Hands On Django

## Installation:

1. Install Python.

Being a Python Web framework, Django requires Python . See [What Python version can I use with Django?](https://docs.djangoproject.com/en/1.10/faq/install/#faq-python-version-support) for details. Get the latest version of Python at <https://www.python.org/download/> or with your operating system’s package manager.

Here we are using python 2.7 which is compatible to django 1.10

1. Install PIP

Command : pip install -U pip

Install [pip](https://pip.pypa.io/). The easiest is to use the [standalone pip installer](https://pip.pypa.io/en/latest/installing/#installing-with-get-pip-py). If your distribution already has pip installed, you might need to update it if it’s outdated. If it’s outdated, you’ll know because installation won’t work.

1. Install a Python Virtual Environment.

Take a look at [virtualenv](https://virtualenv.pypa.io/) and [virtualenvwrapper](https://virtualenvwrapper.readthedocs.io/en/latest/). These tools provide isolated Python environments, which are more practical than installing packages systemwide. They also allow installing packages without administrator privileges. The [contributing tutorial](https://docs.djangoproject.com/en/1.10/intro/contributing/) walks through how to create a virtualenv on Python 3.

1. Install Django.

After you’ve created and activated a virtual environment, enter the command **pip install**

## Starting a Project

Once you’ve installed Python, Django and (optionally) your database server/library, you can take the first step in developing a Django application by creating a *project*.

**A project is a collection of settings for an instance of Django.**

you’ll need to auto-generate some code that establishes a Django [project](https://docs.djangoproject.com/en/1.10/glossary/#term-project) – a collection of settings for an instance of Django, including database configuration, Django-specific options and application-specific settings.

From the command line, cd into a directory where you’d like to store your code, then run the following command:

$ django-admin startproject mysite

This will create a mysite directory in your current directory.

Let’s look at what [startproject](https://docs.djangoproject.com/en/1.10/ref/django-admin/#django-admin-startproject) created:

mysite/  
 manage.py  
 mysite/  
 \_\_init\_\_.py  
 settings.py  
 urls.py  
 wsgi.py

These files are:

* The outer **mysite/** root directory is just a container for your project. Its name doesn’t matter to Django; you can rename it to anything you like.
* **manage.py:** A command-line utility that lets you interact with this Django project in various ways. You can read all the details about manage.py in [django-admin and manage.py](https://docs.djangoproject.com/en/1.10/ref/django-admin/).
* The inner mysite/ directory is the actual Python package for your project. Its name is the Python package name you’ll need to use to import anything inside it (e.g. mysite.urls).
* **mysite/\_\_init\_\_.py:** An empty file that tells Python that this directory should be considered a Python package. If you’re a Python beginner, read [more about packages](https://docs.python.org/3/tutorial/modules.html#tut-packages) in the official Python docs.
* **mysite/settings.py:** Settings/configuration for this Django project. [Django settings](https://docs.djangoproject.com/en/1.10/topics/settings/) will tell you all about how settings work.
* **mysite/urls.py:** The URL declarations for this Django project; a “table of contents” of your Django-powered site. You can read more about URLs in [URL dispatcher](https://docs.djangoproject.com/en/1.10/topics/http/urls/).
* **mysite/wsgi.py**: An entry-point for WSGI-compatible web servers to serve your project. See [How to deploy with WSGI](https://docs.djangoproject.com/en/1.10/howto/deployment/wsgi/) for more details.

## The development server

Let’s verify your Django project works. Change into the outer mysite directory, if you haven’t already, and run the following commands:

$ python manage.py runserver

You’ll see the following output on the command line:

Performing system checks...  
  
System check identified no issues (0 silenced).  
  
You have unapplied migrations; your app may not work properly until they are applied.  
Run 'python manage.py migrate' to apply them.  
  
December 27, 2016 - 15:50:53  
Django version 1.10, using settings 'mysite.settings'  
Starting development server at<http://127.0.0.1:8000/>  
Quit the server with CONTROL-C.

Steps to setup and work with Oracle DB are mention [here](https://docs.google.com/document/d/1qltkDXzqkIVK7PLk1axz6Vn7sNBykvJVQYutKrmU34E/edit?ts=5861163c):

## Creating the raxak app

Goto mysite project folder and in settings.py, change the root\_urlconf to ‘mysite.urls’ (If it’s already inplace ignore this)

ROOT\_URLCONF = 'mysite.urls'

Run the following command to create a raxak app.

$ python manage.py startapp raxak

That’ll create a directory raxak, which is laid out like this:

raxak/  
 \_\_init\_\_.py  
 admin.py  
 apps.py  
 migrations/  
 \_\_init\_\_.py  
 models.py  
 tests.py  
 views.py

This directory structure will house the raxak application.

from django.http import HttpResponse

def index(request):

return HttpResponse("Hey there! Welcome to Raxak app!")

The next step is to point the root URLconf at the polls.urls module. In mysite/urls.py, add an import for django.conf.urls.include and insert an [include()](https://docs.djangoproject.com/en/1.10/ref/urls/#django.conf.urls.include) in the urlpatterns list, so you have:

mysite/urls.py

from django.conf.urls import include, url

from django.contrib import admin

urlpatterns = [

url(r'^raxak/', include('raxak.urls')),

url(r'^admin/', admin.site.urls),

]

The [include()](https://docs.djangoproject.com/en/1.10/ref/urls/#django.conf.urls.include) function allows referencing other URLconfs. Note that the regular expressions for the [include()](https://docs.djangoproject.com/en/1.10/ref/urls/#django.conf.urls.include) function doesn’t have a $ (end-of-string match character) but rather a trailing slash. Whenever Django encounters [include()](https://docs.djangoproject.com/en/1.10/ref/urls/#django.conf.urls.include), it chops off whatever part of the URL matched up to that point and sends the remaining string to the included URLconf for further processing.

The idea behind [include()](https://docs.djangoproject.com/en/1.10/ref/urls/#django.conf.urls.include) is to make it easy to plug-and-play URLs. Since raxak are in their own URLconf (raxak/urls.py), they can be placed under “/raxak//”, or under “/fun\_raxak/” or any other path root, and the app will still work.

You have now wired an index view into the URLconf. Lets verify it’s working, run the following command:

$ python manage.py runserver

Go to<http://localhost:8000/raxak/>

in your browser, and you should see the text “Hey there! Welcome to Raxak app!”, which you defined in the index view.

## 

## 

## Views:

A view is a “type” of Web page in your Django application that generally serves a specific function and has a specific template. For example, in a blog application, you might have the following views:

**Class based views**

Class-based views provide an alternative way to implement views as Python objects instead of functions. They do not replace function-based views, but have certain differences and advantages when compared to function-based views:

* Organization of code related to specific HTTP methods (GET, POST, etc.) can be addressed by separate methods instead of conditional branching.
* Object oriented techniques such as mixins (multiple inheritance) can be used to factor code into reusable components.

Class-based generic views were created with the same objective as function-based generic views, to make view development easier. However, the way the solution is implemented, through the use of mixins, provides a toolkit that results in class-based generic views being more extensible and flexible than their function-based counterparts.

**1) Using class-based Views:**

A class-based view allows you to respond to different HTTP request methods with different class instance methods, instead of with conditionally branching code inside a single view function.

So where the code to handle HTTP GET in a view function would look something like:

*from django.http import HttpResponse  
def my\_view(request):  
 if request.method == 'GET':  
 # <view logic>  
 return HttpResponse('result')*

**In Class-based Views, This would be like:**

from django.http import HttpResponse

from django.views.generic import View

class MyView(View):

def get(self, request):

return HttpResponse('result!! i m here')

Because Django’s URL resolver expects to send the request and associated arguments to a callable function, not a class, class-based views have an [as\_view()](https://docs.djangoproject.com/en/1.10/ref/class-based-views/base/#django.views.generic.base.View.as_view) class method which returns a function that can be called when a request arrives for a **URL matching** the associated pattern.

from django.conf.urls import \*

from raxak.views import MyView

urlpatterns = [

url(r'^about/', MyView.as\_view()),

]

**2) Subclassing Generic Views**

The standard Python way of subclassing and overriding attributes and methods in the subclass. So that if your parent class had an attribute greeting like this:

*from django.http import HttpResponse  
from django.views import View  
class GreetingView(View):  
 greeting = "Good Day"  
  
 def get(self, request):  
 return HttpResponse(self.greeting)*

**You can that in subclass:**

*class MorningGreetingView(GreetingView):  
 greeting = "Morning to ya"*

**Another option is to configure class attributes as keyword arguments to the as\_view() call in the URLconf:**

1)urlpatterns = [

url(r'^about/$', GreetingView.as\_view(greeting="G day")),

Output: G day

2)urlpatterns = [

url(r'^about/$', GreetingView.as\_view()),

Output: *Good Day*

**3) Built-in class-based generic views[¶](https://docs.djangoproject.com/en/1.10/topics/class-based-views/generic-display/#built-in-class-based-generic-views)**

**a) TemplateView:**

The second, more powerful way to use generic views is to inherit from an existing view and override attributes (such as the template\_name) or methods (such as get\_context\_data) in your subclass to provide new values or methods. Consider, for example, a view that just displays one template, about.html. Django has a generic view to do this - [TemplateView](https://docs.djangoproject.com/en/1.10/ref/class-based-views/base/#django.views.generic.base.TemplateView) - so we can just subclass it, and override the template name:

#views.py

*from django.views.generic import TemplateView  
class AboutView(TemplateView):  
 template\_name = "about.html"*

Then we just need to add this new view into our URLconf. TemplateView is a class, not a function, so we point the URL to the as\_view() class method instead, which provides a function-like entry to class-based views:

# urls.py

*from django.conf.urls import url  
from some\_app.views import AboutView  
  
urlpatterns = [  
 url(r'^about/$', AboutView.as\_view()),  
]*

**b) ListView**

* [ListView](https://docs.djangoproject.com/en/1.10/topics/class-based-views/generic-display/) reference Link

**c) Form handling with class-based views**

Django provides a collection of generic class-based views for form processing.

* [Model Forms](https://docs.djangoproject.com/en/1.10/topics/class-based-views/generic-editing/) reference link

**d) Using mixins with class-based views**

* [Mixins](https://docs.djangoproject.com/en/1.10/topics/class-based-views/mixins/) reference link

**Advantages;**

The single most significant advantage i**s inheritance**. On a large project it's likely that you will have lots of similar views. Rather than write the same code again and again, you can simply have your views inherit from a base view.

Also django ships with a collection of generic view classes that can be used to do some of the most common tasks. For example the **DetailView class** is used to pass a single object from one of your models, render it with a template and return the http response. You can plug it straight into your url conf..

***url(r'^author/(?P<pk>\d+)/$', DetailView.as\_view(model=Author)),***

**Django Forms**

Generally, there are two parts to developing a form: the HTML user interface and the backend view code that processes the submitted data.

**Form** [**Ref-1**](https://docs.djangoproject.com/en/1.10/intro/tutorial04/)[**Ref-2**](http://djangobook.com/django-forms/)[**Ref-3**](https://www.tutorialspoint.com/django/django_form_processing.htm)

**Template language**

A template contains the static parts of the desired HTML output as well as some special syntax describing how dynamic content will be inserted.

Django’s template engine provides a powerful mini-language for defining the user-facing layer of your application, encouraging a clean separation of application and presentation logic.

[Reference](https://docs.djangoproject.com/en/1.10/ref/templates/) Link:

* Template Tags and Filters
  + [Reference](https://docs.djangoproject.com/en/1.10/ref/templates/builtins/#ref-templates-builtins-filters) Link
* Template Inheritance
  + [Reference](https://docs.djangoproject.com/en/1.10/ref/templates/language/#template-inheritance) Link

**Managing and deploying static files**

Websites generally need to serve additional files such as images, JavaScript, or CSS. In Django, we refer to these files as “static files”. Django provides django.contrib.staticfiles to help you manage them.

* Managing Static files (eg images, javascript, CSS)

## [**Configuring static files**](https://docs.djangoproject.com/en/1.10/howto/static-files/) **during development**

## [**Deploying static file**](https://docs.djangoproject.com/en/1.10/howto/static-files/deployment/) **in production**

* The [staticfiles app](https://docs.djangoproject.com/en/1.10/ref/contrib/staticfiles/) details and settings.

**Decorators - In built and custom**

**Decorators** provide a simple syntax for calling higher-order functions. By definition, a **decorator** is a function that takes another function and extends the behavior of the latter function without explicitly modifying it.

Django provides several decorators that can be applied to views to support various HTTP features.

@decorator\_func

def func():

Is same as following

func() = decorator\_func(func)

[Reference](https://docs.djangoproject.com/en/1.10/topics/http/decorators/) Link

[Cache decorators](https://docs.djangoproject.com/en/1.10/topics/cache/#the-per-view-cache)

**Managers - In built and custom**

A Manager (an instance of django.db.models.manager.Manager) is described as "the interface through which database query operations are provided to Django models." A model's Manager is the gateway to table-level functionality in the ORM (model instances generally give you row-level functionality). Every model class is given a default manager, called **objects.**

For example:

from django.db import models  
  
*class Person(models.Model):  
 #...  
 people = models.Manager()*

*Person.people.all() will provide a list of all Person objects.*

## **Custom managers**[**¶**](https://docs.djangoproject.com/en/1.10/topics/db/managers/#custom-managers)

### **Default managers**[**¶**](https://docs.djangoproject.com/en/1.10/topics/db/managers/#default-managers)

### **Base managers**[**¶**](https://docs.djangoproject.com/en/1.10/topics/db/managers/#base-managers)

[Managers](https://docs.djangoproject.com/en/1.10/topics/db/managers/) details reference link

[Managers examples](https://www.dabapps.com/blog/higher-level-query-api-django-orm/)  reference link

**Caching**

For medium- to high-traffic sites, it’s essential to cut as much overhead as possible.

That’s where caching comes in. To cache something is to save the result of an expensive calculation so that you don’t have to perform the calculation next time.

[Reference](https://docs.djangoproject.com/en/1.10/topics/cache/) Link

[Memcached](http://memcached.org/) reference link

[Database caching](https://docs.djangoproject.com/en/1.10/topics/cache/#database-caching) reference link

**Filters - in built and custom**

[Template filters](https://docs.djangoproject.com/en/1.10/howto/custom-template-tags/#writing-custom-template-filters) link

**How to use session**

For security reasons, Django has a session framework for cookies handling. Sessions are used to abstract the receiving and sending of cookies, data is saved on server side (like in database), and the client side cookie just has a session ID for identification. Sessions are also useful to avoid cases where the user browser is set to ‘not accept’ cookies.

## **Setting Up Sessions**

In Django, enabling session is done in your project **settings.py**, by adding some lines to

the **MIDDLEWARE\_CLASSES** and the **INSTALLED\_APPS** options. This should be done

while creating the project, but it's always good to know, so **MIDDLEWARE\_CLASSES**

should have −

***'django.contrib.sessions.middleware.SessionMiddleware'***

And **INSTALLED\_APPS** should have −

***'django.contrib.sessions'***

By default, Django saves session information in database (django\_session table or collection), but you can configure the engine to store information using other ways like: in **file** or in **cache**.

Once you have configured your installation, **run manage.py migrate t**o install the single database table that stores session data.

## **Configuring the session engine**[**¶**](https://docs.djangoproject.com/en/1.10/topics/http/sessions/#configuring-the-session-engine)

### **Using database-backed sessions**[**¶**](https://docs.djangoproject.com/en/1.10/topics/http/sessions/#using-database-backed-sessions)

Add ‘django.contrib.sessions‘ to INSTALLED\_APPS

### **Using cached sessions**[**¶**](https://docs.djangoproject.com/en/1.10/topics/http/sessions/#using-cached-sessions)

Add ‘django.contrib.sessions.backends.cache‘ to INSTALLED\_APPS for simple

caching store

For persistent, cached data, add “django.contrib.sessions.backends.cached\_db"

### **Using file-based sessions**[**¶**](https://docs.djangoproject.com/en/1.10/topics/http/sessions/#using-file-based-sessions)

Add ‘django.contrib.sessions.backends.file‘ to INSTALLED\_APPS

### **Using cookie-based sessions**[**¶**](https://docs.djangoproject.com/en/1.10/topics/http/sessions/#using-cookie-based-sessions)

Add ‘django.contrib.sessions.backends.signed\_cookies’ to INSTALLED\_APPS

## **Using sessions in views[¶](https://docs.djangoproject.com/en/1.10/topics/http/sessions/#using-sessions-in-views)**

* [Session](https://docs.djangoproject.com/en/1.10/topics/http/sessions/) Reference Link

**Middleware**

Middleware is a framework of hooks into Django’s request/response processing. It’s a light, low-level “plugin” system for globally altering Django’s input or output.

Each middleware component is responsible for doing some specific function. For example, Django includes a middleware component, [AuthenticationMiddleware](https://docs.djangoproject.com/en/1.10/ref/middleware/#django.contrib.auth.middleware.AuthenticationMiddleware), that associates users with requests using sessions.

During the request phase, before calling the view, Django applies middleware in the order it’s defined in MIDDLEWARE, top-down.

Default value created in settings.py:

*MIDDLEWARE = [  
 'django.middleware.security.SecurityMiddleware',  
 'django.contrib.sessions.middleware.SessionMiddleware',  
 'django.middleware.common.CommonMiddleware',  
 'django.middleware.csrf.CsrfViewMiddleware',  
 'django.contrib.auth.middleware.AuthenticationMiddleware',  
 'django.contrib.messages.middleware.MessageMiddleware',  
 'django.middleware.clickjacking.XFrameOptionsMiddleware',  
]*

## **Activating middleware**[**¶**](https://docs.djangoproject.com/en/1.10/topics/http/middleware/#activating-middleware)

* **Middleware order and layering**[**¶**](https://docs.djangoproject.com/en/1.10/topics/http/middleware/#middleware-order-and-layering)

## **Other middleware hooks[¶](https://docs.djangoproject.com/en/1.10/topics/http/middleware/#other-middleware-hooks)**

* [Built-in middleware classes](https://docs.djangoproject.com/en/1.10/ref/middleware/) reference link

**Django settings**

A Django settings file contains all the configuration of your Django installation. This document explains how settings work and which settings are available.

* [Overview](https://docs.djangoproject.com/en/1.10/ref/settings/) of settings Reference Link
* Full list of settings includes:
  + [Core Settings](https://docs.djangoproject.com/en/1.10/ref/settings/#core-settings)
  + [Auth](https://docs.djangoproject.com/en/1.10/ref/settings/#auth)
  + [Messages](https://docs.djangoproject.com/en/1.10/ref/settings/#messages)
  + [Sessions](https://docs.djangoproject.com/en/1.10/ref/settings/#sessions)
  + [Sites](https://docs.djangoproject.com/en/1.10/ref/settings/#sites)
  + [Static Files](https://docs.djangoproject.com/en/1.10/ref/settings/#static-files)
  + [Core Settings Topical Index](https://docs.djangoproject.com/en/1.10/ref/settings/#core-settings-topical-index)

**Relationships**

**What are the automatically generated admin features and how much customizable - TBD**

[Reference](https://docs.djangoproject.com/en/1.10/topics/db/examples/) Link

## **Create the Django Admin**

### **1) Creating an admin user[¶](https://docs.djangoproject.com/en/1.10/intro/tutorial02/#creating-an-admin-user)**

First we’ll need to create a user who can login to the admin site. Run the following command:

$ python manage.py createsuperuser

Enter your desired username and press enter.

Username: admin

You will then be prompted for your desired email address:

Email address: admin@example.com

The final step is to enter your password. You will be asked to enter your password twice, the second time as a confirmation of the first.

Password: \*\*\*\*\*\*\*\*\*\*  
Password (again): \*\*\*\*\*\*\*\*\*  
Superuser created successfully.

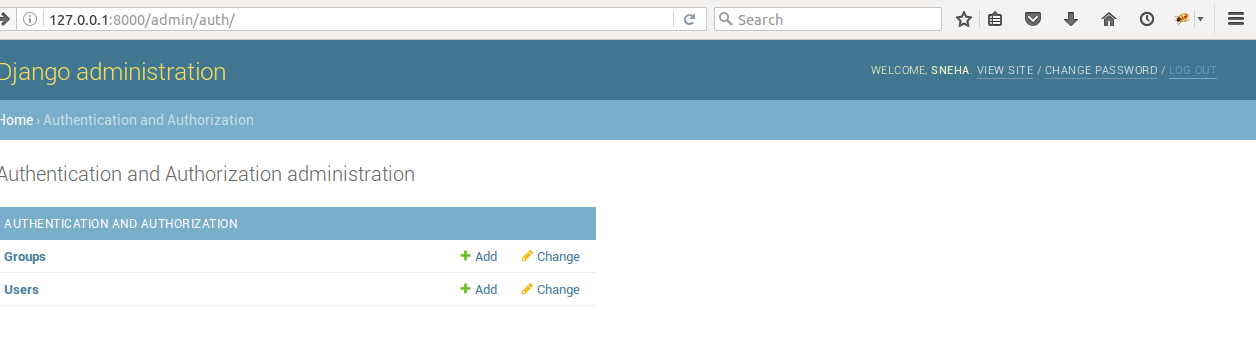
### **2) Start the development server[¶](https://docs.djangoproject.com/en/1.10/intro/tutorial02/#start-the-development-server)**

The Django admin site is activated by default. Let’s start the development server and explore it.

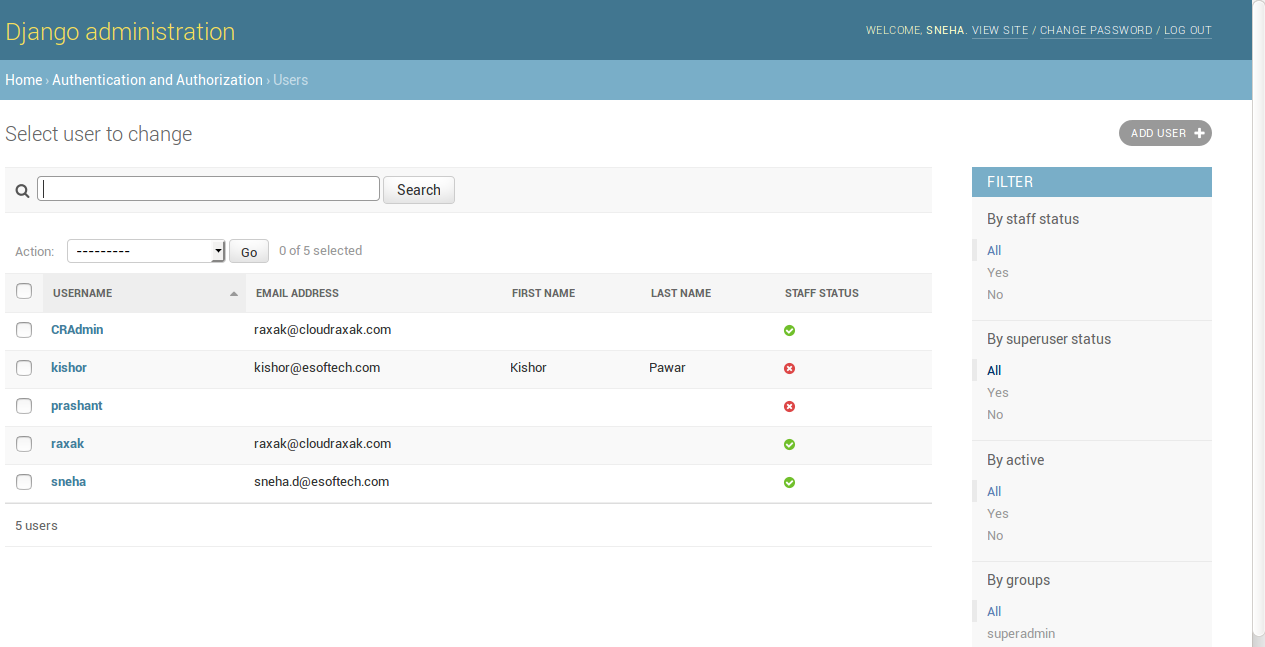
If the server is not running start it like so:

$ python manage.py runserver

Now, open a Web browser and go to “/admin/” on your local domain – e.g.,<http://127.0.0.1:8000/admin/>. You should see the admin’s login screen:



You can view all the registered users



Further information on enhancing the admin is mentioned [here](https://docs.djangoproject.com/en/1.10/intro/tutorial02/)

**3)** [**Admin Actions**](https://docs.djangoproject.com/en/1.10/ref/contrib/admin/actions/)  **reference link**

**4)** [**Admin documentation generator**](https://docs.djangoproject.com/en/1.10/ref/contrib/admin/admindocs/) **reference link**

**5)** [**Django Admin tools**](https://django-admin-tools.readthedocs.io/en/latest/) **reference link.**

**Automated Testing (low prio) (dl)**

[Reference](https://docs.djangoproject.com/en/1.10/topics/testing/) Links

**Internationalization and localization (low prio)**

**Porting to python 3 (low prio)**

**Upgrading django version. (low prio)**

## References:

<https://docs.djangoproject.com/en/1.10/intro/>

<http://djangobook.com/installing-django/>