# Introduction to web development with Python and Django Documentation

***Release 0.1***

**Greg Loyse**

**Apr 23, 2017**

# Contents

Contents:

CHAPTER 1

# Introduction

There are a few things we need to explain before getting stuck in.

We focus on the overall picture. To do this we use a few analogies not to be taken too literally.

## The Internet

The internet is a network of computers. Its goal is to enable communication between them.

A network is composed of nodes and edges. Visually it is a set of dots and connections. The London tube map is an example.

Your family, friends, colleagues, and acquaintances can be thought of as a network of people. (This is how social networks model our relationships.)

To communicate we must have a means by which our messages reach the intended destination.

On the one hand we need something physical to connect the computers. These are the wires.

On the other hand we need some conventions (software) to ensure messages reach their destinations.

One way this is done over the internet is called TCP/IP.

TCP ensures the messages arrive safely with nothing missing. Every computer has an IP which is a unique address.

You can think of TCP as an envelope and IP as the address on it.

## Http and the Request / Response cycle

To communicate effectively the elements of a network need to agree on some protocol. That protocol for humans can be english but there are other ‘protocols’, chinese for example.

Many computers on the internet use Http to communicate.

Every time you click on a link, or type a url and enter into a browser, you are making what is called an http GET request.

Here is an example that uses curl from the command line as a client:

```
$ curl -sv www.example.com -o /dev/null
* About to connect() to www.example.com port 80 (#0)
*   Trying 93.184.216.119...
* Connected to www.example.com (93.184.216.119) port 80 (#0)
> GET / HTTP/1.1
> User-Agent: curl/7.30.0
> Host: www.example.com
> Accept: */*
>
< HTTP/1.1 200 OK
< Accept-Ranges: bytes
< Cache-Control: max-age=604800
< Content-Type: text/html
< Date: Thu, 21 Aug 2014 12:09:46 GMT
< Etag: "359670651"
< Expires: Thu, 28 Aug 2014 12:09:46 GMT
< Last-Modified: Fri, 09 Aug 2013 23:54:35 GMT
< Server: ECS (iad/182A)
< Content-Length: 1270
<
< <!doctype html>
< <html>
< <head>
<     <title>Example Domain</title>
< </head>
< <body>
< <div>
<     <h1>Example Domain</h1>
<     <p>This domain is established to be used for illustrative examples in documents.
↪</p>
< </div>
< </body>
< </html>
```

Note this has been abridged.

The lines starting with:

- '*' is information from the curl program.
- '>' is the http request text that curl is sending.
- '<' is the http response text that curl received.

Note that the response includes the html page that will be rendered in a browser.

Tip:

> Http is just text. We send text requests, we recieve text responses. All complex pretty pages in the browser are created from these text responses.

## The Client Server Architecture

In software development an architecture is a way of organising code you see time and time again. Its also called a pattern. Similar perhaps to how journalists follow a pattern when structuring their articles.

Think about the meaning of the words.

A browser is a great example of a client. It sends http requests to a server. A server returns an http response, which the browser then renders as a web page.

We will see other examples of a client - server architecture when we introduce using databases.

## HTML

Browsers understand how to render HTML.

HTML is a way to structure text.

```
<!doctype html>
<html>
<head>
    <title>Example Domain</title>
</head>
<body>
<div>
    <h1>A Header</h1>
    <p>Here is some text between p elements</p>
</div>
</body>
</html>
```

Note it consists of elements like this: *<el>content<el>*

We won't delve any deeper than this as we don't need to.

## Databases

Data, or information, needs to be stored somewhere.

Typically we save data in files.

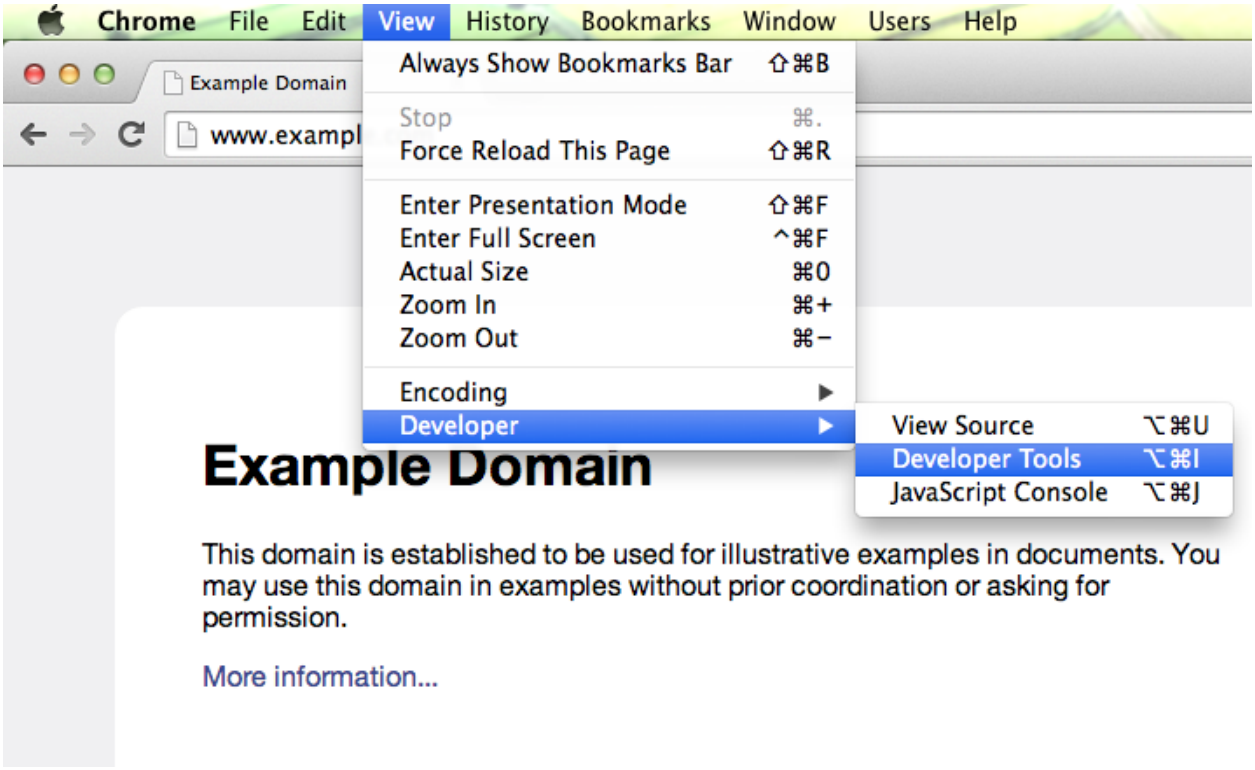
Databases are another way of saving data which has some advantages over plain files.

Web applications often save data in databases rather than files.

You can think of a database much as you would spreadsheet software. It stores information in a collection of tables.

## Exercise

Using Chrome, open developer tools: view/Developer/DeveloperTools

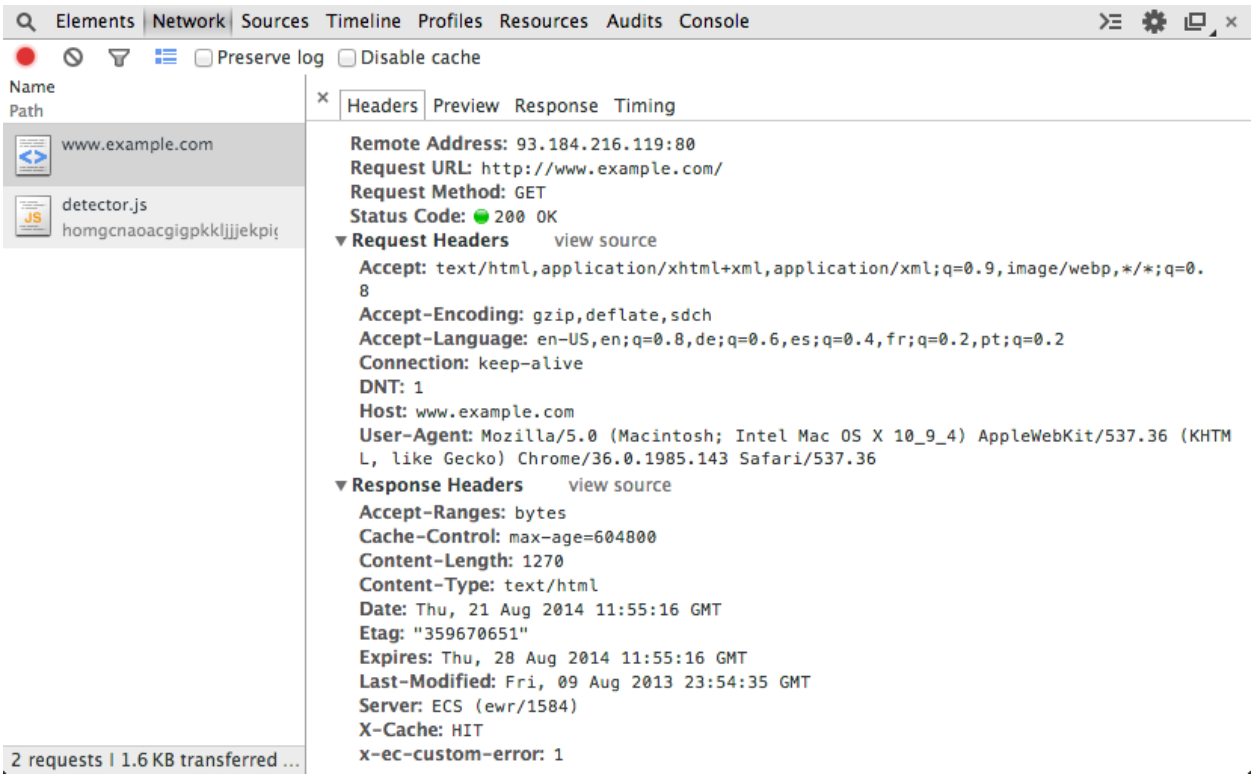



A tab will pop up. Click on the Network tab.

Now type a URL (web address) that is familiar to you.

Inspect the http GET request.

Here we try with *www.example.com*:




Note we have same information we found with *curl* above. It is presented in a more user friendly way however.

Explore one of your favourite websites using the developer tools to inspect what is going on at the http network level.

## Take Away

All internet experiences, online shopping, news, videos, sending texts... boil down to computers sending messages much like what we have described above.

Http is not the only protocol in town, but the concept of computers acting as clients and servers communicating by sending requests and responses is almost universal.

CHAPTER 2

# Setup

## Project folder

Lets create a project directory:

```
mkdir website
cd website
```

## Installing Django

Pip is a way to install python code. Python code is installed as a package.

To list all currently installed python packages:

```
$ pip freeze
```

To install a Django:

```
$ pip install django
```

## Creating Django project

We use a script supplied by django to set up a new project:

```
$ django-admin.py startproject website
```

You should see this folder structure and files generated:

```
website
- manage.py
- website
    - __init__.py
    - settings.py
    - urls.py
    - wsgi.py
```

The important files are *manage.py*, *settings.py*, and *urls.py*.

## settings.py

A lot of configuration is needed to setup a web application.

*website/settings.py* contains a lot of names that define all the configuration for our website. All the defaults are good for now.

Note the INSTALLED_APPS name is defined as a tuple of strings. We will be adding to that tuple shortly.

Note also the DATABASES name is defined as a dictionary.

## Creating the Database

Notice that the current directory doesn't include a db.sqlite3 file.

Django like all web frameworks stores its data in a database. Lets create that database now:

```
python manage.py syncdb
```

You will see some output such as: *Creating table auth_user*

```
(django) website $ ./manage.py syncdb
Creating tables ...
Creating table django_admin_log
Creating table auth_permission
Creating table auth_group_permissions
Creating table auth_group
Creating table auth_user_groups
Creating table auth_user_user_permissions
Creating table auth_user
Creating table django_content_type
Creating table django_session

You just installed Django's auth system, which means you don't have any superusers␣
↪defined.
Would you like to create one now? (yes/no): yes
Username (leave blank to use 'greg'):
Email address:
Password:
Password (again):
Superuser created successfully.
Installing custom SQL ...
Installing indexes ...
Installed 0 object(s) from 0 fixture(s)
```

Now the top level folder website contains a file called *db.sqlite3*. This is your database.

## Inspecting the Database

Download sqlite3 from *www.sqlite.org/download.html*. Choose the sqlite-shell-win32-x86-....zip file. Unzip it by double clicking it. Then drag and drop into C:BOOTCAMPPython34The last step is to add it to a directory on the PATH.

A database application is like a server.

We send requests using clients. The clients in this case aren't the browser but typically programs such as our python website.

We will use another server to independently inspect our database.

You launch the client by typing:

```
sqlite3 db.sqlite3
```

The *sqlite3* program provides a new type of shell which is meant for inspecting our database.

Here is an example interaction:

```
(django)  website  sqlite3 db.sqlite3
SQLite version 3.7.13 2012-07-17 17:46:21
Enter ".help" for instructions
Enter SQL statements terminated with a ";"
sqlite> .tables
auth_group                  auth_user_user_permissions
auth_group_permissions      django_admin_log
auth_permission             django_content_type
auth_user                   django_session
auth_user_groups
sqlite> select * from auth_user;
1|pbkdf2_sha256$12000$YqWBCAkWemZC$+hazwa/dPJNczpPitJ2J0KR8UuAX11txLlSkrtAXk5k=|2014-
↪08-21 14:59:05.171913|1|greg|||1|1|2014-08-21 14:59:05.171913
sqlite>
```

The *.tables* command lists all the tables that exist in the database. We recognise these as being the same that were created earlier by running the *.manage.py syncdb* command.

The *select * from auth_user;* is SQL. SQL is a language dedicated to programming databases. This command means give me everything in the *auth_user* table.

Type:

```
sqlite3> .quit
```

To exit.

## Running the server

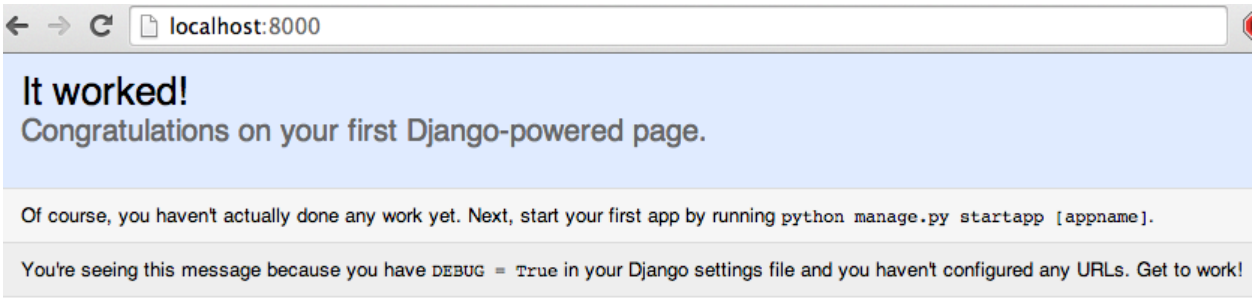
You run the server with:

```
./manage.py runserver
```

Now you can send http requests using your browser as client. Enter:

```
http://127.0.0.:8000/
```

You should see:




You can quit the server at any point by pressing together *cntrl* + *c*

## Creating & installing the Blog App

Tip:

> Django like any framwork, provides a way of organising your code. It provides in effect a proven architecture which you learn to work within.
>
> A good webframework makes a lot of decisions for you. You build on the combined experience of the developpers who created it.

Django introduces the concept of an app as a way to organise code.

Our *Blog* will be an app. We create it thusly:

```
./manage.py startapp blog
```

We now have a foler directory generated looking like:

```
- blog
|    - __init__.py
|    - admin.py
|    - models.py
|    - tests.py
|    - views.py
- db.sqlite3
- manage.py
- website
     - __init__.py
     - settings.py
     - urls.py
     - wsgi.py
```

We now need to tell our website about the *blog* apps' existence. We do this by adding it to the INSTALLED_APPS tuple.

```
INSTALLED_APPS = (
    'django.contrib.admin',
    'django.contrib.auth',
    'django.contrib.contenttypes',
```

```
    'django.contrib.sessions',
    'django.contrib.messages',
    'django.contrib.staticfiles',
    'website',
    'blog',
)
```

CHAPTER 3

# Creating Web Services

We will start by programming the server to return a responses to an http GET request.

We will always need to do two things:

- map a url to a view function
- define the view function

## *website/urls.py*

This file matches urls to view functions.

When the django server receives a url. It searches in this file for one that matches. If it matches it executes the mapped function. If it doesn't find anything you get a 404 - page not found error.

## Saying hello

Django provides us with what it calls view functions.

These are orgindary pythong functions, but they take a request object and they response with a string or what is called an *HTTPResponse* object.

In your blog app, open the *views.py* file.

Add this to it:

```
from django.http import HttpResponse

def hello(request):
    return HttpResponse('hello')
```

Now we need to configure our website with which request will trigger this view function. We do this by adding a line to *website/urls.py*:

```
urlpatterns = patterns('',
    url(r'^hello$', 'blog.views.hello'),
    url(r'^admin/', include(admin.site.urls)),
)
```

In our browser, *http://localhost:8000* responds with 'hello'.

We have responded to a GET request.

We will often follow this pattern of creating a view function and hooking it up to a url.

## GET parameters

http GET requests can pass parameters in the URL.

Here is an example:

```
http://localhost:8000/whoami/?name=greg&sex=male
```

The parameter section is defined by ? followed by & separated keys and values.

Here we have the parameters: - name, equal to greg - sex, equal to male

As usual we need to do two things create a view function and hook it up in *website/urls.py*

First the view function:

```
def whoami(request):

    sex = request.GET['sex']
    name = request.GET['name']

    response = 'You are ' + name + ' and of sex ' + sex

    return HttpResponse(response)
```

Note that we can extract anything passed in the url after the *?* character using the request.GET dictionary.

Now *website/urls.py*:

```
urlpatterns = patterns('',
    url(r'^$', 'blog.views.hello'),
    url(r'^time$', 'blog.views.time'),
    url(r'^whoami/$', 'blog.views.whoami'),
    url(r'^admin/', include(admin.site.urls)),
)
```

You should now get as a response: *You are greg and of sex male*

## Exercises

### A clock service

You can get an exact time by doing the following:

```
>>> import datetime
>>> datetime.datetime.now()
```

Program your server to response the time when it recieves an http GET request to this url:

```
http://localhost:8000/time
```

You will need to create a view function in *blog/views.py*, and hook it up to a url in *website/urls.py*.

## Body Mass Index Service

You have just been contracted by the NHS to provide a service that calculates the BMI. Both other websites and mobile apps will be using your service.

The endpoint (url) will respond successfully to the following type of url:

```
bmi/?mass=75&height=182
```

Look up the BMI equation on wikipedia, and write a bmi view function and hook it up to the website urls.

You may have to revisit the notion of type in Python. Remember there is a difference between *'5'* and *5*.

To transform a number as a string into a number you can cast it using either int() or float():

```
>>> float('5')
5.0
>>> int('5')
5
```

## Your Serivce

By now you have discovered that you can trigger any type of programming sending ba GET request to your server. You simply hook up a url to a view function.

Come up with something that is useful to you!

Anything that involves simple maths is easily explored.

Solutions:

> You can find some suggestions by adding _solutions to the above url.

CHAPTER 4

# Resources

For more use the following:

- The Django Girls tutorial For begginers. Publish a blog in a day!
- The Official Django tutorial is essential.
- Test Driven Development with Python teaches the tools and best practices followed by web professionals.