LECTURE 15

Web Servers

DEPLOYMENT

So, we've created a little web application which can let users search for information about a country they may be visiting. The steps we've taken so far:

1. Writing the backend functionality of our website – factbook_scraper.py
2. Using Flask to write our web application, which calls our factbook_scraper module.

We're going to be terrible software engineers for a moment and assume our web application is ready to be deployed.

DEPLOYMENT

Even though we've been using Flask's development server to serve our application, this should *not* be used publicly. Instead, we should set up a production environment.

We could just pick a platform-as-a-service option such as OpenShift, Gondor, CleverCloud, Heroku, etc. That way, we just need to worry about our application and not the server details.

But we're not going to do that! Let's create our own server (assuming we obtained a machine we could use as the server).

WSGI

Before we begin setting up our server, let's discuss WSGI.

WSGI is the Web Server Gateway Interface.

WSGI is a Python standard, described in PEP (Python Enhancement Proposals) 3333, that details how web servers should communicate with web applications written in Python. It is not a module or server or package. It's just a specification.

WSGI

Before WSGI, a traditional web server did not understand or have any way to run Python applications.

Eventually, an Apache module called mod_python was created to execute arbitrary Python code.

In the late 1990s and early 2000s, most Python web applications were run by Apache configured with mod_python (but also some other custom APIs for specific web servers).

But mod_python was just a small project developed by one person – eventually, a standard method for executing Python code for web applications was necessary.

WSGI

Enter WSGI!

WSGI specifies a uniform standard for invoking Python modules within web applications.

There are two sides to the WSGI specification: “server” and “application”.

- The server side provides environment information and a callback function to the application side.
- The application processes the request, and returns the response to the server side using the callback function it was provided.

If a web framework is WSGI compliant, its applications should be servable by any WSGI-compliant server. This means we can choose our framework and server independently.

WSGI

WSGI-compliant frameworks include Django, Flask, Pylons, Pyramid, etc. (so many choices!)

WSGI-compliant web servers include Gunicorn, Tornado, Waitress, etc.

We also have the option of embedding Python into a non-Python based web server by using mod_wsgi (Apache), ModPython (Apache), Nginx WSGI (Nginx), etc.

DEPLOYMENT

We're going to deploy our little website using Gunicorn fronted by Nginx.

DEPLOYMENT

Why use Nginx? The simple answer is that even though we need a server that can execute Python (WSGI server), there are features of a non-Python based server that we'd like to utilize: we can distribute requests through Nginx to multiple gunicorn instances, we can perform high-performance static file serving, efficiently handle high connection levels, etc.

It's better to use Nginx as a reverse proxy so we can take advantage of its features as a powerful HTTP server, but we're really serving dynamic content from Gunicorn.

DEPLOYMENT

An elegant pictorial representation of our setup (from realpython.com)

GETTING STARTED

The very first thing we need to do is install nginx and gunicorn of course. This class will temporarily turn into a Unix class for a minute.

```
$ sudo apt-get install nginx-full gunicorn
```

CONFIGURING NGINX

This is the easy part. Start up nginx.

```
$ sudo /etc/init.d/nginx start
```

Now, /etc/nginx/sites-enabled/ contains symlinks to configuration files in /etc/nginx/sites-available/. Configuration files in sites-available are not active until they are enabled.

We're going to remove the default symlink, create a new site configuration file for our flask application, and create a link for it in sites-enabled.

CONFIGURING NGINX

```
$ sudo rm /etc/nginx/sites-enabled/default
$ sudo touch /etc/nginx/sites-available/flask_project
$ sudo ln -s /etc/nginx/sites-available/flask_project
             /etc/nginx/sites-enabled/flask_project
```

The first two are self explanatory, but the last command creates a symbolic link between the two file arguments. Essentially, the sites-enabled file will point sites-available file.

Note: be careful when copying/pasting commands. Some characters do not transfer exactly as intended. If all else fails, write the command out manually.

CONFIGURING NGINX

Now, we can actually throw some stuff into the configuration file for our Flask project.

/etc/nginx/sites-enabled/flask_project

```
server {
    location / {
        proxy_pass http://127.0.0.1:8000;
    }
    location /static {
        alias /path/to/travel_app/app/static/;
    }
}
```

When an HTTP request hits the '/' endpoint, reverse proxy to port 8000 on localhost, where gunicorn will be listening.

CONFIGURING NGINX

Now, we can actually throw some stuff into the configuration file for our Flask project.

/etc/nginx/sites-enabled/flask_project

```
server {
    location / {
        proxy_pass http://127.0.0.1:8000;
    }
    location /static {
        alias /path/to/travel_app/app/static/;
    }
}
```

Do not route requests for static files through the WSGI server. Serve directly.

CONFIGURING NGINX

Now, we can actually throw some stuff into the configuration file for our Flask project.

/etc/nginx/sites-enabled/flask_project

```
server {
    location / {
        proxy_pass http://127.0.0.1:8000;
    }
    location /static {
        alias /path/to/travel_app/app/static/;
    }
}
```

Nginx will bind to localhost:80 by default.

CONFIGURING NGINX AND STARTING GUNICORN

And restart nginx.

```
$ sudo /etc/init.d/nginx restart
Restarting nginx: nginx.
```

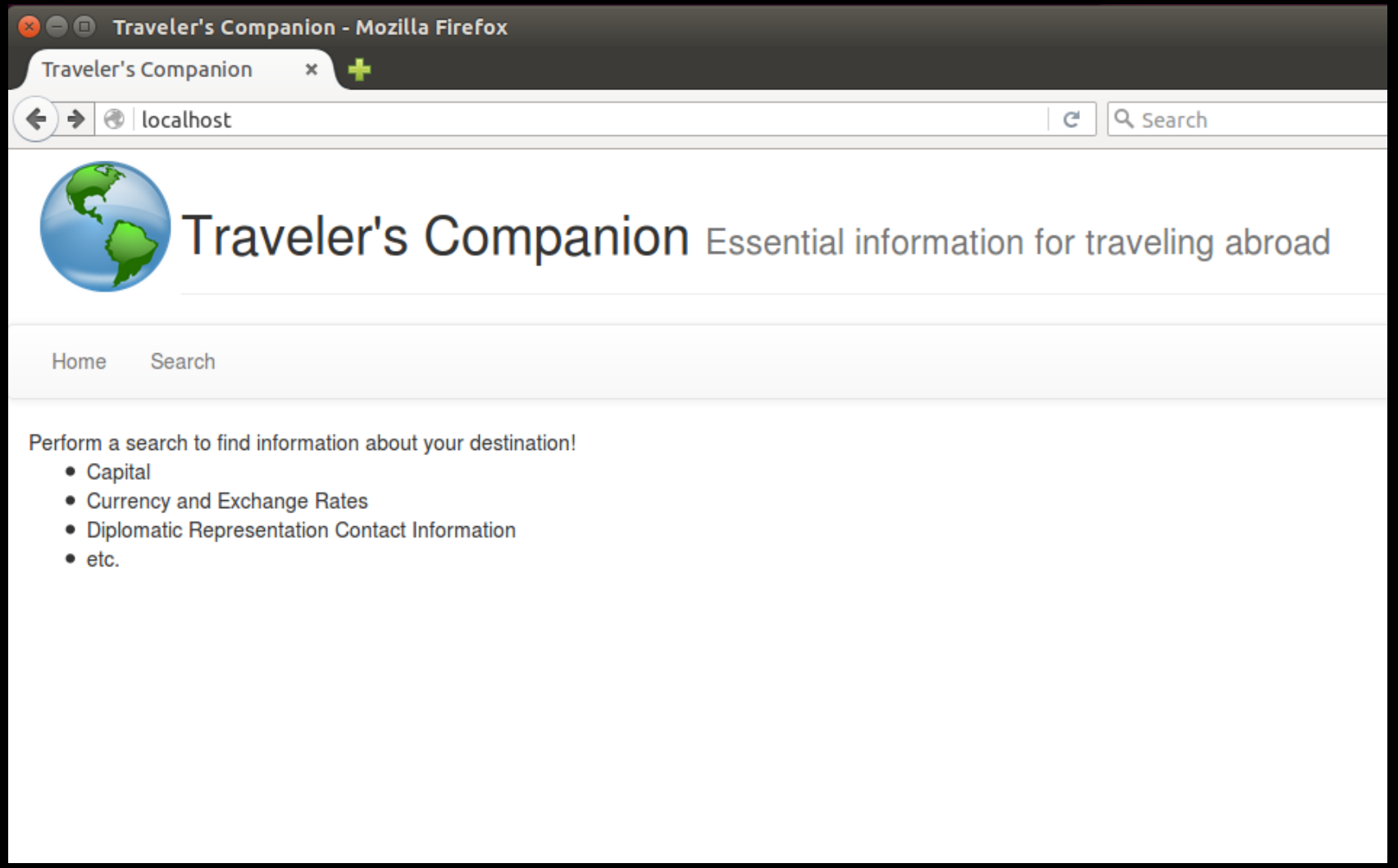
Now start up gunicorn inside of the Flask app directory.

```
travel_app$ gunicorn app:t_app -b localhost:8000
```

What we're doing is telling gunicorn to serve app.t_app on localhost, port 8000.

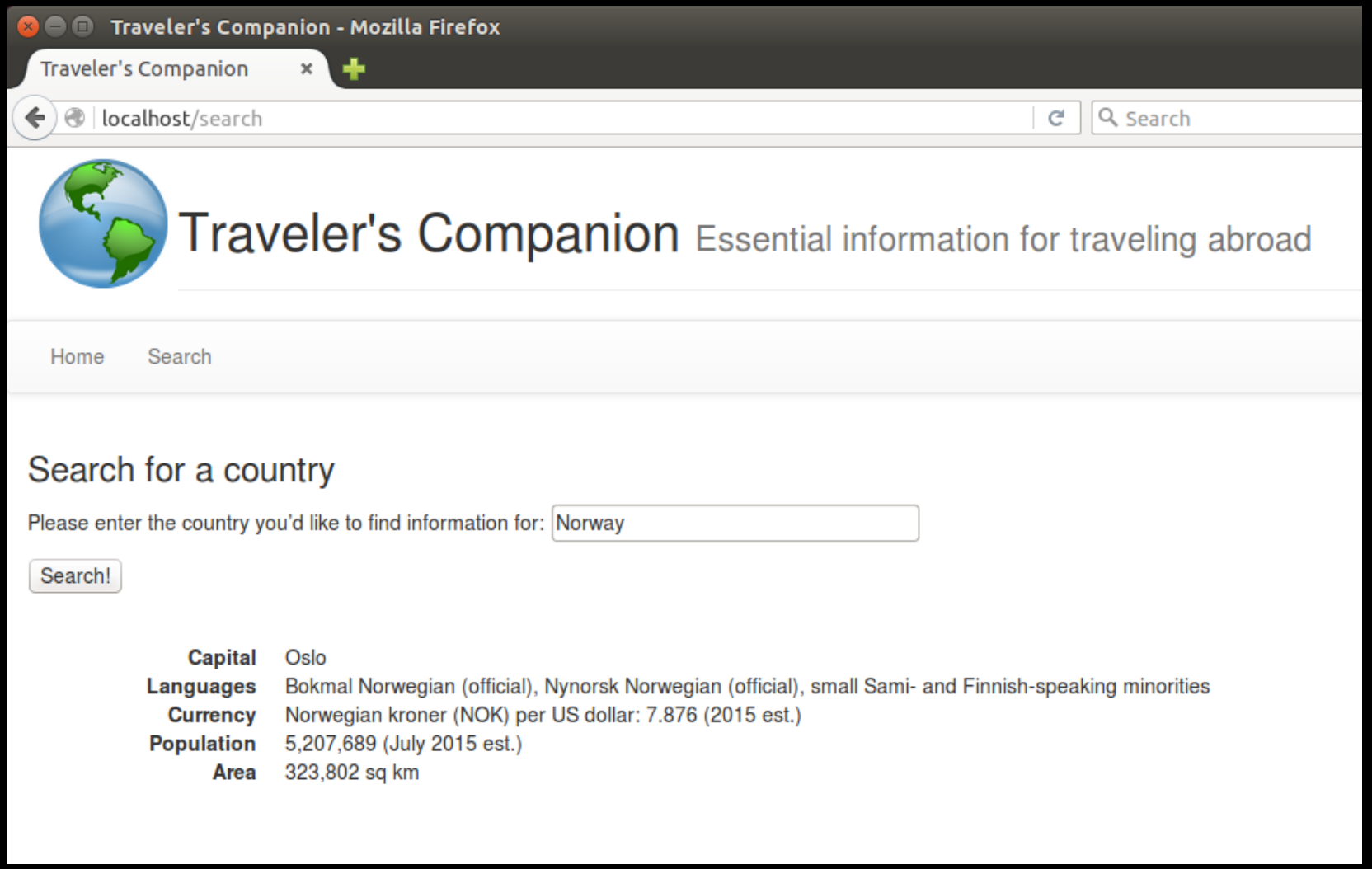
CHECKING OUT OUR SITE

Yay! Let's search for some information.



CHECKING OUT OUR SITE

More yay! Now, let's extend our website to request information by keyword, rather than country.



CHECKING OUT OUR SITE

Noooooo! What's happening!?

CONFIGURING GUNICORN

502 Bad Gateway

The nginx server was acting as a gateway or proxy and received an invalid response from the upstream server.

So, it's Gunicorn's fault. Let's check it out. Gunicorn is nice enough to shout the following at us:

```
[534] [CRITICAL] WORKER TIMEOUT (pid:539)
```

CONFIGURING GUNICORN

Bottom line: it takes too long to scrape the Factbook for every country and Gunicorn is timing out while waiting for the response to be created.

There are some solutions we can put together scraper-side (notably, threading) but let's take a step back and learn about how Gunicorn works.

GUNICORN

Gunicorn is a Python-based WSGI-compliant HTTP server for Unix (obviously).

Gunicorn is based on the pre-fork worker model. There is a central master process which manages a set of worker processes responsible for handling all requests and responses.

There's only one type of master process and it's just a reactor loop that listens for signals and responds accordingly.

There are, however, a bunch of different worker types.

GUNICORN

- Sync Workers
  - Default. Handles a single request at a time.
- Async Workers
  - Called eventlets (requires the eventlet package).
  - Cooperative multi-threading – threads explicitly yield back control after some time.
- Tornado Workers
  - Used for writing applications using the Tornado framework.

GUNICORN

Sync workers are fine for most simple web applications that don't require a lot of resources but they're easy to overwhelm, that's why we use Nginx upfront.

Async workers are required for resource- and time-intensive applications.

In our case, our application makes a long blocking call to scrape the Factbook.

So let's change our Gunicorn configuration. Previously, we just used the default config but now we'll make our own.

CONFIGURING GUNICORN

Let's create a config file which can be passed to the gunicorn command at start-up.

travel_app/gunicorn/gunicorn.py

```
import os

bind = "127.0.0.1:8000"
workers = 2
worker_class = 'eventlet'
logfile = "logs/error.log"
loglevel = 'debug'
timeout=60
```

Async Workers

According to the Gunicorn docs, 4-12 workers should be able to handle hundreds of thousands of requests per second.

SERVING THE APPLICATION

Now, we can issue the following to start up the Gunicorn server:

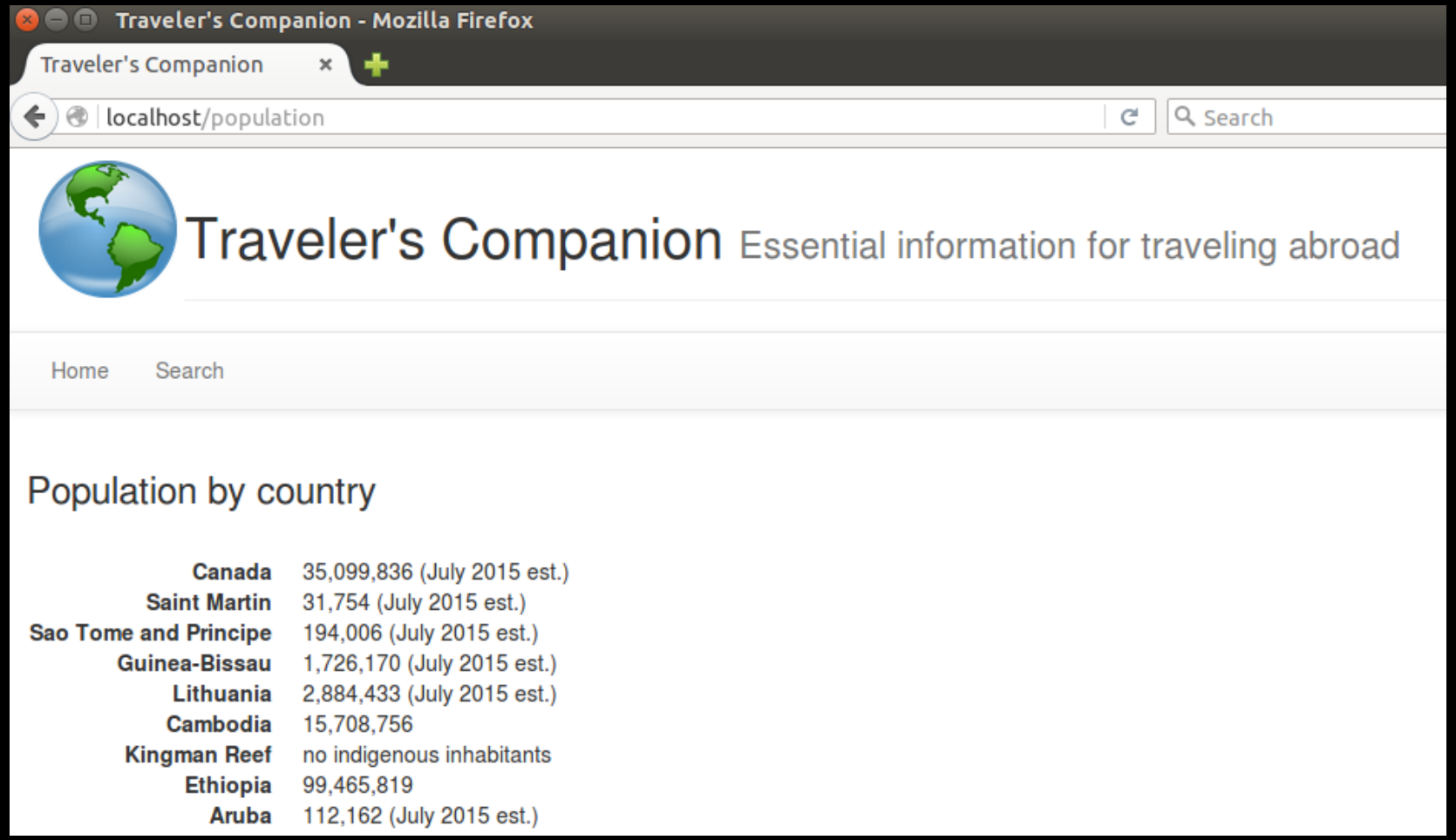
```
$ gunicorn app:t_app -c gunicorn/gunicorn.py
```

Note on requirements: to use async workers, you may need to install python-dev, greenlet and eventlet.

```
$ sudo apt-get install python-dev
$ pip install greenlet
$ pip install eventlet
```

CHECKING OUT OUR SITE...AGAIN

Yay!!!



CONFIGURATION OPTIONS

So that's basic deployment with Gunicorn and Nginx. Some other considerations:

- Explore the Gunicorn and nginx config file settings to see which setup options are available to you.
- Create a supervisor so that you don't have to manually restart your server when changes are made (http://supervisord.org/introduction.html).
- Create git/mercurial hooks to automate application updates.
- Create a fabfile with Fabric to automate the deployment process.