Documentation

Release 0.5

nawab

Nov 21, 2018

Contents

Contents:

Tizen is an open and flexible operating system built from the ground up to address the needs of all stakeholders of the mobile and connected device ecosystem, including device manufacturers, mobile operators, application developers and independent software vendors (ISVs). Tizen is developed by a community of developers, under open source governance, and is open to all members who wish to participate.

CHAPTER 1

About the Linux Foundation

Since its inception in 1991, Linux has grown to become a force in computing, powering everything from the New York Stock Exchange to mobile phones to supercomputers to consumer devices. The Linux Foundation is the nonprofit consortium dedicated to fostering the growth of Linux. Founded in 2007, the Linux Foundation sponsors the work of Linux creator Linus Torvalds and is supported by leading technology companies and developers from around the world. The Linux Foundation promotes, protects and advances Linux by marshaling the resources of its members and the open source development community to ensure Linux remains free and technically advanced.

changed

CHAPTER 2

Toc Item 2

2.1 Toc Item2,2

CHAPTER 3

Deploying nginx + django + python 3

Hello,

due to the lacks of informations about deploying latests version of django (1.6+) with latest nginx (1.6+) using gunicorn (18+) inside virtual environment of python 3 (3.4+), it was really hard for a beginner like me to deploy a django project.

I finally made it and now after several months I decided to share my experiences with all the world. So again:

3.1 What this guide is about

We'll deploy (that means making website available to the world) `django` project with server engine `nginx` using `gunicorn` for that. We'll also use virtual environment of python and installation will be static - it will not depend on your system-wide python installation.

3.2 Prerequisites and informations

I'm using Linux and commands I'm going to introduce are thought to be run in bash shell. Sometimes root privileges might be required and I'll **not** remark that. If you are not familiar with Linux, please read my other guides.

You do not need any special knowledge. But keep in mind that this is not guide how django, nginx or gunicorn work! This is about how it should be brought together to work.

3.3 My choice - why nginx, python 3 etc.

I'm not the one who tried all of possible choices. I've just tried a lot of them and this one was first of them which worked. So I stuck to it.

I've chosen `nginx` over `apache` because this seems to be *trend* today thanks to Apache's age. It's also seems to me easier.

I've chosen `django` because I love python. And I'm quite new to it so when I decided to learn this great language, I started with `python 3`. It was somehow logical because I could choose what I want and the newer one is of course better investment to the future.

Gunicorn is just so easy to use and I've found great documentations and guides for it.

Virtual environment is necessity. You don't want all python projects (not just django websites) depend on one specific configuration. It's not stable (one package can hurt other) nor secure. We'll use **hard** *virtualenv* because it's also safer - you can update this version when you want and not with every time your distribution says to you.

3.4 How the hell all that works

Here is a little model I've made for myself and I think it's not too bad to be a starting point for you ;) . We have five terms: `nginx`, `python`, `virtualenv`, `gunicorn` and `django`.

3.4.1 First layer (nginx)

`nginx` is what cares about requests from the world. It's what catches your request (e.g. Google) and redirects it to according folder (in case of static HTML page with index.html, not our case), or to some application.

3.4.2 Second layer (gunicorn)

This application in our case is `gunicorn`. It's powered by `python` and it basically makes a magical communication channel `nginx``~``django app`. This tunnel is represented by **socket** (we'll get to it). Why can't do this `nginx`? It's just not clever enough (better - it just hasn't do that and that's absolutely OK in Unix philosophy). Gunicorn can make *server* similar to django test server. But it can also **serve** django app content to `nginx` and hence solving nginx's limitation.

3.4.3 Third layer (django)

Then there is just `django` - your project with you pages - this is what your website is about. All previous (and next) is just *working* layer.

3.4.4 Wrapper for second and third layer

`python` is *engine* of `gunicorn` even for `django`. This `python` is running inside *sandbox* called virtualenv. Gunicorn will **activate** this virtualenv for us and *run* django app.

That's all. Not that hard, huh? Basically **nginx~gunicorn~django**. `python` is powering it and `virtualenv` is just wraps python version (it is not necessity to have virtualenv, if it's easier to understand it).

3.5 Let's do it

3.5.1 nginx configuration

Covered in other tutorial, you must be able to make it to the point that you are able to see *nginx welcome page*.

3.5.2 Installing python and virtualenv

About virtual environments there is tons of guides on the internet - feel free to educate yourself :D . Here is my brief guide.

To do that we'll need to install `python-virtualenv`. Install it and then create some folder for our test case. Let it be `/var/www/test`. Also install `python` of version 3, if you haven't done that before.

We'll now create *sandbox* of python for our test case inside this folder (`/var/www/test`). Why `/var/www`? It's just good place to put websites on Unix (and hence - Linux). But it can be anywhere of course. To create a **REAL** copy (and not just a linked variant) of python version 3 we need to use this syntax:

```
virtualenv --python=python3 --always-copy venv
```

What happened here? We created python *sandbox* called **venv** in current directory. It use **python3** as default choice and we copied all necessary files for life of this installation (by default there are only symlinked to the system one).

What now? We need to switch from *system* python installation to *venv* python installation. First try to type `python -c "import sys; print(sys.path)"`. The output is similar to this:

```
['', '/usr/lib/python34.zip', '/usr/lib/python3.4', '/usr/lib/python3.4/plat-linux',
↪'/usr/lib/python3.4/lib-dynload', '/usr/lib/python3.4/site-packages']
```

where you can notice that current default `python` interpreter gets it's config from somewhere in `/usr/lib/...`.

We will now activate our virtualenv by this command: `source /var/www/test/venv/bin/activate`. Now try same command as above (`python -c ...`) and it should print instead of `/usr/lib/...` something starting with `/var/www/test/venv/...`. If yes, it's working :) .

To quit from this environment and get to your system-wide, type `deactivate`.

3.5.3 pip installing django and gunicorn

One of the best advantages of `python 3.4` is a fact that `pip` is installed by default. What is `pip`? `pip` is installer for python packages.

All python packages can be found here. Of course you can find you package there (try for example with `django`), download it and build it with python on your own. But that sound like a lot of work. Let's `pip` do it for us.

Assure yourself you are working in our virtualenv (you can again **activate** it) and type this:

```
pip install django
pip install gunicorn
```

you can specify version just by typing = behind name package:

```
pip install django=1.6.0
```

but of course this version must exists on `pypi.python.org`. If there are errors, try adding `-v` switch for verbose.

To list installed packages type:

```
pip list
```

try if you see `django` and `gunicorn` there :) .

and that's all you need for now with pip (although there isn't much more about pip).

3.5.4 Sample django project

Now we'll need to create django project for our test case. Go inside `/var/www/test` and activate our virtualenv where is `django` and `gunicorn` (you can do that again by `source /var/www/test/venv/bin/activate`).

Create django project by:

```
django-admin3.py startproject ourcase
```

it should create this structure inside `/var/www/test`:

```
ourcase
|-- manage.py
`-- ourcase
    |-- __init__.py
    |-- settings.py
    |-- urls.py
    `-- wsgi.py

1 directory, 5 files
```

check if it's working with local django testing server by `python manage.py runserver`. Check in browser `127.0.0.1:8000` - if there is django welcome page, it's good.

Just for comfort make `manage.py` executable by `chmod +x ourcase/manage.py`.

3.5.5 gunicorn and daemonizing it

Now we'll replace django testing server, which is just for kids (it's just great future :)), with fully mature nginx for adults.

As was previously stated, for that we'll need gunicorn. Gunicorn will have to be running to enable communication between nginx and django project.

First, we'll use just `gunicorn` to display our django test project on `127.0.0.1:8000`. It's incredibly easy. Again - assure yourself you are working in current virtualenv.

Now navigate yourself inside `/var/www/test/ourcase/` and run this magical command:

```
gunicorn ourcase.wsgi:application
```

it will start something like *gunicorn server* - you should be able to see your django welcome page on `127.0.0.1:8000`.

This is just the most stupid configuration, which is enough for this test, but not for deploying on server. For that we'll want to add much more. Create starting script `/var/www/test/gunicorn_start.sh`:

```
#!/bin/bash

NAME="ourcase"                                  #Name of the application (*)
DJANGODIR=/var/www/test/ourcase                 # Django project directory (*)
SOCKFILE=/var/www/test/run/gunicorn.sock        # we will communicate using this unix␣
↪socket (*)
USER=nginx                                        # the user to run as (*)
GROUP=webdata                                     # the group to run as (*)
NUM_WORKERS=1                                     # how many worker processes should␣
↪Gunicorn spawn (*)
```

(continues on next page)

(continued from previous page)

```
DJANGO_SETTINGS_MODULE=ourcase.settings             # which settings file should␣
↪Django use (*)
DJANGO_WSGI_MODULE=ourcase.wsgi                     # WSGI module name (*)

echo "Starting $NAME as `whoami`"

# Activate the virtual environment
cd $DJANGODIR
source /var/www/test/venv/bin/activate
export DJANGO_SETTINGS_MODULE=$DJANGO_SETTINGS_MODULE
export PYTHONPATH=$DJANGODIR:$PYTHONPATH

# Create the run directory if it doesn't exist
RUNDIR=$(dirname $SOCKFILE)
test -d $RUNDIR || mkdir -p $RUNDIR

# Start your Django Unicorn
# Programs meant to be run under supervisor should not daemonize themselves (do not␣
↪use --daemon)
exec /var/www/test/venv/bin/gunicorn ${DJANGO_WSGI_MODULE}:application \
  --name $NAME \
  --workers $NUM_WORKERS \
  --user $USER \
  --bind=unix:$SOCKFILE
```

Wow! A lot happened here compared to our *stupid* variant. Everything marked with `(*)` in comments can be changed (or must be changed if your paths differs).

The most important change here is that we added `SOCKFILE` - socket. This is the magic thingie which will enable `nginx` to server django project (app). Gunicorn will somehow run server as in previous stupid variant and *transfer* this into socket file in language which `nginx` understands. `nginx` is looking to this socket file and is happy to serve everything there is.

It's common practice (and I strongly encouraged it) to run server as some specific user. It's for security reasons. So if you haven't done it before, create some user and group for these purposes (ALSO IN OTHER MY TUTORIAL).

Workers are just how much computing power you enable for this website.

If you are not working as a user which is in script set to `USER` variable, you **won't** be able to run this script (you'll get some errors). That's because of permissions reasons. If you'd like to check or debug this script (and it's recommended), uncomment `--user $USER` line - it should work then even if you run it as another user. Of course you need to make script executable.

See gunicorn documentation for more informations.

This script is laying all over the internet in multiple variants. If you have problems to run it, try to uncomment some other lines in last part of script. For example I wasn't able to run this script with directive `--log-level=warning`.

If it is working, it's great! Now we'll daemonize it by using `systemd`. Of course you can use another init system (like Ubuntu `upstart`. Just search for "how to run script after boot".

Create new service file `/usr/lib/systemd/system/gunicorn_ourcase.service` and insert this:

```
[Unit]
Description=Ourcase gunicorn daemon

[Service]
Type=simple
User=nginx
```

(continues on next page)

(continued from previous page)

```
ExecStart=/var/www/test/gunicorn_start.sh

[Install]
WantedBy=multi-user.target
```

now enable it as with other units:

```
systemctl enable gunicorn_ourcase
```

now this script should be run after boot. Try if it's working (reboot and use `systemctl status gunicorn_ourcase`).

That's all for gunicorn.

3.5.6 django project deployment

Deploying django project is topic for longer tutorial then is this. So I'll make it as small as possible.

If you've just developed django project with test server, it makes a tons of things for you without any notices. In reality it's not as easy - everything isn't done automatically and django is prepared for that - but you need to *activate* this futures, since it's not by default.

Directories

Nice example is with **static** files. There are e.g. some CSS styles for django administration page. These needs to be in special folder and we'll tell nginx that when website asks for file `style.css`, it should looks into `~/var/www/test/ourcase/static/style.css`.

But how to find all this static files? Right now they are sourced from django installation directory (probably something like `/var/www/test/venv/lib/python3.4/django/....` `manage.py` has a special command for this, but first we need to tell him few details in `settings.py`.

The most common configuration is to has a special directory for static files where you can edit them, past them etc. Then there will be static directory, where you won't do any changes - this will be for `manage.py` command - it will collects them from your special directory, from django installation directory etc. In templates, when you want to use e.g. some static image on background, you use `{ STATIC_URL}/static_images/mybgrnd.png`.

To do this we'll add this to `settings.py`:

```
STATIC_URL = '/static/'
STATIC_ROOT = os.path.join(BASE_DIR, "static")
STATICFILES_DIRS = (os.path.join(BASE_DIR, "sfiles"), )
```

all your static files used should now be placed inside `/var/www/test/ourcase/sfiles`. If you just want to try it, create this directory and `touch sfiles/example.png` inside it.

Now run `./manage.py collectstatic`. It should ask you if you really want to do that (and you want). Process will start and after it's finish you'll have collected all static files inside `static` folder. This you need to do every time you change something inside `sfiles` folder.

Websites also usually has `media` folder, which is used for user files - for example images to blog posts. Usually we use `MEDIA_URL` for calling things from media dir in templates.

Configuration should be same as with django testing server and you don't need to do any special changes here. My looks like this:

```
MEDIA_ROOT = os.path.join(BASE_DIR, "media")
MEDIA_URL = '/media/'
ADMIN_MEDIA_PREFIX = '/media/admin/'
```

and all user files (uploaded images, sounds...) are inside */var/www/test/ourcase/media*' directory. You don't need to do something like `collectstatics` here.

Steps for other directories should be same.

Enough for directories. But some other changes are needed to deploy django project. In some cases I don't really know why I need to add this to `settings.py`, but I know what that does and it's just working.

Templates

I had to add this for templates:

```
TEMPLATE_DIRS = (os.path.join(BASE_DIR, 'templates'),)
TEMPLATE_LOADERS = (
'django.template.loaders.filesystem.Loader',
'django.template.loaders.app_directories.Loader',)
```

where I've put my `base.html` which is used in all other templates in whole website (in every app). If you use `flatpages`, you can also make a directory inside `templates` called `flatpages`, where you can copy `base.html` as ``default.html` and use this template as base for flatpages.

SITE_ID

For some purposes is needed to set SITE_ID. In my case it was because of `flatpages`. It's easy:

```
SITE_ID = 1
```

ALLOWED_HOSTS

You need to past all your domains here. If your domain is `www.example.com` and I guess `example.com` also, it should looks like this:

```
ALLOWED_HOSTS = ['example.com', 'www.example.com']
```

DEBUG

This directive should be set to **False**. But when you are configuring your server for first time, let **True** there. It helps you find out bugs on your site.

That's it!

3.5.7 nginx server configuration

Last part is configuring `nginx` to make him listen on socket created by gunicorn. It's not hard.

Edit `/etc/nginx/nginx.conf` and paste this into `http` block:

```
upstream test_server {
  server unix:/var/www/test/run/gunicorn.sock fail_timeout=10s;
}

# This is not neccessary - it's just commonly used
# it just redirects example.com -> www.example.com
# so it isn't treated as two separate websites
server {
        listen 80;
        server_name example.com;
        return 301 $scheme://www.example.com$request_uri;
}

server {
    listen   80;
    server_name www.example.com;

    client_max_body_size 4G;

    access_log /var/www/test/logs/nginx-access.log;
    error_log /var/www/test/logs/nginx-error.log warn;

    location /static/ {
        autoindex on;
        alias   /var/www/test/ourcase/static/;
    }

    location /media/ {
        autoindex on;
        alias   /var/www/test/ourcase/media/;
    }

    location / {
        proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
        proxy_set_header Host $http_host;
        proxy_redirect off;

        if (!-f $request_filename) {
            proxy_pass http://test_server;
            break;
        }
    }

    #For favicon
    location  /favicon.ico {
        alias /var/www/test/test/static/img/favicon.ico;
    }
    #For robots.txt
    location  /robots.txt {
        alias /var/www/test/test/static/robots.txt ;
    }
    # Error pages
    error_page 500 502 503 504 /500.html;
    location = /500.html {
        root /var/www/test/ourcase/static/;
    }
}
```

OK, that's whipping. I'll be fast.

First, we tell *nginx* where is socket file (`gunicorn.sock`) from gunicorn.

Then there is redirect from non-www domain to www domain. This can be omitted or solved in other way (CNAME).

Then there is main body of server configuration: * logs are useful for catching bugs and errors - has multiple parameters, like how much should they bother you. Don't forget to create log directory. * static and media block - these are extremely important - this is why we played all that games with collectstatics etc. It just tells nginx where it should looks when website asks for e.g. `/static/style.css/` or `/media/img/picture_of_my_cat.png`. * Block with all that proxy things is also important and is used for technical background around socket communication and redirecting. Don't care about that. * Favicon and robots.txt is not necessary, but all browsers and web crawlers are still searching for them. So if you don't like errors in your logs, add create these two things. * Last block is telling nginx where it should looks for error pages when something doesn't exists.

Save and exit. Next great future of `nginx` is it's ability of checking configuration. Type `nginx -t` (don't forget root permissions) and you'll see if configuration is syntactically correct. Don't forget about that stupid `;`.

Finally enable `nginx` to be ran after reboot:

```
systemctl enable nginx
```

3.5.8 Some sugar candy

Install with `pip` package called `setproctitle`. It's useful for displaying more info about ran processes like gunicorn in system process managers (`htop`, `ps`, ...).

3.6 Debugging

That's it. Now restart computer and see if it doesn't explode. You can analyse `nginx` or `gunicorn` with `systemctl`, e.g.:

```
systemctl status gunicorn_ourcase
```

and some informations should be also in log files. Try to get to your website from browser and see what happens. Don't forget that browser likes caching and press **CTRL+r** for reload to see changes you've made.

After every change in configuration of nginx you need to restart it by running `nginx -s reload`.

To see what processes are spawned you can use your task manager like `htop` or `ps`.

3.7 Integration with GitHub

For starter it's necessary to say, that GitHub...

...is awesome! If you don't needed and you went through whole process above, you can probably save a lot of headaches just by using GitHub. You don't need any special knowledge for start, but you will need to learn them on the fly while reading this tutorial (there is really a lot about git out there, google is your friend).

The variant I propose here is very easy, scalable and fast. Probably the most easy and effective I've found.

3.7.1 Why to use it

I was so happy when I deployd my first django project. But few weeks later I've found that it's just not feeling right to make changes on live version of the website (sometimes refered as *production*). So I started to use GitHub and found a solution.

Here I will cover this: For every your website you end up with one directory including three subdirectories.

1. First called **production** - it's the one which is live on the internet - the one what `nginx` refers.
2. Second called **mydomain.git** - this one is necessary for our github configuration. You will barely change there anything
3. Last one - **work_dir** - the one where all changes are being made and is connected to GitHub

3.7.2 Workflow

Your work will look like this:

1. Your work_dir contains master branch. This branch can be pushed to production (to go live) anywhen! So when you want to make change to your website, you need create new branch (correctly named based on the change you are doing - e.g. *hotfix_plugin*, *typo_css*...) and when you finish and test this branch, you merge it to master.
2. You push master to your GitHub repository
3. You push master to your production folder on your computer

3.7.3 Set it all up

So how to do it? I suppose you have one working directory as we created in previous chapters.

Now go to the place where you websites are stored. Mine is in `/var/www` and create this structure:

```
mydomain
├── mydomain.git
├── production
└── work_dir
```

Go to `/var/www/mydomain.git` and type this:

```
git init --bare
```

this will create just git repository with some special folders. You don't need to know anything about it. All you need to do is to create this file `/var/www/mydomain/mydomain.git/hooks/post-receive` and add this:

```
#!/bin/sh
git --work-tree=/var/www/mydomain/production --git-dir=/var/www/mydomain/mydomain.git␣
↪checkout -f
```

and make the script runable `chmod +x /var/www/mydomain/mydomain.git/hooks/post-receive`

Go to work_dir and paste there you current *production* code (the one from previous chapters). Now you need to make a GitHub repository from that. The best guide is this one: How to add existing folder to GitHub. (Maybe you'll need to generate SSH key). Is it working? Great.

> Note: It's very good to make git repositories as small as possible, so don't add to repository files which are not necessary or you backup them somewhere else. But virtualenv is a good thing to add there to IMHO.

Now just add another remote, which will point to our created git repository. Every time we'll want to go live with master, we'll push changes to this git repository and it will take care to transfer our files to production. So in work_dir type:

```
git remote add production file::///var/www/mydomain/mydomain.git``
```

and that's all. When you now want to push changes to production, type `git push production master`. Congratulations!

3.8 Finalization

That's all! I hope this guide helped you and you has successfully start up your websites! :)

For further reading, I'd recommend you to look at the part how to set up git for easy deployment.

CHAPTER 4

Development isolation

When you need to maintain and protect a stable main branch, you can branch one or more dev branches from main. It enables isolation and concurrent development. Work can be isolated in development branches by feature, organization, or temporary collaboration.

4.1 Feature isolation

Feature isolation is a special derivation of the development isolation, allowing you to branch one or more feature branches from main, as shown, or from your dev branches.

CHAPTER 5

Making Raspberry Pi usable

5.1 Introduction

After 8 months of using RPi, I decided to make second version of this tutorial for same people as I'm - who looks for easy, understandable way to make RPi as awesome as possible. Several things have changed since last realease of this tutorial, so I decided to rewrite some parts and also to delete some parts which are not necessary today.

In this tutorial I will walk you through whole process of making from Raspberry Pi secure, reliable, efficient, fast and easy to maintain server for variable purposes as is FTP, web hosting, sharing… All that thanks to Arch Linux ARM operating system. The device will be "headless" - it means, there will be no fancy windows etc., just command line. Don't be scared, I will walk you through and you'll thank me then :) . You don't need some special knowledge about computers and linux systems.

5.2 What you get

From "bare" RPi you'll get:

- Safely to connect to your RPi from anywhere
- Possibility of hosting web pages, files, etc.
- Readable and reliable system (it will do what you want and nothing more)

5.3 What you will need

- Raspberry Pi (doesn't matter which model) with power supply
- SD Card as a main hardisk for RPi
- SD Card reader on computer with internet access
- Ethernet LAN cable or USB Wi-Fi bundle

- Other computer (preferably with linux, but nevermind if you use Windows or Mac)
- Possibility to physically get to your router and know credentials to login to it (or have contact to your network administrator :))
- Few hours of work

5.4 What you don't need

- Monitor or ability to connect RPi to some monitor
- Keyboard or mouse connected to your RPi

5.5 Start

So you have just bare RPi, SD card, power supply, ethernet cable (RJ-45). So let's start! There are houndreds of guides, but I haven't found them satisfaing.

5.5.1 Installing Arch Linux ARM to SD card

Go here, choose installation and make first 3 steps. That's it! You have done it. You have you Arch Linux ARM SD card :)

5.5.2 Little networking

I guess you probably have some of "home router" ("box with internet") and when you want to connect e.g by Wi-Fi with your laptop or mobile phone, it just connects (after inserting password). You need to test first what happens, when you try to connect by ethernet cable, for example with your laptop. Turn off Wi-Fi and check it. Did your computer connects to the network (or even internet) as usuall?

If yes, it is great! You can procced. It is what we need - we need RPi, when it boots up, to automatically connect to the network. Then we will able to connect to it. You will need one more thing to find out - which IP address does router assign to you when you connect by cable - it is very probable that RPi will get very similiar. Don't be afraid - it is easy to get IP address. On modern systems, one command :) .

Ok, now you have to insert SD card to RPi and connect it to your router with ethernet cable and then turn RPi on by inserting power supply. The diods start flashing. Now back to your computer and we will try to connect it using **SSH**. SSH is just "magic power" which enables to connect to another computer.

RPi is already ready and waits for SSH connection. How to use SSH is supereasy - you will find a tons of tutorials on the internet (keywords: how to use ssh). IP address is the probably the one you assigned before. It will be something like this: `192.168.0.x`, `10.0.0.14x` or similar. Next thing you need is username. It's just "root" (and password also).

If your RPi haven't got this address (ssh is not working), than there are two options.

1. You will login to your router settings and find out list of all connected devices with IP addresses and try them.
2. Use nmap to find active devices in your network.

Example You have this address assigned: `192.168.0.201`. Then you have to type (in linux): `ssh root@192.168.0.201`.

You should now end up in RPi console.

Enough of networking for now. We'll set a proper network configuration later in this guide, but first some *musthaves*.

5.6 First setup

This is covered over the internet, so I will just redirect you. elinux - from this guide finish these parts (in RPi console):

- Change root password
- Modify system files
- Mount extra partitions (if you don't know what it is, nevermind)
- Update system
- Install sudo
- Create regular user account

My usuall procedure (which is strongly related to my needs!):

```
passwd  # change root password to something important
rm -rf /etc/localtime  # dont care about this
ln -s /usr/share/zoneinfo/Europe/Prague /etc/localtime  # set appropriate timezone
echo "my_raspberry" >  /etc/hostname  # set name of your RPi

useradd -m -aG wheel -s /usr/bin/bash common_user #
groupadd webdata  # for sharing
useradd -M -aG webdata -s /usr/bin/false nginx
usermod -aG webdata common_user

visudo  # uncomment this line:  %wheel ALL=(ALL) ALL

pacman -Syu
```

That's enough for now. Logout from ssh (type `exit`) and connect again, but as user who was created. Similiar to previous: `ssh common_user@ip.address`. From now, you'll need to type "sudo" in front of every command, which is possibly danger. I will warn you in next chapter.

We must be sure that after reboot RPi will reconnect.

Now try if you are connected to the internet. Type `ping 8.8.8.8`. If you don't see `ping: unknown host 8.8.8.8` it's good! If you do, your internet connection is not working. Try to find out why - unfortunately it is not possible to solve it here.

Warning Try also `ping google.com`. It may not work even pinging 8.8.8.8 worked. The reason is bad DNS servers (doesn't matter what it is). To solve this you have to find "DNS servers of your IPS". Try to google it. If you find them, add them to `resolv.conf`.

Reboot you rpi using `systemctl reboot`. You must be able to connect to it again after one minute. If not, somthing is wrong... In that case, you need to find out why connection stoped working - if you have keyboard and monitor, you can repair it. If not, you can try to edit mistake on other computer by inserting SD card. Otherwise, reinstall...

5.7 Installing some sugar candy

For our purpouses we will install usefull things, which will help as maintaing the system. So, run this: `pacman -S vim zsh wget ranger htop lynx`

Do you see:

```
error: you cannot perform this operation unless you are root.
```

Then you need to type `sudo pacman -S ...`. I will not write it in future and it is not in other guides. So sometimes you might be confused whel you'll read some tutorials and autor implicitly use sudo without mentioning it.

We will also need these in next chapters: `pacman -S nginx sshguard vsftpd`

You can notice that is really few packages! And thats true! Isn't it great? No needs of tons of crap in your device.

What are these? Just short summary - you can find more about it in manual pages (`man <name_of_pacakge>`) or find something usefull on the internet. * **vim** - powerfull text editor (that's what you will do 99% of time). First few *days* are horrible, but keep using it :) . * **zsh** - doesn't matter. Just install it and install this * **wget** - just for downloading things without browser * **ranger** - file manager (you can browse files, folders...) * **htop** - task manager - you can see what tasks are running, how much CPU/MEM is used, kill processes and so on * **lynx** - browser - no kidding :)

5.8 Some configurations

I assume you installed `zsh` with `oh-my-zsh` (changed your shell) and also vim. You are connected as created user (from now, I will name him **bob**). You are in Bob's home directory - check it with typing `pwd`. It will print `/home/bob`.

5.8.1 Make vim usable

Edit .vimrc file: `vim .vimrc` and insert this:

```
syntax on
set number
set ruler
set nocompatible
set ignorecase
set backspace=eol,start,indent
set whichwrap+=<,>,h,l
set smartcase
set hlsearch
set incsearch
set magic
set showmatch
set mat=2
set expandtab
set smarttab
set shiftwidth=4
set tabstop=4
set lbr
set tw=500
set ai
set si
set wrap
set paste
set background=dark
vnoremap <silent> * :call VisualSelection('f')<CR>
vnoremap <silent> # :call VisualSelection('b')<CR>
```

it will customize vim a bit, so it will be easier to edit files in it.

5.8.2 Journaling

Journaling is one of the most important things you need to have. It just record everything systemd does. It is part of `systemd` quite customizable. We will save journals in memory, because of limited wear of SD cards. We will also compress them and then limit size for them on 40 MB.

Open file `/etc/system/journal.conf` and uncomment these lines:

```
[Journal]
Storage=volatile
Compress=yes
...
RuntimeMaxUse=40M
```

5.9 Network configuration

For reasons I will mention in future, we need to set RPi to connect with **static ip**. This will assure that the IP address of RPi will be still the same and you can connect it. Right now is probably getting automatically assigned IP address from router (it's called **dhcp**).

We will use `systemd-networkd`.

Type `ip addr`. It should shows something like this:

```
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN group default
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
       valid_lft forever preferred_lft forever
2: ifb0: <BROADCAST,NOARP> mtu 1500 qdisc noop state DOWN group default qlen 32
    link/ether 22:2b:20:5b:8e:b0 brd ff:ff:ff:ff:ff:ff
3: ifb1: <BROADCAST,NOARP> mtu 1500 qdisc noop state DOWN group default qlen 32
    link/ether 6a:68:fb:64:2f:c3 brd ff:ff:ff:ff:ff:ff
4: eth0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast state UP group␣
↪default qlen 1000
    link/ether b8:27:eb:2d:25:18 brd ff:ff:ff:ff:ff:ff
    inet 192.168.0.201/24 brd 192.168.0.255 scope global eth0
       valid_lft forever preferred_lft forever
```

you are interested just in name **eth0**. If it is there, it is ok. In future versions of system it can change to something other, for example *enp0s1*. Don't be afraid of it and just use that instead in next chapters.

In this part you'll need to get address of your router. How to obtain it?

And how to choose static address? As you know your router is assigning IP address automatically (it is called DHCP). But not randomly in full range. It has some range of IP addresses which it can assign. Standard is this: router has standard IP adress `192.168.0.1` and assign addresses from `192.168.0.2` to `192.168.0.254`. Second standard is `10.0.0.138` for router and it assignes addresses from `10.0.0.139` to `10.0.0.254`. But it *can* be anything else.

Interesting - and what the hell should you do that? I suggest to set one the address on the end from this range. You can notice, that my "eth0" has IP address `192.168.0.201`.

Open this file `/etc/systemd/network/ethernet_static.network` (how? just use `vim` as in the previous - but don't forgot to use `sudo` in front of `vim`, or you'll not be able to save it!) and paste this:

```
[Match]
Name=eth0

[Network]
Address=the.static.address.rpi/24
Gateway=your.router.ip.address
```

my example:

```
[Match]
Name=eth0

[Network]
Address=192.168.0.111/24
Gateway=192.168.0.1
```

Now you need to remove old non-static default profile `/etc/systemd/network/eth0.network`. Move it to your home folder just to be safe if something didn't work.

Try to restart RPi and try to SSH again. If you just can't connect, try to find out if RPi hadn't connected at all or it just doesn't use IP specified IP address (try to ssh to old IP, look into your router DHCP table, nmap...). If you want to get it back, just turn off RPi (plug off the power cable), take out SD card, plug in to your PC, move `eth0.network` from home directory to `/etc/systemd/network/`, turn RPi back and try it again.

If you successfuly connected, check how is `systemd-networkd` doing. To find out, type: `systemctl status systemd-networkd`. Does it shows "active (running)" and something like `gained carrier`?

```
â systemd-networkd.service - Network Service
   Loaded: loaded (/usr/lib/systemd/system/systemd-networkd.service; enabled)
   Active: active (running) since Wed 2014-06-11 18:42:13 CEST; 2 weeks 1 days ago
     Docs: man:systemd-networkd.service(8)
 Main PID: 213 (systemd-network)
   Status: "Processing requests..."
   CGroup: /system.slice/systemd-networkd.service
           ââ213 /usr/lib/systemd/systemd-networkd

Jun 17 17:52:01 smecpi systemd-networkd[213]:             eth0: lost carrier
Jun 17 17:52:02 smecpi systemd-networkd[213]:             eth0: gained carrier
```

5.10 Timesynchronization

You've maybe noticed that time is quite weird on your RPi. It is beacuse it does not have real hardware clock. Every time RPi is waken up, it thinks that is June 1970. You don't have to care about it, but after boot it would be fine that time is correctly set. You can do it by using really great part of `systemd`. Go ahead and check service that takes care about that: `systemctl status systemd-timesyncd`.

5.11 Configuring SSH

We will open RPi to world and in that case we need to secure it a bit. Service, which takes care about SSH is called `sshd`. "Where" it is? It is runned by systemd, so `systemctl status sshd` will show you some info :). We will configure it a bit. This is not necessary, but highly recommended! Brutal force attacks are really common (hundreds every day on my little unimportant server).

Open file `/etc/ssh/sshd_config` and edit or add these lines as follows:

```
Port 1234
PermitRootLogin no
PubkeyAuthentication yes
```

that't enough. Restart sshd `systemctl restart sshd`.

Since now, you cannot login as a root by ssh and thats good. Also - we changed the port of ssh. Think about "port" as a tunnel, which is used for ssh. There are about 60 thousands of them and you can choose whatever you want. As default there is port **22** used for ssh. We now changed that to (example) 1234. It is because on port 22 there is to big chance that someone will try to brutal force your credentials.

Since now, only `ssh bob@ipadress` is not enough. You will have to add port which should be used (in default is assumed port 22). `ssh -p 1234 bob@ip.address` will do it for you :) .

If you want to be really safe, the next thing you want to do is set up `sshguard`. More about it here. You don't need more :) . Just remember to use your port (in my case 1234) for settings. Personally I stopped to use it, since just changing port what SSH use was enough to reduce uninvited connections.

It is anoying still typing same username and password when we want to connect to RPi. And now, we have to add "-p 1234" also. We will make it automatic. Here is quite good guide how to do it. On PC from which you are connecting (no RPi), edit `~/.ssh/config` to this:

```
Host my_superpc
  HostName ipaddressofRPi
  IdentityFile /home/yourusername/.ssh/name_of_identityfile
  User bob
  port 1234
```

since now, when you wan't to connect to RPi you can just type `ssh my_superpc` and it will take care about rest.

Screen

You can live without that, but you shouldn't! It makes you more productive and you don't need to be afraid of some mishmash caused by accidently closing terminal during update or lossing connection. Learn more about what the screen is (here, here and here), install it (`pacman -S screen`), use it and love it.

It can be handy to automatically ssh into screen sesion. For that I use this command (from PC I want to connect to RPi):

`ssh my_superpc -t screen -dRS "mainScreen".` You can make some alias to something shorter (for example adding this to `alias ssh_connect_RPI="ssh my_superpc -t screen -dRUS mainScreen"` in .zshrc). Now all you need to do is type `ssh_connect_RPI` - it here is now screen created, it will create new one. If it is, it will attach it.

5.12 Speeding RPi up

Arch Linux ARM for RPi is prepared to be tweaked. And now it is possible to speed RPi up by overclocking it's processor without avoiding your waranty. How to do it? Just edit file `/boot/config.txt` and find this part:

```
##None
arm_freq=700
core_freq=250
sdram_freq=400
over_voltage=0
```

now comment it out. That means to add "#" in front of every line. From now, it will be treated as text and not command. It will look like this:

```
##None
#arm_freq=700
#core_freq=250
#sdram_freq=400
#over_voltage=0
```

and now uncoment this:

```
##Turbo
arm_freq=1000
core_freq=500
sdram_freq=500
over_voltage=6
```

After next boot your RPi will be able to get even to the 1000 MHz. That means it is faster.

5.13 Other tweaks of /boot/config.txt

Since you don't need any of gpu memory - which cares about shiny things like windows etc., you can disable it in favor of the rest of memory which we use. Don't do this if you want to use monitor.

```
gpu_mem=16
#gpu_mem_512=316
#gpu_mem_256=128
#cma_lwm=16
#cma_hwm=32
#cma_offline_start=16
```

5.14 Making RPi visible from outside

Now we need to configure access from outside. You will need to configure you router. You have to make a "port forwarding". Remember port from ssh? I told you to think about them as a tunnels. These tunnels are also handy when you need to find out what is on there end.

What we will do here is this: We want to be able from anywhere on the internet connect to our RPi server.

Example? `ssh -p 1234 bob@what.the.hell.is.here`. You know? There is definetely not your local address (the one with 192.168...). There must be your "public" IP address (more about this in **Domains** - take a look there). But this public address points to your router (if you are lucky). Where does it go next?

With every request there is also a port. With command `ssh smt`, you are sending username, port (standard 22, if not otherwise stated) and IP address. Ip address redirect it to router. Now router takes **port** and looks to it's internal database. In this database are pairs: **port** - **internal_ipaddress**. For some port there is IP address, which it redirects to. In another worlds: if router gets some request from specific port (say, 1234) and it has in it's database IP address

to which it has to redirect, it redirects this request there. In our case, we need to redirect these ports we want (for example 1234 for ssh) to RPi. So find a port forwarding settings for your router (this might be helpful) and set there port forward from port you setted for ssh to RPi. You can check if your port is open (it means it accepts requests here.

Since now, you can ssh from anywhere.

5.15 Webserver

5.15.1 Setting up nginx

Similiar to `ssh` handling *sshish* requests, Nginx is handling almost everything else and even... **WebServers**! Install nginx with `pacman -S nginx`. For security reasons create special user for it, for example using: `useradd -m -G wheel -s /usr/bin/zsh nginx` and also group `groupadd webdata`. Now create some folder for it. It can be `mkdir /var/www/` and now make them owners `chown nginx:webdata /var/www`. Of course, enable and start nginx.

`systemctl enable nginx`. It will start after boot.

Now port forward port number 80 to RPi on your router.

Open `/etc/nginx/nginx.conf`, it can looks like this:

```
user nginx;
worker_processes  1;

error_log  /var/log/nginx/error.log warn;

events {
    worker_connections  1024;
}

http {
    include       mime.types;
    default_type  application/octet-stream;
    server_names_hash_bucket_size 64;

    sendfile        on;

    keepalive_timeout  15;

    server{
        listen  80;
        server_name ~^xxx.xxx.xxx.xxx(.*)$;

        location / {
            root   /var/www/$1;
            index  index.html index.htm;
        }
    }

}
```

next, create `/var/www/test/index.html`:

```
<html>
  <head>
    <title>Sample "Hello, World" Application</title>
  </head>
  <body bgcolor=white>

    <table border="0" cellpadding="10">
      <tr>
        <td>
```

(continues on next page)

(continued from previous page)

```
          <h1>Sample "Hello, World" Application</h1>
        </td>
      </tr>
    </table>

    <p>This is the home page for the HelloWorld Web application. </p>
    <p>To prove that they work, you can execute either of the following links:
    <ul>
      <li>To a <a href="/">JSP page</a>.
      <li>To a <a href="/">servlet</a>.
    </ul>

  </body>
</html>
```

where xxx.xxx.xxx.xxx should be your public address. This will do this: when you type in your browser "youripaddress/test:80", you should see index Hello world example. Try that without `:80` - it will do the same! Default port for webpages is **80** (similiar to 22 for SSH). So it can be omited.

FTP

This will cover the most easy solution for FTP. Don't use this configuration in real, just for test purpouses. If you didn't download `vsftp`, do it now by `pacman -S vsftp`. Now we will create some directory where all files and users will end up after connecting. Let it be in `/var/www/test`. Now edit `/etc/vsftpd.conf` and add on the top this line:

```
anon_root=/var/www/test
```

and make sure that this line is uncommented:

```
anonymous_enable=YES
```

and just start it: `systemctl start vsftpd`.

Now we'll tell nginx about that. Add this to servers confs in `/etc/nginx/nginx.conf`.

```
server{
    listen  80;
    server_name ~^123.123.32.13(.*)$;
    location / {
        ssi on;
        root   /var/www/$1;
        index  index.html index.htm;
    }
}
```

where you need to replace IP address in `server_name` directive to your public IP.

What this little configuration does? It's simple. Every time you type to your brower your IP address and somthing behind it, it will transfer you to this "something" in /var/www/.

Example I created index.html here `/var/www/example/index.html`. I now type `123.123.32.13/test` to my browser and voila!

This nginx configuration isn't neccessary in our ftp example (it could be simpler), but I just like it...

You can now connect to ftp by typing this in your browser: `ftp://your_ip_address` or use your favorite FTP client (e.g. `filezilla`).

CAUTION - again, don't use this settings as default. There are great guides on the internet how to grant access only some users, password protected etc.

5.16 System analyzing and cleaning

Use your friend `systemd-analyze`. It will show you which units load really long time. Also `systemctl status` is great for finding failed units.

5.16.1 Disable things that you dont need

I guess you don't use ipv6 (if you don't know what it is, you don't need it :D). `systemctl disable ip6tables`. In case you use sshguard, you need also edit file `/cat /usr/lib/systemd/system/sshguard.service` and from **Wants** delete `ip6tables.service`.

5.17 Usefull utilites

Simple to use, just install them and run:

- nmon - for internet usage
- htop - for disk usage

5.17.1 Torrents

Your RPi is maybe running 24/7, so why not to use it for torrents? But how, when there is no GUI? It's pretty simple. We will use transmission - popular torrent client. Install it by `pacman -S transmission-cli` Installation should create a new user and group, called transmission. To check that, you can take a look to `/etc/passwd` and `/etc/group`. `transmission` will be runned by `systemd`. Let's see it it's service file is configured properly. Check `/usr/lib/systemd/system/transmission.service`:

```
[Unit]
Description=Transmission BitTorrent Daemon
After=network.target

[Service]
User=transmission
Type=notify
ExecStart=/usr/bin/transmission-daemon -f --log-error
ExecReload=/bin/kill -s HUP $MAINPID

[Install]
WantedBy=multi-user.target
```

`User=transmission` is important here (for security reasons). Next thing we need to do is check, if transmission has place where it will live. By default it is in `/var/lib/transmission(-daemon)`. In this dir should be also config file `settings.json`. There lays configuration for it.Edit it ass you wish. It is covered here and here. Maybe you'll need to forward ports as we did in previous chapters, you should make that again without problems :) . No we can run `transmission` daemon by `systemctl start transmission`. Now you can give it commands using transmission-remote . The most usefull (and that's all I need to know and use :)) are these:

- `transmission-remote <port> -a "magnetlink/url"` - adds torrent and starts download it
- `transmission-remote <port> -l` - list all torrents that are currently running

files should be stored in `/var/lib/transmission/Downloads`. It can be configured in config file :) .

5.18 Backups

For backups I choosed `rdiff-backup`. It's so stupid but works (almost) as expected. More about it's usage you can find in it's manual pages. For my example I'll redirect you to dir with configs in this repo. These are inserted to `cron` (you have it by default installed) to do SSH backup every day in 4AM. If I'm on local network I also do backup to my disc on other PC.

5.19 Final

That's all for now! I will see if this is used by someone and than I will see if I will continue.

5.20 Troubleshooting

- RPi don't boot - unplug everything from USB ports (there may be not enough of power to boot up and supply USB)

CHAPTER 6

Simple GitHub repo and ReadTheDocs set up

I've just wanted to set up a GitHub repository for this guide and I found that it's really ubearable to set so simple thing as this.

Here is short guide which should walk you through:

1. Creating repository on GitHub
2. Cloning it into your local machine
3. Submitting changes from your local machine using SSH
4. Submitting changes from repository on GitHub
5. Generating documentation with ReadTheDocs and *sphinx <http://sphinx-doc.org/>_*.

6.1 Creating repository on GitHub

If you've found this guide, I guess you are inteligent enough to create account on GitHub, so we'll skip this step. Same with installing git and SSH on your machine. Use google if you are lost.

To create a directory just go to your profile (e.g. https://github.com/your_username), click on **repositories** and then click on **NEW**. It's important to make repository **public** (default choice). You can also create README and LICENCE file - do it, if you want.

When repository is created, copy **Subversion checkout URL** which can be found in right panel of repository view. In my case it's:

```
https://github.com/kotrfa/test_repo
```

6.2 Cloning repository

Open terminal and choose folder where you'd like to clone your repository. In my case it is just my home directory. Go to this folder and run:

```
git clone https://github.com/kotrfa/test_repo
```

`cd` inside this folder. You should see LICENCE and README (in case you've created them) but it might be empty if you haven't insert anything through browser to your repository yet.

6.3 Setting up git

Now we need to initialize git folder. To do this run:

```
git init
```

this will create `.git` folder with all important informations. You don't need to mess with that for now.

Next thing we have to do is to setup username. To do this run:

```
git config --global user.email "your_email@your_mail.something"
git config --global user.name "your_username"
```

Let's test it now by creating file:

```
touch test_file.txt
```

now we have to add this file to git's eye - that it has to look if this file changed and in case it does, it will upgrade it on the remote repository on GitHub. Do that by:

```
git add test_file.txt
```

and now tell git that this file is prepared to be upgraded:

```
git commit test.md -m "testing file"
```

`-m` switch is for message and string `"testing file"` is the message which just gives some info about this `commit`.

Now we will send this changes to remote repository on GitHub. It's pretty easy:

```
git push
```

and type your username and password as it asks for it.

If this work, we can set connection without necessity typing our credentials every time.

6.4 Setting up SSH

For some reasons it's not really straigth forward to set up this.

First you have to go on GitHub website and go to **Account settings**. Navigate to **SSH Keys** and click on **SSH Key**.

Title is whatever you want to call it. Key field is what is interesting.

Go again to console and type:

```
ssh-keygen -t rsa -C "your_email_on_GitHub@mail.something"
```

and choose password as you want (or none).

It will generate SSH key inside `~/.ssh` and it has two parts - public (with `.pub` ending) and private. Content of the public must be copied into **Key** field on the GitHub page.

Now, when you'd like to work in github repository you've to run:

```
eval `ssh-agent -s`;ssh-add ~/.ssh/github_private_key
```

this should do the trick. Now you can `git push` as you wish without necessity to insert credentials.

6.5 Submitting changes from repository

To do that just use:

```
git pull
```

6.6 Generating docs with sphinx and RtD

6.6.1 Local sphinx generator

Install `sphinx` using `pip` and navigate yourself into git directory. Create `docs` folder there and go inside. Run:

```
sphinx-quickstart
```

and set to your needs.

Add your source rst files into some directory inside `docs`, for example `source`. Now edit `index.rst` and add there `source/filenames.rst`. In my case:

```
.. toctree::
   :maxdepth: 3

   source/intro
   source/nec_know
   source/domains_ip_servers
   source/ndg
   source/Arch
   source/RPi
```

where `maxdepth` says how much level should TOC has. Another useful directives are `:glob:`. In previous example I should just use `source/*` and it would load all `.rst` files inside `source` dir. If you'd like to have TOC numbered, just add `:numbered:`.

Now just run:

```
make html
```

and it will make a HTML pages for you inside `build/html` directory.

Go to the main git folder (in my case `~/test_repo`) and add, commit and push all changes:

```
git add --all
git commit -a -m "first docs"
git push
```

6.6.2 Read the Docs configuration

Go to the ReadTheDocs and create an account there.

Click on the dasboard and then on **import**. Name your project and add your git url inside **Repo**. In my case it's:

```
https://github.com/kotrfa/test_repo
```

Repository type is **Git** and documentation **Sphinx Html**. Rest is basicaly optional. Now just click on **Create** and wait.

Now you just have to wait :) . RtD will build your project every time it detects changes. Usually it was imediately, but sometimes it takes even several minutes.

CHAPTER 7

Integration with GitHub

For starter it's necessary to say, that GitHub...

...is awesome! If you don't needed and you went through whole process above, you can probably save a lot of headaches just by using GitHub. You don't need any special knowledge for start, but you will need to learn them on the fly while reading this tutorial (there is really a lot about git out there, google is your friend).

The variant I propose here is very easy, scalable and fast. Probably the most easy and effective I've found.

7.1 Why to use it

I was so happy when I deployd my first django project. But few weeks later I've found that it's just not feeling right to make changes on live version of the website (sometimes refered as *production*). So I started to use GitHub and found a solution.

Here I will cover this: For every your website you end up with one directory including three subdirectories.

1. First called **production** - it's the one which is live on the internet - the one what `nginx` refers.
2. Second called **mydomain.git** - this one is necessary for our github configuration. You will barely change there anything
3. Last one - **work_dir** - the one where all changes are being made and is connected to GitHub

7.2 Workflow

Your work will look like this:

1. Your work_dir contains master branch. This branch can be pushed to production (to go live) anywhen! So when you want to make change to your website, you need create new branch (correctly named based on the change you are doing - e.g. *hotfix_plugin*, *typo_css*...) and when you finish and test this branch, you merge it to master.
2. You push master to your GitHub repository

3. You push master to your production folder on your computer

7.3 Set it all up

So how to do it? I suppose you have one working directory as we created in previous chapters.

Now go to the place where you websites are stored. Mine is in `/var/www` and create this structure:

```
mydomain
├── mydomain.git
├── production
└── work_dir
```

Go to `/var/www/mydomain.git` and type this:

```
git init --bare
```

this will create just git repository with some special folders. You don't need to know anything about it. All you need to do is to create this file `/var/www/mydomain/mydomain.git/hooks/post-receive` and add this:

```
#!/bin/sh
git --work-tree=/var/www/mydomain/production --git-dir=/var/www/mydomain/mydomain.git␣
↪checkout -f
```

and make the script runable `chmod +x /var/www/mydomain/mydomain.git/hooks/post-receive`

Go to work_dir and paste there you current *production* code (the one from previous chapters). Now you need to make a GitHub repository from that. The best guide is this one: How to add existing folder to GitHub. (Maybe you'll need to generate SSH key). Is it working? Great.

> Note: It's very good to make git repositories as small as possible, so don't add to repository files which are not necessary or you backup them somewhere else. But virtualenv is a good thing to add there to IMHO.

Now just add another remote, which will point to our created git repository. Every time we'll want to go live with master, we'll push changes to this git repository and it will take care to transfer our files to production. So in work_dir type:

```
git remote add production file::///var/www/mydomain/mydomain.git
```

and that's all. When you now want to push changes to production, type `git push production master`. Congratulations!

CHAPTER 8

Indices and tables

- genindex
- modindex
- search