coventry, 20th September 2014

- Matthew Power https://github.com/mthpower
- Brendan Oates <brenoates@gmail.com>
- Waldek Herka (https://github.com/wherka)
- Stephen Newey (@stevenewey) - https://github.com/stevenewey
- Walter Kummer (work.walter at gmail.com)
- Craig Barnes
- Justin Wing Chung Hui
- Davide Ceretti
- Paul van der Linden https://github.com/pvanderlinden
- Gary Martin https://github.com/garym
- Cedric Da Costa Faro https://github.com/cdcf
- Sebastien Charret <sebastien.charret@gmail.com> https://github.com/moerin
- Nick Smith
- Jonathan Lake-Thomas https://github.com/jonathlt
- Ben Mansbridge
- Glen Davies (@GlenDaviesDev) - https://github.com/glen442

DjangoCon US in Portland, 5th September 2014

- Joseph Metzinger (joseph.metzinger@gmail.com) https://github.com/joetheone
- Abdulaziz Alsaffar (alsaff1987@gmail.com) https://github.com/Octowl
- Patrick Beeson (@patrickbeeson) https://github.com/patrickbeeson
- Vishal Shah - https://github.com/shahv
- Kevin Daum (@kevindaum, kevin.daum@gmail.com) https://github.com/kevindaum
- Nasser AlSnayen (nasser.lc9@gmail.com) https://github.com/LC9
- Nicholas Colbert (@45cali) 45cali@gmail.com
- Chris Cauley https://github.com/chriscauley
- Joe Larson (@joelarson)
- Jeff Kile
- Orlando Romero
- Chad Hansen (chadgh@gmail.com) https://github.com/chadgh

DjangoVillage in Orvieto, 14th June 2014

- Gioele
- Christian Barra (@christianbarra) https://github.com/barrachri
- Luca Ippoliti https://github.com/lucaippo
- @joke2k (https://github.com/joke2k)
- Domenico Testa (@domtes)
- Alessio
- Diego Magrini (http://github.com/magrinidiego)
- Matteo (@loacker) https://github.com/loacker
- Simone (@simodalla) https://github.com/simodalla

DjangoCon Europe on The Île des Embiez, 16th May 2014

- Niclas Åhdén (niclas@brightweb.se) https://github.com/niclas-ahden
- Sabine Maennel (sabine.maennel@gmail.com) http://github.com/sabinem
- JB (Juliano Binder)
- Laurent Paoletti
- Alex Semenyuk (https://github.com/gtvblame)
- Moritz Windelen
- Marie-Cécile Gohier
- Isabella Pezzini
- Pavel Meshkoy (@rasstreli)

Dutch Django Association Sprint in Amsterdam, 22nd February 2014

- Stomme poes (@stommepoes)
- Rigel Di Scala (zedr) <zedr@zedr.com> http://github.com/zedr
- Nikalajus Krauklis (@dzhibas) http://github.com/dzhibas
- Ivo Flipse (@ivoflipse5) https://github.com/ivoflipse
- Martin Matusiak
- Jochem Oosterveen https://github.com/jochem
- Pieter Marres
- Nicolaas Heyning (L1NDA)
- Henk Vos h.vos@rapasso.nl https://github.com/henkvos
- Adam Kaliński @ https://github.com/adamkal
- Marco B
- Greg Chapple http://github.com/gregchapple/
- Vincent D. Warmerdam vincentwarmerdam@gmail.com
- Lukasz Gintowt (syzer)
- Bastiaan van der Weij
- Maarten Zaanen <maarten at PZvK.com><Maarten at Zaanen.net>
- Markus Holtermann (@m_holtermann)

Django Weekend Cardiff, 7th February 2014

- Jakub Jarosz (@qba73) jakub.s.jarosz@gmail.com https://github.com/qba73
- Stewart Perrygrove
- Adrian Chu
- Baptiste Darthenay

PyCon Ireland in Dublin, 14th October 2013

- Vincent Hussey vincent.hussey@opw.ie https://github.com/VincentHussey
- Padraic Harley <@pauricthelodger> <padraic@thelodgeronline.com>
- Paul Cunnane <paul.cunnane@gmail.com> https://github.com/paulcunnane
- Sorcha Bowler <saoili @ github, twitter, gmail, most of the internet>
- Jennifer Parak https://github.com/jenpaff
- Andrea Fagan
- Jennifer Casavantes
- Pablo Porto https://github.com/portovep
- Tianyi Wang <wty52133@gmail.com> @TianyiWang33
- James Heslin <program.ix.j@gmail.com> https://github.com/PROGRAM-IX
- Sorcha Bowler <saoili@gmail.com. saoili on github, twitter, most of the internet>

- Larry O'Neill (larryone)
- Samuel <satiricallaught@gmail.com>
- Frank Healy
- Robert McGivern <Robert.bob.mcgivern@gmail.com>
- James Hickey
- Tommy Gibbons

PyCon UK in Coventry, 22nd September 2013

- Adeel Younas <aedil12155@gmail.com>
- Giles Richard Greenway github: augeas
- Mike Gleen
- Arnav Khare https://github.com/arnav
- Daniel Levy https://github.com/daniell
- Ben Huckvale https://github.com/benhuckvale
- Helen Sherwood-Taylor (helenst)
- Tim Garner
- Stephen Newey @stevenewey (stevenewey)
- Mat Brunt <matbrunt@gmail.com>
- John S
- Carl Reynolds (@drcjar)
- Jon Cage & John Medley (http://www.zephirlidar.com)
- Stephen Paulger (github:stephenpaulger twitter:@aimaz)
- Alasdair Nicol
- Dave Coutts https://github.com/davecoutts
- Daley Chetwynd https://github.com/dchetwynd
- Haris A Khan (harisakhan)
- Chung Dieu https://github.com/chungdieu
- Colin Moore-Hill
- John Hoyland (@datainadequate) https://github.com/datainadequate
- Joseph Francis (joseph@skyscanner.net)
- Åke Forslund <ake.forslund@gmail.com> github:forslund
- Ben McAlister https://github.com/bmcjamin
- Lukasz Prasol <lprasol@gmail.com> github: https://github.com/phoenix85
- Jorge Gueorguiev <yefo.akira@gmail.com> https://github.com/MiyamotoAkira
- Dan Ward (danielward) (dan@regenology.co.uk)
- Kristian Roebuck <roebuck86@gmail.com> https://github.com/kristianroebuck

- Louis Fill tkman220@yahoo.com
- Karim Lameer https://github.com/klameer
- John Medley <john.medley@zephirlidar.com>

DjangoCon US in Chicago, 2nd September 2013

- Barbara Hendrick (bahendri)
- Keith Edmiston <keith.edmiston@mccombs.utexas.edu>
- David Garcia (davideire)
- Ernesto Rodriguez <ernesto@tryolabs.com> https://github.com/ernestorx @ernestorx
- Jason Blum
- Hayssam Hajar <hayssam.hajar@gmail.com> github: hhajar

Cardiff Dev Workshop, 8th June 2013

- Daniel Pass <daniel.antony.pass@googlemail.com>
- Kieran Moore
- Dale Bradley
- Howard Dickins <hdickins@gmail.com> https://github.com/hdickins
- Robert Dragan https://github.com/rmdragan
- Chris Davies
- Gwen Williams
- Chris Lovell <chrisl1991@hotmail.co.uk> https://github.com/polyphant1
- Nezam Shah
- Gwen Williams https://github.com/gwenopeno
- Daniel Pass <daniel.antony.pass@googlemail.com>
- Bitarabe Edgar

DjangoCon Europe in Warsaw, 18th May 2013

- Amjith Ramanujam - The Dark Knight
- @zlatkoc
- larssos@github
- @erccy is my name
- Patrik Gärdeman https://github.com/gardeman
- Gustavo Jimenez-Maggiora https://github.com/gajimenezmaggiora
- Jens Ådne Rydland <jensadne@pvv.ntnu.no> https://github.com/jensadne
- Chris Reeves @krak3n
- Alexander Hansen <alexander@geekevents.org> https://github.com/wckd
- Brian Crain (@crainbf)

- Nicolas Noé <nicolas@niconoe.eu> https://github.com/niconoe
- Peter Bero
- schacki
- Michał Karzyński <djangoconwrkshp@karzyn.com> https://github.com/postrational
- @graup

I followed the tutorial online

- Daniel Quinn - 18th May 2013
- Paul C. Anagnostopoulos - 19 August 2013
- Ben Rowett - 27 August 2013
- Chris Miller, <chris@chrismiller.org> - 5th September 2013
- David Lewis - 7th September 2013
- Josh Chandler - 11th September 2013
- Richie Arnold - <richard@ambercouch.co.uk> - 22nd September 2013
- Padraic Stack - https://github.com/padraic7a
- Patrick Nsukami - <patrick@soon.pro> - lemeteore
- Can Ibanoglu - http://github.com/canibanoglu
- Pedro J. Lledó - http://github.com/pjlledo - 11th October 2013
- Ken Tam - 4th Jan 2014
- Óscar M. Lage - http://github.com/oscarmlage
- Bob Aalsma - https://github.com/BobAalsma/
- Andy Venet - https://github.com/avenet/
- Vathsala Achar - 22nd September, 2014
- Amine Zyad <amizya@gmail.com> http://github.com/amizya

Running a workshop

If you'd like to run a workshop based on this material, please do, and please let me know about it.

Notes on running a workshop

To cover all the workshop material seems to take about four and a half hours.

Any of the following will make it take longer:

- attendees who aren't already a little familiar with the terminal and using a text editor to edit source files
- attendees whose machines aren't already suitably-configured for the workshop and need software installed
- attendees using operating systems you're not familiar with

If you're lucky, you'll find that the majority of attendees have the same expertise and the same gaps in expertise. This makes it much easier to decide which parts to dwell upon and which ones you can skim over. If you're not lucky, they will each have a completely different skillset.

You'll do a lot of running around to look at people's screens, so it helps to be able to get around the room easily.

The *Git on the commandline* section is the one where you will be in most demand - it helps great if at this stage you have one or two helpers who are familiar with Git.

Watch out for wireless network limitations - at one session the promised network turned out to block both github.com and ssh, and we had to rely on an access point created by someone's mobile telephone.

Things that might look odd

If you're experienced with things virtualenv and Git, some of the way things are done here might strike you as odd. For example:

Virtualenvs and code The workshop has users put all the code they're working with into their virtualenv directories. This is done to help associate a particular project and set of packages with each virtualenv, and saves excessive moving around between directories.

Editing and committing on GitHub That's certainly not what we'd normally do, but we do it here for three main reasons:

- GitHub's interface is a friendly, low-barrier introduction to Git operations
- it's easy for people to see the effects of their actions straight away
- they get to make commits, pull requests and merges as soon as possible after being introduced to the concepts

The last of these is the most significant.

Other oddities There may be others, which might be for a good reason or just because I don't know better. If you think that something could be done better, please let me know.

1.3.3 Credits

"Don't be afraid to commit" was created by Daniele Procida. Other contributors include:

- Daniel Quinn
- Brian Crain (@crainbf)
- Paul Grau
- Nimesh Ghelani https://github.com/nims11
- Robert Dragan https://github.com/rmdragan
- David Garcia https://github.com/davideire
- Jason Blum https://github.com/jasonblum
- Kevin Daum https://github.com/kevindaum

Many thanks are also due to the members of #django, #python, #git and #github on irc.freenode.net for their endless online help.

... and if I have neglected to mention someone, please let me know.

Please feel free to use and adapt the tutorial.

1.4 Phase III: Complete Web Development

1.4.1 Introduction

The purpose of the workshop

At this point in the course we have all the tools and workflows we would need to do the following:

- Manually set up a development environment
- Manually set up a stack on any machine
- Collaborate on a project remotely

In this workshop, through the process of building and deploying and actual live web application, we will learn to automate and streamline those processes.

Phase III: Complete Web Development will help put you in a position to collaboratively develop a live web application which can be shared, deployed, and installed by anyone from anywhere to any machine.

What's covered

Vagrant will help you automate setting up multiple, identical virtual development environments and share those settings with your team.

Ansible will help you automate the provisioning and deployment of your application to any machine and share those settings with your team.

Python and **Django** - We will introduce the basics of Python and the Django framework and let you take it from there!

Toy Project - This workshop will revolve around an actual Django toy project that we will develop, deploy and iterate on over the duration of this course.

1.4.2 Contents

Vagrant

In *Phase I* we learned how to:

- Setup individual virtual machines
- Setup an environment with the software packages we needed
- Configure and network our machines
- Configure a basic stack

And we did it all **manually** and **individually**!

Vagrant is a tool that changes all of that by providing us with a way **automate** the entire process using a single file that we can share along with the project code.

Vagrant helps you create, provision, share and destroy virtual machines. Vagrant can interact with different hypervisors e.g. VirtualBox, VMware ...etc. Instead of you going and writing scripts to interact with your hypervisor softwre of choice, it does it for you. It has simple commands to launch and create VM. In this course, we will touch the surface of what it can do.

Prerequisite

You need to install VirtualBox or any hypervisor you wish to use. The following instruction will focus on VirtualBox since it's our presonal preference. After that Download and install Vagrant. After you done the previous two steps move to the next section.

Init

To initialize the vagrant you have to run the following command `$vagrant init`, and you should get the following out put.

```
A `Vagrantfile` has been placed in this directory. You are now
ready to `vagrant up` your first virtual environment! Please read
the comments in the Vagrantfile as well as documentation on
`vagrantup.com` for more information on using Vagrant.
```

Now, you should have a new file created in the same directory called **Vagrantfile**. This is a ruby file and it has the initial settup for the VM. Take a moment and open the file and be familair of the different configuration that you can do with your VM. If you open the file you should see the following.

```
# -*- mode: ruby -*-
# vi: set ft=ruby :

# Vagrantfile API/syntax version. Don't touch unless you know what you're doing!
VAGRANTFILE_API_VERSION = "2"

Vagrant.configure(VAGRANTFILE_API_VERSION) do |config|
  # All Vagrant configuration is done here. The most common configuration
  # options are documented and commented below. For a complete reference,
  # please see the online documentation at vagrantup.com.

  # Every Vagrant virtual environment requires a box to build off of.
  config.vm.box = "base"

  # Disable automatic box update checking. If you disable this, then
  # boxes will only be checked for updates when the user runs
  # `vagrant box outdated`. This is not recommended.
  # config.vm.box_check_update = false

  # Create a forwarded port mapping which allows access to a specific port
  # within the machine from a port on the host machine. In the example below,
  # accessing "localhost:8080" will access port 80 on the guest machine.
  # config.vm.network "forwarded_port", guest: 80, host: 8080

  # Create a private network, which allows host-only access to the machine
  # using a specific IP.
  # config.vm.network "private_network", ip: "192.168.33.10"

  # Create a public network, which generally matched to bridged network.
  # Bridged networks make the machine appear as another physical device on
  # your network.
```

```
  # config.vm.network "public_network"

  # If true, then any SSH connections made will enable agent forwarding.
  # Default value: false
  # config.ssh.forward_agent = true

  # Share an additional folder to the guest VM. The first argument is
  # the path on the host to the actual folder. The second argument is
  # the path on the guest to mount the folder. And the optional third
  # argument is a set of non-required options.
  # config.vm.synced_folder "../data", "/vagrant_data"

  # Provider-specific configuration so you can fine-tune various
  # backing providers for Vagrant. These expose provider-specific options.
  # Example for VirtualBox:
  #
  # config.vm.provider "virtualbox" do |vb|
  #   # Don't boot with headless mode
  #   vb.gui = true
  #
  #   # Use VBoxManage to customize the VM. For example to change memory:
  #   vb.customize ["modifyvm", :id, "--memory", "1024"]
  # end
  #
  # View the documentation for the provider you're using for more
  # information on available options.

  # Enable provisioning with CFEngine. CFEngine Community packages are
  # automatically installed. For example, configure the host as a
  # policy server and optionally a policy file to run:
  #
  # config.vm.provision "cfengine" do |cf|
  #   cf.am_policy_hub = true
  #   # cf.run_file = "motd.cf"
  # end
  #
  # You can also configure and bootstrap a client to an existing
  # policy server:
  #
  # config.vm.provision "cfengine" do |cf|
  #   cf.policy_server_address = "10.0.2.15"
  # end

  # Enable provisioning with Puppet stand alone.  Puppet manifests
  # are contained in a directory path relative to this Vagrantfile.
  # You will need to create the manifests directory and a manifest in
  # the file default.pp in the manifests_path directory.
  #
  # config.vm.provision "puppet" do |puppet|
  #   puppet.manifests_path = "manifests"
  #   puppet.manifest_file  = "site.pp"
  # end

  # Enable provisioning with chef solo, specifying a cookbooks path, roles
  # path, and data_bags path (all relative to this Vagrantfile), and adding
  # some recipes and/or roles.
  #
  # config.vm.provision "chef_solo" do |chef|
```

```
  #   chef.cookbooks_path = "../my-recipes/cookbooks"
  #   chef.roles_path = "../my-recipes/roles"
  #   chef.data_bags_path = "../my-recipes/data_bags"
  #   chef.add_recipe "mysql"
  #   chef.add_role "web"
  #
  #   # You may also specify custom JSON attributes:
  #   chef.json = { mysql_password: "foo" }
  # end

  # Enable provisioning with chef server, specifying the chef server URL,
  # and the path to the validation key (relative to this Vagrantfile).
  #
  # The Opscode Platform uses HTTPS. Substitute your organization for
  # ORGNAME in the URL and validation key.
  #
  # If you have your own Chef Server, use the appropriate URL, which may be
  # HTTP instead of HTTPS depending on your configuration. Also change the
  # validation key to validation.pem.
  #
  # config.vm.provision "chef_client" do |chef|
  #   chef.chef_server_url = "https://api.opscode.com/organizations/ORGNAME"
  #   chef.validation_key_path = "ORGNAME-validator.pem"
  # end
  #
  # If you're using the Opscode platform, your validator client is
  # ORGNAME-validator, replacing ORGNAME with your organization name.
  #
  # If you have your own Chef Server, the default validation client name is
  # chef-validator, unless you changed the configuration.
  #
  #   chef.validation_client_name = "ORGNAME-validator"
end
```

After making your adjustments on the file move the next section.

Up

The `VagrantFile` has all the configuration for the VM, and once you fnish configuring run the following command `$vagrant up`. The command will aply the configuration to your VM and instantiate the machine. The output of the command should look like the following.

```
Bringing machine 'default' up with 'virtualbox' provider...
==> default: Importing base box 'base.box'...
==> default: Matching MAC address for NAT networking...
==> default: Setting the name of the VM: test_default_1416676086074_86409
==> default: Clearing any previously set forwarded ports...
==> default: Clearing any previously set network interfaces...
==> default: Preparing network interfaces based on configuration...
    default: Adapter 1: nat
==> default: Forwarding ports...
    default: 80 => 8080 (adapter 1)
    default: 22 => 2222 (adapter 1)
==> default: Booting VM...
==> default: Waiting for machine to boot. This may take a few minutes...
    default: SSH address: 127.0.0.1:2222
    default: SSH username: vagrant
```

```
    default: SSH auth method: private key
    default: Warning: Connection timeout. Retrying...
    default: Warning: Remote connection disconnect. Retrying...
==> default: Machine booted and ready!
==> default: Checking for guest additions in VM...
==> default: Mounting shared folders...
    default: /vagrant => /private/tmp/test
```

After completing the command, you should have a VM running in the background. The VM is running without GUI, and you should interact with it using vagrant commands. To check if there is a machine or not open up VirtualBox GUI and you should see a machine running.

SSH

`$vagrant ssh` is the command that you should use in order to log in to the machine and run your commands. After running the command, you will be logged in into the machine.

Provision

Provision is the way to set up your development environment. You can put your configuration inside **VagrantFile** and let **Vagrant** run and configure your machine the way you like. There are multiple options to accomplish that you could use Puppet, Chef, Ansible or Shell. In this section will talk about how you can provision using shell commands and then we will talk about Ansible provisioning later in this course. Follow the next instructions to provision using shell

1. Open VagrantFile
2. Append `config.vm.provision "shell", path:  "myscript.sh"`
3. Create a file called **myscript.sh**
4. Copy the following into the file

```
#!/bin/sh
set -e

# installing Nginx
sudo apt-get install -y --force-yes nginx

# starting nginx
sudo service nginx start
```

Note: The script will install and start nginx with the default configuration. It should listen to port 80 on the guest machine and you should `port-forward` that port to a port on the host machine by changing the configuration in the VagrantFile.

Tutorial

We will be following the official vagrant tutorial.

Exercise

Let's create a Vagrantfile to set up the box we will need for our toy project.

Ansible

Python

Python is an general purpose object oriented programming language that was created by Guido Van Rossum. The language has many implementations e.g. CPython, Jython, Pypy, ...etc. We focus on the CPython implementaion in this course, and feel free to check the rest of them out.

Python is widly used programing language, therefore it is being taught by many schools as an introductory programming language in computer science classes. In this course we will try to teach you how to use it.

From now on when we use the word **python** we mean the **CPython** implementation of the **Python** programming language.

Now to start a terminal session using `ALT+CTRL+T` and launch the **python** REPL. Once you started the interpreter you should see the following output

```
Python 2.7.5 (default, Mar  9 2014, 22:15:05)
[GCC 4.2.1 Compatible Apple LLVM 5.0 (clang-500.0.68)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

This is the **python** prompt it, and it shows useful information about the current **python** version and the compiler that **python** was compiled with.

The >>> is the begining of the REPL and anything you type and hit **Enter** it will evaluate on the second line and print out the result. Once it's done executing it will display the same prompt again waiting for the next command.

Now we will learn about the useful operations that **python** offers in the command line.

Integer Operations

```
>>> 3 + 7 # Addition
10
>>> 10 - 9 # Subtraction
1
>>> 9 - 100 # Handles negative numbers
-91
>>> 5 * 6 # Multiplications
30
>>> 15 / 5 # division
3
>>> 14 / 3 # integer division
4
>>> 14 // 3 # forcing integer division
4
>>> 5 ** 7 # Power operator
78125
>>> 10 % 4 # modules operations
2
>>> 1 << 4 # Left bitshift
16
>>> 16 >> 2 # Right bitshift
4
>>> 3 & 2 # Bit And operator
2
>>> 1 | 4 # Bit Or operator
5
```

```
>>> ~ 1 # 2's complement
-2
>>> 1 ^ 10 # XOr operator
11
```

Django

Toy project

CHAPTER 2

Indices and tables

- *genindex*
- *modindex*
- *search*