Web Development with Python

with flask, tornado and nginx

This book is dedicated to Violeta

This book is for

Beginners or advanced in python and total beginners in web programming with python. You should have a basic python knowledge.

Quick test:
1. You know how to convert a string to a number and vice-versa ?
2. Can you write a simple for loop in python ?
3. Can you create a simple class ?

If the answer to all above is “YES” then go ahead. If you don’t know, then you really should start with some basic python. There are lots of nice and free resources on net.

It helps if you know a little html too, like “What does <h1> do ?” for example.

Disclaimer:

This book is written by an **amateur**, and it’s goal is to provide you just with **a starting point** into Python - Flask web programming and giving you **my own version** on how to do things.

An **amateur** (French *amateur* "lover of", from Old French and ultimately from Latin amatorem nom. amator, "lover") is generally considered a person attached to a particular pursuit, study, or science in a non-professional or unpaid manner. Amateurs often have little or no formal training in their pursuits, and many are autodidacts (self-taught). (wikipedia)

Alternatives better than this book.

1. The flask documentation itself is pretty good
2. http://exploreflask.com/
3. Flask Web Development: Developing Web Applications with Python by Miguel Grinberg
4. Miguel Grindberg’s blog - free
5. Instant Flask Web Development by Ron DuPlain
6. There are even youtube movies on flask.

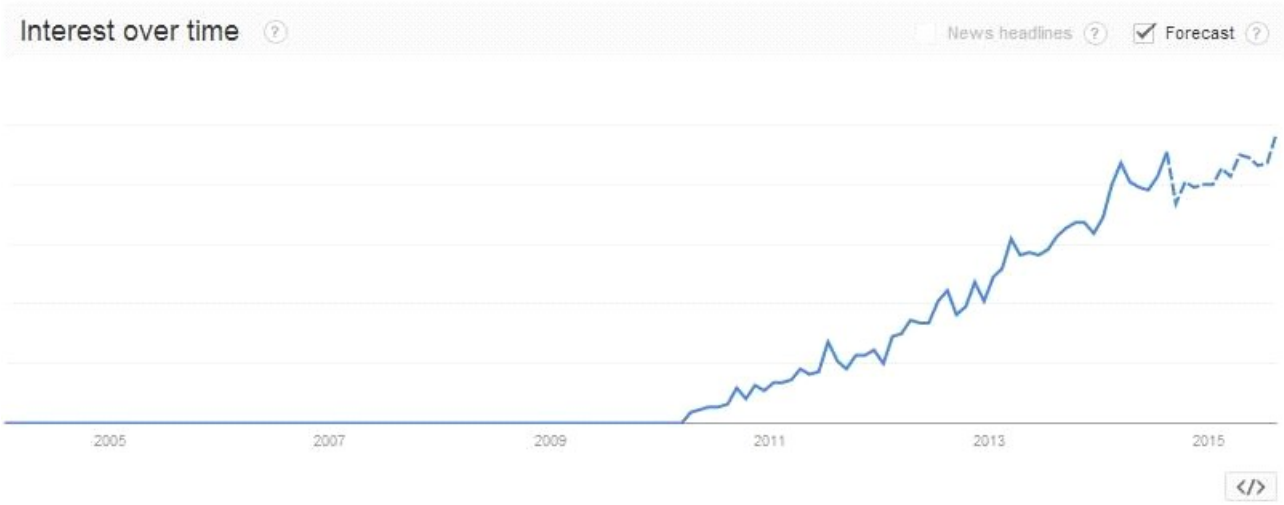
I remind you that **You can do things differently and more optimized!** Here I write my own idea on how things should be. If you’re good in front-end I’m sure you would find better alternatives. If you’re good in web-security I’m sure you’ll find better alternatives for example.

PREFACE

Flask is minimal and simple. You don't get ORM's, Admin Panels and other stuff that Django has out of the box. You can install a very cool admin panel with just 1 line of code: "pip install flask-admin" and integrate it with 3-4 lines in your app.

It is easy to learn, powerful and combined with Tornado it produces awesome performance even on a small VPS of 1Ghz.

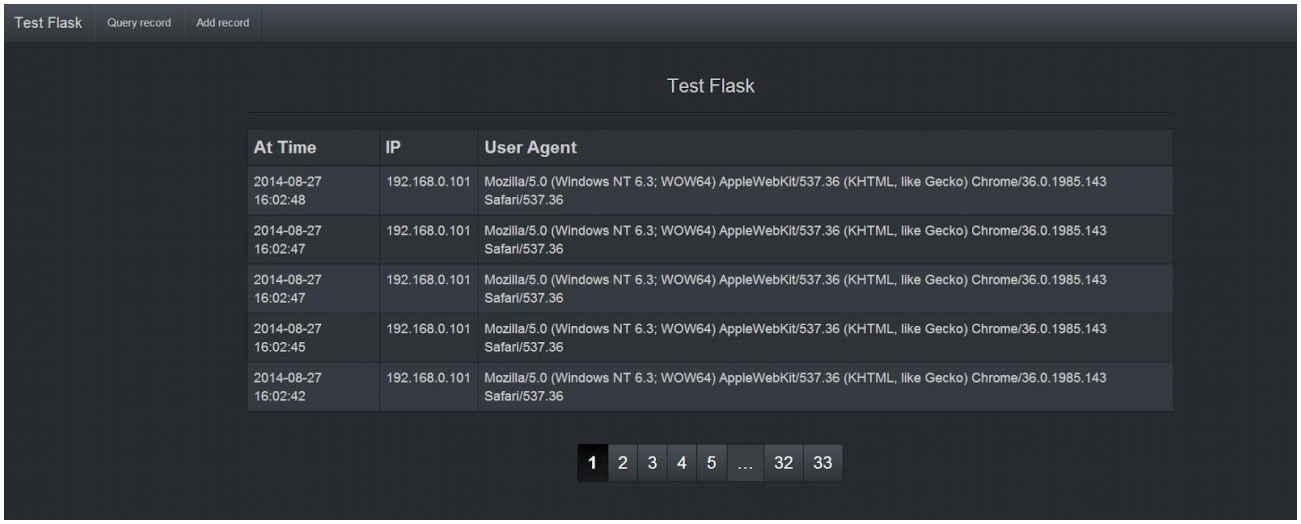
write in http://www.google.com/trends/ "Flask Python"



What you see is the trending of Flask Programming. Pretty cool isn't it ?

Quick preview on what we'll build in this book

A simple user-tracking database management system with pagination, admin panel, login, security.



As extra we'll discuss in the last chapter, best practices for production environment and optimizing it for a high traffic app.

Chapter 1 - Preparation

You can skip this chapter if you already have a linux environment set up and you don't what to setup a new one.

Suggestion:

For $5 per month, I strongly suggest that you get a VPS at digitalocean.com. In fact I will give you a gift of $10 if you sign up now using this link (you'll receive the money in your account after you sign in and get a new droplet) http://goo.gl/ArLvyx (this is a referal link to me from DigitalOcean, this way you can say thank you for this book and get $10)

- you get a static ip, you can add very easy a domain name and it's very fast. Now (August 2014) you can choose locations from: New York, San Francisco, Amsterdam, Singapore, London.

let's get started:

1. Install Ubuntu latest version
2. Login into ssh

This book assumes that you are familiar with terminal commands and running remote commands with putty. If you need help read this article https://www.digitalocean.com/community/tutorials/how-to-log-into-your-droplet-with-putty-for-windows-users

Tip: if you use Windows, install Putty then install https://code.google.com/p/superputty/ and you have a nice putty managment with multiple windows.

Commands to run:

```
df -h
ifconfig –a
ping –c 4 google.com
```

* if you get unknown host * -> sudo nano /etc/resolv.conf nameserver 192.168.1.1 (line down) nameserver 8.8.8.8

```
sudo apt-get update && sudo apt-get upgrade
```

[ssh]

```
sudo apt-get install -y openssh-server
sudo nano /etc/ssh/sshd_config
sudo restart ssh
```

sudo apt-get install -y htop zip rar unrar
sudo apt-get install -y mysql-client mysql-server
sudo apt-get install -y nginx

[webmin]
sudo nano /etc/apt/sources.list
  -> deb http://download.webmin.com/download/repository sarge contrib
wget -q http://www.webmin.com/jcameron-key.asc -O- | sudo apt-key add -
sudo apt-get update
sudo apt-get install webmin

[failtoban]
sudo apt-get install -y fail2ban
sudo nano /etc/fail2ban/jail.conf

check that you have this configuration

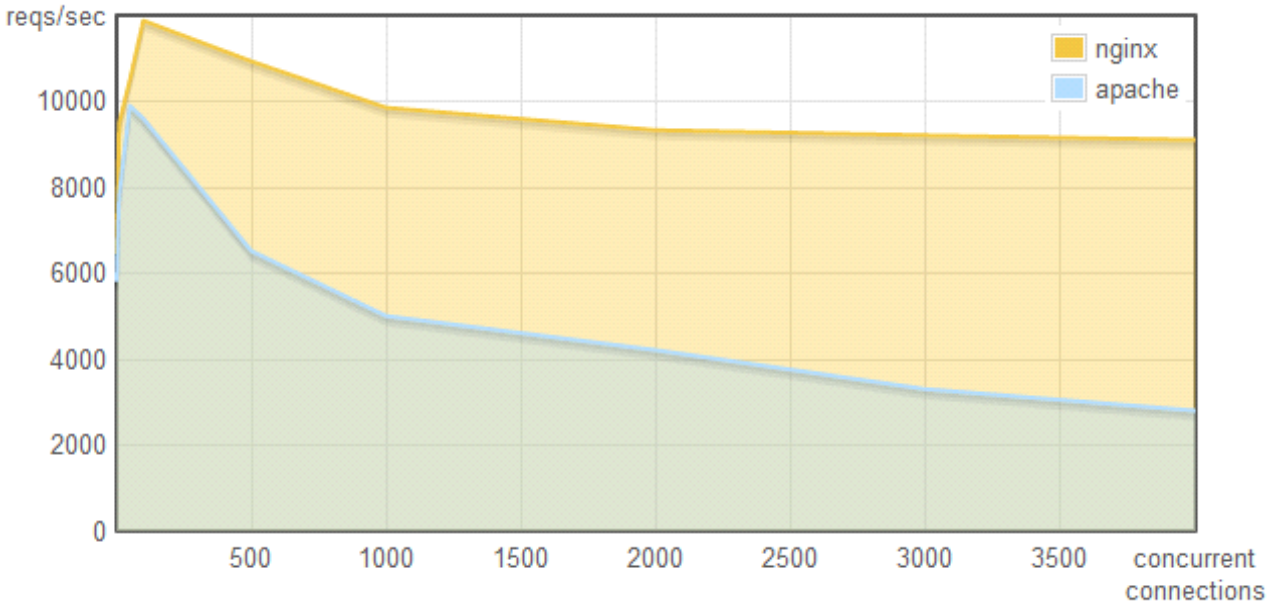
```
[ssh]
enabled  = true
port     = ssh
filter   = sshd
logpath  = /var/log/auth.log
maxretry = 3
```

sudo /etc/init.d/fail2ban restart
sudo fail2ban-client status

sudo apt-get install -y build-essential python python-dev python-pip python-mysqldb libmysqlclient-dev supervisor libmemcached-dev memcached python-memcache

pip install flask flask-login flask-mail sqlalchemy flask-sqlalchemy flask-wtf flask-migrate tornado flask-cache simpleencode
pip install pdfminer flask-admin flask-security

Nginx (pronounced "engine-x") is an open source reverse proxy server for HTTP, HTTPS, SMTP, POP3, and IMAP protocols, as well as a load balancer, HTTP cache, and a web server (origin server). The nginx project started with a strong focus on high concurrency, high performance and low memory usage.



Webmin is a web-based system configuration tool for Unix-like systems, although recent versions can also be installed and run on Windows. With it, it is possible to configure operating system internals, such as users, disk quotas, services or configuration files, as well as modify and control open source apps, such as the Apache HTTP Server, PHP or MySQL.

Fail2ban is software to protect computer servers from single-source brute-force attacks. Fail2ban is an intrusion prevention framework written in the Python programming language. It is able to run on POSIX systems that have an interface to a packet-control system or firewall installed locally (for example, iptables or TCP Wrapper).

pip is a package management system used to install and manage software packages written in Python.

Tornado is a scalable, non-blocking web server and web application framework written in Python.

Tornado is noted for its high performance. It tries to solve the C10k problem affecting other servers. The following table shows a benchmark test of Tornado against other Python-based servers:

Server	Setup	Requests per second
Tornado	nginx, four frontends	8213

Tornado	One single-threaded frontend	3353
Django	Apache/mod_wsgi	2223
web.py	Apache/mod_wsgi	2066
CherryPy	Standalone	785

Chapter 2 - Getting your hands dirty with “Hello World!”

PRO TIP OF THE DAY: the fastest way to learn this book is to type everything manually from it. Copy-paste is not productive for learning programming unless you understand 100% the code. If you make a typo, then the simple action of debugging it will give you a reward in learning.

If you already have a webserver and you skipped the first Chapter, run and install the following:

```
sudo apt-get install -y build-essential python python-dev python-pip python-mysqldb libmysqlclient-dev supervisor libmemcached-dev memcached python-memcache

pip install flask flask-login flask-mail sqlalchemy flask-sqlalchemy flask-wtf flask-migrate tornado flask-cache simpleencode
pip install pdfminer flask-admin flask-security
```

Hello World Application

Create a new directory under /home

```
cd /home
mkdir helloworld
```

create a new file named run.py

nano run.py

run.py

```
from flask import Flask
app = Flask(__name__)

@app.route('/')
def hello_world():
    return 'Hello World!'

if __name__ == '__main__':
    app.run()
```

save it, CTRL+X then “Y”, Enter

now type:

python run.py

Then from your browser open

http://your_server_ip:5000/

If you forgot your server ip, write

wget -qO- http://ipecho.net/plain ; echo

Or if you use digitalocean you can see it after you login on your account after your VPS name.

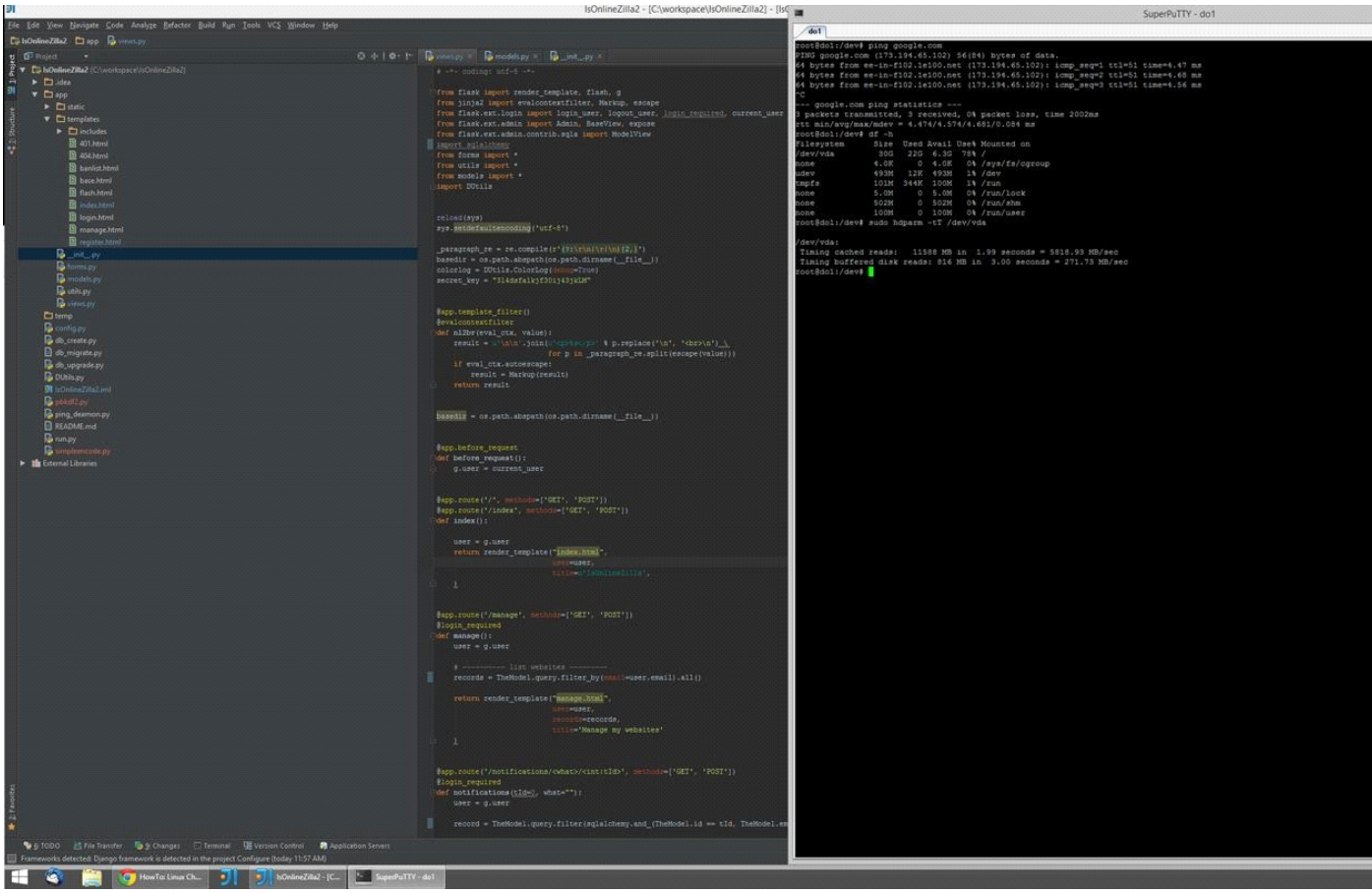
You should see a white page with a **“Hello World!”**. That’s all.
After you are done admiring your first flask application, hit CTRL+C.

PRO TIP OF THE DAY: you can write on top of run.py #!/usr/bin/python then chmod +x run.py so you can just type ./run.py instead of “python run.py”.
If you get error like: -bash: ./run.py: /usr/bin/python^M: bad interpreter: No such file or directory
apt-get install dos2unix then write “dos2unix run.py”. [and configure your IDE to use Line separator Unix and OSX. code style->general in IntelliJ]
If you already knew this, and I offended you with this time wasting info, I apologize!

Chapter 3 - Setting up your development environment

The settings are just my preferences. You are free to use whatever you want of course.

Using nano to edit scripts is not productive. Here's a screenshot on how I do it. You are free to choose whatever method suits you however.



You see IntelliJ Studio and SuperPutty.

The text is so small because I have a 2560x1440 resolution (best money spend ever on a good monitor), I have DELL U2713HM (now it's about $600)
I themed IntelliJ with a dark theme, so my eyes don't hurt from so much white. You prefer another color -> http://ideacolorthemes.org/home/ they have quite a few themes.

Python: https://www.python.org/download
IntelliJ Studio: http://www.jetbrains.com/idea/download/
Putty: http://the.earth.li/~sgtatham/putty/latest/x86/putty.exe

SuperPuty: https://code.google.com/p/superputty/

Start IntelliJ, configure whatever it asks you, choose "New Project" -> Python -> (don't check Django, Google App Engine etc) just click next. next. Python interpreter. Name your project.

Skip this step if you plan to use your own computer for development. But eventually you'll end up doing this in the end.

Go to Tools -> Deployment -> Configuration -> Add a new server. Select SFTP, fill the detals host, port, username, password. Click test SFTP connection. You should see an Successfully message. Select root path. Create a new folder in /home (for example).
Click the next tab from top Mappings
Type "/" in the Deployment path on the server ''server_name''
Click "Use this server as default (a button on top)

Go to Tools -> Deployment -> Options. Check "Create empty directories". And select "Upload changed files automatically to the default server "Always".

Now, to have IntelliJ show you nicely the code you need to install the packages on your computer too.

Get the pip for windows
https://pypi.python.org/pypi/pip#downloads (link might change, so if it doesn't work, just google "install pip on windows"

start the shell on windows and write:

pip install flask flask-login flask-mail sqlalchemy flask-sqlalchemy flask-wtf tornado flask-cache flask-admin flask-security

You need to install them both on windows and on the server because IntelliJ will use them too.

Now let's test if everything is going ok.

Remove the HelloWorld directory. You are not a beginner anymore :)

Let's now structure the app a little.
Create a new directory named flask_tutorial.

```
mkdir flask_tutorial
cd flask_tutorial
mkdir app
mkdir app/static
mkdir app/templates
```

In the main directory "flask_tutorial" create a file named run.py
Inside app create a **__init__.py** and **views.py**

Here's the standard folder structure of a Flask App.
So everything should look like this:

```
flask_tutorial/
        app/
                __init__.py
                static/
                templates/
                views.py
        run.py
```

The app folder is containing the bread and butter. Static folder is for css, js, jpg etc. files. The __init__.py is where we will create our app object. The run.py is where the server will be.

Edit: **app/__init__.py**

```
from flask import Flask
# Define the WSGI application object

app = Flask(__name__)

from app import views
```

Here we just create the app object and import the views in it. In views we keep all the logic on how the app responds to url requests.

Edit: **app/run.py**

```
import tornado
from tornado import autoreload
from tornado.wsgi import WSGIContainer
from tornado.httpserver import HTTPServer
from tornado.ioloop import IOLoop
from tornado.log import enable_pretty_logging
from app import app
enable_pretty_logging()

http_server = HTTPServer(WSGIContainer(app))
http_server.listen(1337)
ioloop = tornado.ioloop.IOLoop().instance()
autoreload.start(ioloop)
ioloop.start()
```

This is a default configuration on 1 core for the tornado server. We'll start like this because you'll just write it once and you'll use it until the end of the book.

If you want to go into details with tornado, at the end of the book I'll give you a better config, but for now this is more than enough.

Pretty logging for a nice display on the terminal. Notice that we start the app on port **1337**!

[offtopic]

If 1337 doesn't tell you anything then here's the wikipedia intro to it.

Leet (or "**1337**"), also known as **eleet** or **leetspeak**, is an alternative alphabet for the English language that is used primarily on the Internet. It uses various combinations of ASCII characters to replace Latinate letters. For example, leet spellings of the word *leet* include *1337* and *l33t*; *eleet* may be spelled *31337* or *3l33t*.

The term leet is derived from the word *elite*. The leet alphabet is a specialized form of symbolic writing. Leet may also be considered a substitution cipher, although many dialects or linguistic varieties exist in different online communities. The term **leet** is also used as an adjective to describe formidable prowess or accomplishment, especially in the fields of online gaming and in its original usage – computer hacking.
[/offtopic]

Edit: **app/views.py**

```
from app import app

@app.route('/')
@app.route('/index')
def index():
```

```
    return "Hello, World2!"
```

Here we map the / and the /index to our “index()” function, that just returns a simple text.

Now go in the main folder and run “**python run.py**” then with our browser test the app at

http://your_server_ip:1337

you should see a “Hello, World2!” on it.

Chapter 4 - Making the app look good.

Twitter **Bootstrap** has evolved as an efficient tool kit and is widely used today for creating websites. The reason that we do this now, is because it’s nicer to learn on something that looks good. Let’s integrate bootstrap in our app

Probably at the time you read this, the libraries have different versions. You can just google each one to get the latest and the greatest.

[don’t feel like typing ? go to the github page of this book https://github.com/AndreiD/FlaskBook]

In the templates folder create a file called “**base.html**”. In it put:

```
<!DOCTYPE html>
<html lang="en" class="no-js">
{% set bootstrap_version = '3.2.0' %}
{% set modernizer_version = '2.8.2' %}
{% set jquery_version = '2.1.1' %}
{% set bootswatch_version = '3.2.0' %}
{% set bootswatch_theme = 'slate' %}

<head>
    <meta charset="utf-8">
    <meta name="viewport" content="width=device-width, initial-
scale=1.0" />
    <meta http-equiv="X-UA-Compatible" content="IE=Edge,chrome=1" />
    <title>{% block title%}Flask Testing{% endblock %}</title>
```

```
    <link href="//netdna.bootstrapcdn.com/bootstrap/
{{ bootstrap_version }}/css/bootstrap.min.css" rel="stylesheet" />

 <link href="//netdna.bootstrapcdn.com/bootswatch/
{{ bootswatch_version }}/{{ bootswatch_theme }}/bootstrap.min.css"
rel="stylesheet" >


    <link href="/static/css/main.css" rel="stylesheet" />
    <link rel="shortcut icon" href="/static/img/favicon.ico" />
    {% block style_block %}{# page-specific CSS #}{% endblock %}
    <script src="//cdnjs.cloudflare.com/ajax/libs/modernizr/
{{ modernizer_version }}/modernizr.min.js"></script>{# Modernizr must be
here, above body tag. #}
    {% block head_script %}{# defer-incapable JS block #}{% endblock %}
</head>
<body>

        {% include 'includes/nav.html' %} {# pull in navbar #}

    <div class="container" id="maincontent">
        {% include 'includes/flash_message.html' %} {# page-level
feedback notices #}
        <div id="body_content">
            {% block content %}{# main content area #}{% endblock %}
        </div>
    </div><!-- /container -->
    <footer>
        <div id="footer" class="container">
            {% block footer %}{% endblock %}
        </div><!-- /footer -->
    </footer>
    <script src="https://ajax.googleapis.com/ajax/libs/jquery/{{ jquery_version
}}/jquery.min.js"></script>
    <script src="//netdna.bootstrapcdn.com/bootstrap/
{{ bootstrap_version }}/js/bootstrap.min.js"></script>
    <script src="/static/js/main.js"></script>
</body>
</html>
```

Pro tip of the day: instead of the bootswatch - "**slate**" theme, replace it with the one of your liking.

You notice some tags like
{% include 'includes/nav.html' %}

As you probably suspect in the templates folder create another folder named **includes**. We will use it to structure our code in a nice way, adding navigation to our site much more easy.

The **{{ variable }}** means dynamic content. Something like: Total users: **<?php $total_users; ?> in php**

/app/templates/includes/nav.html

```
<nav class="navbar navbar-default" role="navigation">
  <div class="navbar-header">
    <button type="button" class="navbar-toggle" data-toggle="collapse"
data-target="#bs-example-navbar-collapse-1">
      <span class="sr-only">Toggle navigation</span>
      <span class="icon-bar"></span>
      <span class="icon-bar"></span>
      <span class="icon-bar"></span>
    </button>
    <a class="navbar-brand" href="/">Test Flask</a>
      <ul class="nav navbar-nav navbar-right">
        <li><a href="#">Example Nav</a></li>
      </ul>
  </div>
</nav>
```

Because we'll be using after some time a nice way to give feedback to users called flash let's create it now

/app/templates/includes/flash_message

```
{% with messages = get_flashed_messages(with_categories=true) %}
  {% if messages %}
    {% if category == 'error' %}
      {% set icon = 'icon-exclamation-sign' %}
    {% elif category == 'success' %}
      {% set icon = 'icon-ok-sign' %}
    {% else %}
```

```
            {% set icon = 'icon-info-sign' %}
        {% endif %}
        {% for category, message in messages %}
            <div class="alert alert-{{ category }}">
                <i class="{{ icon }}"></i> 
                <a class="close" data-dismiss="alert">×</a>
                {{ message }}
            </div>
        {% endfor %}
    {% endif %}
{% endwith %}
```

In the static folder create 3 folders: **css, img, js.** In the folder css create a file named **main.css** and in the js one **main.js.** You can use the to add custom things to your template.

After all this preparation, let's finally create our **index.html**

/app/templates/index.html

```
{% extends "base.html" %}
{% block content %}
    <div style="font-size: 1.5em;text-align: center">
        <h3>Testing Flask</h3>
        <hr>
    </div>
{% endblock content %}
```

This is all! Now we have everything in place. Just need to tell that on "/" or "/index" url the view should display the "index.html" template. To do so update

/app/views.py

```
from app import app
```

```
from flask import render_template
import logging

@app.route('/')
@app.route('/index')
def index():
    return render_template("index.html")
```

Now go in the main folder and run "**python run.py**"

Because learning something should be fun, let's quickly test our "webserver" to see how it performs. Ideally you should use another server or machine for this.

Copy paste some random loren ipsum in the index.html so you load the page some more. Keep in mind that this is a simple static page, no database queries, no other funny stuff is inside.

sudo apt-get install apache2-utils
now for the benchmark write

ab -n 1000 -c 100 http://server_ip:1337/

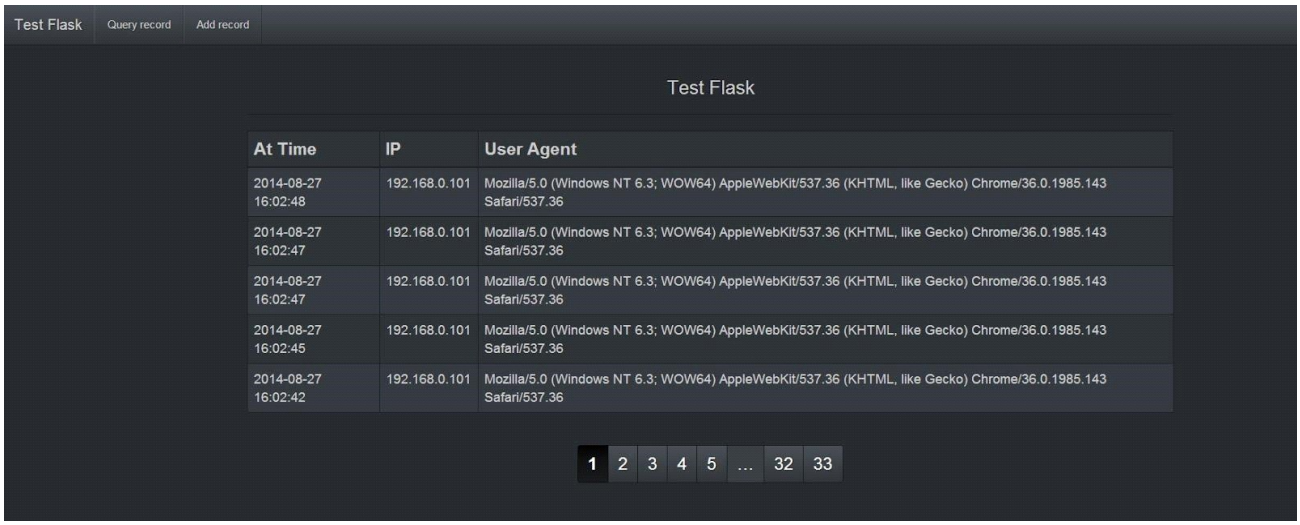
Look at **Time per request** value. I get 126.564 [ms] (mean) on an i3 laptop with 4GB ram. Try playing with increasing the -c to 250 for example.

Pro tip of the day: This testing is a little useless if you are running the test from localhost and you are not interacting with any database. You should run it from a different machine to get more accurate results. Also the tornado is set up to use just 1 core of your machine. But this is for the last chapter.

Later on, we'll use nginx to make it even faster.

Chapter 5 - Databases - made simple

What we'll build next:



SQLAlchemy is the Python SQL toolkit and Object Relational Mapper that gives application developers the full power and flexibility of SQL.
An ORM is good for abstracting the datastore (sqlite, mysql, oracle, etc etc) in order to provide an interface that can be used in your code.
If you didn't installed it from the first chapter go ahead and write

pip install flask-sqlalchemy

Because we'll use lots of configuration constants in our app. Let's organize them in a config file.

In the main folder **flask_tutorial,** (where the **run.py** is) create a new file called **config.py**

/config.py

```
import os
basedir = os.path.abspath(os.path.dirname(__file__))

class Config(object):
    DEBUG = False
    TESTING = False
    SQLALCHEMY_DATABASE_URI = ''
    APP_NAME = 'Flask Test'
    SECRET_KEY = 'thisisaveryhardsecret!1234!1234'



class ProductionConfig(Config):
    SQLALCHEMY_DATABASE_URI =
'mysql://username:password@server_ip/db'
    DEBUG = False


class DevelopmentConfig(Config):

    SQLALCHEMY_DATABASE_URI = 'sqlite:///' + os.path.join(basedir,
'db.sqlite')
    SQLALCHEMY_MIGRATE_REPO = os.path.join(basedir, 'db_repository')
    DEBUG = True
```

```
class TestingConfig(Config):

    SQLALCHEMY_DATABASE_URI = 'sqlite:///' + os.path.join(basedir,
'db.sqlite')
    SQLALCHEMY_MIGRATE_REPO = os.path.join(basedir, 'db_repository')
    TESTING = True
```

Now we should let know that the app will use this configuration

/app/__init__.py

```
from flask import Flask
from flask.ext.sqlalchemy import SQLAlchemy
import logging

app = Flask(__name__)
app.config.from_object('config.DevelopmentConfig')
db = SQLAlchemy(app)

logger = logging.getLogger(__name__)
logger.setLevel(logging.INFO)
```

```
from app import views, models
```

Now a model for the database is required, which are a collection of classes that we'll use to interact with the db.

A **database model** is a type of data model that determines the logical structure of a database and fundamentally determines in which manner data can be stored, organized,

and manipulated. The most popular example of a database model is the relational model, which uses a table-based format. [wikipedia]

create a **model.py** in /app

/app/model.py

```
import datetime
from app import db

class Tracking(db.Model):

    __tablename__ = "tracking"

    id = db.Column(db.Integer, primary_key=True)
    user_ip = db.Column(db.String(46))
    user_agent = db.Column(db.String(100))
    at_time = db.Column(db.DateTime, default=datetime.datetime.now)

    def add_data(self, user_ip, user_agent):
        new_user = Tracking(user_ip=user_ip, user_agent=user_agent)
        db.session.add(new_user)
        db.session.commit()

    def list_all_users(self):
        return Tracking.query.all()

    def __repr__(self):
        return '<Tracking %r>' % (self.id)
```

Update views.py to

/app/views.py

```
from flask import render_template, request
from models import *
from flask.ext.admin import Admin, BaseView, expose
```

```
from flask.ext.admin.contrib.sqla import ModelView
from app import *
import logging

# Executes before the first request is processed.
@app.before_first_request
def before_first_request():
    logging.info("---------- initializing everything ----------")
    db.create_all()

@app.route('/')
@app.route('/index')
def index():

        new_tracking = Tracking()
        new_tracking.add_data(request.remote_addr,request.headers.g
        et('User-Agent'))

        list_records = new_tracking.list_all_users()

        for record in list_records:
         logging.info(record.user_ip + " " + record.user_agent)

    return render_template("index.html")
```

The @app.before_first_request

This creates the database when you first access your website.

You can test the database with a quick view: nano db.sqlite ….you should see your table there.

Using **request** we get the visitor ip and the user-agent and we store it each time we refresh the page.

Open your browser http://server_ip:1337/
And “python run.py”
You should see in the terminal something like this

[I 140827 11:22:29 views:18] 192.168.0.123 Mozilla/5.0 (Windows NT 6.3; WOW64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/36.0.1985.143 Safari/537.36

and each time you refresh it, more records are added.

Chapter 6 - Making the app tick.

No need to write now, just read.
So the url routing is done like this:

```
@app.route('/welcome')
def welcome():
    return render_template('welcome.html')  # render a template
```

This means that if you write **you_website.com/welcome** you will get the **"welcome.html"** from the templates directory.

You can also use the routing to display static content like this:

```
@app.route("/favicon.ico")
def favicon():
    return app.send_static_file("img/favicon.ico")
```

or other static content:
see http://stackoverflow.com/a/14054039/1031297

```
from flask import Flask, request, send_from_directory
@app.route('/robots.txt')
@app.route('/sitemap.xml')
def static_from_root():
    return send_from_directory(app.static_folder, request.path[1:])
```

You can also make the url dynamic and add variables to it like this (from flask documentation)

```
@app.route('/user/<username>')
def show_user_profile(username):
    # show the user profile for that user
    return 'User %s' % username
```

Or with an integer

```
@app.route('/post/<int:post_id>')
def show_post(post_id):
    # show the post with the given id, the id is an integer
    return 'Post %d' % post_id
```

You can combine them

```
@app.route('/hello/')
@app.route('/hello/<name>')
def hello(name=None):
    return render_template('hello.html', passed_name=name)
```

Notice the variable name with the default None. You can use it in your templates using {{ … }}
like Greeting **{{ passed_name }}**!

and you can specify http methods

```
@app.route('/login', methods=['GET', 'POST'])
def login():
    if request.method == 'POST':
        do_the_login()
    else:
        show_the_login_form()
```

Now let's modify the index.html template to show nicely the records in our database and add a new url to display a record by id.

/app/views.py

```
from flask import render_template, request
from app import *
from models import *


@app.route('/')
@app.route('/index')
def index():
    new_tracking = Tracking()
    new_tracking.add_data(request.remote_addr, request.headers.get('User-
```

```
Agent'))
    list_records = new_tracking.list_all_users()
    return render_template("index.html", list_records=list_records)
```

PRO TIP OF THE DAY: You can use **CTRL+ALT+L** to re-format your code in IntelliJ. Try it.

The return Tracking.query.all() returns a list with all records. We passed the list of all records to our index template.

Quick intro into Jinja2 (the templates that flask is using)

read more here http://jinja.pocoo.org/docs/dev/templates/

for the if we have:

```
  {% if True %}
      it's true
  {% endif %}
```

a for statement is done like this

```
{% for item in list_of_items %}
 do something with {{ item }}
{% endfor %}
```

and for if/else:

```
{% if kenny.sick %}
    Kenny is sick.
{% elif kenny.dead %}
    You killed Kenny!  You bastard!!!
{% else %}
    Kenny looks okay --- so far
{% endif %}
```

you can also use else with for like this:

```
<ul>
{% for user in users %}
    <li>{{ user.username }}</li>
```

```
{% else %}
   <li><em>no users found</em></li>
{% endfor %}
</ul>
```

[great examples taken from jinja2 documentation]

so we modify our index.html

/app/templates/index.html

```
{% extends "base.html" %}
{% block content %}
   <div style="font-size: 1.5em;text-align: center">
      <h3>Test Flask</h3>
      <hr>

   {% for record in list_records  %}
      {{ record.user_ip }}
      <br />
      {{ record.user_agent }}
   {% endfor %}

   </div>
{% endblock content %}
```

Test it in your browser. Works! but wait, it looks **ugly**. Let's wrap it up in a bootstrap nice table.

```
{% extends "base.html" %}
{% block content %}
   <div style="font-size: 1.5em;text-align: center">
      <h3>Test Flask</h3>
      <hr>
      <table class="table table-striped table-bordered table-hover">
         <thead>
         <tr>
            <th>IP</th>
            <th>User Agent</th>
         </tr>
         </thead>
         {% for record in list_records %}
```

```
            <tr>
               <td> {{ record.user_ip }}</td>
               <td>{{ record.user_agent }}</td>
            </tr>
         {% endfor %}
      </table>
   </div>
{% endblock content %}
```

and edit the

/app/static/css/main.css

```
td {
   font-size: 0.8em;
   text-align: left;
}
```

Test it. Looks much better now isn't it ?

Going even further

Problem: if you refresh the page couple of times you get bigger and bigger listing. Let's add pagination to it. With SQLAlchemy pagination is piece of cake. Here's the recipe since I'm supposing you don't have time to read the official documentation.

In /config.py in the main Config(object) add **LISTINGS_PER_PAGE = 5**

```
class Config(object):
   DEBUG = False
   TESTING = False
   SQLALCHEMY_DATABASE_URI = ''
   APP_NAME = 'Flask Test'
   SECRET_KEY = 'thisisaveryhardsecret!1234!1234'
   LISTINGS_PER_PAGE = 5
```

Change the **/app/models.py** to

```
   def list_all_users(self,page, LISTINGS_PER_PAGE):
      return Tracking.query.paginate(page, LISTINGS_PER_PAGE, False)
```

/app/views.py

```
from flask import render_template, request
from app import *
from models import *


@app.route('/')
@app.route('/index')
@app.route('/index/<int:page>')
def index(page=1):
    new_tracking = Tracking()
    new_tracking.add_data(request.remote_addr, request.headers.get('User-Agent'))
    list_records =
new_tracking.list_all_users(page,app.config['LISTINGS_PER_PAGE'])
    return render_template("index.html", list_records=list_records)
```

Now we have to add the pagination 1, 2, 3….8,9 small buttons at the bottom of the page.

/app/templates/index.html

```
{% extends "base.html" %}
{% block content %}
    <div style="font-size: 1.5em;text-align: center">
        <h3>Test Flask</h3>
        <hr>
        <table class="table table-striped table-bordered table-hover">
            <thead>
            <tr>
                <th>At Time</th>
                <th>IP</th>
                <th>User Agent</th>
            </tr>
            </thead>
            {% for record in list_records.items %}
                <tr>
                    <td> {{ record.at_time }}</td>
                    <td> {{ record.user_ip }}</td>
                    <td>{{ record.user_agent }}</td>
                </tr>
            {% endfor %}
```

```
    </table>

    <ul class="pagination">
      {%- for page in list_records.iter_pages() %}
        {% if page %}
          {% if page != list_records.page %}
            <li> <a href="{{ url_for('index', page = page) }}">{{ page }}</a> </li>
          {% else %}
            <li class="active"> <a href="#"><strong>{{ page }}</strong></a>
</li>
          {% endif %}
        {% else %}
          <li> <span class=ellipsis>…</span> </li>
        {% endif %}
      {%- endfor %}
    </ul>

  </div>
{% endblock content %}
```

Test it, and see the greatness of your work!
[If something is wrong, copy-paste the file from the github]

Optional Improvements:

add a time formatting:

```
        <td> {{ record.at_time.strftime('%Y-%m-%d %H:%M:%S')  }}</td>
```

order the records by created time:

/app/models.py

```
[...]
from sqlalchemy import asc, desc
[...]
   def list_all_users(self,page, LISTINGS_PER_PAGE):
       return
Tracking.query.order_by(desc(Tracking.at_time)).paginate(page,
LISTINGS_PER_PAGE, False)
```

Add some test data

Since you've been connecting just from your computer, add a new different record to the database by accessing it from a web proxy.

https://simple-proxy.com/
[note: if you develop from a local network you can open the port from your router]

[note2: if you don't like this just add some dummy test data]
dummy test data;

somewhere in /index

```
new_user = Tracking(user_ip="100.100.100.100", user_agent="My browser")
db.session.add(new_user)
db.session.commit()
```

How to add url for displaying just one record

What you need to modify.

/app/models.py

```
    def track_user_ip(self, user_ip, page, LISTINGS_PER_PAGE):
        return Tracking.query.filter(Tracking.user_ip ==
user_ip).order_by(desc(Tracking.at_time)).paginate(page, LISTINGS_PER_PAGE,
False)
```

let's add the view now

/app/views.py

```
@app.route('/track/<user_ip>')
@app.route('/track/<user_ip>/<int:page>')
def track_user_ip(user_ip="", page = 1):

    new_tracking = Tracking()
    list_records = new_tracking.track_user_ip(user_ip, page,
app.config['LISTINGS_PER_PAGE'])
    return render_template("track_ip.html", list_records=list_records)
```

and duplicate the **index.html** to **track_ip.html**

[**optional**: you can modify it's title from Test Flask to Track IP or something]

Small instant homework. If you add more test data, you see that the pagination doesn't work. Fix it.

[---------- spoiler alert ----------]
[---------- spoiler alert ----------]
[---------- spoiler alert ----------]
[---------- spoiler alert ----------]
[---------- spoiler alert ----------]
[---------- spoiler alert ----------]
[---------- spoiler alert ----------]
[---------- spoiler alert ----------]
[---------- spoiler alert ----------]
[---------- spoiler alert ----------]
[---------- spoiler alert ----------]
[---------- spoiler alert ----------]

[---------- spoiler alert ----------]
[---------- spoiler alert ----------]
[---------- spoiler alert ----------]
[---------- spoiler alert ----------]
[---------- spoiler alert ----------]

change the {{url_for('index')}} in the pagination at the bottom to url_for("track_user_ip")]

Test it: **http://server_ip:1337/track/91.121.21.58**

Alternative: with Flask-Restless create an API for adding/deleting/updating a record.

Flask-Restless provides simple generation of ReSTful APIs for database models defined using SQLAlchemy (or Flask-SQLAlchemy). The generated APIs send and receive messages in JSON format.

Chapter 7 - Forms

Let's create some forms to add new records to our database.

create a new file **/app/forms.py**

/app/forms.py

```
from app import *
from wtforms.validators import Required, Length
from wtforms import Form, TextField


class TrackingInfoForm(Form):
    user_ip = TextField('user_ip', validators=[Required(), Length(max=46,
message='max 46 characters')])
    user_agent = TextField('user_agent', validators=[Length(max=46, message='max
46 characters')])
```

We are using WTF Forms. You can play with them like adding different fields, validators. But's let's keep it basic for now.

change the imports of /app/views.py to

/app/views.py

```
import logging
from flask import render_template, request, flash
from models import *
from forms import *
```

We're importing logging to do some debuging in the app, the forms and "flash" to display some feedback to the user.

and add this to views.py

/app/views.py

```
[...]

@app.route('/add_record', methods=['GET', 'POST'])

def add_record():

    form = TrackingInfoForm(request.form)

    if request.method == 'POST':
        if form.validate():
            new_tracking = Tracking()
            user_ip = form.user_ip.data
            user_agent = form.user_agent.data
            logging.info("adding " + user_ip + " " + user_agent)
            new_tracking.add_data(user_ip, user_agent)
            flash("added successfully", category="success")







    return render_template("add_record.html", form=form)
```

Update the navigation bar:

/app/templates/includes/nav.html

```
<nav class="navbar navbar-default" role="navigation">

    <div class="navbar-header">

        <button type="button" class="navbar-toggle" data-toggle="collapse" data-target="#bs-example-navbar-collapse-1">

            <span class="sr-only">Toggle navigation</span>

            <span class="icon-bar"></span>
```

```
            <span class="icon-bar"></span>
            <span class="icon-bar"></span>
        </button>
        <a class="navbar-brand" href="/">Test Flask</a>
        <ul class="nav navbar-nav navbar-right">
          <li><a href="/add_record">Add record</a></li>
        </ul>
    </div>
</nav>
```

Test it by adding few records to the db. **You should also see in the terminal the debug message.**

The most important part of this chapter are 2 homeworks that you really should do. They are simple and should take 2-3 minutes for the first one and about 10 minutes for the second.

Homework 1.

Add a validator for ip address in the form.

HINT: You should search the documentation of WTFForms

[spoiler alert]

[spoiler alert]

[spoiler alert]

[spoiler alert]

[spoiler alert]

[spoiler alert]

[spoiler alert]

[spoiler alert]

[spoiler alert]

[spoiler alert]

[spoiler alert]

[spoiler alert]

[spoiler alert]

[spoiler alert]

[spoiler alert]

[spoiler alert]

[spoiler alert]

[spoiler alert]

[spoiler alert]

[spoiler alert]

[spoiler alert]

from wtforms.validators import Required, Length, **IPAddress**

user_ip = TextField('user_ip', validators=[Required(), **IPAddress(message="Invalid IP Address")**])

Homework 2.

Remember the **track_user_ip** to display a record filtered by ip ? Add to the **track_ip.html** page a form with one input text where you can enter the IP and a button submit. Display the filtered ip

page after.

[spoiler alert]

[spoiler alert]

[spoiler alert]

[spoiler alert]

[spoiler alert]

[spoiler alert]

[spoiler alert]

[spoiler alert]

[spoiler alert]

[spoiler alert]

[spoiler alert]

[spoiler alert]

add

/app/forms.py

```
class QueryOneForm(Form):

    user_ip = TextField('user_ip', validators=[Required(), IPAddress(message="Invalid
IP Address")])
```

/app/templates/includes/nav.html

```
<li><a href="/track">Query record</a></li>
```

/app/views.py

modify track_user_ip function to:

```
@app.route('/track', methods=['GET', 'POST'])
@app.route('/track/<user_ip>', methods=['GET', 'POST'])
@app.route('/track/<user_ip>/<int:page>', methods=['GET', 'POST'])
def track_user_ip(user_ip="", page=1):

    form = QueryOneForm(request.form)

    if request.method == 'POST':
        if form.validate():
            user_ip = form.user_ip.data


    new_tracking = Tracking()
    list_records = new_tracking.track_user_ip(user_ip, page,
app.config['LISTINGS_PER_PAGE'])

    return render_template("track_ip.html", list_records=list_records, form=form,
user_ip = user_ip)
```

and

/app/templates/track_ip.html

```
{% extends "base.html" %}

{% block content %}

        <h3>Track IP</h3>

       <hr>
```

```
    <div class="row">


        <div class="well bs-component">
            <form class="form-horizontal" method="post" action="">
                <fieldset>
                    <legend>Search by IP</legend>
                    <div class="form-group">
                        <label for="user_ip" class="col-lg-4 control-label">User
IP</label>
                        <div class="col-lg-6">
                            {{form.user_ip (class="form-control")}}
                            {% for error in form.errors.user_ip %} <br/>
                                <div class="alert alert-danger" style="display:
inline-block">
                                    {{error}}
                                </div>
                            {% endfor %}
                        </div>
                    </div>




                    <div class="form-group">
                        <div class="col-lg-4  col-lg-offset-4">
```

```
                        <button type="submit" class="btn btn-
primary">Search</button>
                    </div>
                </div>
            </fieldset>
        </form>
    </div>
```

```
<table class="table table-striped table-bordered table-hover">
    <thead>
    <tr>
        <th>At Time</th>
        <th>IP</th>
        <th>User Agent</th>
    </tr>
    </thead>
    {% for record in list_records.items %}
        <tr>
            <td> {{ record.at_time.strftime('%Y-%m-%d %H:%M:%S')  }}</td>
            <td> {{ record.user_ip }}</td>
```

```
            <td>{{ record.user_agent }}</td>
        </tr>
    {% endfor %}
</table>

<ul class="pagination">
    {%- for page in list_records.iter_pages() %}
        {% if page %}
            {% if page != list_records.page %}
                <li> <a href="{{ url_for('track_user_ip', user_ip = user_ip, page = page) }}">{{ page }}</a> </li>
            {% else %}
                <li class="active"> <a href="#"><strong>{{ page }}</strong></a> </li>
            {% endif %}
        {% else %}
            <li> <span class=ellipsis>…</span> </li>
        {% endif %}
    {%- endfor %}
</ul>

</div>
{% endblock content %}
```

Chapter 8 - User login - with management

To write yourself the user registration, email confirmation, forgot password etc. would take some time and you'd have to be very good to write it without bugs and optimized.

We are grateful for having **Flask-Security** to save our souls.
https://pythonhosted.org/Flask-Security/index.html

Let's get our hands dirty. We want a user register, confirmation by email, forgot password and a /secret_page with a ultra-secret information inside.

We'll use yahoo, that has a limit of 100 emails per day. For our testing purposes it will be ok.
If you have another mail server, just change the config.
For large volumes you can check http://sendgrid.com

Let's clear our previous database first.
rm -rf db.sqlite

To use **Flask-Security**, first let's configure it

add this values in the main Config

/config.py

```
SECURITY_REGISTERABLE = True
SECURITY_RECOVERABLE = True
SECURITY_TRACKABLE = True
SECURITY_PASSWORD_HASH = 'sha512_crypt'
SECURITY_PASSWORD_SALT = 'add_salt_123_hard_one'
SECURITY_CONFIRMABLE = True
MAIL_SERVER = 'smtp.mail.yahoo.com'
MAIL_PORT = 465
MAIL_USE_SSL = True
```

```
    MAIL_USE_TLS = False
    MAIL_USERNAME = 'email@yahoo.com'
    MAIL_PASSWORD = 'password'
    DEFAULT_MAIL_SENDER = 'email@yahoo.com'
    SECURITY_EMAIL_SENDER = 'email@yahoo.com'
```

we need to add the flask-mail object

/app/__init__.py

```
[...]
from flask_mail import Mail
[...]
mail = Mail(app)
```

We need to update our models.py too

/models.py

```
from flask.ext.security import UserMixin, RoleMixin,
SQLAlchemyUserDatastore

roles_users = db.Table('roles_users',
                    db.Column('user_id', db.Integer(), db.ForeignKey('user.id')),
                    db.Column('role_id', db.Integer(), db.ForeignKey('role.id')))


class User(db.Model, UserMixin):
    id = db.Column(db.Integer, primary_key=True)
    email = db.Column(db.String(255), unique=True)
    password = db.Column(db.String(255))
    active = db.Column(db.Boolean())
    confirmed_at = db.Column(db.DateTime())
    roles = db.relationship('Role', secondary=roles_users,
                            backref=db.backref('users', lazy='dynamic'))
    last_login_at = db.Column(db.DateTime())
    current_login_at = db.Column(db.DateTime())
    last_login_ip = db.Column(db.String(255))
    current_login_ip = db.Column(db.String(255))
    login_count = db.Column(db.Integer)

    def __repr__(self):
        return '<models.User[email=%s]>' % self.email
```

```
class Role(db.Model, RoleMixin):
    id = db.Column(db.Integer(), primary_key=True)
    name = db.Column(db.String(80), unique=True)
    description = db.Column(db.String(255))


user_datastore = SQLAlchemyUserDatastore(db, User, Role)
```

And the views.py

/views.py

```
[...]
from flask.ext.security import Security, login_required, logout_user
[...]

security = Security(app, user_datastore)

@app.route('/secret')
@login_required
def secret():
    return render_template('secret.html')
```

now create a simple **secret.html** in templates and write "**secret page**" in the content.

Flask-security uses some templates. Because They are simple, and self explanatory, I won't copy paste them here.
I've customized my own templates with bootstrap, also you'll find email templates.
Here is the all chapter with all the code. You just have to **edit** the **config file** with **your mail credentials**.
https://github.com/AndreiD/FlaskBook/tree/master/flask_book/chapter_8

Chapter 9 - An Admin Panel to save us all

Flask-Admin

pip install Flask-Admin

https://github.com/mrjoes/flask-admin

Flask-Admin is a batteries-included, simple-to-use Flask extension that lets you add admin interfaces to Flask applications. It is inspired by the django-admin package, but implemented in such a way that the developer has total control of the look, feel and functionality of the resulting application. (official description)

Working with flask admin is very simple. Here's the way to do it with SQLAlchemy.

```
from flask.ext.admin.contrib.sqla import ModelView

# Flask and Flask-SQLAlchemy initialization here

admin = Admin(app)
admin.add_view(ModelView(User, db.session))
```

And this is all! It creates an admin panel for the User, with all the goodies that come with it.

However, we want to customize it and to **protect it with flask-security.**

```
from flask.ext.admin.contrib.sqla import ModelView

# Flask and Flask-SQLAlchemy initialization here

class MyView(ModelView):
    # Disable model creation
    can_create = False

    # Override displayed fields
    column_list = ('login', 'email')

    def __init__(self, session, **kwargs):
        # You can pass name and other parameters if you want to
        super(MyView, self).__init__(User, session, **kwargs)

admin = Admin(app)
admin.add_view(MyView(db.session))
```

To the MyView class we can add

```
    def is_accessible(self):
```

```
        return current_user.has_role('admin')
```

So here's the code for our app. There's no need to copy it, just read it line by line and try to understand it

```
# ------------------------- ADMIN PART ----------------------------------
class MyView(BaseView):
    @expose('/')
    def index(self):
        return self.render('admin/index.html')


class TrackingAdminView(ModelView):
    can_create = True
    def is_accessible(self):
        return current_user.has_role('end-user')
    def __init__(self, session, **kwargs):
        super(TrackingAdminView, self).__init__(Tracking, session, **kwargs)

class UserAdminView(ModelView):
    column_exclude_list = ('password')

    def is_accessible(self):
        return current_user.has_role('admin')

    def __init__(self, session, **kwargs):
        super(UserAdminView, self).__init__(User, session, **kwargs)


class RoleView(ModelView):
    def is_accessible(self):
        return current_user.has_role('admin')
    def __init__(self, session, **kwargs):
        super(RoleView, self).__init__(Role, session, **kwargs)

admin = Admin(app,  name="Flask Test Admin")
admin.add_view(TrackingAdminView(db.session))
admin.add_view(UserAdminView(db.session))
admin.add_view(RoleView(db.session))
```

---------- ADMIN PART END ----------

Chapter 10 - Prepare for the production environment

Time has come to let the world know on what we've been spending our resources...especially time.

Before we venture into spending $$$ into advertising it :) let's make sure the production release is good.

Everything I know, I share it with you. If any of you are more knowledgeable than me and you want to share this with the people that will read this book, please email me your suggestions using the http://androidadvance.com contact form.

What we plan to do:

- Time to leave sqlite and move to mysql.
- If by some mysterious force our app breaks, we want to restart automatically
- Let's serve the app from nginx with a modified tornado

In order to tune the app for performance, let's first make it do some hard work. Go into config and modify the **LISTINGS_PER_PAGE = 500**

Now for the testing let's use apache benchmark. The idea is to use it from a different machine than the one you are hosting the app. If you have 2 servers just use another one, if you want to have it on windows

Go to http://www.apachehaus.com/cgi-bin/download.plx and download **Apache 2.4.10 x64 (you have a x64 OS don't you ?). Inside /bin you find ab.exe. Run it with PowerShell or cmd just like on linux**

Testing with:

.\ab.exe -n 100 -c 50 http://server_ip:1337/ (replace .\ab.exe with ab in linux)

On an old i3 laptop, 4GB RAM, with a shitty HDD

/run.py

```
http_server = HTTPServer(WSGIContainer(app))
http_server.listen(1337)
ioloop = tornado.ioloop.IOLoop().instance()
autoreload.start(ioloop)
ioloop.start()
```

We'll call it single threaded, 1 core.

Testing on single core, sqlite database

Requests per second: **5.89** [#/sec] (mean)
Time per request: 8495.497 [ms] (mean)
Time per request: 169.910 [ms] (mean, across all concurrent requests)

Improvements list

/run.py

```
http_server = HTTPServer(WSGIContainer(app))
http_server.bind(1337)
http_server.start(0)
ioloop = tornado.ioloop.IOLoop().instance()
autoreload.start(ioloop)
ioloop.start()
```

We'll call it 4 cores tornado

Changing the config to Production, with a mysql server hosted on another machine. Make sure the Debug is set to False

4 core tornado:
Requests per second: **13.39** [#/sec] (mean)
Time per request: 3735.012 [ms] (mean)
Time per request: 74.700 [ms] (mean, across all concurrent requests)

Alright, now we're talking. With multithread and mysql the performance skyrocketed.

$sudo apt-get install nginx
nginx + 4 core tornado + mysql
Requests per second: **13.45** [#/sec] (mean)
Time per request: 3716.845 [ms] (mean)
Time per request: 74.337 [ms] (mean, across all concurrent requests)

How to configure nginx with flask.

Clean and install nginx if you already played with it.

sudo apt-get purge nginx nginx-common nginx-full

then reinstall:

sudo apt-get install nginx

Test if nginx is running. Open the ip of your server. You should see a "hello world from nginx" message.
Read this article https://www.digitalocean.com/community/tutorials/how-to-optimize-nginx-configuration

[Note: all config files are in https://github.com/AndreiD/FlaskBook last chapter]
now let's put flask in nginx

/etc/nginx/sites-enabled/default

```
server {
	listen   80 default;
	server_name  domain.com;
	server_name www.domain.com;

	access_log  /var/log/nginx/domain.com.access.log;

	root /home/your_flask_project;


	location /static/ {
		expires max;
		add_header Last-Modified $sent_http_Expires;
		alias /home/your_flask_project/app/static/;
```

```
        }


        location / {
                try_files $uri @tornado;
        }

        location @tornado {
                proxy_set_header Host $host;
                proxy_set_header X-Real-IP $remote_addr;
                proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
                proxy_pass      http://127.0.0.1:1337;
        }
}
```

Now you should change the port on your testing machine.

.\ab.exe -n 100 -c 50 http://server_ip/ (nginx server is on another port)

Want more testing ? check out http://www.slashroot.in/httperf-web-server-performance-test

Make the app autorestart on crash:

apt-get install supervisor

Supervisor:
(from the official website)

Features

Simple
Supervisor is configured through a simple INI-style config file that's easy to learn. It provides many per-process options that make your life easier like restarting failed processes and automatic log rotation.
Centralized
Supervisor provides you with one place to start, stop, and monitor your processes. Processes can be controlled individually or in groups. You can configure Supervisor to provide a local or remote command line and web interface.
Efficient
Supervisor starts its subprocesses via fork/exec and subprocesses don't daemonize. The operating system signals Supervisor immediately when a process terminates, unlike some solutions that rely on troublesome PID files and periodic polling to restart failed processes.
Extensible
Supervisor has a simple event notification protocol that programs written in any

language can use to monitor it, and an XML-RPC interface for control. It is also built with extension points that can be leveraged by Python developers.
Compatible
Supervisor works on just about everything except for Windows. It is tested and supported on Linux, Mac OS X, Solaris, and FreeBSD. It is written entirely in Python, so installation does not require a C compiler.
Proven
While Supervisor is very actively developed today, it is not new software. Supervisor has been around for years and is already in use on many servers.

Note that: Supervisor will not run at all under any version of Windows.

let's now insert our program into supervisor. Terminate the app if you have it running and

/etc/supervisor/supervisord.conf

```
; supervisor config file

[unix_http_server]
file=/var/run/supervisor.sock   ; (the path to the socket file)
chmod=0700                      ; sockef file mode (default 0700)

[supervisord]
logfile=/var/log/supervisor/supervisord.log ; (main log file;default
$CWD/supervisord.log)
logfile_maxbytes=50MB
pidfile=/var/run/supervisord.pid ; (supervisord pidfile;default supervisord.pid)
childlogdir=/var/log/supervisor            ; ('AUTO' child log dir, default $TEMP)
; the below section must remain in the config file for RPC
; (supervisorctl/web interface) to work, additional interfaces may be
; added by defining them in separate rpcinterface: sections
[rpcinterface:supervisor]
supervisor.rpcinterface_factory =
supervisor.rpcinterface:make_main_rpcinterface

[supervisorctl]
serverurl=unix:///var/run/supervisor.sock ; use a unix:// URL  for a unix socket

; The [include] section can just contain the "files" setting.  This
; setting can list multiple files (separated by whitespace or
; newlines).  It can also contain wildcards.  The filenames are
; interpreted as relative to this file.  Included files *cannot*
; include files themselves.
```

[include]
files = /etc/supervisor/conf.d/*.conf

[program:mysuperapp]
command=python /home/the_path_to_your_project/run.py
stderr_logfile = /var/log/supervisor/mysuperapp-stderr.log
stdout_logfile = /var/log/supervisor/mysuperapp-stdout.log
autostart=true
autorestart=true
stdout_logfile_maxbytes=10MB
stderr_logfile_maxbytes=10MB
startsecs=5
startretries=20

If you want, you can read more about supervisor config at
http://supervisord.org/configuration.html
You have so many users that your server can't hold them ?

- find out where's the bottleneck
- check Flask-Cache for a nice way to implement caching in your app
- split the logic into multiple servers
- ask for help on stackoverflow.com
- see http://www.maxcdn.com/
- money are no problem ? Get a 64GB / 20 CPUS 640GB SSD DISK 9TB TRANSFER MONTHLY $640.00 from digitalocean...or 2, 3…

Chapter 11 - Going online

My Method!

When it comes to buying domain names I go with namecheap
http://www.namecheap.com/?aff=64507 (please use the link if you want to say thanks for this book)
1. Buy a domain from namecheap.
2. Get a VPS from digitalocean.com http://goo.gl/ArLvyx (my referal link again and you get +$10 on your account after you make it)
3. Now in your namecheap domain manager
My Account → Manage Domains → Modify Domain

Specify Custom DNS Servers (Your own DNS Servers)
NS1.DIGITALOCEAN.COM
NS2.DIGITALOCEAN.COM
NS3.DIGITALOCEAN.COM

4. Now go to your digitalocean.com dashboard
create a new droplet (while the minimum droplet should be enough for start, get the 1Ghz one if +$5 per month is not too much for you)

Click the droplet.

Go to DNS

From the top button click “Add Domain”

Select the droplet from the left. The IP address should be added automatically

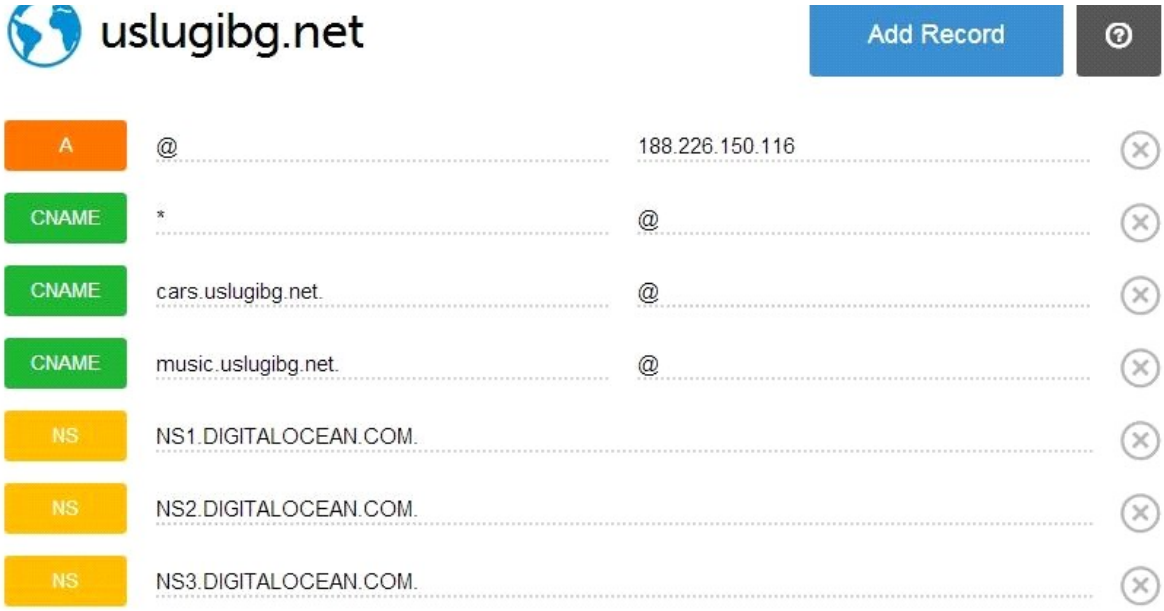
Enter the domain name (Ex: androidadvance.com)

Add an “A” record … pointing to the droplet ip
@ …….. ip_of_the_droplet

[this might be already added for you]

Add a CNAME
*..... @
[optional] Add CNAMEs for a subdomain
subdomain.domain.com……..@

Example:



Zone File

```
$TTL    1800
@               IN      SOA     NS1.DIGITALOCEAN.COM.   hostmaster.uslugibg.net. (
                        1401878999 ; last update: 2014-06-04 10:49:59 UTC
                        3600 ; refresh
                        900 ; retry
                        1209600 ; expire
                        1800 ; ttl
                        )
             IN      NS      NS1.DIGITALOCEAN.COM.
                     NS      NS2.DIGITALOCEAN.COM.
                     NS      NS3.DIGITALOCEAN.COM.
@       IN A    188.226.150.116
*       CNAME   @
cars.uslugibg.net.      CNAME   @
music.uslugibg.net.     CNAME   @
```

Now login in your droplet, install supervisor, install nginx, deploy your super flask app etc.

Chapter 15 - A list of addons that might interest you

skipping the ones that we already used.

- **Flask-Babel** - translate your website easy
- **Flask-Cache** - Adds cache support to your Flask application.
- **Flask-Login** - Flask-Login provides user session management for Flask. It handles the common tasks of logging in, logging out, and remembering your users' sessions over extended periods of time.
- **Flask-MongoAlchemy** - Add Flask support for MongoDB using MongoAlchemy
- **Flask-OAuth** - guess what it does :)
- **Flask-OpenID**
- **Flask-Restless** - provides simple generation of ReSTful APIs for database models defined using SQLAlchemy (or Flask-SQLAlchemy). The generated APIs send and receive messages in JSON format.
- **Flask-RESTful** - is an extension for Flask that adds support for quickly building REST APIs.
- **Flask-Testing** - The Flask-Testing extension provides unit testing utilities for Flask.
- **Flask-Uploads** - Flask-Uploads allows your application to flexibly and efficiently handle file uploading and serving the uploaded files.

I wish you good luck and if you have any feedback please use http://androidadvance.com contact page.

Chapter 16

This is intended to be sort of "blog posts", regarding flask, python, admin stuff that might help you on your road ahead.

Logging.

What you want to do:
Log serios errors to a file. Log debug messages on the console. Enable colors on the console so they appear pretty.

1. create a folder called "utils". inside it create an empty file called __init__.py
this is called a package. and the __init__.py tells python that there are modules to be imported from this "folder".

2. create colorstreamhandler.py

Google **mooware / colorstreamhandler.py** and copy-pate it from his **gist.**

[sidenote: Gist is a simple way to share snippets and pastes with others. All gists are Git repositories, so they are automatically versioned, forkable and usable from Git. You can create two kinds of gists: public and private.

3. create general_utils.py

```
# -*- coding: utf-8 -*-
import logging
import logging.handlers

import colorstreamhandler


LOG_FILENAME = '../LOCATION/the_log.out'
my_logger = logging.getLogger('MyLogger')
my_logger.setLevel(logging.DEBUG)

file_handler = logging.handlers.RotatingFileHandler(LOG_FILENAME,
maxBytes=10000, backupCount=0)
file_handler.setLevel(logging.ERROR)
my_logger.addHandler(file_handler)

stderr_log_handler = colorstreamhandler.ColorStreamHandler()
stderr_log_handler.setLevel(logging.NOTSET)
my_logger.addHandler(stderr_log_handler)


def cool_log(message, category="debug"):
    if category == "debug":
        my_logger.debug(message)
    if category == "info":
        my_logger.info(message)
    if category == "warning":
        my_logger.warning(message)
```

```
    if category == "error":
        my_logger.error(message)
```

[edit the LOCATION for "the_log.out"]

"# -*- coding: utf-8 -*-" is good if you work with russian,chineese characters.
"LOG_FILENAME = '../folder/the_log.out'" si where the_log.out will be
RotatingFileHandler will keep it small...so you don't end up with 200GB log files.
file_handler.setLevel(logging.ERROR) - we only log ERRORS to the file

now we add another handler and set the logging level to NOTSET so we see everything in the console.

Now we call our log with:

/some_file.py

```
#!/usr/bin/python
#-*- coding: utf-8 -*-
from utils import general_utils

general_utils.cool_log("hello from error","error")
general_utils.cool_log("hello from debug","debug")
```

you should see both messages displayed in color on your console. and just the error message logged to file.

Homework:
add time to the log in the file (hint: use "%Y-%m-%d %H:%M:%S") and level name in the console display.

[------ spoiler alert ----------]
[------ spoiler alert ----------]
[------ spoiler alert ----------]
[------ spoiler alert ----------]
[------ spoiler alert ----------]
[------ spoiler alert ----------]
[------ spoiler alert ----------]
[------ spoiler alert ----------]
[------ spoiler alert ----------]
[------ spoiler alert ----------]
[------ spoiler alert ----------]
[------ spoiler alert ----------]
[------ spoiler alert ----------]
[------ spoiler alert ----------]
[------ spoiler alert ----------]
[------ spoiler alert ----------]
[------ spoiler alert ----------]
[------ spoiler alert ----------]
[------ spoiler alert ----------]
[------ spoiler alert ----------]
[------ spoiler alert ----------]
[------ spoiler alert ----------]
[------ spoiler alert ----------]
[------ spoiler alert ----------]
[------ spoiler alert ----------]
[------ spoiler alert ----------]
[------ spoiler alert ----------]

```
file_handler.setFormatter(logging.Formatter("%(asctime)s ### %(message)s", "%Y-%m-%d %H:%M:%S"))

stderr_log_handler.setFormatter(logging.Formatter("%(levelname)s ### %(message)s", "%Y-%m-%d %H:%M:%S"))
```

Backups done easy

Option 1: write your backup scripts, add them in cron.
Option 2: use webmin

If you don't have webmin installed already, you can scroll to the begining of the book to see how it's done or google it.

After you logged in to webmin, let's configure it to be able to send us notification mails.

Webmin Configuration > Sending Email (on the bottom of the screen)
Configure "Send email using"

Note: I use sendgrid, so I put Via SMTP to remote mail server smtp.sendgrid.net port 587 and my username and password, SMTP authentication method: "Login". But if you prefer another setting please configure it.

One of the most important things to do is backup the database. Here's how:

Open MySQL Database Server on servers (or in unused modules)
enter root and password

on the bottom you have "backup databases" with all the nice options, including Scheduled backup enabled and sending mail in case the backup fails. Sweet

Problem. If you chose to make backup every day, the webmin overwrites the files. We want to create a new folder everytime a backup is done.

To enable this return to the main mysql server module. Click module config link (upper part). Set Do strftime substitution of backup destinations? YES

Now in the backup database screen write the backup folder with /home/backups/%d-%m-%Y/ (example if you want daily backups). Now each time a backup is done, it will create a new folder.

Note: in case your database is a big one, you might want to consider other methods.

Homework:

Webmin has lots of other cool things too. Check tab "Filesystem Backup" and "Scheduled Cron Jobs"